Carl von Ossietzky Universität Oldenburg
Fakultät II - Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Cooperative Software Verification

Von der Fakultät für Informatik, Wirtschafts- und Rechtswissenschaften der Carl von Ossietzky Universität Oldenburg zur Erlangung des Grades und Titels eines

Doktor der Naturwissenschaften (Dr. rer. nat.)

angenommene Dissertation von

Jan Frederik Haltermann
geboren am 19.11.1994 in Herford

Gutachter
Prof. Dr. Heike Wehrheim,
Prof. RNDr. Jan Strejček, Ph.D.
Tag der Disputation:
21.02.2025

Abstract

Software plays an important role in our daily lives. To prove that a program fulfills specific requirements, *software verification* is used. There exist various approaches for software verification, implemented in verifiers, each having individual strengths and weaknesses. To combine their strengths, different concepts for combining them in a white-box manner have been developed.

In contrast to white-box integrations, *cooperative software verification* aims at combining existing components on a conceptual level, working loosely coupled together to solve a task. Consequently, components are used as black-boxes, exchanging information via verification artifacts. Thereby, integrating novel components and thus innovation becomes at best a configuration task. There already exist a few concepts for cooperative software verification, which demonstrate the advantages of black-box cooperation. In most of these approaches, the components are executed in a sequence, where each is using intermediate results computed by the prior component.

This thesis presents a systematic analysis of the effects of using cooperative software verification. To this end, we develop and evaluate three concepts for cooperative software verification: a sequential, a cyclic, and a parallel concept. The sequential concept, called CoVEGI, uses a verifier in cooperation with components that generate invariants. Invariant generation is a core task in software verification and using these externally generated invariants allows the verifier to solve tasks that could not be solved without cooperation. The second, cyclic concept for cooperative verification is based on the decomposition of the existing and widely used CEGAR scheme. In the decomposed version C-CEGAR, three stateless components communicate via existing verification artifacts. Additionally, we decompose a tightly coupled implementation of CEGAR and demonstrate its feasibility, and that using other off-the-shelf components allows for solving tasks that were not solvable before. Lastly, we propose the concept of ranged program analysis, a parallel composition of verifiers, that generalizes the idea of ranged symbolic execution to arbitrary verification approaches. Ranged program analysis divides the verification task using ranges, where each range can be solved in parallel by different verifiers. We develop two different methods for restricting the exploration of off-the-shelf analyses to a given range and show experimentally, that using two different verifiers increases the overall performance.

Zusammenfassung

Software spielt in unserem täglichen Leben eine wichtige Rolle. Um zu zeigen, dass ein Programm bestimmte Anforderungen erfüllt wird Softwareverifikation verwendet. Es existieren verschiedene Ansätze zur Softwareverifikation, implementiert in Verifizierern, wobei jeder Ansatz individuelle Stärken und Schwächen besitzt. Um die Stärken der einzelnen Ansätze zu kombinieren, wurden verschiedene Konzepte zur Kombination mithilfe von Whitebox-Integration entwickelt.

Das Ziel der *kooperativen Softwareverifikation* ist im Gegensatz zu Whitebox-Integrationen, existierende Komponenten auf einer konzeptionellen Ebenen zu kombinieren, damit diese im losen Verbund zusammen die Aufgabe lösen können. Dabei werden Komponenten konsequenterweise als Blackbox verwendet und die Informationen über Verifikationsartefakte austauscht. Dadurch wird die Integration von neuen Komponenten und somit auch von Innovation im Idealfall eine Konfigurationsaufgabe. Es gibt bereits erste Konzepte für kooperative Softwareverifikation, die die Vorteile der Blackbox-Kooperation demonstrieren. In den meisten dieser Ansätze werden die Komponenten sequentiell ausgeführt, wobei jede die Zwischenergebnisse der Vorgänger verwendet.

In dieser Arbeit wird eine systematische Analyse der Auswirkungen der Verwendung von kooperativer Softwareverifikation durchgeführt. Dafür entwickeln und evaluieren wir drei Konzepte der kooperativen Softwareverifikation: ein sequenzielles, ein zyklisches und ein paralleles. Im sequenziellen Konzept CoVEGI wird einen Verifizierer in Kombination mit Komponenten zur Invarianten-Generierung verwendet. Die Invarianten-Generierung ist eine der Kernaufgaben in der Softwareverifikation und die Verwendung von extern generierten Invarianten erlaubt den Verifizierer Aufgaben zu lösen, die ohne Kooperation nicht lösbar sind. Das zweite, zyklische Konzept für eine kooperative Softwareverifikation basiert auf der Zerlegung des existierend und weit verbreiteten CEGAR-Schemas. In der zerlegten Version C-CEGAR kommunizieren drei zustandslose Komponenten über existierenden Verifikationsartefakten. Darüber hinaus haben wir eine existierende Implementierung des CEGAR-Schemas zerlegt und damit sowohl die Machbarkeit von C-CEGAR demonstriert als auch dass die Nutzung von anderen Standardkomponenten das Lösen von vorher nicht lösbaren Aufgaben ermöglicht. Zuletzt haben wir das Konzept der Ranged Program Analysis vorgestellt, eine parallele Komposition von Verifizieren, das die Idee des Ranged Symbolic Execution für beliebige Standard-Verifizierer generalisiert. Dabei wird das Verifikationsproblem anhand von Ranges aufgeteilt, sodass die Teilprobleme parallel von verschiedenen Verifizieren gelöst werden können. Wir haben zwei Methoden entwickelt, um die Analyse eines Standard-Verifizier auf eine gegebene Range einzuschränken und gezeigt, dass die Kombination von verschiedenen Verifizieren die Performance signifikant erhöht.

Acknowledgment

When I started writing this part of my thesis, so many wonderful people came to my mind, who have helped me along the way. I want to express my deepest gratitude to each and every one of them Without them, this thesis would probably not have been done so easily.

First and foremost I want to thank my supervisor Heike Wehrheim. I am extremely grateful for the enormous support, guidance and patience I received during the time in your research group. You gave me room to follow my own ideas, supported me from the first day on when I started as a student assistant in 2018 and always provided valuable feedback, new insights or even a Plan B when needed. Thank you for giving me the opportunities to attend that many conferences, workshops and summer schools to discuss and promote my ideas.
I would like to thank my PhD committee members, Andreas Peter and Christian Schönberg, as well as to the reviewer of my thesis, Jan Strejček. Their valuable feedback and insightful comments have significantly improved the quality of this thesis.

I would also like to thank all the co-authors and collaborators I have worked with: Dirk Beyer and Thomas Lemberger for their cooperation in the DFG project "Cooperative Software Verification" and especially Thomas for the many and long discussions about C-CEGAR. I would also like to thank Marie-Christine Jakobs and Cedric Richter for their cooperation and support with ranged programs analysis. Cedric was always a fantastic partner to discuss ideas with during the whole time of my PhD.

I am also grateful to my colleagues Arnab, Cedric, Felix, Jürgen, Lara, ManueNjkL1l and Nicola. You all always had an open door and an open ear when I needed someone to ask for help and were a reason for the great office days in Paderborn and Oldenburg. In addition, you provided great feedback on the draft version of my thesis, which helped a lot to improve its quality.
This thesis would not have been possible without the support of my friends, those who stood by me during our bachelor and master studies in Paderborn and those who have supported me even longer. Thank you Andi, Felix, Jan, Laura, Marcel, Mirko, Nils and Timo.

Last but not least, I would like to thank my family, my parents and Bastian, for their support from the first moment I became interested in computer science and throughout my whole life, no matter what happened. And most importantly, I am really grateful to my loving wife Meike, who supported me throughout my entire PhD, especially when I was working in Oldenburg on weekdays or when some important deadlines were approaching.

Contents

1 Introduction

Software is used everywhere in our daily lives. It is not only executed on computers and mobile devices, software also controls rockets, self-driving cars, or all kinds of medical devices. Especially in these safety-critical systems, the correctness of software is crucial, as bugs can lead to critical issues, ranging from severe financial damage to personal injuries or death. In 1987, a race condition in the control software of the Therac-25, a computer-controlled radiation therapy machine for cancer, caused a massive overdose of radiation, killing at least three patients [LT93]. More recently, a bug in the app of an insulin pump caused the pump's battery to run out faster than expected and thereby stopping the patient's insulin treatment, which led to 224 injuries [FDA24]. In 1996, the Ariane 5 rocket started its self-destruction shortly after its launch, as the internal navigation system was stopped due to an overflow in a variable [Lio+96].

All prior examples demonstrate that even software in safety-critical systems contain bugs, which can remain undetected until the systems' productive use. Normally, bugs do not lead to such drastic consequences. In practice, developers often use testing methods to test the functionality of software and to identify bugs. Testing is very effective in finding bugs, however, it can never show their absence [Dij72]. In contrast, methods such as software verification aim at finding proofs for the correctness of the (software) system, that are valid for all possible executions of it [GS18; JM09]. In order to do so, a formal specification of the system's correctness is needed. Safety properties are often used for this purpose, for instance stating that certain error states are unreachable or postconditions are satisfied [CHVB18; JM09]. Given a formal specification, verification approaches work on a model of the system rather than computing the proofs directly. Such models are for example Büchi automata [Büc62], Kripke structures [Kri59] or Control-flow automata [ASU86; BHJM07] and can be constructed at any point in the development process, but are constructed in most cases based on existing code [CHV18]. As the verification approaches are working on models, they are applicable to different types of systems as operating systems and drivers [KMPZ09; BR02b; Man+12],

(industrial) applications and mobile apps [PNRR15; Pau23; Bau+21] or even hardware [SSS00; JKSC08; GS20]. Having the model and the specification at hand, the verification approach analyzes the full state space to guarantee that no execution of the system violates the safety property.

Within the last years, software verification has made enormous progress. New ideas and techniques as symbolic execution [Kin76; CS13], predicate abstraction [GS97], (bounded) model checking [CE82; JM09; Bie+03], k-induction [DHKR11], property-directed reachability [Bra11], or dataflow analyses [Kil73] are developed. In addition, some large companies like NASA [GMKC13], Facebook [Cal+15; OHe18], Amazon Web Services [Coo18], Google [Sad+18], or Airbus [Cou+05] use such techniques. The progress is also observable in annual competitions as SV-COMP [Bey24] or VeriFyThis [EHMU19], where so-called verifiers realizing these techniques compete. These competitions also show that existing verification approaches have individual strengths and weaknesses. There is no approach that is superior to all others. Hence, either an approach that fits the needs of the use case best is chosen or different approaches are combined. The latter idea is successfully employed in many approaches, e.g., [Roc+17; AABC21; Hus+17; DGH16; NKP18; AGC12; GD17; Nol+20], to either increase the number of correct results computed (*effectiveness*) or reduce the time taken to compute a solution (*efficiency*). The combinations employ two or more (conceptually different) approaches and combine their strengths by exchanging information or partial results or by dividing the task. Most existing combinations use a conceptual integration with multiple components. In these white-box integrations, information is exchanged using internal formats, method calls, or accessing shared data structures. Therefore, replacing one component with another is almost impossible without building a new integration.

1.1 The Idea of Cooperative Software Verification

In 2012, Beyer et al. [BHKW12] and Christakis et al. [CMW12] presented two concepts for black-box cooperation of verification tools. Both concepts are based on the same idea, namely that several components are executed as black-boxes in a sequence, where each reuses the partial verification result computed by the previous tools. The information computed is exchanged using *verification artifacts* that can be exchanged among the black-box components. Hence, their ideas combine the strengths of different components on a conceptual level, cooperating loosely coupled to solve a task. As there are multiple approaches based on these two ideas [CJW15; BJLW18; BJ20; CMW16], they can be seen as a starting point for black-box cooperation in software verification. The ultimate goal of such black-box cooperation is to make the integration of new concepts, and thus innovation, a configuration task, by avoiding strong cohesion between existing components. In 2020, Beyer and Wehrheim termed this idea as *cooperative verification* [BW20]. Following them, we state four requirements for cooperative software verification approaches in Characterization 1.1.

Characterization 1.1. *A verification approach is called* cooperative *if (1) there are at least two identifiable actors that (2) solve the verification task together by (3) exchanging information using verification artifacts, and (4) most actors are used off the shelf.*

We call any component that participates in the process of computing a solution an *actor*. Using an actor off-the-shelf means that it is not modified for the specific cooperative approach. A *verification artifact* is a standardized format that allows encoding information and is exchangeable among different actors. At best, all off-the-shelf actors are able to use the verification artifacts as inputs or generate them as output.

In contrast to the significant number of approaches using a conceptual integration, it turns out that there are only a few other cooperative approaches according to Characterization 1.1. Most approaches are sequential and based on the ideas of [BHKW12; CMW12], or are tailored to a specific domain, e.g., Android app analysis [Pau23]. Consequently, there is currently a research gap regarding cooperative software verification, as it remains unclear whether ideas successfully used within non-cooperative combinations are also suited for being used in a cooperative way, how such cooperative concepts look like and if we can develop novel concepts. Moreover, there is no systematic evaluation of the positive and potential negative effects when using cooperative software verification. In this thesis, we thus address the following high-level research question:

Research Question

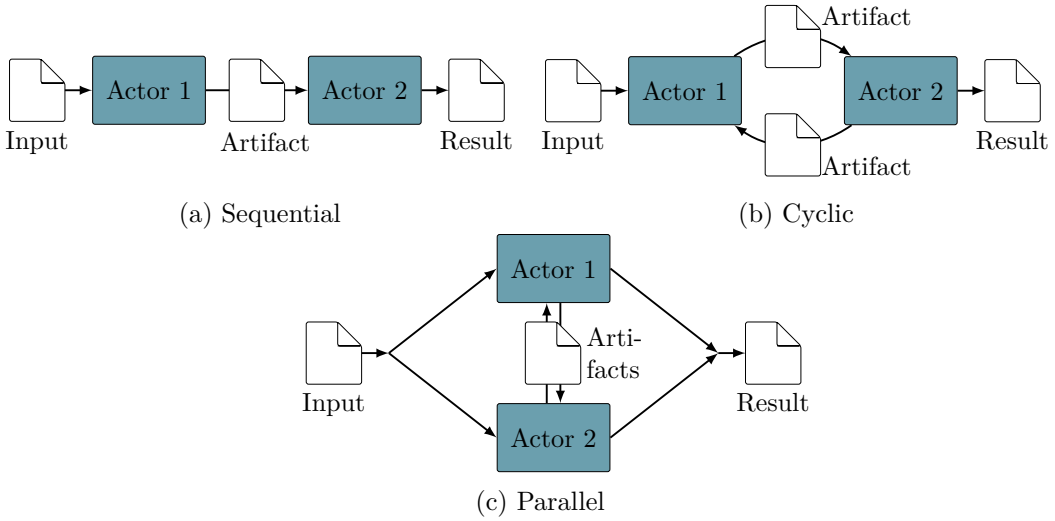
What are potential forms of cooperative software verification and is it beneficial using them?

1.2 Contribution of this Thesis and Outline

To answer this question systematically, we first have a look at potential forms of cooperation. In general, we can differentiate three fundamental ways of establishing cooperation between actors: sequential, cyclical, or parallel. The three forms are depicted in Figure 1.1 on a conceptual level, with actors colored in blue.

In order to compute a proof for the correctness of the system, software verification approaches have to analyze the full state space of the model of the system. As the state space is usually large or even infinite, we develop three different concepts for cooperative software verification throughout this thesis, aiming to simplify the analysis of the state space. Within the sequential concept, actors cooperate using invariants, which abstract parts of the state space and thereby reduce it. The cyclic concept aims to generate a suitable abstraction, where different actors cooperate through solving different subtasks and exchanging the results. The parallel concept allows for dividing the state space into disjoint parts, such that each can be analyzed independently and in parallel. We realize and evaluate all three concepts using seven off-the-shelf tools.

In Chapter 2 of the thesis, we explain the fundamental concepts of software verification used in this thesis, especially defining the verification task that needs to be



(a) Sequential (b) Cyclic

(c) Parallel

Figure 1.1: Three different ways to build cooperative verification approaches

solved, and introducing existing concepts for software verification and artifacts. Thereafter, we start in Chapter 3 the systematic analysis of cooperative software verification. First, we define a novel concept for sequential cooperative software verification. In contrast to the two sequential ideas proposed by Beyer et al. [BHKW12] and Christakis et al. [CMW12], we develop a novel concept, called Cooperative Verification via Externally Generated Invariants (CoVEGI), that does not exchange intermediate verification results but allows for delegating one core task of verification, namely invariant generation. CoVEGI uses a verifier in cooperation with a component that generates invariants. In order to compute such invariants, we additionally present the Modular Framework for Invariant Generation using Machine Learning (MIGml), which allows for the construction of tools for invariant generation with machine learning and enables the comparison of existing ones on equal grounds.

In Chapter 4, we focus on a cyclic cooperative verification. In contrast to the first concept, we do not propose a novel form of cooperation. Instead, we follow the idea of decomposing existing schemes of software verification to make them ready to be used in cooperative settings. To demonstrate and evaluate the feasibility of such decomposition, we exemplarily decompose the Counterexample-Guided Abstraction Refinement (CEGAR) scheme. Furthermore, we also have a closer look at existing verification artifacts, particularly discussing if one artifact with fixed semantics exists that is usable in different scenarios of cooperative software verification. As a result, we develop the Generalized Information Exchange Automaton (GIA).

In Chapter 5, we develop a parallel approach for cooperative verification. The main challenge in parallel cooperation is to split the task among the actors, such that the resulting subtasks can be executed in parallel. Our approach, called ranged program analysis, uses a divide-and-conquer approach and generalizes the idea of ranged symbolic execution. It allows the division of the verification task such that the actors can

solve the assigned subtasks in parallel and the intermediate results are joined to a final answer. We present two methods based on Configurable Program Analyses and instrumentation to ensure that off-the-shelf tools only work on the assigned subtasks.
In each chapter, we experimentally evaluate the concepts and compare them to the non-cooperative actors to demonstrate that cooperative verification is beneficial. We conclude this thesis by summarizing and discussing the results and providing an outlook for future work in Chapter 6.

1.3 Publication Details and Personal Contribution

The ideas and most results presented in this thesis are based on articles published in conference proceedings or in journals. All publications are authored in cooperation with my supervisor Heike Wehrheim and partially with other colleagues. We also submitted artifacts, in case the conference accepted or required a submission, and additionally archived all experimental data and replication packages at Zenodo for all publications.

The ideas presented in Chapter 3 are based on two publications. The concept of CoVEGI was first published in 2021 at FASE [HW21a] (artifact [HW21b]), and MIGml was first published in 2022 at ICST [HW22b] (artifact [HW21c]). Both publications are co-authored with Heike Wehrheim and Jan Haltermann is the main author. In the same year, we published a paper on decomposition in software verification yielding the concept of cyclic cooperation, which is applied to CEGAR, as discussed in Chapter 4 at ICSE [BHLW22a] (reusable artifact [BHLW22b]). The publication is authored in cooperation with Dirk Beyer, Thomas Lemberger, and Heike Wehrheim, where the concept, implementation, and evaluation were developed and conducted in close cooperation with Thomas Lemberger, where both authors contributed equally. The idea of GIAs, the novel verification artifact, is based on a publication at SEFM [HW22a] and an article in the Journal on Software and Systems Modeling [HW24] (unified artifact [HW23]). Both publications are co-authored with Heike Wehrheim and Jan Haltermann is the main author.

The ideas presented in Chapter 5 are based on three different publications. In 2023, we published a paper introducing the core ideas and the CPA-based approach at FASE [HJRW23b] (reusable artifact [HJRW23a]). Thereafter, we published further ideas, including the concepts of work stealing in the journal Science of Computer Programming [HJRW24b] (artifact [HJRW24c]). The instrumentation-based approach was published at SEFM [HJRW23c] in 2023 (reusable artifact [HJRW23d]). All three publications are authored in cooperation with Marie-Christine Jakobs, Cedric Richter, and Heike Wehrheim, and Jan Haltermann is the first author. Different implementation aspects were divided among Marie-Christine Jakobs, Cedric Richter, and Jan Haltermann.

2 Fundamentals

Before developing different forms of cooperation for software verification, we set the stage by defining the *verification task* that needs to be solved. The goal of software verification is to determine whether a given program satisfies the predefined correctness property. As conducting these proofs by hand is time-consuming and error-prone, automatic tools, so-called *verifiers* are employed.

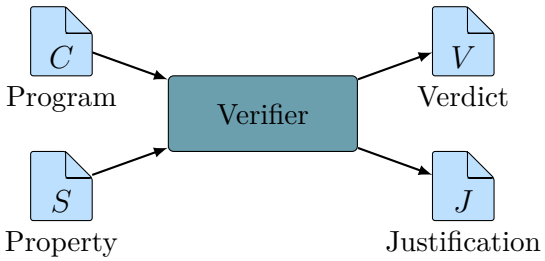


Figure 2.1: Conceptual view of a verifier

As shown in Figure 2.1, a *verifier* receives a program C and the property S as input and computes a verdict V. It may also provide a justification J for this verdict, ideally in the form of a standardized verification artifact. In this chapter, we define and explain all elements in Figure 2.1 from left to right: We first formalize the syntax and semantics of programs as well as the property to be verified in Section 2.1. We then provide a brief overview of existing verification techniques that are used in the cooperative verification approaches presented in this thesis in Section 2.2. Thereafter, we present existing verification artifacts that serve as justification like correctness and violation witnesses, and that are usable for the information exchange among actors in a cooperative setting in Section 2.3. Next, we introduce existing verification tools that are used as actors in cooperative verification concepts in Section 2.4. We conclude this chapter by introducing a tool for building cooperative verification approaches and describing the methodology used for benchmarking the performance of the cooperative approaches in Section 2.5.

2.1 Programs and Properties

Within this thesis, we focus on the verification of programs written in GNU C. We first define the syntax and semantics for the programs and then provide a formal definition of the safety properties used for verification.

2.1.1 Program Syntax and Semantics

For representation purposes only, we will introduce a simplified While programming language using only bounded integer variables. All implementations and concepts presented in this thesis cover a rich subset of C programs. Figure 2.2 shows the abstract syntax, which is based on the formalization given in [NNH99]:

$$
\begin{aligned}
P &:= \ \mathsf{type}\ \mathsf{NAME}((\mathsf{type}\ x)^*)\{S\}\ (\mathsf{type}\ \mathsf{func}((\mathsf{type}\ x)^*)\{S\})^* \\
S &:= \ assign \mid assume \mid S_1;\ S_2 \mid \texttt{skip} \\
assign &:= \ \mathsf{type}\ x = a \mid x = a \mid x = \mathsf{func}(a^*) \mid \texttt{return}\ x \\
assume &:= \ \texttt{if}(b)\ \{S_1\}\ \texttt{else}\ \{S_2\} \mid \texttt{if}(b)\ \{S\} \mid \texttt{while}(b)\ \{S\}
\end{aligned}
$$

Figure 2.2: Abstract syntax of the While programming language

For the syntax, we use the set of all arithmetic expressions *AExpr* with $a \in AExpr$, the set of all boolean expressions *BExpr* with $b \in BExpr$, and the set of all program variables *Var* with $x \in Var$. The type is a keyword for an arithmetic integer datatype in C (`char, short, int, long`) which can additionally be `unsigned`. The language contains calls to functions with an arbitrary number of arguments via func, where the arguments are assigned to the parameters x. NAME is the name of the main function of the program P. S is a sequence of statements, an assignment *assign*, an assume statement *assume* or an `skip` statement. An assignment *assign* is a variable declaration, a variable assignment, a function call assignment, or a return statement, and an assume statement *assume* is either an if statement[1] or a `while` loop. We denote by *Assign* the set of all possible assignment statements. We use the set *Ops* to denote all operations of the program, more formally we define $Ops = BExpr \cup Assign$. Note that programs generated by this abstract syntax are deterministic except for the input. It is also possible to model non-deterministic behavior caused by user input using specific functions such as `rand()` or `nondet()`.

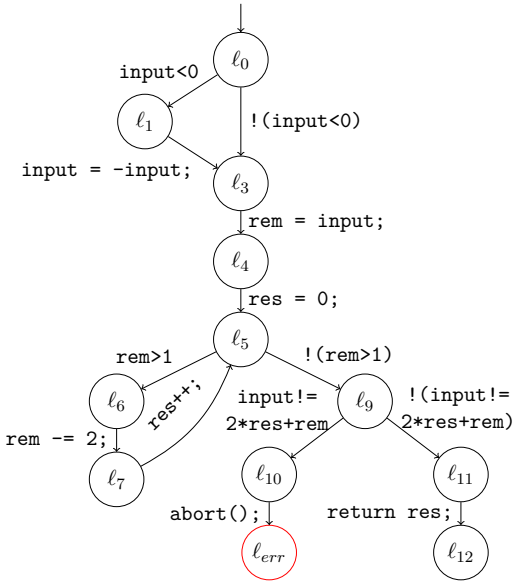
An example of a program generated by the abstract syntax is given in Figure 2.3a. It divides the absolute value of the input value *input* by two and calculates the reminder using only subtraction and addition. The program also contains a call to the function `abort();` in line 10., that is treated as an error location.

Most verification techniques represent programs as automata. We use a Control-Flow automaton [ASU86; BHJM07] to model programs, an automaton in which edges are labeled with program statements:

[1] Note that $\texttt{if}(b)\ \{S_1\}$ is short for $\texttt{if}(b)\ \{S_1\}\ \texttt{else}\ \{\texttt{skip}\}$.

```
   int div2WithReminderAbs(short input){
0      if (input < 0){
1          input = -input;
2      }
3      int rem = input;
4      int res = 0;
5      while (rem > 1) {
6          rem -= 2;
7          res++;
8      }
9      if(input != 2*res + rem)
10         abort();
11     return res;
12 }
```

(a) The running example program



(b) The CFA C_1 for the running example

Figure 2.3: The running example for the thesis, a program that computes the division by two with a reminder for the absolute of given input using only addition and subtraction

Definition 2.1. *A* Control-Flow automaton (CFA) $C = (L, \ell_0, G)$ *is an automaton, where* $L \subseteq Loc$ *is a finite set of location names, a subset from all possible locations* Loc*. The CFA has the initial location* $\ell_0 \in L$ *and* $G \subseteq L \times Ops \times L$ *is the set of control flow edges. A control flow edge* (ℓ_i, g_i, ℓ_j) *describes that the execution of statement* g_i *at location* ℓ_i *leads to location* ℓ_j*.*

We denote by $\mathcal{C}$ the set of all CFAs. We present the CFA C_1 for our running example from Figure 2.3a in Figure 2.3b. Throughout this thesis we generate a CFA for a program P. Thus, the CFA is deterministic, i.e., $\forall(\ell_i, g_i, \ell_j), (\ell_i, g_i, \ell'_j) \in G : \ell_j = \ell'_j$, and that branching in the CFA occur only at edges labeled with assume statements. Hence, we can construct during the transformation an indicator function $B_C : G \to \{T, F, N\}$ for a CFA C that indicates for each statement $g \in G$ if g is not an assume statement (N), is an assume statement that represents the true evaluation of the boolean condition (T) or the false evaluation (F). In the CFA C_1 shown in Figure 2.3b, we have for example $B_{C_1}(\texttt{rem>1}) = T$, $B_{C_1}(\texttt{!(rem>1)}) = F$, and $B_{C_1}(\texttt{res=0;}) = N$.

After having defined the syntax for programs, we need to define their semantics. First, we define a *state* σ as a mapping from the integer variables to integers, i.e., $\sigma : Var \to \mathbb{Z}$. We can lift the definition of a state to also contain the evaluation of arithmetic and boolean expressions so that σ maps $AExpr$ to $\mathbb{Z}$ and $BExpr$ to $\mathbb{B}$, which is the boolean domain. The set of all possible states is denoted by Σ. The combination of state and program location $\langle \ell, \sigma \rangle$ is called *program state*. A *program path* $\pi = \langle \ell_0, \sigma_0 \rangle \xrightarrow{g_1} \langle \ell_1, \sigma_1 \rangle \dots \xrightarrow{g_n} \langle \ell_n, \sigma_n \rangle$ of a CFA C is a sequence of program states and operations, such that for each transition $\langle \ell_{i-1}, \sigma_{i-1} \rangle \xrightarrow{g_i} \langle \ell_i, \sigma_i \rangle$ the condition $(\ell_{i-1}, g_i, \ell_i) \in G$ holds and that the path π starts at ℓ_0. The initial state σ_0 assigns the

$$\frac{g_i \in \mathit{BExpr} \qquad \sigma_{i-1} \models g_i \qquad \sigma_i = \sigma_{i-1}}{\langle \ell_{i-1}, \sigma_{i-1} \rangle \xrightarrow{g_i} \langle \ell_i, \sigma_i \rangle} \tag{2.1}$$

$$\frac{g_i \in \mathit{Assign} \setminus \{\texttt{return } x, x = \mathsf{func}(a^*)\} \qquad g_i \equiv x = a \qquad \sigma_i = \sigma_{i-1}[x \mapsto \sigma_{i-1}(a)]}{\langle \ell_{i-1}, \sigma_{i-1} \rangle \xrightarrow{g_i} \langle \ell_i, \sigma_i \rangle} \tag{2.2}$$

$$\frac{\begin{array}{l}\sigma_i = \sigma_{i-1}[x_1 \mapsto \sigma_{i-1}(a_1), \ldots, x_n \mapsto \sigma_{i-1}(a_n)]^2 \qquad \sigma_{m+1} = \sigma_m[x \mapsto \sigma_m(x')] \\ g_{i+1}, \ldots, g_m \in \mathit{BExpr} \cup (\mathit{Assign} \setminus \{\texttt{return } y, y = \mathsf{func}(a^*)\}) \\ \langle \ell_i, \sigma_i \rangle \xrightarrow{g_{i+1}} \langle \ell_{i+1}, \sigma_{i+1} \rangle, \ldots, \langle \ell_{m-1}, \sigma_{m-1} \rangle \xrightarrow{g_m} \langle \ell_m, \sigma_m \rangle \text{ adhere to (2.1),(2.2)}\end{array}}{\langle \ell_{i-1}, \sigma_{i-1} \rangle \xrightarrow{x = \mathsf{func}(a_1, \ldots, a_n)} \langle \ell_i, \sigma_i \rangle \xrightarrow{g_{i+1}} \ldots \xrightarrow{g_m} \langle \ell_m, \sigma_m \rangle \xrightarrow{\texttt{return } x'} \langle \ell_{m+1}, \sigma_{m+1} \rangle} \tag{2.3}$$

Figure 2.4: The operational semantics for transitions

input values to the input variables and 0 to all other variables.

Next, we define the semantics of a program path. The formal definition of the operational semantics is given in Figure 2.4. A transition adheres to the semantics, if the state changes adhere to these rules. Intuitively, for a transition $\langle \ell_{i-1}, \sigma_{i-1} \rangle \xrightarrow{g_i} \langle \ell_i, \sigma_i \rangle$ where g_i is a boolean condition, σ_{i-1} has to satisfy the condition (2.1). For assignments, the value of a variable x in σ_i has to be equal to the value of the expression a on the right-hand side in σ_{i-1}, cf. (2.2). The notation $\sigma[x \mapsto c]$ means that the value of x is updated to c in σ and all other values stay unchanged. For function calls, we consider all statements between the function call and the return statement. The values of the arguments $a_1, \ldots, a_n$ in σ_{i-1} given as input to the function func has to be equal to its parameters $x_1, \ldots, x_n$. Moreover, the value of x' in σ_{m-1} returned by func has to be equal to x, the variable it is assigned to, cf. (2.3). We call a path π *feasible* if every transition adheres to the semantics. Otherwise, we call it *infeasible*. We denote the set of all feasible program paths of a CFA C by $\mathit{paths}(C)$. If there is no feasible program path leading to a certain location $\ell \in L$, we call ℓ *unreachable*, and otherwise *reachable*.

2.1.2 Safety Properties for Programs

After having defined the syntax and semantics of the programs, we can define a correctness criterion. According to Lamport, safety properties ensure that nothing bad happens [Lam77] and the Handbook of Model Checking defines safety in terms of (non-)reachability of certain error states [CHVB18]. We, and many other approaches, follow this idea:

Definition 2.2. *A* safety property $S = (\ell, \omega)$ *is a pair of location* $\ell \in L$ *and a boolean formula* $\omega \in \mathit{BExpr}$*, where* ω *has to hold in all program states reaching* ℓ*.*

Thus, a CFA C *violates* S, if there exists a feasible program path $\pi = \langle \ell_0, \sigma_0 \rangle \xrightarrow{g_1} \langle \ell_1, \sigma_1 \rangle \ldots \xrightarrow{g_i} \langle \ell_i, \sigma_i \rangle \xrightarrow{g_{i+1}} \ldots$, such that $\ell_i = \ell$ and $\sigma_i \not\models \omega$. A CFA can have multiple safety properties. Note that we are interested only in the partial correctness

[2] where $x_1, \ldots, x_n$ are the parameters of func

of a program since we focus on checking the violation of the safety property from Definition 2.2 and do not consider termination.

In our running example, we use the safety property $S = (\ell_{err}, \mathit{false})$, i.e., the program violates S when the function `abort()` in line 10 is called. Thus, verifying that a CFA satisfies the safety property S is equivalent to checking the reachability of the *error location* ℓ_{err}. For verifiers working with assertions, we can replace lines 9 and 10 with the function call `assert(input == 2*res + rem)`. This encoding results in an equivalent safety property $S' = (\ell_9, \mathit{input} = 2 * \mathit{res} + \mathit{rem})$.

2.1.3 SSA and Strongest Postcondition

When analyzing a program, keeping track of the variable values at different points in the program is needed. To ease that, we usually transform the program first in the so-called *Single Static Assignment (SSA) form* (see [Cyt+91]). In a program in SSA form each variable is equipped with an index. The indices are increased if a variable is (re-)assigned. Whenever the control flow of the program branches, e.g., for if statements, there are nodes in the CFA having multiple ingoing edges. In case a variable is assigned a new value within the if- or the else-block, two different SSA indices may be valid for the variable afterward, depending on which block is executed. To ensure that all SSA indices that are valid on some paths reaching a location are correctly merged, *ϕ-nodes* are introduced. A ϕ-node unifies all indexed variables defined on the paths leading to the location by reassigning the variable. We present in Figure 2.5 the CFA C_1' of Figure 2.3a in SSA form. Our example contains three ϕ-nodes: The first unifies the indices of input_0 and input_1 after the if-statement in line 0. The other two ϕ-nodes unify the indices of the variables used before the loop and within the loop body.

To be able to encode the effect of the operations from Figure 2.2 on a boolean formula, we use the *strongest postcondition* [DS90]. It is used in verification techniques and we employ the strongest postcondition for defining loop invariants. Intuitively, it models the effect of executing an operation g on a formula φ.

Definition 2.3. *The* strongest postcondition operation $sp : \mathit{Ops} \times \mathit{BExpr} \to \mathit{BExpr}$ *is defined for non-loop statements of the abstract syntax given in Figure 2.2 as follows, assuming that the program is transformed in SSA form:*

$$
\begin{aligned}
sp(S_1; S_2, \varphi) &\equiv sp(S_2, sp(S_1, \varphi)) \\
sp(x_i = a, \varphi) &\equiv \varphi \wedge x_i = a \\
sp(x_i = \phi(x_j, x_k), \varphi) &\equiv \varphi[(x_j{=}a) \mapsto (x_i{=}x_j \wedge x_j{=}a), (x_k{=}a) \mapsto (x_i{=}x_k \wedge x_k{=}a)] \\
sp(\texttt{if}\ (b)\ \{S_1\}\ \texttt{else}\ \{S_2\}, \varphi) &\equiv (b \wedge sp(S_1, \varphi)) \vee (\neg b \wedge sp(S_2, \varphi)) \\
sp(\texttt{return}\ x_i, \varphi) &\equiv \varphi \wedge \mathit{retVal}_{\mathsf{func}} = x_i \\
sp(x_i = \mathsf{func}(a_1, \ldots, a_n)\{S\}, \varphi) &\equiv \varphi \wedge x_1 = a_1 \wedge \ldots \wedge x_n = a_n \wedge sp(S) \wedge\ x_i = \mathit{retVal}_{\mathsf{func}} \\
&\quad (\textit{where } \mathit{retVal}_{\mathsf{func}} \textit{ is the value returned by } \texttt{func} \\
&\quad \textit{and } \texttt{func} \textit{ is defined as } \texttt{func}(x_1, \ldots, x_n))
\end{aligned}
$$

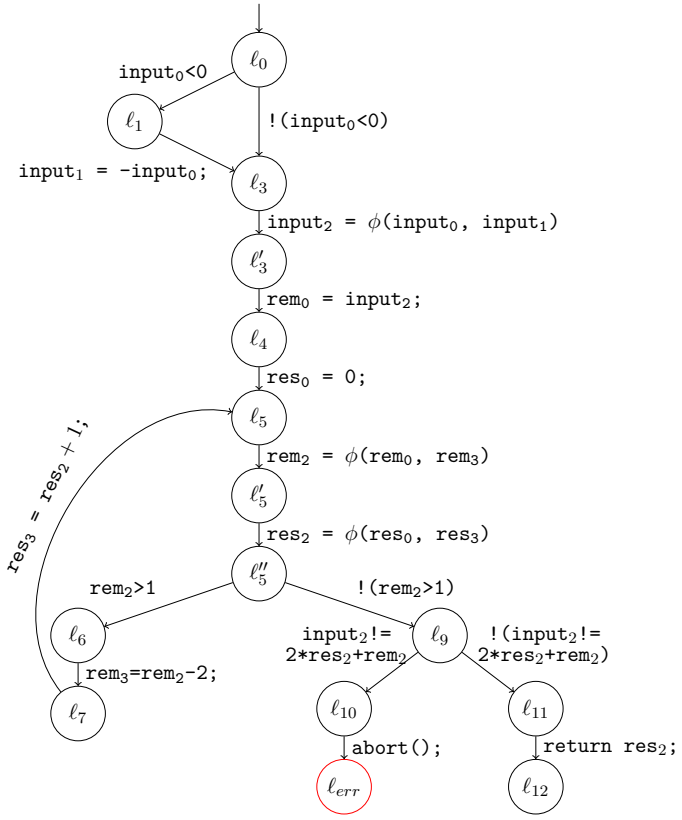


Figure 2.5: The CFA C'_1 for the program from Figure 2.3a in SSA form

Next, we define the strongest postcondition for loops. One option could be to dynamically unroll the loop and "move the next loop iteration before the loop" using:

$$\texttt{while}(b)\ \{S_1\}; S_2 \quad \equiv \quad \texttt{if}(b)\ \{S_1;\ \texttt{while}(b)\ \{S_1\}\ \}\ \texttt{else}\ \{S_2\}$$

Then, we could apply the *sp*operation for if-statements. Nevertheless, such a transformation causes two problems. The resulting program may be infinitely long and each transformation requires to re-compute the SSA form for the program. Thus, we use a different approach and define the strongest postcondition operation for loops using loop invariants, that we introduce next.

2.1.4 Loop Invariants

When verifying a program, loops can hamper verification, because the CFA becomes cyclic and the number of paths becomes large or infinite. A loop invariant summarizes the effect of a loop and is therefore helpful for verifying a program. We first define loop invariants for programs with a single loop and later on generalize the definition.

Definition 2.4. *A formula inv is called a* valid loop invariant *(or just* loop invariant*), if for a program in SSA of the form* $\texttt{S}_1$; `while(b){S}`; $\texttt{S}_2$ *with a single loop the following*

$$\begin{aligned} sp(S_1, true) \Rightarrow inv_{\text{SSA}(S_1)} \equiv & (((input_0 < 0 \wedge input_1 = -input_0 \wedge input_2 = input_1) \vee \\ & (input_0 \geq 0 \wedge input_2 = input_0)) \wedge \\ & rem_0 = input_2 \wedge res_0 = 0) \Rightarrow \\ & input_2 = rem_0 + 2 * res_0 \end{aligned}$$

$$\begin{aligned} sp(S, b \wedge inv_{\text{SSA}(S_1)}) \Rightarrow inv_{\text{SSA}(S)} \equiv & (input_2 = rem_2 + 2 * res_2 \wedge rem_2 > 1) \wedge \\ & (rem_3 = rem_2 - 2 \wedge res_3 = res_2 + 1) \Rightarrow \\ & input_2 = rem_3 + 2 * res_3 \end{aligned}$$

Figure 2.6: Establishment and preservation condition for the loop of C_1

two conditions are fulfilled:

$$sp(S_1, true) \Rightarrow inv_{SSA(S_1)} \tag{2.4}$$

$$sp(S, b \wedge inv_{SSA(S_1)}) \Rightarrow inv_{SSA(S)}, \tag{2.5}$$

where $inv_{SSA(S)}$ is an instantiation the variables in inv with the index of the last variable assignments in S. We call (2.4) *the establishment and* (2.5) *the preservation condition.*

A candidate loop invariant for the loop in C_1' in a non SSA form is $inv \equiv input = rem + 2 * res$. For checking if inv is in fact a valid loop invariant for the loop of C_1' in Figure 2.5, we have to check (2.4) and (2.5) instantiated for C_1' as shown in Figure 2.6. As inv fulfills both conditions, it is a valid loop invariant. Since *true* always satisfies (2.4) and (2.5), it is always a valid loop invariant and called *trivial* loop invariant.

Now, having the definition of a valid loop invariant at hand, we use the following rule for the strongest postcondition for loops in SSA form by abstracting its behavior with a valid loop invariant inv:

$$sp_{inv}(\texttt{while}(b)\ \{S\}, \varphi) \ \equiv \ \varphi \wedge \neg b_{\text{SSA}(S_1)} \wedge inv_{\text{SSA}(S_1)}, \tag{2.6}$$

where S_1 is the part of the program prior of the loop. In the example C_1', the variable indices used in $inv_{\text{SSA}(S_1)}$ are $\texttt{rem}_2$ and $\texttt{res}_2$. Because of the SSA form of the program, we can also keep all φ's which hold somewhere in the program within the formula for the strongest postcondition.

For programs with several consecutive loops, we check the validity of loop invariants in the order the loops appear in the CFA. We first check establishment and preservation for the first loop in the CFA and, if inv is valid, use (2.6) to replace the first loop. Now, the paths from the initial location of the CFA to the second loop do not contain a loop and we can check establishment and preservation for the second loop. For nested loops, we first check the validity of the loop invariant for the innermost loop. We use the loop-free paths from the initial location of the CFA to the innermost loop to check

establishment and check preservation for the acyclic loop body. If the loop invariant for the innermost loop is valid, we use (2.6) to replace the loop and repeat the process, until the validity of the outermost loop's invariant is checked.

As loop invariants are used to ease the verification problem, we need a way to classify their quality. Intuitively, invariants forming a good abstraction of the loop and being sufficient to prove the program safe with respect to the verification problem are helpful. To define this property formally, we use a program of the form $\texttt{S}_1$; `while(b){S}`; $\texttt{S}_2$; `assert`(ω) with safety property $S = (\ell_{err}, \omega)$, where ℓ_{err} is the assert statement after S_2.

Definition 2.5. *A valid loop invariant inv is a* helpful loop invariant *(sometimes also* safety invariants *[McM06]), if*

$$sp(S_2, sp_{inv}(\texttt{while}(b)\ \{S\}, sp(S_1, true))) \Rightarrow \omega_{SSA(S_2)} \tag{2.7}$$

holds. We call (2.7) *the check condition.*

For programs with several consecutive loops, we check if the combination of all loop invariants for the program is helpful by replacing all loops using sp_{inv}. For programs with nested loops, it suffices to check if the loop invariant for the outermost loop is helpful, as the loop body of the outermost loop and thus all inner loops are replaced by the invariant for the outermost loop.

The definition of loop invariant used ensures that all invariants are 1-inductive, meaning that the preservation condition takes a single execution of the loop body into account, and establishment ensures that the 1-inductive loop invariant holds before the first loop iteration. There are programs where finding a 1-inductive loop invariant that is a good summary of the loop is hard. In contrast, finding a *k-inductive loop invariant* may be easier. A k-inductive loop invariant is a generalization of a 1-inductive invariant. For establishment, we take the path leading to the loop and the first $k-1$ loop iterations into account, for preservation we take k loop iterations into account. For example, finding a good 2-inductive loop invariant for a function computing the Fibonacci numbers is easier than finding a 1-inductive one, as explained in [Wah13].

2.2 Program Verification Techniques

In 1953, Rice showed that all non-trivial semantic properties of programs are in general undecidable [Ric53], meaning that program verification is also undecidable in general. Nevertheless, for specific instances of program and property, a solution can be computed [JM09]. Today, there is a plethora of different verification approaches that can be employed for solving a given verification task. Most of them are based on some fundamental underlying technique such as model checking [CE82; JM09], abstract interpretation [CC77] or dataflow analyses [Kil73].

Since the analysis of exactly all feasible paths of a program is a challenging task, program verification techniques make use of approximation. More precisely, they compute and analyze either an *over-approximation* or an *under-approximation* of the set of feasible program paths. Over-approximation here means that at least all feasible paths are analyzed, thus locations identified as unreachable in the abstraction are unreachable in the program. Locations marked as reachable may still be unreachable, because the over-approximation may be too coarse. Hence, an over-approximating verifier can correctly prove a program safe but may generate infeasible or spurious counterexamples. In contrast, under-approximating techniques analyze at most all feasible paths. Thus, every property violation reported is a real counterexample.

In general, verification techniques compute analysis information that is associated with program locations of the CFA. There may be multiple elements for a single location, especially if the location is reachable on multiple paths. This analysis information depends on the type of verifier. As we are interested in finding out whether certain program locations, especially error locations, are reachable, the computed analysis information is used to answer these questions. Thus, most program verification techniques compute an over-approximation of the set of all reachable program locations, whereas approaches like testing compute an under-approximation.

As we repeatedly make use of certain verification techniques as actors within the cooperative concepts that we develop in this thesis, we provide a brief overview by introducing value analysis, symbolic execution, predicate abstraction, Bounded Model Checking, and k-induction. In advance, we explain the concept of CEGAR, a scheme used for finding a suitable abstraction. At the end of the section, we introduce Abstract Reachability Graphs, a way to store the information computed by the verification approaches. For a more recent and more comprehensive overview of existing tools and techniques, we refer the reader to recent surveys [BP22; Bal+18].

2.2.1 Counterexample-Guided Abstraction Refinement

For techniques that make use of abstract interpretation, i.e., computing an abstraction of concrete program states, finding a suitable abstraction that is precise enough to prove the program correct is a challenging task. Therefore, the Counterexample-Guided Abstraction Refinement (CEGAR) [Cla+00; Cla+03] scheme is used to iteratively compute a suitable abstraction for proving the correctness of a program.
It consists of the three steps *Abstract Model Exploration*, *Feasibility Check* and *Precision Refinement* that are visualized in Figure 2.7. Starting with an initial abstraction, the analysis technique explores in the first step the model using the given abstraction. If no violation of the safety property is found, the program is correct. Otherwise, a potential counterexample is generated. Since the abstraction may be too coarse to prove the correctness, the potential counterexample is checked for feasibility in the second step. If the counterexample is feasible, the program is incorrect. If it is infeasible, the third

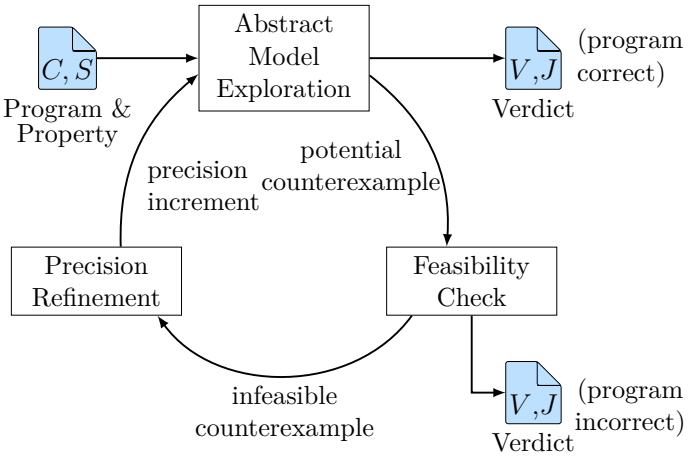


Figure 2.7: Concept of the Counterexample-Guided Abstraction Refinement scheme, adapted from [BHLW22a]

step is performed and the precision is refined to exclude the infeasible counterexample. The newly generated precision increment is then used in the *Abstract Model Exploration* in the next iteration.

2.2.2 Value Analysis

Value Analysis [NNH99] is a verification technique analyzing the concrete variable values within the program. For this, it tracks the exact values for each variable along all program paths, storing information on whether a variable has a concrete constant value z, is undefined ($\bot$), or has no constant value ($\top$). In the running example C_1, a value analysis may compute the fact that the variable res has the value 0 on the path $\ell_0, \ell_3, \ell_4, \ell_5$. When multiple paths are reaching a location, the value analysis may or may not merge the information. Information is merged per variable, i.e., if a variable has the same value on both paths merged this value is kept, if they have different constant values $\top$ is used. In practice, merging information leads especially for large programs to a too coarse over-approximation. Hence we use a configuration of the value analysis that does not merge analysis information but keeps them separately. To enhance the performance for large programs, the analysis does initially not track the values of all variables but uses a CEGAR based refinement scheme [BL13].

2.2.3 Symbolic Execution

Symbolic execution [Kin76; CS13] is a technique similar to value analysis. The main difference is the handling of unknown input values. Instead of using an unknown value $\bot$ as in value analysis, symbolic execution introduces a new symbolic value for each unknown value. Thereby, relations between variables can be expressed using formulae containing the symbolic values, stored in the symbolic memory. The computed analysis information contains per location the variable values using symbolic values and a path

condition. The symbolic memory is updated for each variable assignment. A *path condition* is a formula containing the branching conditions on a path leading to a certain program location using the symbolic values of the variables. It thus describes the conditions that must be satisfied by the symbolic values if the path leading to the location is feasible. Whenever a branching point is reached, the path condition is checked for satisfiability. If the path condition is not satisfiable, the path is infeasible and its exploration is stopped. In the running example, a symbolic execution computes the information that at location ℓ_5 reached on the path $\ell_0, \ell_1, \ell_3, \ell_4$ the variable values are $\{input \mapsto -\alpha, rem \mapsto -\alpha, res \mapsto 0\}$ where α is a symbolic value and the path condition is $\alpha < 0$. As the path condition is satisfiable, the path is feasible.

2.2.4 Predicate Abstraction

Predicate abstraction [GS97] is an analysis technique that uses a set of predicates called precision Π as abstractions for the concrete program states. Initially, the abstraction *true* is used for the initial location ℓ_0. The new abstraction for a location ℓ_i reached via the transition $\langle \ell_{i-1}, \sigma_{i-1} \rangle \xrightarrow{g_i} \langle \ell_i, \sigma_i \rangle$ is computed by using all predicates (and their negations) from the precision that are implied by $sp(g_i, \varphi)$, where φ is the abstraction of σ_{i-1}. In case a location has multiple predecessors, the disjunction is used to combine the abstractions. For example, if the precision $\Pi = \{input \geq 0\}$ is used, the abstraction of the location ℓ_1 contains all predicates from Π implied by $sp(input < 0, true)$, which is $\neg(input \geq 0)$.

In practice, there exist multiple optimizations for predicate abstraction. First and foremost, the abstraction is usually computed using CEGAR [Cla+00]. In addition, instead of computing the abstraction for each location of the CFA, a *large-block-encoding* [BKW10] is used. Therein, the abstraction is computed only at certain *abstraction points* (by default, e.g., loop heads or branching points) and instead of using the strongest postcondition for a single operation, the strongest postcondition is computed for the complete path between two abstraction points. A third optimization used is called *lazy refinement* [HJMS02]. When lazy refinement is used, the new precision computed in each CEGAR iteration is used to recompute the analysis information only for the counterexample path that was found in the previous iteration, instead of recomputing all analysis information. The predicate abstraction that we use in this thesis computes new predicates in the precision refinement step using Craig-interpolation [HJMM04].

2.2.5 Bounded Model Checking

Unlike the other techniques, *Bounded Model Checking (BMC)* [Bie+99; Bie+03] computes an under-approximation of the program. More precisely, it encodes all paths leading to the error location using the strongest postcondition *sp*, where each loop is unrolled up to a bound of k iterations. If there are multiple paths leading to an error

location, they are disjoined. The resulting formula is then checked for satisfiability. If the formula is satisfiable, the error location is reachable, otherwise, the bound k is increased in the next iteration. The program is reported as safe, only if all paths of the program are analyzed and none of them reaching an error location is feasible, happening when the bound k is sufficiently large and all loops are completely unrolled. Since the variables in programs are not unbounded integers but have a datatype limiting the variable values, BMC will eventually explore all paths of programs without non-terminating loops or infinte srecursions. In our running example, BMC reports the program as safe if k is greater than 16 384, the maximal number of loop unrollings.

2.2.6 k-Induction

The basic idea of k-induction [DHKR11] is to generalize BMC by induction. In the first step, the program is proven safe for a bound k using BMC. Next, the proof for the program with a bounded number of unrollings is extended. For this, the approach maintains a collection of auxiliary invariants and tries to generate k-inductive invariants. If such invariants are found, it is checked whether the invariants are strong enough to prove the program safe by abstracting the loops using the sp_{inv} rule. Otherwise, k is increased and the process starts anew. The auxiliary invariants can be generated by a different analysis and are in general continuously refined. In the employed configuration, a CEGAR-based dataflow analysis using intervals is used to compute them [BDW15].

In the running example, we assume that k-induction starts with $k = 1$. First, BMC is executed, unrolls the loop once, and thereby proves that the program is correct for the bound k. Next, k-induction tires to generalize the result of BMC. Let us assume that the 1-inductive invariant $inv \equiv (input = rem + 2 * res)$ is generated. As inv is helpful, the program is proven correct for an arbitrary number of loop unrollings with respect to the safety property S.

2.2.7 Abstract Reachability Graph

The information computed by a verification approach is associated with the locations of the CFA. The Abstract Reachability Graph [BHJM07; BJ21] represents the abstract state space computed by an analysis and contains the analysis information computed.

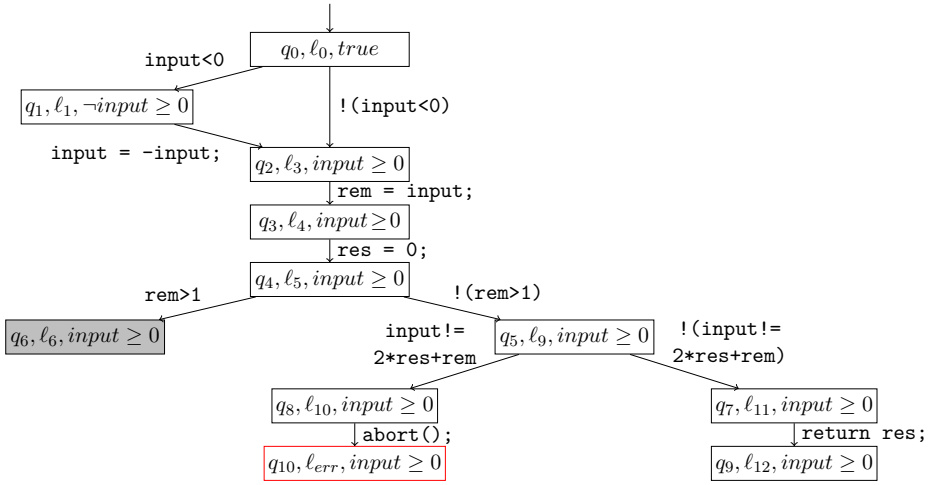


Figure 2.8: ARG A_1^{ARG} generated by a predicate abstraction using the predicate $\Pi = \{input \geq 0\}$ for the running example C_1 and safety property $S = (\ell_{err}, false)$

Definition 2.6. *An* Abstract Reachability Graph (ARG) $A^{ARG} = (N, n_0, succ, F, prec)$ *for a CFA* $C = (L, \ell_0, G)$ *is a directed graph with a set of abstract states* N*, a successor relation* $succ \subseteq N \times G \times N$*, initial node* $n_0 \in N$*, a set of frontier nodes* $F \subseteq N$ *that need to be explored and a precision* $prec$ *that describes the abstraction level of each state.*

Each *abstract state* comprises a unique name q_i, a location ℓ_i from the CFA, and analysis information that is valid in the current state. We present an ARG that is generated by a predicate abstraction using the precision $\Pi = \{input \geq 0\}$ in a CEGAR iteration in Figure 2.8. Frontier states, that are not explored, are marked in gray. Hence, node q_6 has not yet been explored. The node q_{10} violates the safety property $S = (\ell_{err}, false)$ and is therefore marked in red.

2.3 Verification Artifacts

Verification artifacts are used within (cooperative) verification to encode and exchange information between actors. The *verdict* V provides an answer to the question if the program satisfies the property and is computed by a verifier. Accordingly, a verdict is either *true*, *false*, or *unknown*. The verdict *unknown*can be used if the verifier is unable to compute a solution, exceeds the memory or time limitations, or in case of an internal error.

The other verification artifacts serve as a *justification* J for the verdict, i.e., they state why the property is fulfilled or violated. In the latter case, providing a counterexample in the form of a concrete path of the CFA that leads to a property violation suffices [Bey+15]. For correct programs, more detailed reasoning is needed, as it must be justified why all paths do not violate the property. As comprehending such a justification most of the time requires a recomputation of the proof, it is necessary for

the justification to contain predicates and especially loop invariants that are crucial for this [BDDH16]. In case the verifier returns the verdict *unknown*, it is also possible to provide partial results, in case a part of the program is already verified.

There are already some standardized formats available, serving as justification, namely the *violation witness* [Bey+15], the *correctness witness* [BDDH16] and the *condition automaton* [BJLW18]. In the following, we present the three artifacts. Further details on witnesses can be found in [Bey+22].

2.3.1 Protocol Automaton

All three formats can be defined as instances of a *protocol automaton*. The protocol automaton, initially introduced in [Bey+15] and later extended in [BW20] is an automaton that accepts a set of program paths from a CFA C.

Definition 2.7. *A* protocol automaton $A^{\text{PA}}=(Q, \Sigma, \delta, q_0, F)$ *for a program represented as CFA* $C = (L, \ell_0, G)$ *is a non-deterministic automaton that consists of:*

- *a finite set of states* $Q \subseteq \Omega \times BExpr$*, each being a pair of a name out of some set* Ω *and a state invariant,*
- *an alphabet* $\Sigma \subseteq 2^G \times BExpr$*,*
- *a transfer relation* $\delta \subseteq Q \times \Sigma \times Q$*,*
- *an initial state* $q_0 \in Q$*, and*
- *a set* $F \subseteq Q$ *of final states.*

Automaton states have (arbitrary) names and potentially a *state invariant* φ associated with them which come in the form of boolean expressions over program variables. Transitions are labeled over the alphabet Σ with elements being sets of transitions of the CFA plus additional *assumptions* ψ about program variables describing conditions when executing these transitions. The connection between a program path π in the CFA C and paths that are described by a protocol automaton A^{PA} is established via matched paths. An A^{PA} *matches* a path $\pi = \langle \ell_0, \sigma_0 \rangle \xrightarrow{g_1} \langle \ell_1, \sigma_1 \rangle \dots \xrightarrow{g_n} \langle \ell_n, \sigma_n \rangle$ if there is a sequence $\rho = (q_0, \varphi_0) \xrightarrow{(G_1,\psi_1)} \dots \xrightarrow{(G_k,\psi_k)} (q_k, \varphi_k)$, $0 \leq k \leq n$ in A^{PA}, with $(q_{i-1}, \varphi_{i-1}) \xrightarrow{(G_i,\psi_i)} (q_i, \varphi_i) \in \delta$, such that

1. $g_i \in G_i$ for all $i \in \{1, \dots, k\}$,
2. $\sigma_i \models \psi_i$ for all $i \in \{1, \dots, k\}$,
3. $\sigma_i \models \varphi_i$ for all $i \in \{0, \dots, k\}$.

An A^{PA} *covers* π, if A^{PA} matches π, $k = n$ and $q_k \in F$. Three protocol automata are shown in Figures 2.9 to 2.11: A correctness witness, a violation witness, and a condition automaton. Each of these protocol automata covers a set of paths from the CFA C_1 of

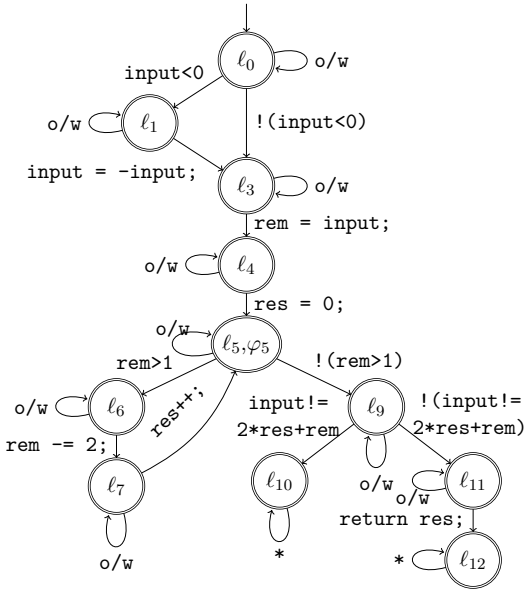


Figure 2.9: Correctness Witness A_1^{CW} for C_1, with $\varphi_5 \equiv \ input = rem + 2 * res$

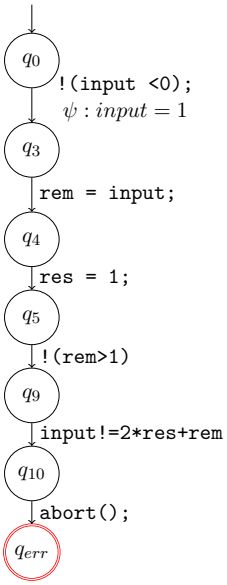


Figure 2.10: Violation Witness A_1^{VW} for C_1''

Figure 2.3b. Note that we use $*$ to denote any operation from *Ops* and `o/w` as shortcut to denote all transitions other than the one explicitly depicted.

To be able to represent different artifacts as protocol automata, a *context-dependent* semantics is used, meaning that the semantics is fixed per artifact instance. Depending on the artifact, matched paths may, among others, encode paths leading to a property violation (in Figure 2.10), or paths not reaching any property violation (in Figure 2.9).

2.3.2 Correctness and Invariant Witness

A correctness witness [BDDH16] is used to encode that a program is safe and thus no path exists violating the safety property. Intuitively, it is a CFA that is equipped with invariants that explain why the program is safe.

Definition 2.8. *A* correctness witness $A^{\text{CW}} = (Q, \Sigma, \delta, q_0, Q)$ *is a protocol automaton, where all states are final states* $(F = Q)$ *and each edge is labeled with trivial assumptions, i.e.* $\psi = true$ *holds for all* $(q, (G, \psi), q') \in \delta$.

States may contain a *state invariant* φ that justifies why the program is correct. Semantically, paths covered by an A^{CW} do not contain a property violation. We colorize correctness witnesses A^{CW} in light green. As an example, we depict in Figure 2.9 the correctness witness A_1^{CW} for the CFA C_1 from Figure 2.3b with $S = (\ell_{err}, false)$. In A_1^{CW}, we define the state invariant $\varphi_5 \equiv \ input = rem + 2 * res$, and all other state invariants are *true*. As ℓ_{err} is unreachable, the correctness witness A_1^{CW} can be generated and all paths of the CFA C_1 are accepted by A_1^{CW}. The state invariant φ_5 is a valid loop invariant and can be used as justification for the unreachability of ℓ_{err}.

In some cases, we want to exchange (candidates for) loop or program invariants between actors. As these predicates may not be valid on all paths, e.g., they do not form a valid loop invariant but can be extended to be valid, we need a new artifact with slightly modified semantics compared to correctness witnesses. We call this artifact *invariant witness* [BHLW22a]. A tool can generate an invariant witness, even though the verification task is not completely solved and the program is thus not fully verified.

Definition 2.9. *A* invariant witness $A^{\text{IW}} = (Q, \Sigma, \delta, q_0, Q)$ *is a protocol automaton, where each edge is labeled with trivial assumptions only, i.e.* $\psi = \mathit{true}$ *holds for all* $(q, (G, \psi), q') \in \delta$.

Hence, an invariant witness may serve as justification only for a subset of the program paths of a CFA, that can be empty if $F = \emptyset$ holds. We colorize invariant witnesses A^{IW} in light orange.

Recently, a new YAML-based format for correctness witnesses (version 2.0) has been released [Aya+24]. Instead of constructing an automaton, the information encoded within the format is directly associated with the source code lines of the C program. It allows the attachment of loop invariants and location invariants, whereas the latter are equivalent to state invariants. Note that the semantics of the YAML-based format are defined based on the semantics of the programs, making it independent of CFAs. Thus, the format is likely easier to generate, but is less expressive, as predicates that hold only on a specific path or after a fixed number of loop unrollings cannot be encoded directly. As the YAML-based format is not designed to encode partial correctness of a program, it cannot represent all information encoded within an invariant witness, specially defined for this purpose.

2.3.3 Violation Witnesses

A violation witness [Bey+15] is used to encode a set of feasible program paths that lead to a property violation.

Definition 2.10. *A* violation witness $A^{\text{VW}} = (Q, \Sigma, \delta, q_0, F)$ *is a protocol automaton, where each state has only a trivial state invariant, i.e.* $\varphi = \mathit{true}$ *holds for all* $(q, \varphi) \in Q$.

The assumptions in an A^{VW} can contain constraints on the variable values. Semantically, at least one program path covered by an A^{VW} contains a property violation. We colorize violation witnesses A^{VW} in light red. As the program from Figure 2.3 is correct with respect to the safety property $S = (\ell_{err}, \mathit{false})$, we construct a CFA C_1'' assigns an initial value of 1 to `res`. Formally, for $C_1 = (L_1, \ell_0, G_1)$, we construct $C_1'' = (L_1 = L, \ell_0, G_1 \setminus \{(\ell_4, \texttt{res=0;}, \ell_5)\} \cup \{(\ell_4, \texttt{res=1;}, \ell_5)\})$. Then, the program violates the safety property S. Hence, we can construct a violation witness A_1^{VW} depicted in Figure 2.10, where we add the assumption that $\mathit{input} = 1$ holds. The only path

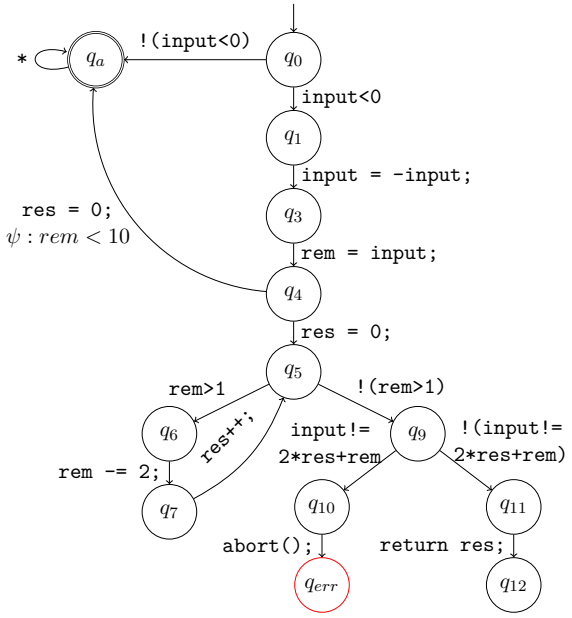


Figure 2.11: Condition Automaton A_1^{CA} for C_1

covered in C_1'' by A_1^{YW} is

$$\begin{aligned}\pi = \langle \ell_0, \{input \mapsto 1\}\rangle &\xrightarrow{!(input<0)} \langle \ell_3, \{input \mapsto 1\}\rangle \\ &\xrightarrow{rem=input;} \langle \ell_4, \{input \mapsto 1, rem \mapsto 1\}\rangle \\ &\xrightarrow{res=1;} \langle \ell_5, \{input \mapsto 1, rem \mapsto 1, res \mapsto 1\}\rangle \\ &\xrightarrow{!(rem>1)} \langle \ell_9, \{input \mapsto 1, rem \mapsto 1, res \mapsto 1\}\rangle \\ &\xrightarrow{input!=2*res+rem} \langle \ell_{10}, \{input \mapsto 1, rem \mapsto 1, res \mapsto 1\}\rangle \\ &\xrightarrow{abort();} \langle \ell_{err}, \{input \mapsto 1, rem \mapsto 1, res \mapsto 1\}\rangle\end{aligned}$$

Thus, following π in C_1'' leads to a property violation.

There is also a YAML-based format for violation witnesses (version 2.0) [Aya+24]. It describes paths directly by sequences of branching decisions and inputs, e.g., that the if-branch of a condition in line 9 of Figure 2.3a should be taken and the else-branch avoided.

2.3.4 Condition Automaton

A condition automaton [BJLW18] states which semantic paths of the program are already successfully verified and under which condition.

Definition 2.11. *A condition automaton $A^{CA} = (Q, \Sigma, \delta, q_0, F)$ is a protocol automaton, where each state has only trivial state invariants, i.e. $\varphi = true$ for all $(q, \varphi) \in Q$: and accepting states cannot be left, i.e. $q_f \in F \Rightarrow q \in F$ holds for all $(q_f, \cdot, q) \in \delta$.*

Semantically, the paths covered by an condition automaton do not contain a prop-

erty violation. Unlike a correctness witness, a condition automaton can contain assumptions that specify the unreachability of program locations under those assumptions. In Figure 2.11, we depict A_1^{CA}, a condition automaton A^{CA} for our running example, where $*$ again denotes any operation from Ops. The partial result encoded within A_1^{CA} covers all paths of the CFA C_1 where the input is positive (accepted by the path from q_0 to q_a), as well as all paths where the input is negative and greater than -10 (accepted by the path q_0, q_1, q_3, q_4, q_a). Therefore, the remaining task is to prove that the program is safe for all inputs less or equal to -10.

2.4 Existing Verification Tools

After having explained the verification techniques used in this thesis and the input and output artifacts consumed respectively generated, we introduce some existing verification tools, that we use as components in our cooperative verification approaches. These tools include CPAchecker, UltimateAutomizer, Klee, Symbiotic, VeriAbs, and SeaHorn.

2.4.1 CPAchecker

CPAchecker [BK11] is a configurable tool for software verification. To analyze a program, CPAchecker computes a CFA of the program and employs a Configurable Program Analysis (CPA) [BHT07; BHT08]. A CPA provides an abstract domain for analysis information that forms a semilattice. It defines a transfer relation modeling the abstract state changes when analyzing a program statement, a merge operator for combining analysis information, a stop operator for deciding if an abstract state is already covered by another state, and a precision adjustment operator. CPA-based analyses can be composed. In that case, the analysis information computed by the composed analyses can be employed for a dynamic precision adjustment. The analysis information for each location of the CFA are computed using the CPA-algorithm, that conducts a fixpoint computation. The computed analysis information is stored in an ARG, e.g., as shown in Figure 2.8.

CPAchecker provides an implementation of various verification techniques such as value analysis [BL13], predicate abstraction [BKW10], symbolic execution [BL18], BMC [BDW18] or k-induction [BDW15], as well as an implementation of CEGAR.

In addition to providing a collection of analyses, CPAchecker allows for an easy extension, as building a new program analysis can be done by reusing existing analyses in composition with a new analysis. The modular structure and the rich set of configuration options allow for either tailoring the CPAchecker to the user's needs or to reuse internal components as standalone elements within a cooperative approach. Information computed during the analysis, especially predicates and loop invariants, are exported in different formats, among others using correctness witnesses. In addition,

CPAchecker has proven its strength in program verification and falsification in many editions of the SV-COMP by winning several medals [Bey22b; Bey23; Bey24].

2.4.2 UltimateAutomizer

UltimateAutomizer [Hei+23; Hei+18; Hei+17; HHP13] is a software verifier using a verification approach based on automata theory. The program is represented as a finite automaton $\mathcal{A}$ with error locations as accepting states. UltimateAutomizer applies a CEGAR-based iterative refinement for an over-approximation of a program's error paths, which is represented by the paths accepted by the automaton $\mathcal{A}$. In each CEGAR iteration, an error path is selected from the automaton $\mathcal{A}$ and checked for feasibility. If the chosen path is infeasible, a Floyd-Hoare automaton [HHP13] is constructed, which explains the infeasibility of the chosen path and may also show that similar paths of the automaton $\mathcal{A}$ are infeasible. The Floyd-Hoare automaton is constructed using predicate abstraction wherein the new predicates are computed using Craig interpolation or Newton refinement [Die+17; Hei+23]. The over-approximation is then refined by removing all paths from $\mathcal{A}$ that are accepted by the Floyd-Hoare automaton. An answer to the verification problem is found if either a feasible counterexample is discovered or the over-approximation contains no path violating the property. The predicates computed in the abstraction are exported within the generated justification and UltimateAutomizer generates helpful loop invariants in many cases. In recent years, UltimateAutomizer has successfully participated in the SV-COMP, e.g., winning the category `overall` in 2023 [Bey23] and 2024 [Bey24].

2.4.3 Klee

Klee [CDE08] is a dynamic symbolic execution tool, working on the LLVM-intermediate representation (IR) language. Klee aims for (1) finding inputs covering each executable line of code and (2) finding any input that leads to a property violation, i.e., a violated assertion. To achieve this goal, Klee first generates path conditions by symbolically executing the program, then computes concrete values for the program's inputs using the symbolic values, and finally dynamically executes the program using these inputs. As not all path conditions can be generated in one step, Klee uses, among others, probability-based or heuristic-based strategies for the exploration. As the number of paths in a program and thus path conditions are typically large, Klee optimizes the path conditions using techniques such as constraint simplification (removing unnecessary constraints) or value concretization (computing implied values in advance). Klee employs dynamic symbolic execution, which under-approximates the state space. Therefore, it has to show that all program paths leading to the safety property are infeasible to verify that the program satisfies the safety property.

2.4.4 Symbiotic

Symbiotic [Cha+22] is a verifier that combines slicing [Wei84] with dynamic symbolic execution. The program is first translated to LLVM-IR and sliced with respect to the safety property. Thereby, all statements that do not influence the reachability of error locations are removed. Next, a modified and enhanced version of Klee is executed for at most one-third of the available overall time to find violations of the safety property. In case Klee is unable to find a program violation, SlowBeast is used. By default, a backward symbolic execution [CFS09] with loop folding [CS21] is conducted by SlowBeast. Instead of analyzing the program paths from the initial location to the error location as in symbolic execution, backward symbolic execution analyzes the program in reverse. In case loops are found, the loop folding attempts to generate loop invariants proving the backward path from the error location to the loop head infeasible. In case this technique fails within a given time limit, Symbiotic runs forward symbolic execution again using SlowBeast. In recent years, Symbiotic has successfully participated in the SV-COMP, e.g., winning the category `overall` in 2022 [Bey22b].

2.4.5 VeriAbs

VeriAbs is a portfolio-based verification tool, employing four different verification techniques, where each may consist of several sequentially composed components [Afz+19]. The strategy selection is based on the loop structure and the value ranges of the variables related to the loop. Based on the result, VeriAbs selects a strategy: (1) random fuzz testing using Afl for loops with unstructured control flow, (2) techniques to abstract arrays followed by the default strategy afterward for programs with loops working on arrays, (3) explicit state model checking followed by standalone invariant generation techniques for programs with potentially few loop iterations or (4) the default and fallback strategy, a fixed sequence of different verification approaches like interval analysis, loop abstraction and BMC. In a more recent version, VeriAbs employs a slicing technique to generate multiple program fragments that can be analyzed in parallel [DAV21].

2.4.6 SeaHorn

SeaHorn [GKKN15] is a verifier that works on LLVM-IR and that generates *Constrained Horn-Clauses (CHCs)* [Gur22] to solve the verification task. The CHCs generated for each statement contains predicates for modeling the data and the control dependencies of the program. They also encode the safety properties of the program. In case the control flow of the program is cyclic, the set of generated CHCs is recursive. To solve the computed CHCs, the solver Spacer [KGC14] is employed. Spacer tries to prove the unsatisfiability of the CHCs, being equivalent to proving the program safe,

by searching for interpretations of the predicates present in the CHCs. SeaHorn provides a front end for C programs, but it does not automatically translate the computed information for the LLVM-IR program back to C.

2.5 Benchmarking and Existing Tool Support

To answer the question of whether cooperation can increase the performance in software verification, we need easily measurable criteria. In this thesis, we generally use *effectiveness* and *efficiency* as criteria. Intuitively, we are interested in the total number of tasks correctly or incorrectly solved with a fixed set of resources per task for effectiveness, and in the time taken to compute these solutions per task for efficiency. Before we describe the setup and the metrics in more detail, we first explain the general evaluation setup used. Lastly, we describe an existing tool called CoVeriTeam facilitating the building and evaluation of new forms of cooperative software verification.

2.5.1 General Evaluation Setup

A *verification task* comprises a C program and a safety property, which defines the error locations of the program, whose (non)-reachability needs to be checked. To achieve a fair comparison between different concepts, such as cooperative and non-cooperative approaches, we must guarantee that the set of tasks used is representative and widely accepted. Therefore, we decided to use the SV-Benchmarks, which is currently the largest publicly available benchmark for C program verification [SVB24]. It comprises, among others, around 10 000 tasks for reachability verification, that are organized in 18 categories, ranging from tasks with loops, arrays, or recursion over tasks adapted from hardware verification to tasks based on Linux device drivers or AWS applications. Moreover, for each task, a ground truth is given. It states if the task is correct and satisfies the specifications or is incorrect and violates them. The benchmark is extended on an annual basis. We conducted most of the experiments using the version published in 2023 [SVB23]. In some cases, we used a subset of the available tasks, as we necessitate that the program has certain properties like a loop for the generation of loop invariants.

The SV-Benchmarks are also used to conduct the annual SV-COMP [Bey24], wherein verifiers compete in different categories, e.g. `reach-safety`. In 2024, 59 verifiers competed in SV-COMP [Bey24], trying to correctly solve as many verification tasks as possible. The SV-COMP requires each participating tool to generate a correctness witness (cf. Section 2.3.2) or a violation witness (cf. Section 2.3.3) as justification for a verification task. Hence, these two formats are widely used.

In addition to the fixed set of tasks, we need to ensure that each verifier has the same fixed set of resources available. For most experiments, each tool has in total 15 minutes CPU time on 4 cores and 15 GB of RAM available. We employed BenchExec [BLW19] as the tool for conducting all experiments, as BenchExec guarantees that the provided

resource limitations are ensured and also measures the consumed resources (CPU time, overall time, and memory). Thereby, we get reliable benchmarking results as well as easing the reproduction of the conducted experiments. This is a setup that is comparable to those used in the Competition on Software Verification (SV-COMP) [Bey24], where each tool can access 8 CPU cores. The setup is comparable, as the overall CPU time is still fixed to 15 minutes, which are divided among at most 4 cores and not 8.

2.5.2 Evaluation Criteria

To evaluate the performance of a verification approach we use *effectiveness* and *efficiency*. In terms of effectiveness, BenchExec counts the number of correct proofs (tasks that are correct and where the verifier computes a proof), of correct alarms (where the task is incorrect and the verifier raises an alarm), incorrect proofs and alarms, and cases where no answer is given (unknowns). Moreover, we calculate the total number of correct answers (correct proofs + correct alarms) and analogously the total number of incorrect answers. For efficiency, we measure the time consumed by the verifier to compute a correct result. Rather than comparing the CPU time that is taken to compute an answer, we decided to compare the overall time elapsed for solving the task, known as *wall time*. Thereby, we account for potential parallelization that is introduced in cooperative approaches. Note that the resources are limited for the verifier, regardless if only a single actor is employed or multiple actors are working together cooperatively. Thus, cooperative approaches share the resources among all components used. Thereby, the cooperative approaches do not get any additional resources when using several instances in parallel. They may only compute a result more efficiently if subtasks are solved in parallel.

2.5.3 Tool Support for Realizing Cooperative Verification Concepts

According to Characterization 1.1, an approach for cooperative software verification is exchanging verification artifacts among off-the-shelf components. To facilitate such an exchange, some form of orchestration is necessary. The basic operations performed by such an orchestration are executing actors, gathering artifacts and verdicts, and executing different operations based on the collected verdicts and artifacts in the form of branches or loops. In 2022, Beyer and Kanav present an open-source framework called CoVeriTeam [BK22] that is designed to realize cooperative verification approaches by providing a language and execution framework: Within the language, verification artifacts serve as objects, and actors like verifiers perform operations on artifacts. By default, CoVeriTeam provides more than ten predefined artifacts, including program, safety property, or verdict, and ten predefined actors like verifiers. These sets can be extended by users in case new artifacts or actors are needed. The description of the actors contains the input and output artifacts that are consumed and respectively produced by the actors. For artifacts, operations such as comparing them for equality

or merging two artifacts of the same type are defined. For using a specific tool, an *actor definition* is used, defining where to find the executable binaries of the tool and commands needed for the tool execution.

To realize a new cooperative combination of tools, CoVeriTeam allows the execution of actors sequentially, branch executions in an if-then-else style, repeat the execution of actors until a certain condition is fulfilled, or execute several actors of the same type in parallel, until the first computes a result. The main advantage of using CoVeriTeam for realizing new forms of cooperative verification is that the users do not need to take care of downloading and executing the tools in isolation or collecting the results, as CoVeriTeam relies on mechanisms that are provided by BenchExec for the execution of the actors. In addition, building multiple instances of the same cooperative approach using different tools is easy in CoVeriTeam, as changing the off-the-shelf tool that is used within the configuration can be done by updating a single line of code.

3 Sequential Cooperation

When combining two or more verifiers, using a sequential combination is likely the most intuitive approach. Executing the verifiers or analysis configurations non-cooperatively one after another, i.e., restarting from scratch, is a commonly used approach, as in tools like CPAchecker [Wen13] or Symbiotic [Cha+22]. These approaches do not reuse information computed by previous instances. Thus, there is no need for encoding, exchanging, and importing the information. In contrast, it is necessary for sequential approaches employing cooperation to carry over the information computed by a previous tool to the next tool and enable it to use the information.

In 2012, Beyer et al. and Christakis et al. present two similar cooperative approaches: Beyer et al. introduce Conditional Model Checking (CMC) [BHKW12], a technique where multiple so-called *conditional verifiers* are executed sequentially, each with a fixed time limit assigned. After the time limit is exceeded and no final answer is computed, an intermediate answer in the form of a condition is generated. The condition states which parts of the program have already been successfully verified. The next conditional verifier can use the condition, which can be represented as condition automaton (see Definition 2.11), to safely discard the parts of the program that have already been proven safe by prior verifiers.

Another approach proposed by Christakis et al. [CMW12] allows verifiers to directly add assumptions under which a program is verified into the code, such as certain operations do not cause an overflow. The next verifier only needs to confirm that the assumptions are valid to prove that the full program is correct. The exchanged artifact is herein the annotated program containing the assumptions.

We propose a novel idea to complement existing concepts for sequential cooperative verification. Instead of sequentially executing different instances of the same kind of actor (as in CMC), we propose to delegate subtasks to components specialized in solving that subtask. More precisely, we let tools cooperate on invariant generation, one core task of software verification. As a high-level idea, we let a verifier delegate the task

of generating loop invariants to a specialized invariant generation tool. This concept, called CoVEGI, is introduced in Section 3.1. Within CoVEGI, tools for generating invariants are employed. In Section 3.2, we take a more detailed look at a specific class of tools for generating invariants using machine learning (ML). More precisely, we present the modular framework MIGml for experimenting with and comparing of ML based invariant generators. It generalizes the ideas of existing concepts and comprises a *teacher* and a *learner* as instantiable components with clear-cut interfaces. We describe in Section 3.3 the realization of both approaches. We evaluate in Section 3.4 the concept of CoVEGI to determine, whether a sequential cooperation wherein the task of invariant generation is delegated to specialized components increases the verification performance. Additionally, we investigate if our framework MIGml could potentially be used as a component within CoVEGI. We conclude this chapter by discussing the results in Section 3.5 and providing an overview of related work for cooperative sequential combinations and on invariant generation in general in Section 3.6.

3.1 Cooperation using Externally Generated Invariants

In general, the existing verification approaches perform good in solving verification tasks. This is for example visible in the increasing number of correctly solved tasks in annual verification competitions [Bey24; EHMU19]. As invariant generation is one core task of verification, many approaches also perform well in finding loop invariants helpful for verification. However, there are tasks for which a verifier fails to compute a solution. This may happen in case no loop invariant is found or valid loop invariants are generated that do not ease the verification problem. For those cases, we propose to let the verifier delegate the task of *invariant generation* to a dedicated loop invariant generator. Our novel sequential cooperation approach is called Cooperative Verification via Externally Generated Invariants (CoVEGI). The idea behind CoVEGI is to obtain invariants inferred using conceptually different approaches and thereby increasing the variety of generated predicates. Thus, we use the strength of specialized invariant generation tools to help the verifier in finding a solution for the verification task.

The high level idea of CoVEGI is depicted in Figure 3.1 and comprises two types of actors, both shown in blue: A *main verifier* and a number of *helper invariant generators*. The main verifier has the overall control over the verification process and receives the program and the property as input. It can delegate tasks to helpers as well as continue its own verification process with (partial) results provided by helpers. The helpers run in parallel as black-boxes without cooperation. The generated invariants are given to the main verifier in the form of invariant witnesses. They are particularly well suited for our intended use because their format is standardized and several verifiers already produce invariant witnesses. The generated witnesses are then *injected* into the verification run of the main verifier using the *Witness Injector*. Some off-the-shelf invariant generators are not able to process the given task or to generate invariant witnesses. To

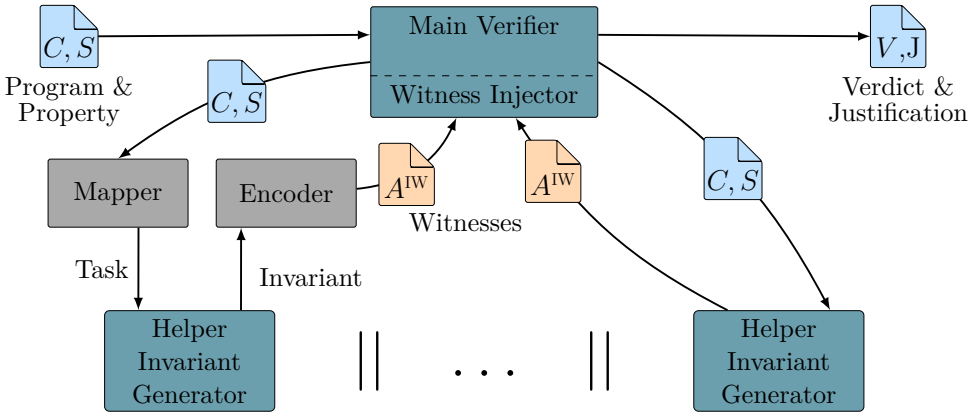


Figure 3.1: Overview of CoVEGI

allow using them, the framework thus foresees the two additional components *mapper* and *encoder*, depicted in gray. A mapper maps the program and property into a task format understandable by the helpers and an encoder incorporates an invariant into an invariant witness. The framework can be arbitrarily configured with different main verifiers and helpers, in case suitable encoders and mappers are given.

3.1.1 Motivating Example

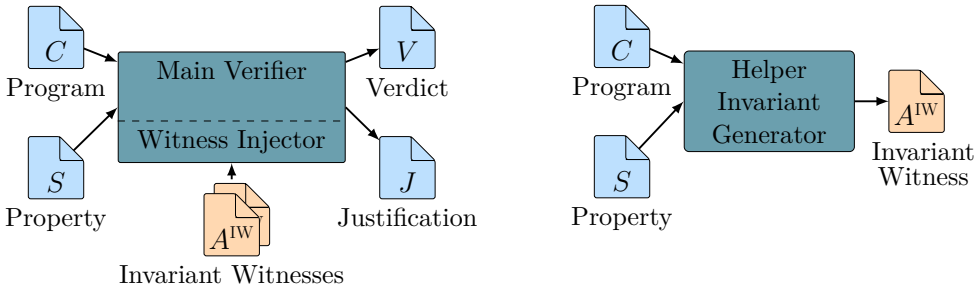
To ease understanding, we exemplify the overall workflow and the idea behind CoVEGI using the running example presented in Figure 2.3: We instantiate CoVEGI with a main verifier and a helper invariant generator. First, the main verifier runs standalone and generates the invariant $inv_1 \equiv (rem \geq 0)$, until it eventually realizes that it "got stuck" in solving the task standalone and requests for help[1]. The helper invariant generator is started, computes an invariant $inv_2 \equiv (input = 2 * res + rem)$, and generates an invariant witness, that is injected into the main verifier. The main verifier now can use inv_1 and inv_2, where the conjunction suffices to prove the program correct with respect to a given safety property S.

3.1.2 Actors in CoVEGI

Next, we explain the two sorts of actors participating in the CoVEGI framework, namely the main verifier and helper invariant generator. For both, we state the requirements for them and describe their functionality:

Main Verifier The *main verifier*, conceptually depicted in Figure 3.2a, is responsible for coordinating the verification process and, if needed, it can request support from the helper invariant generator in the form of invariants. Hence, the main verifier is steering the cooperation. It receives as input the program C as CFA and a safety property S. It

[1]There are multiple ways to detect such a situation, e.g., using a timeout or monitoring the analyses behavior [THW23].



(a) Main Verifier (b) Helper Invariant Generator

Figure 3.2: Conceptual view of the actors in CoVEGI

computes as output a verdict, stating whether the property holds, and possibly (but not necessarily) provides a justification in the form of correctness or violation witness. To be able to process the provided support in the form of invariants stored inside invariant witnesses, a main verifier is required to offer a *witness injector*. The witness injector loads a witness, extracts the invariants encoded, and injects them into the analysis of the main verifier. The witness injection can either happen before (re-)starting the analysis or during runtime.

Helper Invariant Generator A *helper invariant generator*, conceptually depicted in Figure 3.2b, receives as input the program C as CFA and a safety property S. It computes as output a set of invariants, stored in an invariant witness A^{IW}. The generated invariants are neither required to be helpful for the main verifier nor to be correct. Thus, helper invariant generators are also allowed to generate trivial invariants or invariant candidates which might turn out to be wrong.[2] Due to this design decision, we can employ a wider range of tools generating invariants, e.g., those generating invariants using heuristics, machine learning, or Large Language Models.

3.1.3 Witness Injection for Main Verifiers

The concept of CoVEGI requires that each main verifier allows for a witness injection. There are existing verifiers that do not have such functionality. In the following, we thus explain how to realize witness injection for two widely used verification techniques, namely predicate abstraction and k-induction, and provide ideas for a realization in other approaches. The core idea for both analysis techniques is to make the invariants available in all abstraction locations associated with the loop head the invariant is generated for. The analyses may compute different analysis information for the same location, e.g., when unrolling a loop. Hence, witness injection has to update all elements that contain information for a location, for which an invariant is contained in the invariant witness.

[2] Recently, Saan et al. have shown (in the context of witness validation) that their tool Goblint significantly benefits from invariants that could not be verified, i.e., that these invariants ease the verification problem [Saa+24].

Predicate Abstraction Predicate abstraction uses a set of predicates, often called *precision*, for computing an abstraction of concrete program states [GS97], in most cases iteratively refined using the CEGAR scheme (cf. Section 2.2.1). Therefore, the witness injection adds the loop invariants to the precision. Thereby, the predicate abstraction will employ these predicates within the next CEGAR iteration. As we allow helper invariant generators to generate loop invariant candidates, we cannot take their correctness for granted. If predicates within the loop invariant are conjoined or disjoint, they are furthermore split up into atoms and the atoms are inserted individually. Splitting predicates increases the performance, as Satisfiability modulo theories (SMT) solvers perform better using many small predicates than few larger ones[3]. By adding the predicates to the precision, the abstraction function of the predicate abstraction itself will reestablish the loop invariants. Invalid loop invariants are ignored, but valid atoms are still used and may help to complete the verification task.

In case that lazy abstraction [HJMS02] is used, each abstraction point is equipped with an individual precision. Hence, we add the loop invariant and its atoms only to the precisions of those abstraction points associated with the loop heads for which the loop invariant is generated. We can also employ the same procedure when the main verifier is configured to be restarted. Instead of adding the invariants and the atoms to the current precision, we add them to the initial precisions of predicate abstraction, which is otherwise empty.

k-Induction k-Induction maintains a set of auxiliary invariants, used for constructing the inductive correctness proof. Hence, we can make use of the same idea in case that the verifier is restarted or not: Whenever an invariant witness is made available to the analysis, the encoded predicates and the program locations are added as candidates to the set of auxiliary invariants. k-induction checks each element from that set periodically for validity. Thus, externally generated invariants that are valid are conjoined with the predicates stored in the analysis abstract states, corresponding to the invariant's location. Invalid invariants are thus ignored.

General Methods For other verification methods, we need to distinguish whether the technique uses predicates for abstraction or encodes paths using boolean formulas, like BMC or symbolic execution. In the former case, we follow the idea used for predicate abstraction and k-induction: The goal is to add the invariants and all atoms used to the set of predicates maintained by the analysis. For analysis techniques that employ abstract interpretation, the technique proposed by Saan et al. making use of "unassume" operations might also be applicable [Saa+24]. In the latter case, we can follow an idea inspired by CEGAR: First, it has to be verified that the invariants encoded in the invariant witness are valid, as we allow the helper to generate candidates. Next, the rule sp_{inv} (cf. (2.6)) can be used with the invariant to reduce the number of

[3]This has been reported by tool developers and has also shown in our experiments.

paths within the program. If the program is verified using the invariant, the approach can use the generated verdict as the final answer, as the loop invariant is valid and over-approximates the loop body and the program thus satisfies the specification. In case a violation is found, the counterexample has to be analyzed in the original program. If the counterexamples is feasible, the program violates the specification. Otherwise, it is infeasible, due to a too coarse over-approximation through the loop invariant. In this latter case, the invariant is not helpful and the verification approach continues its verification without the invariant.

3.1.4 Using Off-the-shelf Helper Invariant Generators

As we are interested in building instances of CoVEGI using off-the-shelf tools, we can neither expect that all existing tools generating invariants which we want to use as helpers can process the safety property defined nor that they are able to produce invariant witnesses. Next, we explain the components mapper and encoder, that can be used in these situations.

3.1.4.1 General Idea of Mapper and Encoder

Intuitively, the mapper's task is to map the program and property into a format that can be processed by the off-the-shelf invariant generator and the encoder's task is to encode the information computed in a non-standardized format into an invariant witness. We depict this construction for building a helper invariant generator using an *off-the-shelf invariant generator*, a *mapper*, and an *encoder* in Figure 3.3. *Translated task* and *invariant* are visualized using a white file symbol with a dotted border to indicate that this information is encoded in a non-standardized, internal format.

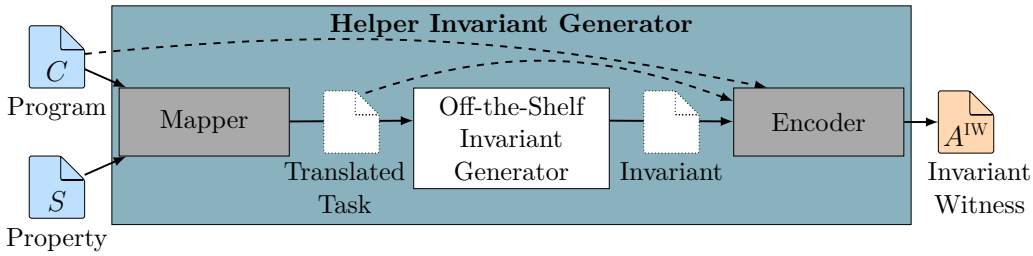


Figure 3.3: Construction of an helper invariant generator using an off-the-shelf invariant generation tool

Next, we explain the components mapper and encoder in greater detail:

Mapper A *mapper* transforms the safety property and the program into an input format understood by the invariant generation tool. In general, these transformations are simple syntactic code replacements. For instance, some tools may not support specifying certain error locations, but rather require the use of specific function calls like `verifier_error()` or specific return values to encode error locations. Moreover,

some tools may also require to denote, for which location or loop an invariant should be generated.

Encoder An *encoder* generates an invariant witness out of the computed loop invariants of a helper, that may be represented using an internal format. In case a mapper is used, it may conduct syntactical transformations on the task. Firstly, the encoder has to ensure that the invariant witness is generated for the original task. In addition, some helper invariant generators work on IR of the C language (e.g., LLVM-IR) or intermediate verification languages (e.g., Boogie). In such cases, the computed invariants (formulated in terms of IR-variables) first of all need to be translated back to the namespace of the C program. We exemplify the construction of such an encoder for LLVM in Appendix A.1.1.

3.1.5 Cooperation within CoVEGI

Having all components at hand, we formalize their interaction and cooperation within CoVEGI. The CoVEGI algorithm, shown in Algorithm 1, orchestrates the cooperation and is configurable. Within the algorithm, we assume that main verifier and all helpers run as threads and can be started and stopped. We furthermore employ methods `wait` for waiting until some conditions are satisfied and `join` for waiting for a specific thread to complete.

Initially, the main verifier is started without any helper invariant generators running in parallel (line 1), providing the opportunity to verify programs alone. It runs standalone until it computes a result that is subsequently returned (line 3), or until it requests help. Currently, the main verifier sets its flag `requestsForHelp` after a timer has expired, which is configurable using the option `timerM`. We use an explicit flag to allow using other mechanisms to enable the cooperation[4]. In case help is requested, all helpers are started for a user-configurable time limit (`timeoutH`) in parallel without cooperation (lines 4 and 5). CoVEGI allows the use of several helper invariant generators, as we aim to combine complementary approaches to leverage their strengths. In case the first non-trivial invariant is found, it is added to the set of witnesses (line 8). Depending on the configuration option `termAfterFirstInv`, the algorithm either injects the first invariant found directly or waits for all helpers to compute a solution or exceed their time limit. If invariant witnesses have been computed, they are injected into the main verifier (line 15). If the `restartMain` option is set, the main verifier is stopped before injection and restarted afterward. This option is useful to reduce the number of available predicates within the main verifier and thereby speed up the verification with the externally generated invariants. Finally, the main verifier continues and completes its verification (without any further request for help) and the result is returned.

[4]One approach monitors the main verifier's internal state to predict if it will run into a timeout [THW23].

Algorithm 1 CoVEGI Algorithm

Input: C ▷ *CFA*
S ▷ *safety property*
M ▷ *main verifier*
Helpers ▷ *set of helpers*
conf ▷ *configuration*
Output: V ▷ *verdict*
J ▷ *justification*

```
1:  M.start(C, S, conf.timerM);                          ▷ start M standalone
2:  wait until (M.requestsForHelp ∨ M.hasSolution());
3:  if (M.hasSolution()) then return M.getSolution(); end if      ▷ return V, J
4:  for each H ∈ Helpers do parallel                      ▷ run helpers in parallel
5:    H.start(C, S, conf.timeoutH);
6:    wait until (H.timeout() ∨ H.hasSolution() ∨ H.stopped());
7:    if (H.hasSolution() ∧ nonTrivial(H.getSolution())) then
8:      witnesses := witnesses ∪ H.getSolution();
9:      if (conf.termAfterFirstInv) then
10:       for each H' ∈ helpers \{ H } do parallel
11:         H'.stop();                                    ▷ stop other helpers
12: if (M.hasSolution()) then return M.getSolution(); end if      ▷ return V, J
13: if (witnesses ≠ ∅) then                               ▷ invariants found
14:   if (conf.restartMain) then M.stop(); end if
15:   M.inject(witnesses);                    ▷ inject witnesses into main verifier
16:   if (conf.restartMain) then M.start(C, S, ∞); end if
17: join(M);                                              ▷ wait for M to finish
18: return M.getSolution();                               ▷ return V, J
```

3.1.6 Example Application of the CoVEGI Algorithm

After having described all components and the cooperation within CoVEGI, we exemplify Algorithm 1 applied to the running example in more detail. We instantiate CoVEGI with a main verifier employing k-induction and four helper invariant generators[5]. The first one called *Helper A* can process a program and a property as input and generate invariants in the form of invariant witnesses. The second one, called *Helper B* is an off-the-shelf invariant generator working on the LLVM-IR. Hence, we make use of the construction depicted in Figure 3.3, using a mapper and the encoder explained in Appendix A.1.1. We additionally use two further helper invariant generator *Helper C* and *Helper D*. We configure the framework as follows: We set `restartMain` to *true*, `terminateAfterFirstInv` to *false*, `timerM` to 50 seconds, and `timeoutH` to 300 seconds.

Initially, the main verifier runs standalone and the component within k-induction for invariant generation computes predicates enumerating concrete values for *input*

[5]In practice, having more than two helpers is not practical [HW21a]. We use four helpers to exemplify the most important cases of Algorithm 1 within a single example.

($input = 0, 1, \ldots$). As they do not suffice to prove the program correct, the main verifier sends a request for help after 50 seconds of runtime. Next, the four helper invariant generators are started and run in parallel with the main verifier.

Let us assume that *Helper C* returns only trivial invariants (after 10s), *Helper A* the invariant $input = 2 * res + rem$ (after 50s), the *Helper B* the invariant $0 = input - rem - 2 * res$ (after 100s, transformed from LLVM-IR) and *Helper D* the invariant $0.5 * (input - rem) = res$ (after 500s). The trivial invariant is ignored (see check in line 7) and when *Helper A* returns a solution, *Helper B* and *Helper D* are still not stopped, due to the chosen configuration. The algorithm waits until the *Helper B* computes the invariant. As *Helper D* can compute an invariant only after 500s, it exceeds the given time limit and is thus stopped. As configured, the running main verifier is stopped, and thereby all not helpful predicates are removed. The freshly started main verifier uses the two invariants $input = 2 * res + rem$ and $0 = input - rem - 2 * res$, which enable it to prove the program correct.

3.2 Machine Learning based Invariant Generation

As just exemplified, the quality of the invariants generated by the helper invariant generators is the key to success in CoVEGI. If they do not generate (helpful) loop invariants, the main verifier does not receive any new information and the cooperation is useless. Nevertheless, finding such invariants is a challenging endeavor. For a long time, loop invariant generation was dominated by template- and logic-based techniques (e.g., [HJMM04; BR02a; Bra11; BDFW08; BHJM07]). Most template-based approaches are data-driven, meaning that a set of data of the values of program variables observed at a program location is generated and a template is used to find the best matching invariant. These templates range from simple non-relational predicates as $x \leq 4$ or $x \neq 0$ for $x \in Var$ over more complex predicates from relational domains, that allow stating relations between program variables. Examples for relational domains are Difference Bound Matrices (DBMs) [DKW08], which contain conjunctions of inequations of the form $x_1 - x_2 \leq c$ or $\pm x_1 \leq c$, the octagon domain [Min06], that contains equations of the form $b_1 * x_1 + b_2 * x_2 \leq c$, or from the domain of Polyhedra [CH78], that contains equations of the form $a_1 * x_1 + \ldots + a_n * x_n \leq c$, for $x_i \in Var$, $c, a_i \in \mathbb{Z}$ and $b_1, b_2 \in \{-1, 0, 1\}$. Logic-based approaches use for example interpolation along program paths to generate predicates that can serve as (part of) a loop invariant.

In 2012, one of the first data-driven approaches to generate predicates, more precisely interpolants, using ML was proposed by Sharma, Nori, and Aiken [SNA12]. Therein, the predicates are inferred using a combination of Support-Vector Machines (SVMs) rather than a template. Later, more approaches focusing on the generation of loop invariants, employing different learners or different ways to generate data were introduced (e.g., [GLMN14; KPW15; ZMJ18; PSM16; Rya+20; Yao+20]). Most ML-based approaches for generating loop invariants follow the structure of the learning

process for regular languages proposed by Angluin [Ang87], except for the fact that they have a different learning objective. The Angluin-style learning process is divided into two components, a *Teacher* and a *Learner*, as depicted in Figure 3.4:

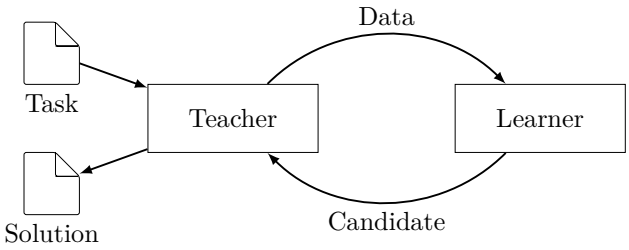


Figure 3.4: The Angluin-style learning process using Teacher and Learner

The Teacher is given the task that needs to be solved and generates a set of data that are given to the Learner. The Learner learns a candidate for the solution and returns the candidate to the Teacher. Important to note that the Learner is not aware of the task and is working only on the given data. The Teacher receives the candidate from the Learner and checks, if it is a valid solution. If the solution is invalid, the data is refined and the process starts anew, otherwise the candidate is returned as the process' final answer. One advantage of that learning process is the clear separation of concerns: The Teacher is responsible for validating candidates and generating the data, and the Learner's task is to generate candidates given data. Note that this concept is also sometimes called Counterexample-Guided Inductive Synthesis (CEGIS) [Alu+13].

In the setting of invariant learning using ML, the task comprises a program C for which an invariant needs to be generated and the property S, the candidate is a potential loop invariant and the solution thus a valid loop invariant *inv*. The idea of using the concept of Teacher and Learner for invariant generation is formalized by Garg et al. in the ICE-framework [GLMN14; GNMR16]. A point in the set of data given to the Learner contains values for the program variables that might or might not be observed at the program location for which a loop invariant needs to be generated. They also may lead to a violation of the safety property. The candidate generated by the Learner is herein a candidate loop invariant.

There exist different approaches for ML-based invariant generation, thus it seems reasonable to employ some within CoVEGI. Having a closer look at four existing approaches ([SNA12; KPW15; ZMJ18; GLMN14]), we observe similarities and differences, e.g., same classification approach used and different methods for generating an initial set of data. Although all approaches are evaluated experimentally, it is still challenging to find out which approach might likely be the best one for our chosen setting. A thorough comparison is hindered by several facts:

- Some implementations of the approaches are not publicly available (anymore), require specific preprocessing or input formats or only run in specific environments.
- The results reported in articles rely on specific optimizations, tuning of hyper-

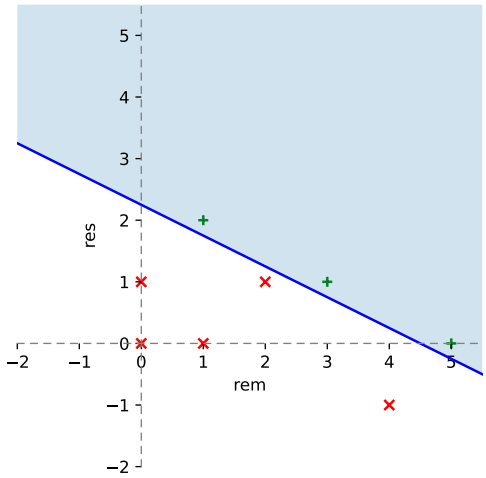


Figure 3.5: A linearly separable set of training data for two variables *res* and *rem* and the function $f_1 = res + 0.5 * rem - 2.25$

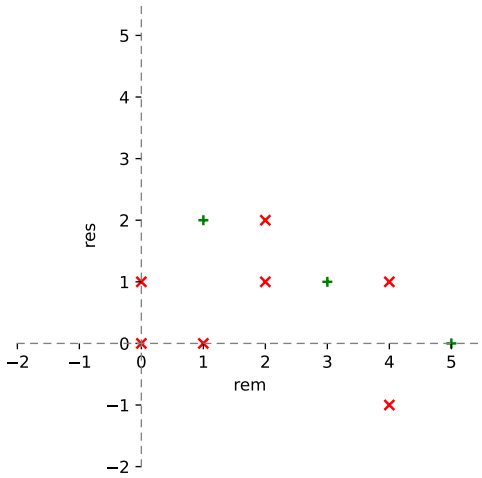


Figure 3.6: A set of training data that is not linearly separable

parameters, or additional preprocessing in corner cases that are not documented (or only within the code).

- The authors employ different benchmarks for their evaluation, often with a small intersection among the benchmark tasks used.
- Seldomly, techniques only work on the presented benchmark and are not ready to be used in other contexts.

To overcome this unsatisfactory situation, we develop a framework allowing for a conceptual and experimental comparison of ML-based invariant generation approaches. Therein, we provide *re-implementations* of the core components of ML-based invariant generation as proposed in the four selected articles on ML invariant generation ([SNA12; GLMN14; KPW15; ZMJ18]), allowing for the first time for an unbiased comparison, disregarding any specific optimizations or test case specific tunings.

Before we introduce the Modular Framework for Invariant Generation using Machine Learning (MIGML) in Section 3.2.3, we first provide some fundamentals on different ML classification techniques that we use within the framework next.

3.2.1 Fundamentals on Machine Learning Classification Techniques

Machine Learning comprises a huge variety of techniques to process data, like classification, regression, or clustering. We focus on the *classification* of data into two classes, called *binary classification*. Each data point in the set of training data provided to a ML algorithm is labeled with one of the two classes, called *supervised learning*. Conceptually, supervised learning comprises two phases: a training phase and a prediction

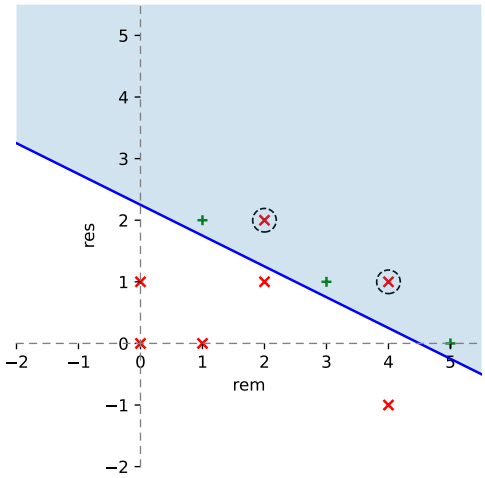


Figure 3.7: Dataset with two misclassified negative data points

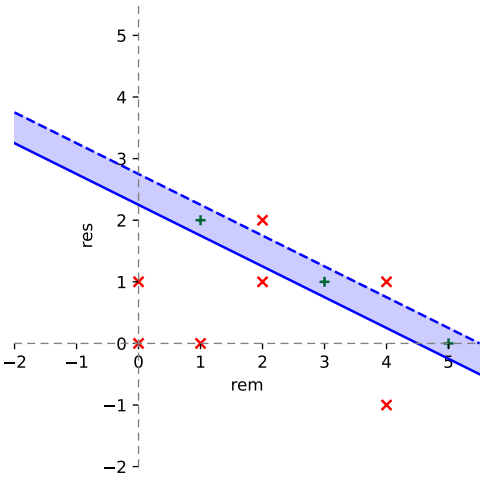


Figure 3.8: A conjunction of two linear functions correctly classifying the set of training data

phase. In the training phase, the learner is given a so-called *training set*, which it generalizes in order to obtain a model that captures the relationship between the two classes. The data in the training set are labeled according to the two classes. In our setting, the training set comprises a set of *positive* points X^+, depicted in green and a set of *negative* points X^-, depicted red. In the prediction phase, the model is used to classify new data points, not present in the training set. There are many machine learning algorithms, called *classifiers*, usable for binary classification in supervised learning. Two classifiers widely used are *Support-Vector Machines (SVMs)* and *Decision Tree (DT)* learners. As we use both within MIGml, we briefly introduced them next:

Support-Vector Machine A (linear) SVM learns a linear function $f(x) = a_1x_1 + \ldots + a_nx_n + b$, $x_i \in Var$, $a_i, b \in \mathbb{Q}$ separating the positive and negative points, i.e., $f(x) > 0$ holds for all $x \in X^+$ and $f(x) \leq 0$ holds for all $x \in X^-$. Hence, the hyperplane $f(x) = 0$ separates positive and negative points. The SVM aims to find a function that separate the points maintaining a maximal margin. For a detailed description of how to compute such a function, we refer to Bishop [Bis07]. We depict in Figure 3.5 a set of training data and the function $f_1 = res + 0.5 * rem - 2.25$ learned by the SVM[6]. As all points x in the set of training data for which $f_1(x) > 0$ holds are classified positively, the model learned by the SVM classifies all points in the light blue region positively.

A set of training data is called *linearly separable*, if there exists a linear function separating all positively labeled and negatively labeled points. We depict in Figure 3.6 a set of training data that is not linearly separable. In these cases, multiple solutions exist: One can make use of the *kernel-trick* [TK09], where the set of training data is mapped to a higher-dimensional space such that it is linearly separable. A different method that

[6]More precisely, we depict the hyperplane $f_1(x) = 0$, which is for the two-dimensional case a line.

is for example used by Sharma et al. [SNA12] is to use a combination of multiple models learned by a SVM. In case negatively labeled points are misclassified, an intersection of multiple linear functions is used. Therefore, all negative points correctly classified in the first iteration are removed from the set of training data and a new model is learned, that is conjoined with the model learned in the first iteration. Assuming that the SVM is given the set of training data depicted in Figure 3.7 and generates the linear function $f_1 = res + 0.5 * rem - 2.25$ in the first iteration, the two data points above the linear function that are marked are misclassified. In the second iteration, the updated set of training data consists of all positive points and the two negative points misclassified in the first iteration. The SVM now learns the linear function $f_2 = -res - 0.5 * rem + 2.75$, depicted as a dashed blue line in Figure 3.8 which in conjunction with f_1 correctly classifies all data in the set. The SVM has learned a model that classifies all points x positively, for which $f_1(x) > 0 \wedge f_2(x) > 0$ holds, which is equivalent to $res + 0.5 * rem = 2.5$ or $2 * rem + res = 5$ for rem, res being integers. The case that some positive points are misclassified can be handled analogously using a union of multiple models by disjoining them.

Decision Tree Learner A Decision Tree (DT) learner generates a Decision Tree as a model which is a compact way of representing a boolean formula. A *decision tree* is a binary tree, where each inner node has two successors and is labeled with predicates of the form $x > b, x \in Var, b \in \mathbb{Q}$. The edges to the successors are labeled with *true* (depicted by solid edges) or *false* (depicted by dashed edges). Each leaf is labeled with a class, in our setting either positive (1) or negative (0). A point is classified by traversing the decision tree and evaluating each predicate in the inner nodes until a leaf is reached. The learner starts with an empty tree and iteratively selects heuristically a feature that classifies the data best. For a detailed explanation of decision tree learners we refer to Quinlan [Qui93] and Breiman et al. [BFOS84].

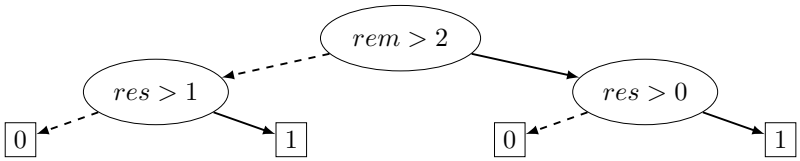


Figure 3.9: Decision tree classifying the training data of Figure 3.5

We present in Figure 3.9 a decision tree that is learned for the points given in Figure 3.5. It contains three inner nodes and four leaves. For example, for a classification of the point $(2, 1)$ it is first checked if $rem > 2$ is *true*. As $2 > 2$ is *false*, the dashed edge is followed and $res > 1$ is checked. As this predicate is also not satisfied, the point is classified negatively.

3.2.2 Motivating Example

Before we explain the details of MIGml, we exemplify the overall process to ease understanding. We make use of an SVM within the Learner and aim to learn an invariant for the program of Figure 2.3. At first, the Teacher generates a set of training data that is depicted in Figure 3.5, comprising positively and negatively labeled points. Please note that each point in the set of training data contains values for res, rem, and $input$. As the value for $input$ is 5 for all points in the set of training data, we do not show the dimension for $input$ in Figure 3.5 for representation purposes. The training data is given to the Learner, which generates the linear function $f_1 = res + 0.5 * rem - 2.25$, which is transformed into the predicate $inv_1 \equiv 2 * res + rem > 4.5$ and given to the Teacher. The Teacher checks if inv_1 is a valid loop invariant and computes counterexamples. They are used to extend the set of training data, e.g., by the two points $(2, 2, 5)$ and $(4, 1, 5)$, yielding the set depicted in Figure 3.6. As the training data is not linearly separable, the SVM generates the two functions: $f_1 = res + 0.5 * rem - 2.25$ and $f_2 = -res - 0.5 * rem + 2.75$, correctly separating the training data. The conjunction of the two functions is transformed into the predicate $inv_2 \equiv 2 * res + rem = input$. The transformation is correct, as all variables are integers. The Learner again checks the candidate inv_2 and confirms that inv_2 is a valid loop invariant and thus the final answer of the learning process. The learned invariant could then be used within CoVEGI.

3.2.3 MIGml and its Components

To be able to compare different existing approaches for invariant generation and to ease the evaluation of new ideas, we develop a Modular Framework for Invariant Generation using Machine Learning (MIGml). The framework is based on the idea of using Teacher and Learner as introduced in the ICE approach by Garg et al. [GLMN14], but we refine each to comprise two individual components to enable the realization of approaches which do not strictly follow the ICE approach. As a result, MIGml comprises four components, namely example generator, predicate generator, classifier, and model validator, depicted in blue in Figure 3.10.

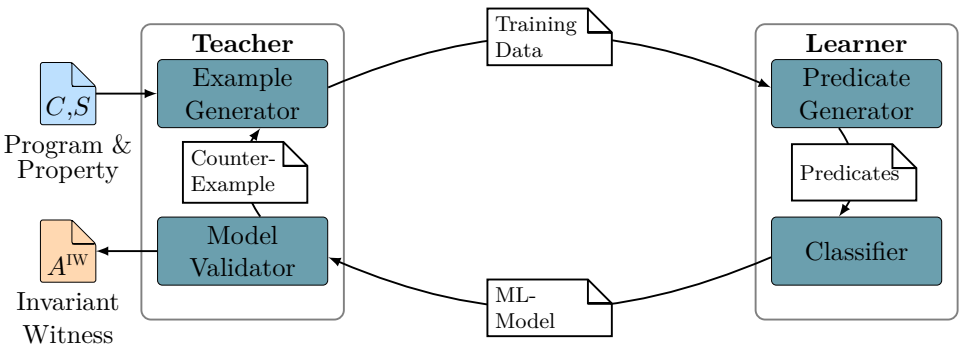


Figure 3.10: Overview of the MIGml-framework

They communicate via precisely defined interfaces which facilitate a clear decoupling and independence of components. Each component can be instantiated arbitrarily, allowing for the implementation of and experimentation with different techniques. Before describing Teacher and Learner and the four components within the framework, we first explain which information is exchanged and which formats are used for the information exchange.

3.2.3.1 Exchange Formats

Teacher and Learner exchange information using *training data* and *ML-model*. Internally, the Teacher uses a *counterexample* and the Learner a set of *predicates*. To account for components realized using different techniques and to facilitate reusing existing components, the information exchange needs to be realizable using files, that at best follow the prevalent format. These formats and the encoded information are presented next.

Training Data The training data is used for guiding the Learner towards a valid loop invariant and therefore contains variable values at the loop head, i.e., labeled program states. It comprises of *positive samples* X^+ and *negative samples* X^-. Furthermore, as employed by one approach, it may contain *implication samples* $X^{\Rightarrow}$. Intuitively, positive samples are observable in the program and negative ones are either not observable or lead to a property violation. To precisely define the elements within the training data, we assume a program is given as CFA C having a single loop at location ℓ_h. To account for the latter definition of negative samples, we require the same form of safety property as for helpful loop invariants defined in Definition 2.5, namely, having the safety property $S = (\ell_{err}, \omega)$, where ω is asserted at ℓ_{err}. A state σ_h is labeled *positively* whenever there exists a path $\pi \in paths(C)$ such that $\pi = \langle \ell_0, \sigma_0 \rangle \xrightarrow{g_1} \ldots \xrightarrow{g_h} \langle \ell_h, \sigma_h \rangle$. Depending on the approach, the definition of negative states differs slightly. A state σ_h is labeled *negatively* if (in some approaches) there is no path $\pi' \in paths(C)$ such that $\pi' = \langle \ell_0, \sigma_0 \rangle \xrightarrow{g_1} \ldots \xrightarrow{g_h} \langle \ell_h, \sigma_h \rangle$ or (in other approaches that assume that programs are correct) there is a path $\pi \in paths(C)$ such that $\pi = \langle \ell_h, \sigma_h \rangle \xrightarrow{g_{h+1}} \ldots \xrightarrow{g_j} \langle \ell_{err}, \sigma_{err} \rangle$ and $\sigma_{err}(\omega) = \mathit{false}$. Finally, an implication sample [GLMN14] is a pair of states, where the second one is labeled positively whenever the first one gets classified as positive by the learned model. If the first state is classified as negative, the second one can be ignored. Implication samples are used when generating new samples based on the preservation condition (2.5), as it is in advance unclear, whether the first point, referring to the state before the loop execution, is observable. In the following, we refer to these states or state pairs jointly as *data points*.

Predicates The set of predicates over the variables contained in the training data is used to increase the expressiveness of the invariants generated by the Learner. Predicates can either be given as additional input to the classifier or are directly encoded

within the training data. In the former case, an SMT-2-LIB [BST10] conformal notation is used. In the latter case, the training data are extended by calculating the concrete value for the terms used within the predicates for each data point. For example, if the training data contains the data point $(x \mapsto 3, y \mapsto 2)$ for the variables and the term $x - y$ for the predicate $x - y > 0$, a new entry with the term $x - y$ and the calculated value is added to the data point (hence $(x - y) \mapsto 1$).

ML-Model The ML-model is used for communication of the result of the learning process to the Teacher and thus contains the candidate invariant computed by the Learner. It is generated by transforming the learned model into a boolean formula. The formula is encoded in the standardized SMT-2-LIB [BST10] format.

Counterexample The counterexample is used for extending the current training data in case the ML-model is not a valid loop invariant. It contains the formula for the violated condition and satisfying assignments for the variables and uninterpreted functions present in the formula.

3.2.3.2 Learner

The Learner comprises the two components *predicate generator* and *classifier*. Its task is to learn a classification for the training data separating the positive and negative points, that does not contradict any implications. The first component, the predicate generator, may generate additional predicates that serve as additional inputs for the classifier. The predicate generator can employ multiple techniques: Using templates (e.g., predicates from the octagon domain [Min06] of the form $\pm x \pm y \leq c;\ x,y \in Var, c \in \mathbb{Q}$), logic-based approaches, or using the ML-model that is generated by a different classifier.
The training data are labeled using two labels. Thus, ML algorithms for binary classification in supervised learning are required. Implication points do not match the binary classification setting. Thus, the learning algorithms used need to be adjusted to be able to process them. Note that we require in the setting of learning invariants that the learned model perfectly classifies all given points inside the training data. After the learning phase is completed, the learned model is transformed into an *ML-model* containing the candidate invariant using a boolean combination of predicates over the program variables. For this, we require the use of classifiers which generate models having a (compact) boolean representation. Hence, we need a mechanism to translate the classification learned by the classifier into a boolean formula.

3.2.3.3 Teacher

The Teacher's main objective is to guide the Learner towards finding an invariant by providing suitable training data. It comprises of an *example generator* and *model val-*

Algorithm 2 MIGml-Algorithm

Input: C ▷ *CFA*
S ▷ *safety property*
Teacher = (ExampleGenerator, Validator) ▷ *Teacher*
Learner = (PredicateGenerator, Classifier) ▷ *Learner*
Output: A^{IW} ▷ *Invariant witness*

```
1: data := ExampleGenerator.generateInitial(C, S);
2: term := false;
3: while (¬term) do
4:   pred := PredicateGenerator.gen(C, S);
5:   ml_model := Classifier.learn(data, pred);
6:   term, cex := Validator.validate(ml_model, C, S);
7:   if (¬term) then
8:     data := ExampleGenerator.update(data, cex);
9: return Teacher.generateWitness(ml_model); ▷ Return witness
```

idator. The model validator is given the ML-model and may check if the candidate encoded is a valid or even helpful loop invariant, e.g., using verification techniques based on CHCs or strongest postcondition. In case the candidate is not approved a counterexample is generated and given to the example generator.
The example generator uses the information of the counterexample to extend the training data. Intuitively, if the candidate violates the establishment condition (2.4), additional positively labeled data is added. In case the loop invariant is not helpful, new negatively labeled data is generated and added, and if the preservation condition (2.5) is violated, new implication points are generated. Moreover, the example generator may produce an initial set of training data for the Learner.

3.2.3.4 Learning Process

The overall learning process for a MIGml instance, using all four components, is formalized in Algorithm 2. The process is completed when the model validator does not generate new counterexamples anymore. Then, the last ML-model contains the final invariant and an invariant witness A^{IW} that is returned as the learning process' final answer is generated.
Having a learning process comprising the four components yields multiple advantages: It allows us to realize components in MIGml that are not described as instances using Teacher and Learner, create new combinations, and model implicit execution steps explicitly. Next, we explain how to realize existing approaches for ML-based invariant generation using MIGml.

Table 3.1: Overview of the concepts for learning invariants used with their realization as instances in MIGml

		SNA12	GNMR16	KPW15	ZMJ18
General:	**Objective**	Learn Interpolants	Learn Loop Invariant	Learn Loop Invariant	Verify Programs
	Detect property violations	✗	✗	✗	✓
	Training Data	X^+, X^-	$X^+, X^-, X^\Rightarrow$	X^+, X^-	X^+, X^-
Teacher:	**Example Generator**	Logic (sp)	Logic(sp)	Execution	Logic(CHC)
	Initial Training Data	✓	✗	✓	✗
	Training Data Refinement	✓	✓	✗	✓
Learner:	**Predicate Generator**	✗	Octagon-template	Octagon-template	Classification &Octagon-template
	ML-Classifier	SVM	DT-learner	DT-learner	DT-learner

3.2.4 Existing Approaches for Invariant Generation in MIGml

The modular structure of MIGml allows its instantiation with existing concepts for machine learning-based invariant generation. Thereby, it facilitates a conceptual comparison of different approaches as well as an experimental evaluation of them on equal grounds. We selected four existing and conceptually different approaches, published by Sharma, Nori and Aiken (SNA12) [SNA12], Garg, Neider, Madhusudan and Roth (GNMR16) [GNMR16], Krishna, Puhrsch and Wies (KPW15) [KPW15] and Zhu, Magill and Jagannathan (ZMJ18) [ZMJ18]. The four approaches have slightly different objectives and employ different techniques within the four MIGml components. Due to the conceptual differences, we consider our selection to be reasonable w.r.t. the variety of existing machine learning approaches for invariant generation. Next, we shortly introduce each of the four approaches and summarize the results in Table 3.1. We provide a more detailed discussion and a conceptual comparison that is possible due to the modular structure of MIGml in Appendix A.1.3.

SNA12 To learn interpolants and likely invariants, two formulas are generated for programs with a single loop: The first formula, representing establishment and preser-

vation, cf. (2.4) and (2.5), summarizes the path from the program entry with one loop iteration. The second one encodes the paths from the loop head to the assertion at the program's end, used for generating helpful loop invariants (cf. (2.7)). These formulae are used for model validation and example generation, where ten positive and negative data points are sampled. In case the learned candidate violates one of the conditions, new data points are inferred. Within the Learner, an SVM is used as classifier, learning predicates from the Polyhedra domain [CH78; DKW08]. If the training data is not linearly separable, a combination of SVMs is used, as explained in Section 3.2.1. No additional predicates are used as input to the SVM.

GNMR16 The idea of *ICE learning* [GLMN14; GNMR16] aims at generating loop invariants that help in proving the program correct. For model validation and example generation three formulas representing establishment (2.4), preservation (2.5), and check (2.7) are generated. Initially, an empty set of training data is used. If the candidate generated by the Learner violates establishment (respectively check), a new positively (respectively negatively) labeled data point is generated. Whenever an ML-model violates the preservation condition, the training data is extended with a new implication point. On the Learner side, the approach employs a decision tree learner that is able to also process implications [GNMR16]. The classifier is also able to handle predicates as additional input. In this approach, all predicates from the octagon domain [Min06] are generated using a template.

KPW15 To learn loop invariants, a similar Learner instantiation as in GNMR16 is used, applying a decision tree as classifier and generating predicates from the octagon domain. The main idea of this approach is to generate a single, rich set of training data and use the decision tree learner only once, not iteratively as in the other approaches. Therefore, the program is executed with input values from a predefined interval, and observed states are used as positive data points, negative ones are obtained by mutating the positive data points and checking, if executing the program leads to a property violation. If the model validator confirms that an ML-model is a valid loop invariant, an invariant witness is generated. Otherwise, the process aborts without a result.

ZMJ18 To verify a program, CHCs are generated (cf. Section 2.4.6) that contain uninterpreted functions. The machine learning approach is asked to learn a predicate for each uninterpreted function symbol, where one function is generated per loop that models the loop invariant. Initially, an empty set of training data is used and the learned interpretations for the uninterpreted function in the CHCs are checked in the validation step. The CHC generated correspond (among others) to the initialization (2.4) establishment (2.5) and check (2.7). Positive points are generated and all negative ones are discarded whenever the uninterpreted functions within the CHC corresponding to the initialization condition (2.4) is violated. Negative points are added if a CHC

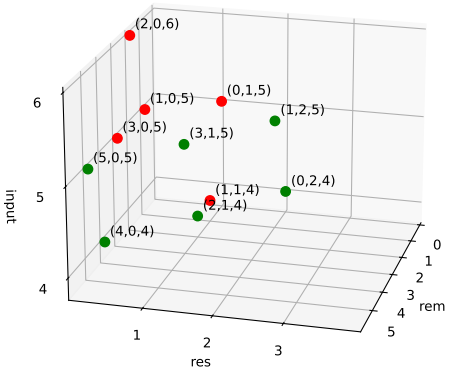


Figure 3.11: Initial set of training data

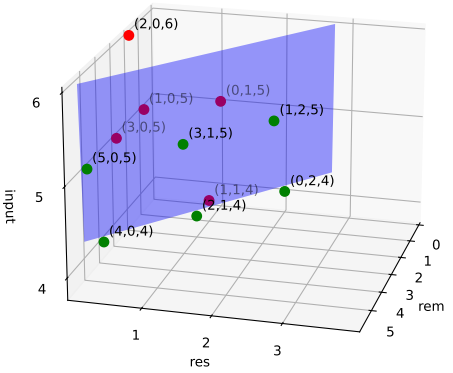


Figure 3.12: Classification of the initial set of training data using the SVM

corresponding to establishment (2.5) or if check (2.7) is violated and it cannot be decided if the new data point is observable at the loop head using the set of training data. For the Learner, a decision tree is used as classifier. Predicates are generated using an octagon-based template as well as the classification learned by an SVM.

Artifact Availability of the Approaches As we aim to use existing approaches off the shelf, we are interested in the availability and reusability of the provided software artifacts. For ZMJ18, the artifact is available at GitHub [ZGNK18], but we were not able to execute the tool due to issues while building it. Both artifacts of KPW15 and GNMR16 are not publicly available anymore, but the authors provided them and we included them in our own artifact [HW21c]. The tool developed by Krishna et al. is only applicable to tasks used in their evaluation, as it requires some manual transformation. The implementation of Garg et al. is applicable to Boogie programs only and it is only executable on Windows-32Bit. For SNA12, no artifact is available anymore.

3.2.5 Example Application of MIGml

To exemplify the advantages of the MIGml framework, we apply a newly built configuration to the running example from Figure 2.3. Let the configuration make use of X^+ and X^- in the training data, an example generator using the strongest postcondition for the validation of a ML-model (as in GNMR16), an example generator executing the program to sample data points for the initial set of training data (as in KPW15), and the Learner that is used by ZMJ18, i.e., using the classification of a SVM and an octagon template for the predicate generator and a DT-learner as classifier.

In the first step, the initial set of training data is computed by the Teacher. The set generated is shown in Figure 3.11 and comprises in total of seven positive points depicted in green and five negative points depicted in red. This set is given to the Learner. The predicate generator uses an SVM to classify the data and generate additional predicates. The hyperplane learned by the SVM is depicted in Fig-

ure 3.12, which is $f_1 \equiv rem + 2 * res - input + 0.5 = 0$ and is transformed in the predicate $p_1 \equiv rem + 2 * res - input + 0.5$. The set of predicates comprises $\{p_1,\ \pm rem \pm res \leq c,\ \pm rem \pm input \leq c,\ \pm res \pm input \leq c\}$ and their negations are given as additional input to the classifier using a DT learner. The DT that is learned is depicted in Figure 3.13 and is transformed into the invariant candidate $inv_1 \equiv input \leq 5 \wedge rem + 2 * res - input + 0.5 > 0$, that is encoded within the ML-model and given to the Teacher.

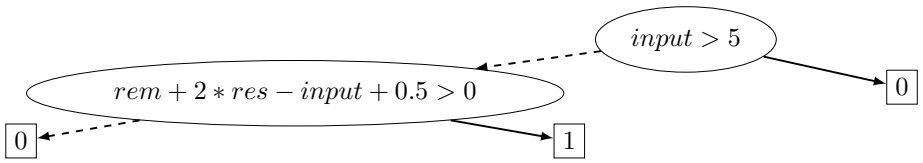


Figure 3.13: Decision tree classifying the training data of Figure 3.11

The Teacher computes within the model validator for an invariant inv the three conditions initialization (I_C), preservation (P_C) and check (C_C) in SSA form for the program C_1. To increase readability, we have simplified the conditions by removing or inlining the ϕ nodes and other SSA-related transformations:

$$\begin{aligned} I_C \equiv &\big((input_0 < 0 \wedge input_1 = -input_0) \vee (input_0 \geq 0 \wedge input_1 = input_0) \\ &\wedge (rem_0 = input_1 \wedge res_0 = 0)\big) \Rightarrow inv \\ P_C \equiv &\big(inv \wedge rem_1 > 1 \wedge rem_2 = rem_1 - 2 \wedge res_2 = res_1 + 1\big) \Rightarrow inv' \\ C_C \equiv &\big((input_0 < 0 \wedge input_1 = -input_0) \vee (input_0 \geq 0 \wedge input_1 = input_0) \\ &\wedge rem_1 = input_1 \wedge res_1 = 0 \wedge inv \wedge rem_1 \leq 1\big) \Rightarrow input_1 = 2 * res_1 + rem_1 \end{aligned}$$

Within the equations, inv makes use of the variables $rem_1, res_1, input_1$ and inv' of $rem_2, res_2, input_1$. Now, the candidate inv_1 encoded in the ML-model is validated. Unfortunately, all three conditions I_C, P_C, and C_C are violated and the counterexamples are given to the example generator. Depending on its configuration, new data points are generated. Let us assume that three additional data points $(2, 2, 6)$ for X^+ and $(4, 1, 5)$, $(5, 1, 6)$ for X^- are generated. The second iteration starts with the extended training data that is given to the Learner. Within the predicate generator an SVM is trained on the data first. As the data is not linearly separable, no hyperplane perfectly classifying the data can be found. The generated hyperplane that classifies most data points correctly is again $f_1 \equiv rem + 2 * res - input + 0.5 = 0$. The two negative data points $(4, 1, 5)$, $(5, 1, 6)$ are missclassified, as depicted in Figure 3.14. Hence, the SVM is asked to classify a reduced set comprising of X^+ and $X^- = \{(4, 1, 5), (5, 1, 6)\}$. The classification of the dataset that is learned is $f_2 \equiv -rem - 2 * res + input + 0.5 = 0$, as depicted in Figure 3.15. The conjunction of the two halfspaces and hence of the predicates generated using f_1 and f_2 is the learned predicate:

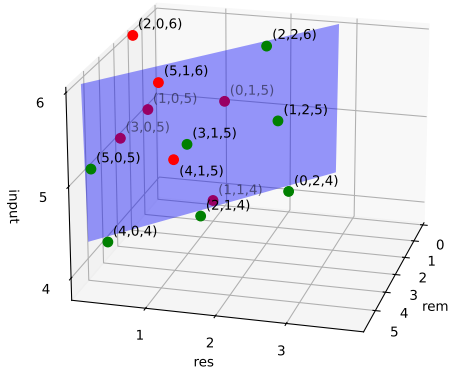


Figure 3.14: Classification with two misclassified data points

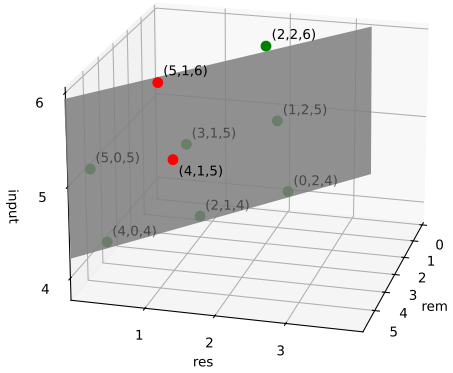


Figure 3.15: Classification of the reduced set of training data

$$p_2 \equiv rem + 2 * res - input + 0.5 > 0 \wedge -rem - 2 * res + input + 0.5 > 0 \tag{3.1}$$

$$\Leftrightarrow rem + 2 * res - input > -0.5 \wedge rem + 2 * res - input \leq 0.5 \tag{3.2}$$

$$\Leftrightarrow -0.5 < rem + 2 * res - input \leq 0.5 \tag{3.3}$$

$$\Leftrightarrow rem + 2 * res - input = 0 \tag{3.4}$$

The predicate that is generated is thus $rem + 2 * res - input$. The transformation from 3.3 to 3.4 is valid, as all variables used in the program are integers. Thus, the set of predicates given as additional input to the classifier is $\{p_2,\ \pm rem \pm res \leq c,\ \pm rem \pm input \leq c,\ \pm res \pm input \leq c\}$. The Decision Tree-learner generates a DT that consists of two decision nodes and represents the formula $inv_2 \equiv rem + 2 * res = input$, as depicted in Figure 3.16.

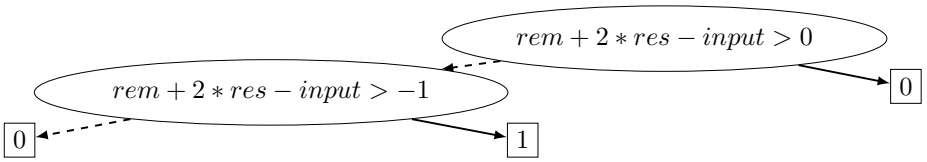


Figure 3.16: Decision tree representing the loop invariant $rem + 2 * res = input$ classifying the training data of Figure 3.14

As the invariant inv_2 encoded within the ML-model is a valid and helpful loop invariant, thus satisfying I_C, P_C and C_C, the model validator confirms the invariant and an invariant witness A^{IW} is generated and returned as the learning process' final answer.

3.3 Implementation

To analyze the feasibility of CoVEGI and MIGml we build a prototype for CoVEGI and MIGml described next. Both implementations are available and software artifacts are archived at Zenodo [HW21b; HW21c].

3.3.1 Implementation of CoVEGI

To be able to analyze the performance of CoVEGI, we build a prototype within the CPAchecker. We implemented Algorithm 1 within CPAchecker 1.9.1, as a configurable algorithm. Thereby, we can instantiate the algorithm with different main verifiers and helper invariant generators. For the evaluation, we aim to use CPAchecker's implementation of predicate abstraction and k-induction. To ensure that witness injection as explained in Section 3.1.3 is fully supported, we build some minor extensions within the existing implementation.

To the best of our knowledge, there existed no standalone and publicly available invariant generators, that generate invariants for both, global and local variables, without doing a full verification, when we built and evaluated the prototype of CoVEGI in 2020. To be able to still evaluate CoVEGI, we decided to use off-the-shelf verifiers as invariant generators instead. We ignore the computed verdict and only use the invariant witness generated. We selected the tools SeaHorn (cf. Section 2.4.6), UltimateAutomizer (cf. Section 2.4.2) and VeriAbs (cf. Section 2.4.5), where the latter two participated in the SV-COMP [Bey20]. Both UltimateAutomizer and VeriAbs achieved excellent results in 2020's SV-COMP, being the reason to chose them. To exemplify that CoVEGI can employ arbitrary off-the-shelf tools as helper invariant generator, we select as third tool SeaHorn, a verification tool neither currently participating in the SV-COMP nor producing witnesses. It operates on LLVM-IR, therefore we used the mapper and encoder exemplified in Appendix A.1.1. The three helper invariant generators are used as black-boxes and employ verification techniques complementary to those of both the other helpers and the two main verifiers. For VeriAbs and UltimateAutomizer we use the versions evaluated in the SV-COMP 2020[7]. Since there is no precompiled binary of SeaHorn, we employ the docker container of the 2020 version[8]. All three helper invariant generators are used in their default configuration.

3.3.2 Implementation of MIGml

One goal of MIGml is to allow us to compare different approaches on equal grounds. Some implementations of existing approaches are (1) not publicly available, (2) only executable in a specific execution environment or for a limited set of tasks, and (3) apply hyperparameter tuning, preprocessing, or additional optimizations for certain tasks. Hence, such a comparison cannot rely on the existing implementations. The Association for Computing Machinery (ACM) provides criteria for reproducing experiments [Ass20], requiring among others that only parts of the artifacts provided by the authors are used. We therefore decided to build a prototype for the MIGml framework as an independent standalone tool.

The MIGml framework itself comprises the execution skeleton, steering the ex-

[7] https://gitlab.com/sosy-lab/sv-comp/archives-2020/tree/master/2020
[8] suggested by the developers; used docker seahorn/seahorn-llvm5 (4c01c1d)

ecution and implementing Algorithm 2 and instances for Teacher and Learner. We employ existing tools (like the CPAchecker [BK11]), that are able to process a rich set of tasks, but do not primarily focus on efficiency while developing the prototype. As we aim to allow easy creation of new combinations and extensions of existing ones with new components, each instance is configured using configuration files. Next, we shortly introduce the components realized within MIGml. A more detailed description containing implementation details is given in Appendix A.1.4.

Beneath instances of Teacher and Learner, we provide utility functions to collect information on variables (*Default variable collector*), to execute a program with input values from intervals (*Interval-based program executor*), or symbolically (*Symbolic execution based program executor*) and to determine, whether a certain state results in a property violation (*Injectable value analysis*). The *Symbolic execution based program executor* is realized using Klee [CDE08; CN20], the other functions using CPAchecker [BK11].

3.3.2.1 Learner

In contrast to most application areas of machine learning, the data points in our training data are not noisy, meaning that *all* data points need to be classified correctly by the learned ML-model. Moreover, a precise transformation of the generated models into a boolean representation is required. We implement two different classification algorithms, namely SVM and decision tree learner.

SVM We have already exemplified in Section 3.2.1, how to represent a model learned by an SVM using boolean formulae. Due to the learning algorithm employed by the SVM, the coefficients present in the learned model are real values, where the variables present in the program are (in most cases) integers. Consequently, an invariant containing integer coefficients like 2 or reals like 0.5 is more likely correct than an invariant containing 1.999974 or 0.5192. Based on the SVM implementation from the scikit-learn library [Sci20], we have implemented two different rounding mechanisms, called *close rounding* (Svm-C) inspired by SNA12 and *scaled rounding* (Svm-S), inspired by ZMJ18. Svm-C rounds only reals that are close[9] to the next integer, whereas Svm-S searches a scaling factor such that all coefficients are close to an integer.

Decision Tree Learner The decision tree learner based on the scikit-learn library is called Dt-Skl. The boolean formula generated out of a tree is the disjunction of the formulae for every path from the root to a leaf labeled with 1, which themselves are conjunctions of the boolean conditions on the nodes. Additional predicates are integrated into the training data as proposed in [KPW15] and explained in Section 3.2.3.1.

[9] We define close as a configurable parameter denoting the maximal distance to the nearest integer, by default 0.1.

To be able to employ implication points, we integrated the existing decision tree learner (called Dt-Ice) proposed in 2016 by Garg et al. [GNMR16].

Predicate Generator MIGml contains a template-based predicate generator based on the octagon domain, called Pg-Oct.

3.3.2.2 Teacher

The teacher's main task is to guide the learner towards finding an invariant. It comprises two components, model validator and example generator.

Model Validator For the model validator, we implement three different components: Val-Ip and Val-SP both make use of the strongest postcondition operator and are realized using CPAchecker, where Val-Ip computes the two formulas described by SNA12, and Val-SP, used by GNMR16, generates establishment, preservation, and check (see (2.4), (2.5) and (2.7)). Val-CHC is based on CHCs, which is used by ZMJ18. The CHCs are generated using the tool Korn [Ern20; Ern23], that generates CHCs for C programs directly and exporting them as SMT-2-LIB-code. Inside the model validators we employ the Java-SMT library [KFB16; BBF21], providing support for widely used solvers, where we currently use Z3.

Example Generator For the initial example generation, MIGml contains a logic-based (EX-Log) and execution-based example generator (EX-Exec). The logic-based example generator generates a fixed number of satisfying assignments for each formula and extracts the data points. In contrast, EX-Exec generates data points using the interval-based (EX-I) or symbolic execution-based program executor (EX-S). These points are mutated, resulting in a set of candidate negative points, that is checked for violating the postcondition using the injectable value analysis, as proposed by KPW15.

3.4 Evaluation

After having introduced the concepts of CoVEGI for cooperative verification using externally generated invariants and MIGml for invariant generation using machine learning, we evaluate both concepts next. The goal of the evaluation is two-fold: Firstly, we want to investigate, whether externally generated invariants ease the verification problem, focusing on effectiveness and efficiency. We analyze this in RQ1, RQ2 and RQ3, cf. Sections 3.4.2 to 3.4.4. Secondly, we are interested in answering in RQ 4 (cf. Section 3.4.5), if these invariants could be generated using ML, i.e., using our novel MIGml framework.

3.4.1 Experimental Setup

To answer the first three research questions regarding the use of externally generated invariants, we evaluate the general performance of the concept of CoVEGI as well as compare different configurations. As explained in Section 2.5, we focus on both effectiveness and efficiency, generally aiming at checking whether the use of CoVEGI can increase the number of correctly solved verification tasks within the same resource limits. We explain the slightly different setup used for RQ 4 (cf. Section 3.4.5) later.

Configuration The CoVEGI algorithm presented in Algorithm 1 offers four configuration options. During our evaluation, we set `termAfterFirstInv` and `restartMain` to *true*, the `timerM` to 50s, and the `timeoutH` to 300s. Other configurations using different values for `timerM` and `termAfterFirstInv` are evaluated in [HW21a] and for `restartMain` in Appendix A.1.5. We use the abbreviations *SH* for SeaHorn, *UA* for UltimateAutomizer, and *VA* for VeriAbs.

Benchmark Tasks The verification tasks used are taken from the set of SV-COMP 2020 benchmarks [SVB20]. As we are interested in generating loop invariants externally, we selected all tasks from the category `ReachSafety-Loops`. The programs used as tasks in this category of the SV-Benchmarks contain at least one loop. To obtain a more broad distribution of tasks, we randomly selected 55 additional tasks from the categories `ProductLines`, `Recursive`, `Sequentialized`, `ECA`, `Floats` and `Heap` that contain loops, yielding in total 342 tasks.

Computing Resources We conducted the evaluation on three virtual machines, each having an Intel Xeon E5-2695 v4 CPU with eight cores and a frequency of 2.10 GHz and 16GB memory, running an Ubuntu 18.04 LTS with Linux Kernel 4.15. We run our experiments using the same setting as in the SV-COMP, giving each task 15 minutes of CPU time on 8 cores and 15GB of memory and use BenchExec (cf. Section 2.5.1).

Availability The implementations of CoVEGI and MIGml as well as all experimental data are publicly available and archived at Zenodo. We archive in [HW21b] CoVEGI and the data for RQs 1-3, and in [HW21c] MIGml and the data for RQ 4.

3.4.2 RQ 1: Can CoVEGI Increase the Effectiveness of a Main Verifier?

Evaluation Plan We let the CoVEGI framework run with a single invariant generator and compare the results to a standalone run of the main verifier. In total, we evaluate eight different configurations.

Table 3.2: Comparison of the two main verifiers running standalone, using a single and two helper invariant generators

k-induction	**alone**	**+SH**	**+UA**	**+VA**	**+SH-UA**	**+SH-VA**	**+UA-VA**
correct overall	146	148	158	163	153	156	163
correct proof	102	104	114	119	109	112	119
additional proof	-	3	13	19	7	11	19
correct alarm	44	44	44	44	44	44	44
additional alarm	-	0	0	0	0	0	0
incorrect overall	1	1	1	1	1	1	1

predicate abstr.	**alone**	**+SH**	**+UA**	**+VA**	**+SH-UA**	**+SH-VA**	**+UA-VA**
correct overall	116	122	132	125	130	130	136
correct proof	78	84	94	87	92	92	98
additional proof	-	6	16	9	9	9	15
correct alarm	38	38	38	38	38	38	38
additional alarm	-	0	0	0	0	0	0
incorrect overall	1	1	1	1	1	1	1

Experimental Results Before having a detailed look at our experimental results, we can already state that the cooperative idea used within CoVEGI itself is feasible: For our proof of concept implementation, we only had to build minor extensions within the main verifiers implementation to fully support witness injection. To be able to employ the three helper invariant generators within CoVEGI, we do not change a single line of code of the tools, except for adding encoders and mappers if needed. Integrating helpers like VeriAbs, which do not require a mapper nor an encoder, can be done within a few lines of code. Although the implementation is a proof of concept, this shows that it is applicable to all kinds of off-the-shelf helper invariant generators, those producing verification witnesses as well as those generating invariants in an IR.

To evaluate whether a main verifier benefits from the support of a helper in the form of invariants, we execute a combination of a main verifier and a helper in the default configuration and compare it to the main verifier running standalone. Here, we are interested in the number of correct answers overall, correct proofs and alarms, as well as in the number of incorrect answers. In addition, we also state the number of additional proofs and alarms, i.e., those tasks that are solved by one instance of CoVEGI but not be the main verifier running standalone.

We summarize the results in the left half of Table 3.2. Running standalone, k-induction can correctly solve 146 of the verification tasks, predicate abstraction 116. Both tools compute one incorrect alarm, also reported by any of the framework configurations used. When comparing the number of correctly solved tasks for the analyses running standalone compared with instances using CoVEGI, we observe that the effectiveness is increased, as each instance can solve more tasks. Using k-induction with

VeriAbs leads to 19 additional proofs, using predicate abstraction with UltimateAutomizer to 16 additional proofs. As expected, this applies to verification tasks matching the safety property only, as the invariant generators can help in proving the correctness, but cannot help in refuting properties (as none of the helpers employed is generating invariants in these cases). Besides the additionally solved tasks, there are also one (for SH and UA) and two (for VA) tasks, respectively, which cannot be correctly solved anymore when using k-induction as main verifier. In these cases, the main verifier consumes most of the CPU time available, hence sharing resources in cooperation with the helpers results in a timeout. For predicate abstraction, all tasks solved standalone are also solved by the main verifier.

Results

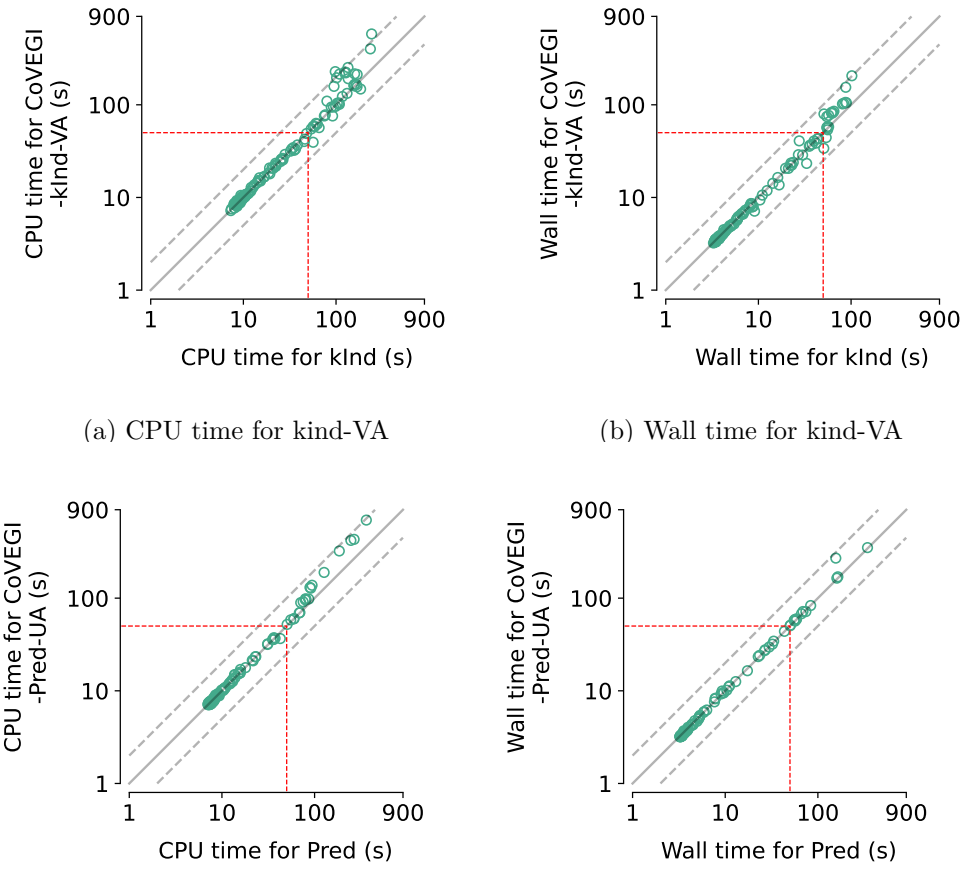
On our data set, the total number of correctly solved tasks using CoVEGI increases by 12% for k-induction and 14% for predicate abstraction as main verifier compared to the main verifier used standalone.

3.4.3 RQ 2: How does CoVEGI Impact the Verifier's Efficiency?

Evaluation Plan Next, we evaluate the efficiency of CoVEGI. Therefore, we compare the CPU time and the overall time (wall time) spent solving the verification tasks. A CoVEGI instance runs the main verifier standalone within the first 50 seconds and eventually shares the resources between main verifier and helper invariant generators. Thus, we expect that more CPU time is needed to compute a correct result after the helper invariant generator is started for tasks that can also be solved by the main verifier standalone.

Experimental Results We present in Figure 3.17 four log-scaled scatter plots comparing the CPU time and the wall time for the best performing instances of CoVEGI for predicate abstraction and k-induction with the standalone analyses per task for tasks that are solved by both verifiers. A point (x, y) in the plot means that the analysis running standalone needs x seconds CPU resp. overall time to compute a solution, whereas the CoVEGI instance needs y seconds. For points on the solid, diagonal line both analyses take the same time, while for points on the dashed lines below and above one analysis takes twice as long as the other. All tasks solved within 50 seconds are contained in the red dashed box.

When analyzing the scatter plots, we, first of all, observe that the standalone analyses and the CoVEGI instances behave equally within the first 50 seconds CPU time (cf. Figures 3.17c and A.3e). After 50 seconds, CoVEGI is faster with respect to CPU and wall time for some tasks compared to the standalone verifier. For other tasks, we observe that the cooperative approach is slower, as expected. With respect to the overall time depicted in Figures 3.17d and A.3f, the difference between both tools is even smaller, as executing main verifier and helper invariant generator in parallel do



(a) CPU time for kind-VA

(b) Wall time for kind-VA

(c) CPU time for Pred-UA

(d) Wall time for Pred-UA

Figure 3.17: Scatter plots comparing the execution time of CoVEGI instances with the main verifier running standalone

not negatively influence the overall execution time. All instances of CoVEGI need on average at most five seconds of wall time more than the standalone verifier, the difference between the median values is within a second.

Results

On our dataset, collaborative invariant generation does not significantly negatively impact the efficiency. We even see in some cases small improvements.

3.4.4 RQ 3: Does Using Invariant Generators in Parallel Pays Off?

Evaluation Plan After having compared the default configuration using a single helper invariant generator, we want to analyze whether it is beneficial to run two invariant generators in parallel and to find the best combination. We thus studied the effectiveness and efficiency of CoVEGI using two helpers in parallel, yielding three combinations per main verifier.

Experimental Results The results for effectiveness for the three combinations are summarized on the right-hand side of Table 3.2. For checking whether a parallel execution of two helpers is beneficial, we compare these numbers to the results of CoVEGI using only a single helper invariant generator and the main verifier running standalone, which are given in the left-hand side of the table. We see that predicate abstraction benefits from using two helpers in parallel, especially using UltimateAutomizer and VeriAbs. Using CoVEGI with these tools perfectly combines their strengths, thereby increasing the number of correctly solved tasks in total by 17%. For k-induction, none of the combinations of two helpers outperforms CoVEGI using VeriAbs only, which can solve 163 tasks. For UltimateAutomizer and VeriAbs as helper invariant generators, also 163 tasks are correctly solved. Compared to CoVEGI using only VeriAbs, the set of solved tasks deviates. For instance, nearly 50% of the additional tasks solved using both as helpers are not solved using only UltimateAutomizer as a helper and vice versa. This result is based on the fact that two helper invariant generators have to share the available CPU time in the combination. Hence, tasks that are solved using one of them as a helper alone could not be solved anymore in a combination, because the instance of CoVEGI runs into a timeout. With respect to efficiency, it turns out that using two helpers in parallel yields a higher CPU time for correctly solved tasks, whereas the overall consumed time does not change when using one or two helpers in parallel (cf. Appendix A.1.2).

When using all three helper invariant generators in parallel we notice that the resource sharing among all components becomes even more an issue. The CoVEGI instance using k-induction as main verifier and all three helpers in parallel can solve only 154 tasks, using predicate abstraction only 129 tasks. We also evaluated different configurations that wait for all helpers to finish (using `termAfterFirstInv` as *false* and 100s, 200s for `timeoutH`) and that do not restart the main verifier (`restartMain` is *false*) in [HW21a; HW20]. We observe that none of the configurations using all three helper invariant generators yield a better effectiveness.

Results

On our dataset, CoVEGI can increase the total number of correctly solved tasks using UA and VA in parallel; in general waiting for the other tool to also finish its computation does not pay off.

3.4.5 RQ 4: Can MIGml Generate Loop Invariants?

Evaluation Plan: Next, we answer the question of whether machine learning can be used to learn loop invariants for program verification and thus validate if MIGml could be used as a component within CoVEGI. Therefore, we evaluate the four instances that represent the approaches presented in [SNA12; KPW15; ZMJ18; PSM16] using the same setup and thus on equal grounds. The feasibility of MIGml to reproduce and partially confirm the reported results is analyzed in detail in Appendix A.1.6. We

Table 3.3: Realization of existing concepts in MIGml

	ExGen	Validator	PredGen	Classifier
SNA12	EX-Log	Val-Ip	✗	Svm-C
GNMR16	EX-Log	Val-SP	Pg-Oct	Dt-Ice
KPW15	EX-I	✗	Pg-Oct	Dt-Skl
ZMJ18	EX-Log	Val-CHC	Pg-Oct, Svm-L	Dt-Ice

focus during this research question mainly on effectiveness, i.e., the number of correctly generated loop invariants, rather than on efficiency, as efficiency was not the primary focus when developing the prototype of MIGml. As the learning process involves randomness, we run each tool ten times and aggregate the results as follows: An instance generates a helpful loop invariant if at least one answer contains a helpful invariant, a valid (but not helpful) invariant if at least one answer contains a valid invariant, and invalid if at least one invalid is generated; otherwise the result is unknown/timeout (helpful $\succ$ valid $\succ$ invalid, cf. (2.4), (2.5) and (2.7)). The invariants given as the final answer are evaluated using the strongest postcondition semantics.

Configuration. To reproduce the result from the four approaches, we create configurations making use of the core components explained in Section 3.3.2, that are summarized in Table 3.3. We contacted all authors and discussed our findings and our implementation. Each configuration is used for all tasks. Within their implementation, KPW15 uses for EX-I different, hard-coded values for each task, which is not applicable for new tasks. We thus use the median values $L = [0, 2]$ for the interval, $I = 100$ for the number of loop unrollings, and $M = 2$ as parameters for the permutation. We also evaluated other parameters (e.g., maximal value) and obtained similar results, cf. [HW22b].

Computing Resources. The experiments are conducted on the same machines as for research questions 1,2 and 3 but with a decreased available CPU time of 5 minutes, which is the largest timeout used within the four evaluations and as default for CoVEGI.

Benchmark Tasks For the evaluation of MIGml, we first collected all benchmark tasks from the four approaches that are still available, in summary, 147. To enhance the validity of our results, we additionally added all tasks from the `Loops` category of the SV-COMP 2020 benchmarks [SVB20]. We excluded tasks that contain arrays and recursion, as they were not supported by Korn (that we use to generate the CHC) or for which we were not able to generate the strongest postcondition, that is needed to validate the generated invariants. In total, we obtained a benchmark set of 263 tasks, where 222 are correct and 41 are incorrect programs.

Evaluation Result Table 3.4 provides in the upper part an overview of the quality of the generated invariants for the four approaches SNA12, GNMR16, KPW15, and ZMJ18. It contains the number of tasks, where each approach was able to generate

Table 3.4: Quality of generated invariants for different MIGml configurations

	invariant generated			others	
	helpful	**valid**	**invalid**	**unknown**	**timeout**
SNA12	17	5	109	30	102
GNMR16	55	2	0	155	51
KPW15	24	13	0	209	17
ZMJ18	38	4	4	66	146
KPW15-Val	46	3	7	60	141
ZMJ18-Oct	44	4	16	53	136

invariants that are helpful, valid, or invalid, report no answer, or run into a timeout. The best approach for generating helpful loop invariants on this benchmark is GNMR16, generating 52 helpful invariants for verification and 3 for falsification. ZMJ18 can generate 38 helpful invariants. Surprisingly, the approach is not able to compute an answer within the given time limit of 5 minutes in more than 50% of all tasks. Both, GNMR16 and ZMJ18 can generate invariants that are strong enough to show that the property is valid in the program. In addition, both generate valid or invalid invariants, for incorrect tasks, where the approaches do not detect the violation. For our re-implementation KPW15, we observe that a non-iterative approach can also be used to generate helpful loop invariants (24), although the number is lower compared to GNMR16 and ZMJ18. As it is non-iterative, it generates a lot more valid but not helpful invariants (13), and invalid invariants (167), that are reported as unknown by the approach. SNA12 generates 17 helpful and 5 valid invariants. In addition, a high number of timeouts is observed for SNA12 (102) and ZMJ18 (146). With a few exceptions, all reported results are computed in all ten runs conducted.

As MIGml ease building new instances especially the combination of existing components, we present in the lower part of Table 3.4 the two new instances KPW15-Val and ZMJ18-Oct[10]. We realize that KPW15 generates many invalid invariants. Thus, we additionally apply the model validator Val-CHC, (instance is called KPW15-Val). We additionally asked ourselves, if a SVM and Pg-Oct within ZMJ18 pays off. Hence, we also evaluated the configuration ZMJ18-Oct, which only makes use of Pg-Oct within the predicate generator. Comparing the absolute numbers, we see that KPW15-Val computes 46 helpful invariants, whereas the number of valid and invalid answers (which are subsumed in unknown for KPW15) reduced drastically. Having a look at the last row in Table 3.4, we observe that ZMJ18-Oct is superior among ZMJ18. In contrast, ZMJ18 can solve four tasks that ZMJ18-Oct is not able to solve. For example, it generates the invariant $xa{+}2*ya{>}1$, which is not expressible using the octagon template. We see that MIGml allows for easily building new configurations that can increase the effectiveness.

[10] A more in-depth analysis of other instances is given in [HW22b].

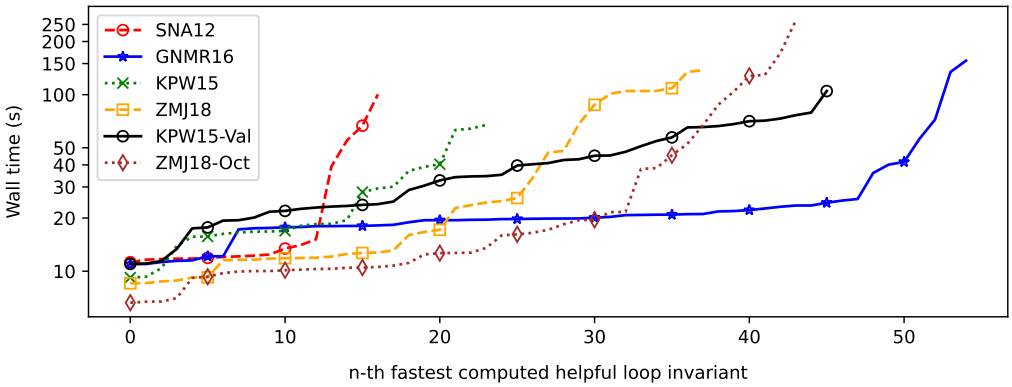


Figure 3.18: Quantile plot for different instances of MIGml

After having analyzed the effectiveness, we also have a look at efficiency. To be potentially usable within CoVEGI, the invariants need to be generated within 300 seconds (the default value for `timeoutH` is 300), at best significantly faster. As the CoVEGI algorithm is using the overall time for tracking the timeouts, we present in Figure 3.18 a quantile plot comparing the overall time taken to compute helpful invariants. A data point (x, y) in the plot means that the instance computes the x-fastest helpful loop invariant in at most y seconds. Therein, we can observe that all instances except KPW15 and ZMJ18 compute at least 80% of the generated loop invariants within 50 seconds wall time. For the two best-performing instances we see that ZMJ18-Oct computes 36 (82%) and GNMR16 51 (93%) of the helpful loop invariants within 50 seconds.

Results

MIGml can generate helpful loop invariants. The configuration GNMR16 generates the largest number of helpful invariants on our dataset and more than 90% of these invariants are computed within less than 50 seconds.

3.5 Discussion

Our experimental evaluation demonstrates that using externally generated invariants eases the verification task, resulting in a higher number of correctly solved tasks on our data set. Moreover, we observe that MIGml can be used to generate helpful loop invariants. We first discuss the validity of our experimental evaluation before discussing the approach itself:

For the evaluation on CoVEGI in RQ 1-3, we used all tasks from the category `Loops` and a random sample from other categories. Although this guarantees some diversity, our findings may not completely carry over to arbitrary real-world programs. The experiments are conducted using the reliable framework BenchExec, guaranteeing comparable and reproducible results. However, as SeaHorn is used within a docker-container, its CPU usage could not be measured by BenchExec. To obtain a lower

bound for the correctly solved tasks, we measured the CPU usage externally and added it to the CPU time measured by BenchExec. Thereby, all the results stay valid, especially of the best-performing instantiations of CoVEGI, as they do not use SeaHorn.

Our implementation of CoVEGI relies on the correctness of the used main verifiers and helpers (which are given) as well as on the encoders (which we build). However, an incorrectly translated invariant may only have a negative impact on the performance. The implementation of the CoVEGI algorithm from Algorithm 1 models both `timerM` and `timeoutH` using real time values instead of CPU time. A main verifier or a helper using more CPU time than defined does not impair the validity of the results, as the overall consumed CPU time of a CoVEGI instance is fixed and monitored by BenchExec. Both main verifiers used as well as UltimateAutomizer and VeriAbs participate in the annual SV-COMP, so they might be tuned to the tasks employed. Nevertheless, this does not influence the validity of the results since our focus is on the number of tasks that can be solved *additionally* when employing cooperation, and not the overall solved tasks.

For RQ 4, we use a set of benchmarks that has a large overlap with the one used for the previous evaluation (the category `Loops`). In addition, we included all available benchmarks from four different papers on machine learning-based invariant generation. Nevertheless, the findings may not completely carry over to arbitrary real-world C programs. We discussed the re-implementation and intermediate results of the four approaches used within RQ 4 with the original authors, to ensure that we did not miss any important aspect. Nevertheless, we cannot guarantee that we did not miss minor implementation details (as we did not analyze the full code of the original approaches) or that the prototype of MIGml is free of errors. Such wrong configurations of MIGml instances or errors within MIGml affect the results only negatively.

Our evaluation indicates that it could be promising to employ MIGml as helper invariant generator within CoVEGI, as the best-performing instance of MIGml generates most of the loop invariants within 50 seconds. In addition, the valid but not helpful loop invariants may also support the main verifier, as a combination with other discovered predicates may suffice to prove the program correct. Nevertheless, evaluating a combination of both approaches is subject to future work.

The obtained results indicate that CoVEGI increases the effectiveness of the two evaluated main verifier for finding proofs. The concept of CoVEGI works only for tasks that contain at least one loop. By using invariant witnesses for exchanging the computed loop invariants, we also gain some additional flexibility: The CoVEGI algorithm does not require that the invariant witnesses contain loop invariants (only). To use it for loop-free programs, we could generalize the task for the helper invariant generator's to generate arbitrary state invariants or predicates that might be helpful for the verification.

Within the experiments, the use of CoVEGI enables the main verifiers to compute additional proofs but not to raise additional alarms. The evaluation of MIGml, as reported in Section 3.4.3, shows that some instances of MIGml generate loop invariants that are helpful for falsification, meaning that the invariant is valid and helps to verify that the safety property is violated. Beneath that, a valid loop invariant could potentially reduce the number of CEGAR iterations needed to compute a valid counterexample and thereby speed up the falsification of a program. An experimental evaluation that analyzes how CoVEGI in combination with MIGml can be used for increasing the number of correctly raised alarms is another subject for future work.

3.6 Related Work

The approach for sequential cooperative software verification presented in this chapter is based on using externally generated invariants. Next, we provide a brief overview of related work sequentially combining tools or techniques for software verification. Thereafter, we focus on other existing approaches for generating invariants, of which most are also based on machine learning.

3.6.1 Sequential Combinations

In the following, we focus on concepts that combine the strengths of different approaches and exchange information between them in a sequential manner. We first have a look at cooperative approaches and then at approaches using conceptual integration. Many tools make use of a sequential portfolio of different analyses or analysis configurations [DLW15; BKR22; LD22; Ern23; RW19; RHJW20; CSV20; Cha+22; HHP13], where the instances are executed one after another until a result is computed. As these combinations do not foresee an information exchange, we do not discuss them in detail.

Sequential Cooperative Approaches As stated in Characterization 1.1, cooperative approaches use components as black-boxes and exchange information using clearly defined verification artifacts. One of the earliest ideas for cooperative software validation in a cooperative manner is CMC [BHKW12]. Therein, so-called conditional model checkers are executed sequentially. Each generates a predicate specifying under which condition the program adheres to the specification. This sequential process continues, until the full program is proven to be correct, i.e., the condition generated is *true*. The conditions are exchanged via a condition automaton (cf. Definition 2.11). The idea of CMC is extended and refined in multiple lines of work: Czech et al. propose in 2015 an extension of CMC wherein the second model checker is replaced using a testing tool [CJW15]. The information from the condition automaton is transformed into a reduced program, allowing the use of arbitrary testing approaches. In 2018, Beyer et al. formalized the idea of reducing the program based on a condition automaton

in [BJLW18]. They define so-called reducers, that remove paths already marked as safe within the condition automaton from the CFA and generate a residual program using the reduce CFA. Different reduction and folding strategies are proposed in [BJ20]. A similar concept for sequential cooperative verification is presented by Christakis et al., where a verification tool can add conditions under which the program is safe directly into the code. These conditions can also be used to model implicit assumptions made by the verifier explicitly [CMW12]. The task of the next verifier is to validate that the encoded condition is valid. In 2016, the technique of encoding explicit assumptions directly in the code is used to guide a dynamic symbolic execution tool towards unverified parts of the program, i.e., to abort the exploration of already analyzed program parts [CMW16]. Thereby, test cases are generated only for unverified or conditionally verifier parts of the program. In both lines of work, all employed tools are working on the same task, that is to verify the remaining parts of the program. In contrast, we have two conceptually different components working on two different tasks, namely program verification and invariant generation.

Pauck and Wehrheim proposed CoDiDroid, a framework for cooperative taint flow analysis for Android apps [PW19; Pau23]. Within their framework, different analysis tools with specialized capabilities (e.g. native code analysis or inter-procedural analysis) are combined as black-boxes to conduct a taint flow analysis for Android apps. CoDiDroid is however tailored to the needs of Android taint flow analysis, thus the exchanged information differs. Thus CoDiDroid is not able to orchestrate or exchange information on safety analysis with shared invariant generation.

Witness validation [Bey+15; BDDH16; Bey+22; BS22], aiming to ensure that the results computed by a verifier are correct, is also a sequential form of combining tools. It either makes use of witness validators [BS20; HM22; BDLT18; AS23; WSC22] or is realized as an internal combination of a verifier and a testing approach, e.g., in [CS05; CSX08]. In contrast to these ideas, CoVEGI aims for solving the verification task.

Sequential Conceptual Integrations Instead of combining different approaches cooperatively, there are also several approaches where different approaches are connected as conceptual integration, meaning that information is either exchanged by direct method calls to certain tools or using internal formats: In [GTXT11], a sequential combination of a verifier that uses a predicate abstraction technique followed by a dynamic symbolic execution engine that uses the previously computed information for detecting property violations is presented. Similarly, a verifier is combined with a tool for robustness testing [Hus+17], which is used to test assumptions made or conditions stated during verification. In FuSeBMC different components for test case generation and for interval analysis are combined [AABC21; Ald+23]. In the first phase, a fuzzing and a bounded model checking tool are run in parallel trying to cover all test goals. Afterwards, the covered goals and the inputs for covering them are given to a selective fuzzer, that uses them to cover the remaining test goals in the second phase.

These approaches combine tools working on the same task. In contrast, we employ two conceptually different components working on two different tasks in CoVEGI.

As finding helpful loop invariants is one key challenge in software verification, tools combine a verification technique with some invariant generation tool as conceptual integration. For example, DepthK combines ESBMC with the invariant generators PAGAI and PIPS, where the invariants are written as ESBMC-annotations directly into the code [Roc+17]. These combinations are conducted in a *white-box* manner using strong coupling between the components, making the addition of a new approach a challenging task. CoVEGI conceptually decouples the invariant generation from the verification, making it more flexible. In addition, using a black-box integration using standardized verification artifacts, as in CoVEGI, allows for an easy exchange or integration of new approaches.

Frama-C is a framework for code analysis, aiming for analyzing industrial size code [Kir+15; Bau+21]. The framework contains different plugins, each implementing a verification or testing technique. The plugins can exchange information in the form of ASCL source code annotations. Within Frama-C, the analyzers can collaborate for example by being composed sequentially. For this, partial results produced by an analysis can be completed by a second one, or several partial results computed in parallel are combined to a complete result. For instance, a value analysis result is used as input for a plugin applying verification using a weakest precondition calculus. Frama-C offers the general possibility to define cooperation between existing plugins. To the best of our knowledge, Frama-C does however not provide a conceptual collaboration of a verification approach and tools for invariant generation driven by the verification approach's demand for support.

The approach of using continuously refined invariants for k-induction [BDW15; BD20] uses a lightweight dataflow analysis [BDW15] or the Property-directed reachability (PDR) algorithm [BD20] for generating auxiliary invariants for k-induction, which can be considered to be a helper for verification. Therein, the supporting invariant generator runs in parallel to the k-induction analysis. Note that an analogous approach is proposed by Brain et al. [BJKS15]. Compared to our framework, the main difference is the form of cooperation used. Both approaches use a white-box integration for the cooperation between k-induction and the invariant generator, building hardly wired connections between both analyses and sharing the information *inside* the tool. Thus, integrating external tools is hard to achieve. Moreover, the approach is designed to work for k-induction only.

To summarize, there are a lot of existing approaches for cooperative verification, but most of them are white-box combinations, and the existing black-box combinations do not allow for cooperation on externally generated invariants.

3.6.2 Invariant Generation

To the best of our knowledge, there is no existing approach for machine learning-based invariant generation that provides the modularity of MIGml. Ideas and tools for generating loop invariants (for verification) are however numerous. Most approaches are iterative and can be classified by the way new predicates and new candidate invariants are generated, either using logic, templates, machine learning, or a combination. Especially for the ML-based invariant generation, several approaches using Large Language Models (LLMs) for a non data-driven invariant generation were recently proposed. Often, approaches that make use of a template are referred to as syntax-guided synthesis (SyGuS)-approaches [Alu+13], as the template is a grammar.

Logic-based approaches Within the logic based approaches, many verification tools [AMK18; BK11; KRSS16; RW19; HHP13; BDFW08; GDP17; BHJM07; Hei+18] employ a CEGAR scheme, where predicates are generated based on Craig interpolation [Cra57a; McM03; HJMM04] or Newton refinement [BR02a]. The IC3 approach proposed by Bradly [Bra11] generates a sequence of candidates, inductive relative to the predecessor until a 1-inductive loop invariant is found. The approach presented by Dillig et al. [DDLM13] generates inductive invariants using logical abduction. Chalupa and Strejček propose a method for generating loop invariants using backwards symbolic execution with loop folding to generate inductive loop invariants [CS21].

Template-based approaches In contrast to logic-based approaches, most template-based approaches are data-driven: Daikon [ECGN99; Ern+07] executes the program and uses the variable values observed to heuristically select a subset of predicates from a set of templates. These predicates can contain, among others, constant values $x = c$, in range expressions $a \leq x \leq b$, or linear expression $y = a * x + b$. InvGen [GR09] uses dynamic and static analysis techniques to generate a set of constraints, that are used for instantiating an invariant template over linear inequalities. Sharma et al. [Sha+13] present a technique called Guess-and-Check, that generates a set of data and guess an invariant using linear algebra techniques from the domain of polynomial with bounded degrees. NumInv [NARH17] generates template-based polynomial equalities and octagon-based inequalities, where the variable values used for instantiation are computed using program traces. The generated invariants are analyzed using Klee, which may generate additional counterexamples for invalid invariants. c2i [SA14] uses an iterative approach, wherein a randomized search generates invariant candidates based on templates, that are verified in a second step. Invariants generated using these approaches are limited by the predicates abstractly described in the templates, often from the polyhedral or octagon domain.

Machine learning-based approaches Using machine learning approaches for generating invariants is currently a very active field of research. Beneath the data-driven

approaches that generate data and mostly follow the idea of ICE learning, approaches that employ LLMs for generating invariants have also been used recently. We first give an overview of data-driven approaches:

An extension of the ICE-framework of [GNMR16] proposed by Ezudheen et al. employs a validator using CHCs, additionally giving the learner access to a separate CHC-solver [Ezu+18]. Thereby a more robust algorithm with guaranteed termination is obtained. The authors of SNA12 published a different concept for learning invariants, called Pie [PSM16]. The predicate generator works on an expressive and extendable template, only generating a subset of all predicates, sufficing to separate the good and bad points and the classifier employs probably approximately correct (PAC) learning. In contrast to Pg-Oct, a subset of all available predicates that suffice to separate the positive and negative data points is generated. The tool LoopInvGen is an efficient implementation of this idea [PM17]. The tool Zilu presented by Li et al. [Li+17] employs an active learning approach based on an SVM, allowing the learner to request a classification for certain data points and generates invariants path-sensitive with respect to the path inside the loop. Ryan et al. present a new architecture for neural networks to learn SMT formulas, called Continuous Logic Networks (CLN) [Rya+20]. The network aims at learning an instantiation of a given template, using data points that are generated based on program execution traces. A generalization of CLN, so-called gated-Continuous Logic Networks are proposed in [Yao+20], that does not require templates as inputs.

In general, the approaches that follow the idea of ICE-learning can be integrated directly within MIGml. Extending MIGml to also be able to handle requests of active learning approaches as well as allowing the Learner to generate path-sensitive invariants is another point planned for future work.

Instead of generating data containing variable values for the machine learning approach as input, some approaches use (a representation of) the program to learn an invariant directly as input: Si et al. [Si+18; Si+20] developed the tool Code2Inv, that represents the program using a Graph Neural Network (GNN) and apply reinforcement learning with an attention mechanism in an iterative manner. The approach models the current invariant as a decision tree and new predicates are inferred using an attention mechanism, working on an abstract program representation. The learner learns a strategy where (which leaf) and how (disjunction or conjunction) to attach the newly generated predicates to the decision tree.

LIPuS [YWW23] is using two different program representations as well as concrete variable values as input for a reinforcement-learner. Instead of learning a loop invariant directly, the learner's task is to reduce a general template over the non-linear integer arithmetic as much as possible. The reduced template is instantiated using a SMT solver, that also checks if the instance is a valid loop invariant. Counterexamples generated by the SMT solver are used as additional input for the reinforcement learner.

A similar approach is proposed by Wang and Wang, wherein the search space of the reinforcement learner is reduced based on previous counterexamples generated by the SMT solver [WW22].
As both Code2Inv and LIPuS are using among data points other representations of the program, they do not fit directly into the idea of MIGml, as they do not provide the clear separation into Teacher and Learner as the other ML-based approaches.

Recently, LLMs are also employed for generating invariants: Pei et al. trained a LLM to predict a list of loop invariants for Java tasks based on invariants generated using Daikon [Pei+23]. The advantage of using a LLM is that the invariants are generated based on the full program instead of using a set of data points. Instead of using a model that is specifically trained for generating invariants, Janßen et al. have shown that one of the most prominent LLMs, namely ChatGPT, can directly be used in some scenarios for invariant generation [JRW24]. In [Kam+23], multiple models are evaluated and a list of invariants is generated by the LLMs, where each invariant is analyzed for being valid. Chakraborty et al. propose to use a ranking mechanism called iRank, that ranks the list of invariants with respect to the probability of being correct. Thereby, the number of invariants that need to be validated using Z3 can be reduced significantly [Cha+23]. The approaches that employ LLMs for learning invariants are not data-driven, thus they do not fit in the MIGml-framework.

4 Cyclic Cooperation

After having discussed how verifiers can cooperate sequentially, we focus on cyclic forms of cooperation. In a cyclic combination of tools, each tool is executed in an alternating way, trying to solve a (specific) part of the overall task. Most cyclic combinations are built as strongly cohesive software units. These tools are typically made up of tightly coupled, stateful components that operate on shared data structures. Instead of designing a novel concept for cooperation, as we did in Chapter 3, we aim to decompose concepts widely used in software verification processes. Thereby, we allow for building cooperative software verifiers using the decomposed concepts based on off-the-shelf components. Simply speaking, we enable concepts for software verification to be used in a cooperation setting. Such a decomposition facilitates the reuse of components, impacts scalability (e.g., parallelization), and eases the exchange and integration of new components. As the ultimate goal, we want to reduce the integration of innovations and novel verification techniques to a configuration task. To do so, we avoid a strong cohesion between existing components. To investigate the feasibility of such ideas, we exemplarily realize such a strict decomposition on the CEGAR scheme, presented in Section 2.2.1, that is not only used for software verification by many tools (e.g., [RU17; AMK18; BK11; KRSS16; RW19; HHP13; BDFW08; GDP17; Afz+19; YDLW18; Cas+17; RE14; BHJM07; Wan+16; Hol+17]), but also successfully employed in other areas, like probabilistic or timed-automata model checking [HWZ08; HSW13].

CEGAR is well suited for decomposition, as the process comprises the three steps *Abstract Model Exploration*, *Feasibility Check* and *Precision Refinement*, as depicted in Figure 2.7, but most tools that make use of the scheme are built as strongly cohesive software units. There exist different concepts for realizing the three components, each having individual strengths and weaknesses. Whenever novel research ideas arise, they are integrated into existing strongly cohesive software, which is costly regarding the required implementation effort. Hence, using components built by others or comparing

them is challenging, as well as evaluating new concepts in isolation. This situation is best illustrated by proposals of and discussions on precision refiners [Die+17; HM20; BLW15b; BLW15a]. Precision refinement techniques rely on heuristics, and hence their effectiveness can only be evaluated through experiments. Due to the current situation, multiple re-implementations of precision refiners exist: A vast amount of tools [RU17; AMK18; BK11; KRSS16; RW19; HHP13; BDFW08; GDP17; Cas+17; BHJM07] contain implementations of a refiner based on Craig interpolation [Cra57a; McM03; HJMM04] and at least three tools [Hei+18; BK11; HM20] contain (re-)implementations of so-called Newton refinement [BR02a].

We aim for a decomposition of CEGAR, which we call component-based CEGAR ((C-CEGAR) - see Section 4.1). Within C-CEGAR, we make use of standardized artifacts for information exchange between the individual components, namely correctness, violation, and invariant witnesses. The question arises if these artifacts are best suited for this purpose. We show how a single artifact having a unified semantics (for the use in cooperation between over- and under-approximative tools) eases cooperative verification and propose such a format, called Generalized Information Exchange Automaton (GIA) in Section 4.2. Next, we describe the implementation of C-CEGAR and GIAs in Section 4.3 and evaluate the performance of both concepts in Section 4.4. Therein, we again focus on both, effectiveness and efficiency. We conclude the chapter by discussing the results in Section 4.5 and present related work on cyclic cooperative approaches in Section 4.6.[1]

4.1 Component-based CEGAR

To make CEGAR ready for being used within cooperative software verification, we need to define actors that solve the verification task together and exchange information by using clearly identifiable artifacts. Moreover, the tools used within the decomposed version of CEGAR should be usable off-the-shelf. To achieve the former goals we present in Figure 4.1 the decomposed workflow of CEGAR, called component-based CEGAR (C-CEGAR). It comprises three independent components (or actors) communicating using clearly defined interfaces based on verification artifacts. For the interfaces, we employ the existing standardized artifacts *violation witnesses* (to encode the potential counterexamples), and *invariant witnesses* (to encode the precision increment). We additionally define *path witnesses* to encode infeasible counterexamples, that are based on violation witnesses but have different semantics. These witness formats are already produced and processable by many verifiers, which allows us to partially reuse tools. Before explaining the artifacts, especially the information encoded, as well as the actors in more detail, we motivate the advantages of C-CEGAR next.

[1]Note that C-CEGAR, more precisely the concept (Section 4.1), the implementation (Section 4.3), and evaluation of C-CEGAR (Section 4.4) was developed and conducted in close cooperation with Thomas Lemberger from LMU Munich, whereas both authors contributed equally.

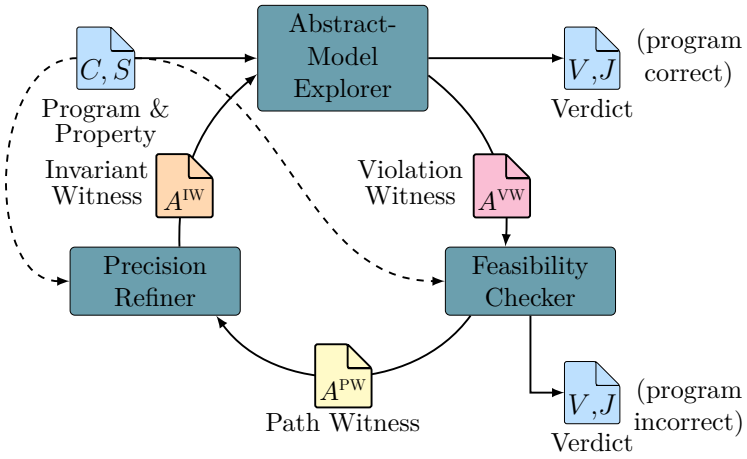


Figure 4.1: Workflow of component-based CEGAR

4.1.1 Motivating Example

To exemplify why an easy exchange of components within C-CEGAR can be beneficial, we present in Figures 4.2a and 4.2b the two programs `prog1` and `prog2` taken from the SV-Benchmarks [SVB22], where we use the safety properties $S_1 = (\ell_4, \mathit{false})$ for `prog1` and $S_2 = (\ell_6, \mathit{false})$ for `prog2`. Both programs satisfy the specifications. Let us assume that we analyze these two programs using a verifier that employs a predicate abstraction using CEGAR for computing new predicates as precision increments.

```
  int prog1() {
0   unsigned int y = 1;
1   while (1) {
2     y = y + 2U * nondet();
3     if (y == 0)
4       abort();
5 }
```

(a) Craig interpolation finds the more meaningful predicate ($y \mathit{mod}\ 2 = 1$), Newton finds the equivalent, but more complex predicate $1 \leq y + 2 * \lfloor((y * -1 + 1)/2)\rfloor$.

```
  int prog2() {
0   unsigned int x = 0;
1   unsigned short N = nondet();
2   while (x < N) {
3     x += 2;
4   }
5   if (x % 2 != 0)
6     abort();
7 }
```

(b) Newton refinement finds the more meaningful predicate $x \leq 2 * (x/2)$, Craig interpolation enumerates all valid assignments for x explicitly.

Figure 4.2: Code examples for Craig interpolation and Newton refinement

When using the implementation of Craig interpolation from CPAchecker for `prog1`, the predicate $p_1 \equiv y \mathit{mod}\ 2 = 1$ is computed, which is a helpful 1-inductive loop invariant. Newton refinement (implemented in UltimateAutomizer [Die+17]) computes the predicate $p_2 \equiv 1 \leq y + 2 * \lfloor((y * -1 + 1)/2)\rfloor$, that is equivalent to p_1 (see [BHLW22a]), but is more complex and increases the verification overhead, such that the tool fails to compute a solution. For `prog2`, Craig computes in each iteration a new predicate enumerating possible values for x, namely $x = 0, 2, 4$, and so on. In contrast, Newton computes immediately (during the first iteration) the predicate

$x \leq 2*(x/2)$, which encodes the helpful loop invariant $x \; mod \; 2 = 0$. These two examples show that there is no single technique that is optimal for all programs. Hence, having the ability to compare different approaches and select the one potentially best suited is achievable using C-CEGAR.

4.1.2 Exchange Formats in C-CEGAR

The information exchange within C-CEGAR is realized using the tool-independent interfaces violation witnesses, path witnesses, and invariant witnesses to pass (infeasible) counterexamples and precision increments among the components. We explain their usage in the context of C-CEGAR next:

Potential Counterexample - Violation Witnesses The abstract model explorer may aim to encode a set of program paths abstractly or a single path precisely. In the former case, the abstract representation may ease encoding long paths as well as give the feasibility checker and precision refiner a greater degree of freedom in computing information, which may lead to a more generic precision increment. In the latter case, the abstract model explorer can add information on variable values, hence program states, at certain points of the path. Such a description avoids imprecision but may be very large. We decided to use the existing format of violation witnesses (cf. Definition 2.10), as they allow for encoding abstract sets of paths as well as a single path. In addition, the semantics of violation witnesses foresees encoding at least one program path that contains a property violation.

Infeasible Counterexample - Path Witnesses In contrast to the prior artifact, we cannot use violation witnesses to encode an infeasible counterexample. The feasibility checker has rejected the violation witnesses given, hence the violation witness does not contain a path violating the specification. We thus define a novel artifact called path witness, abbreviated A^{PW} and depicted in light yellow.

Definition 4.1. *A* path witness $A^{\text{PW}} = (Q, \Sigma, \delta, q_0, F)$ *is a protocol automaton, where each state has only a trivial state invariant, i.e.* $\varphi = true$ *holds for all* $(q, \varphi) \in Q$.

Semantically, program paths covered by an A^{PW} contain no property violation. Hence, a violation witness rejected by the feasibility checker is a path witness.

Precision Increment - Invariant Witnesses To guarantee progress in any of the CEGAR iterations, the precision increment contains an explanation of the infeasibility of the counterexample generated in that iteration. For model explorers using predicates as abstraction, the precision increment contains at least one new predicate, preventing the exploration of the same counterexample again by proving its infeasibility. We use invariant witnesses as defined in Definition 2.9, which contain predicates justifying that a set of paths but not necessarily the full program does not violate the specification.

4.1.3 Components of C-CEGAR

After having explained which artifacts are exchanged among the components, we describe the components, especially the task that needs to be solved in more detail. As depicted in Figure 4.1, component-based CEGAR (C-CEGAR) comprises the three components *abstract model explorer*, *feasibility checker*, and *precision refiner*. As each of the components is defined stateless, all information needed for solving the tasks is given as input. In addition, C-CEGAR makes use of a controlling unit, that checks if a final answer is computed and is responsible for handling the information exchange among the components. The controlling unit additionally ensures that the precision increment given to the abstract model explorer contains the increments computed in all prior iterations, as each component in C-CEGAR is defined stateless.

Abstract Model Explorer The task of an *abstract model explorer* is to compute the abstraction of the program using a given precision. It receives as input the task comprising of the program as CFA C and the specification S as well as the precision in the form of an invariant witness A^{IW}. After extracting the relevant information from the invariant witness, it checks whether the program satisfies the given specification and is thus correct. If the program is correct, the verdict V is *true* and a justification J in the form of a correctness witness is returned by the abstract model explorer. In this case, the CEGAR cycle is terminated, as the verification problem is solved and the verdict and witness are returned as the final answer. Otherwise, the verdict V is *false* and the justification J contains a violation witness A^{VW}.

Feasibility Checker The task of a *feasibility checker* is to analyze the potential counterexamples for feasibility. It also receives the task as input and the potential counterexample encoded within a violation witness A^{VW}. As there may be multiple paths covered by A^{VW}, each has to be checked for feasibility. In the case at least one is feasible, the verdict V is *false* and A^{VW} is returned. The CEGAR cycle is terminated and the feasible counterexample and V are returned as the final answer. Otherwise, all counterexamples encoded are spurious. The feasibility checker returns a path witness A^{PW} and the verdict *true*. Note that violation and path witness are usually syntactically equivalent and cover the same paths but have different semantics.

Precision Refiner The task of a *precision refiner* is to generate a precision increment such that the infeasible counterexamples encoded within the path witness are not explored anymore. It also receives the task as input and the path witness A^{PW}. To ensure that the infeasible paths encoded within A^{PW} are not explored in the next iteration by the abstract model explorer, the precision increment has to contain a reason explaining the infeasibility. This can for example be computed using interpolation or by generating an invariant for (some of) the paths. It returns an invariant witness that contains a precision increment for at least one path encoded in A^{PW}.

4.1.4 Usage of Off-the-Shelf Components

To realize C-CEGAR as a cyclic cooperative verification approach we implement each of the three components. The task of the feasibility checker is almost the same as in violation witness validation [Bey+15; Bey+22]. Thus, there exist already multiple tools like FShell-witness2test [BDLT18], Symbiotic-Witch [AS23], Witch [AS24] or UltimateAutomizer that can be applied off-the-shelf. In the following, we propose a method for building instances of the components using an off-the-shelf verifier (cf. Figure 2.1) for abstract model explorer and feasibility checker. In addition, we explain how a precision refiner can be constructed using an invariant generator.

Abstract Model Explorer As visualized in Figure 4.3, we can turn any off-the-shelf verifier into an abstract model explorer. Therefore, we use MetaVal [BS20] to encode the invariants or predicates in the invariant witness as additional code (assertions) in the program using a so-called *strengthener*. The obtained CFA is thus enriched with invariants and analyzed in combination with the safety property. [2] Some verifiers as CPAchecker [BK11] and UltimateAutomizer [Hei+18] natively support using invariant witnesses as additional inputs when used as an abstract model explorer.

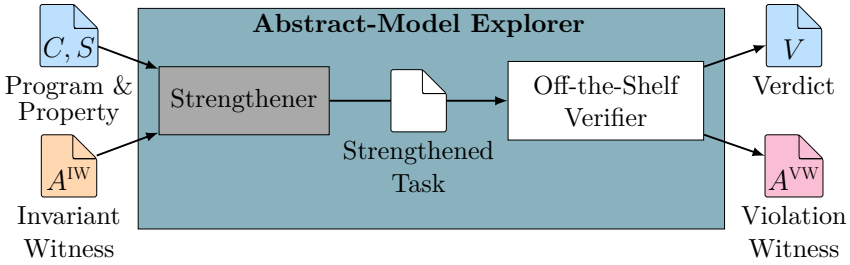


Figure 4.3: Construction of an abstract model explorer using off-the-shelf tools

Feasibility Checker As previously stated, each violation witness validator can be used as a feasibility checker. In case an off-the-shelf verifier should be employed, we can use a *reducer* [BJLW18], as depicted in Figure 4.4, to restrict the program to only those paths that are also encoded in the violation witness A^{VW}, called *path program*. We can employ MetaVal [BS20] or the reducer presented in [BJLW18] for generating a path program.

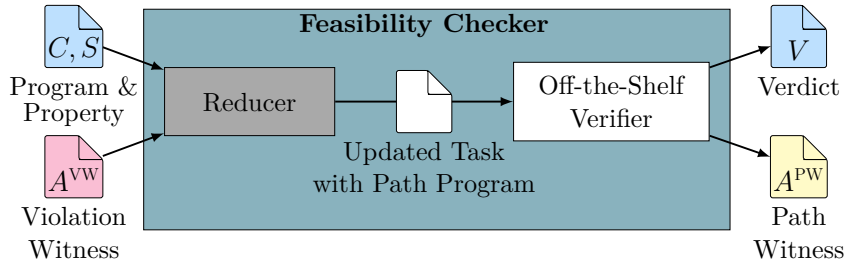


Figure 4.4: Construction of a feasibility checker using off-the-shelf tools

[2] Note that the violation witness is computed for the strengthened task and needs to be modified by removing the additionally added assertions.

Precision Refiner To build a precision refiner using an off-the-shelf invariant generator (e.g., MIGml), we again need to reduce the program to a path program using the reducer before employing the invariant generator on the updated task, as depicted in Figure 4.5. In case the invariant generator is not able to compute invariant witnesses, we can employ the encoder presented in Section 3.1.4.1. In case the invariant generator does not compute any invariant, C-CEGAR is aborted.

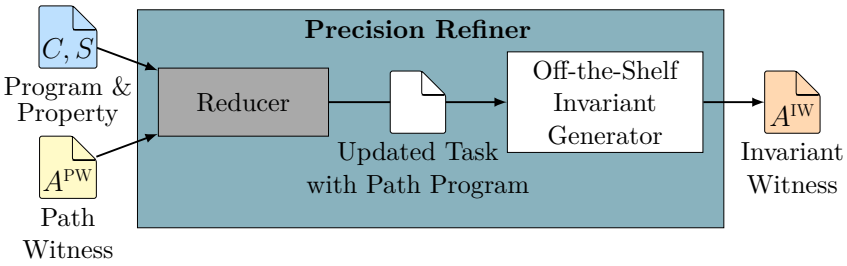


Figure 4.5: Construction of a precision refiner using off-the-shelf tools

Having a component-based instance of CEGAR ready to use, we can build new combinations using different tools by simply plugging different components into C-CEGAR, without the need for re-implementation. Using the presented constructions we can even use off-the-shelf tools that are not able to process the used exchange formats as additional inputs. We exemplify this advantage next in more detail.

4.1.5 Example Application of C-CEGAR

To exemplify the advantages of C-CEGAR in more detail, especially to emphasize the use of off-the-shelf tools, we revisit the program given in Figure 4.2b and present its CFA C_2 in Figure 4.6. Our example instance of C-CEGAR comprises an abstract model explorer that uses predicate abstraction, an off-the-shelf verifier as a feasibility checker, and a precision refiner that computes Craig interpolants. We assume that none of the components can process witnesses as additional inputs to exemplify the constructions described in Section 4.1.4.

Initially, the abstract model explorer starts with empty precision and computes a counterexample A_2^{VW} that does not contain a loop iteration, as depicted in Figure 4.7. The violation witness covers a single path in C_2. Paths that enter the loop or do not enter the if-condition in line 5, are not covered. As we assume that our feasibility checker is not able to process violation witnesses as additional inputs, we employ the construction depicted in Figure 4.4 and generate the path program shown in Figure 4.8 and the updated safety property $S_2' = (\ell_7, \mathit{false})$. The path program contains exactly one path that reaches the call to `abort()` in line 7. In addition, the condition $\psi : N = 0$ present in A_2^{VW} is encoded as an assumption in line 2. The verifier analyzes the program and detects no property violation, hence returning the verdict *true* and a path witness A_1^{PW}, that is identical to A_2^{VW} depicted in Figure 4.7. Hence, the path program based on A_1^{PW} is the same as for A_2^{VW}. It is given to the precision refiner, which computes the

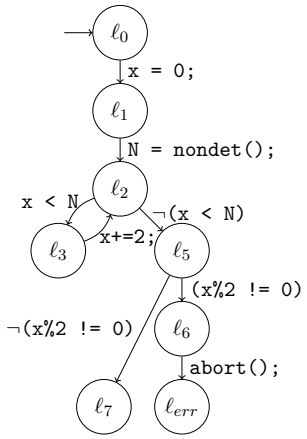


Figure 4.6: CFA C_2 for `prog2`

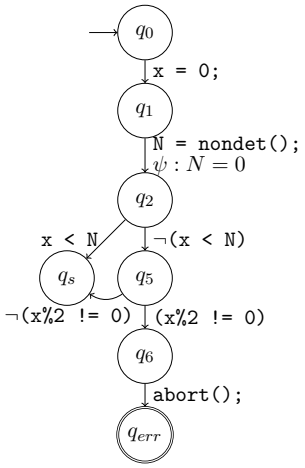


Figure 4.7: Invalid violation witness A_2^{VW} for C_2

```
   int prog2() {
0    unsigned int x = 0;
1    unsigned short N=nondet();
2    assume(N == 0);
3    if (x < N){
4      return 0;
5    }else{
6      if (x % 2 != 0){
7        abort();
8      }else{
9        return 0;
10     }
11   }
12 }
```

Figure 4.8: Path program for A_2^{VW} and C_2

invariant $x = 0$, which suffices to prove that the path program adheres to the safety property S_2'. The resulting invariant witness is depicted in Figure 4.9, containing the precision increment $\varphi_1 \equiv x = 0$. The only path that is covered by A_1^{IW} is the one also encoded in A_2^{VW}. This precision increment is used by the abstract model explorer within the next iteration during its exploration, ensuring that the counterexample from the last iteration (A_2^{VW}) is not computed again. The strengthened program containing the precision increment that is used in the second iteration by the abstract model explorer is given in Figure 4.10.

In this example, the precision increment does not suffice to prove the program correct, hence another counterexample is computed, now containing one loop iteration. The refinement cycle thus starts anew, the feasibility checker rejects the spurious violation witness and the precision refiner computes a new precision increment contained in the invariant witness depicted in Figure 4.11. It contains the precision increment $\varphi_2 \equiv x = 2$, which is associated with the state reachable after one loop iteration. Note that A_2^{IW} contains only the precision increment that is computed in the second iteration. Thus, the controlling unit needs to ensure that A_1^{IW}, the invariant witness computed in the first iteration, and A_2^{IW} are combined to ensure that no information is lost. As the combination of A_1^{IW} and A_2^{IW} is still not sufficient to prove the program correct, the next CEGAR iteration starts, until after 16.384 iterations the program is proven safe.

Now let us consider a slightly different configuration, where we replace the precision refiner and use a tool that computes a Newton refinement. As the abstract model explorer and feasibility checker are not changed, the first two steps do not differ from the first configuration, hence the path program from Figure 4.8 is given to the precision refiner. As we use Newton refinement in this configuration, the invariant witness generated in the first iteration differs from A_1^{IW} regarding the state invariant, as now the state invariant $\varphi_1 \equiv x \leq 2 * (x/2) \equiv x \; mod \; 2 = 0$ is included. φ_1 is a helpful loop invariant, thus it suffices to prove that `prog2` matches the safety property.

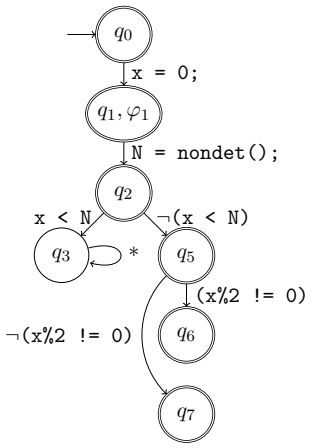


Figure 4.9: Invariant witness A_1^{IW} for C_2, $\varphi_1 \equiv x \leq 2 * (x/2) \equiv x \ mod \ 2 = 0$

```
  int prog2() {
0   unsigned int x = 0;
1   assert(x == 0);
2   unsigned short N=nondet();
3   while (x < N) {
4     x += 2;
5   }
6   if (x % 2 != 0)
7     abort();
8 }
```

Figure 4.10: Strengthened program for A_1^{IW} and C_2

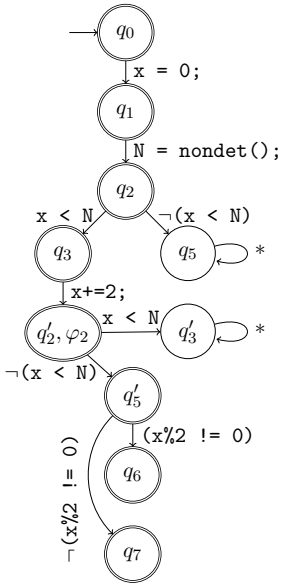


Figure 4.11: Invariant witness A_2^{IW} for C_2, $\varphi_2 \equiv x = 2$

4.2 Generalized Information Exchange Automaton

In C-CEGAR, we use violation witnesses, path witnesses, and invariant witnesses to exchange information among the components. All three automaton formats are based on protocol automata and can, in combination, encode information on feasible counterexamples, infeasible paths (i.e., program paths that are known to contain no property violation), and precision increments in the form of state invariants. Invariant and path witnesses are formats specifically designed for being used in the context of cooperative verification, more precisely in the context of C-CEGAR[3]. The question arises if each new form of cooperative software verification requires defining new artifacts from scratch or redefining existing artifacts by adding new semantics. This may hinder the reuse of actors, as either existing actors have to be updated to work with the novel formats or the novel semantics, or a transformation from one format into another is needed.

To avoid such situations in the future, we would need one artifact for information exchange that has uniform semantics and can encode all information (potentially) exchanged during cooperative software verification among conceptually different actors. In order to answer the question if such an artifact already exists or if we can define such a generalized one, we have a look at the information exchanged in existing concepts for combining software verification tools. Therefore, we distinguish two classes of components or actors: Over-approximating and under-approximating analyses. Over-approximating (OA) analyses build an over-approximation of the state space

[3]Invariant witnesses are also used in Section 3.1, but were originally introduced for C-CEGAR.

of a program while under-approximating (UA) analyses inspect specific program paths. An UA analysis typically aims at finding errors; an OA analysis aims at proving the program correct. For example, the abstract model explorer from C-CEGAR is an OA analysis, whereas the feasibility checker is an UA analysis.

Next, we summarize the type of information exchanged in existing (cooperative and non-cooperative) combinations of OA or UA analyses. As we not only want to look at software verification but also at testing, we propose a uniform way of describing the goal of testing and verification: For *verification*, the goal is to show that the specified safety properties are fulfilled, i.e., the non-reachability of certain error locations (cf. Definition 2.2). For *test case generation*, the goal is to find paths from ℓ_0 reaching all locations from a set L_{cover}, containing e.g., each branch or statement in the program (branch-, statement-coverage) or certain function calls, e.g., `abort()`. To specify these paths, a sequence of return values (called *test suite*) for the program inputs or the calls to `nondet()` suffices (if `nondet()` models inputs to programs).

For cooperation, we prefer a uniform way of describing these tasks which we get by introducing the notion of *target nodes*, denoted by TN, $TN \subseteq L$.

Definition 4.2. *For a CFA $C = (L, \ell_0, G)$ a* target node *$\ell_t \in TN$ is a node either present in a safety property S (for verification) or $\ell_t \in L_{cover}$ (for test case generation).*

We can now reformulate the two tasks: The goal of verification is to show that *no* target node is reachable, the goal of test case generation is to find a test suite such that *all* target nodes are reached. For our running example C_1 from Figure 2.3 the set of target nodes is $TN = \{\ell_{err}\}$ for verification using the safety property $S = (\ell_{err}, \mathit{false})$. In case of test case generation where the goal is branch coverage, we use $TN = \{\ell_1, \ell_3, \ell_5, \ell_6, \ell_7, \ell_9, \ell_{10}, \ell_{11}\}$.

Our analysis of existing artifacts shows that none of them is perfectly suited for this purpose. Hence, we define a new verification artifact in the form of a Generalized Information Exchange Automaton (GIA). A GIA can express information generated by over- and under-approximating analyses in the context of software verification and also testing, which we unified call *software validation*. More precisely, based on the idea of target nodes and inspired by three-valued logic, we can define the semantics of the verification artifact GIA in such a way that it can encode the different information exchanged in software validation while maintaining uniform semantics. Before we have a detailed look at the information exchanged and formally define GIAs, we exemplify the advantage of having a generalized artifact for information exchange with fixed semantics.

4.2.1 Motivating Example

To exemplify the advantages of GIAs, we use an instance of C-CEGAR that employs GIAs for the information exchange, resulting in the C-CEGAR scheme that is depicted

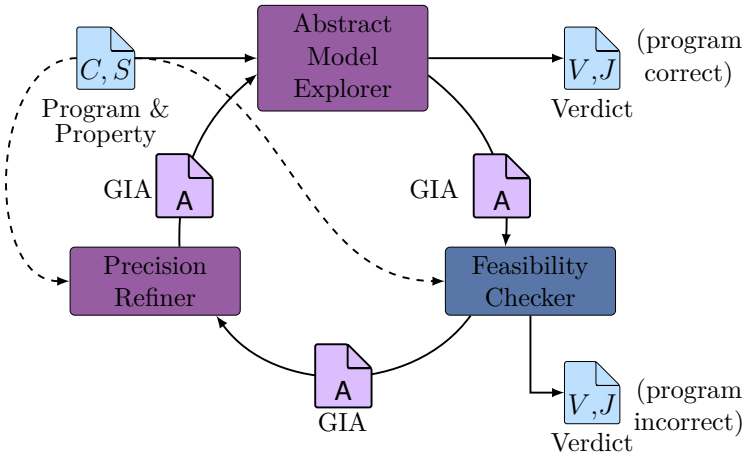


Figure 4.12: C-CEGAR using only GIAs for the information exchange. Components that are over-approximating are shown in purple, and under-approximating actors in dark blue. GIAs are colored in light violett.

in Figure 4.12. It comprises the same three actors as in Figure 4.1, but now we only use GIAs for the information exchange instead of violation witness, path witness, and invariant witnesses. Moreover, we classify the abstract model explorer and the precision refiner as over-approximating (colored in purple), as both can prove that certain paths of the program are infeasible and the feasibility checker as under-approximating, colored in dark blue.

Before emphasizing the advantages of GIAs, we first revisit the example from Section 4.1.5, where we use an instance of C-CEGAR to verify the program `prog2` from Figure 4.2b. We first have a look at the example without using GIAs. After some iterations, the abstract model explorer has proven that some paths of the program are safe, e.g., the path without a loop iteration encoded in A_2^{VW}. As the given precision (respectively the predicates) are not sufficient to prove the complete program safe, the abstract model explorer computes a different (spurious) counterexample. As all components within C-CEGAR are stateless and the violation witness only allows to encode paths that lead to a property violation, the information on paths already proven safe is lost and needs to be reestablished in the next iteration. Moreover, the precision increment computed in the current iteration has to be merged with the precision increments computed in the previous iteration to ensure that no information is lost.

Our goal is to define a format with uniformly applicable and fixed semantics that is able to encode safe paths and paths (potentially) leading to a property violation. Especially, we aim to be able to encode and combine all information computed by different components within a single instance of the artifact GIA. Now, we have a look at C-CEGAR using only GIAs for information exchange. The abstract model explorer can now store all computed information on safe paths within one GIA and reuse it later, e.g. in case the counterexample is spurious. Thus, no information computed in previous iterations by the abstract model explorer needs to be reestablished. The

precision increments computed in previous iterations are also retained in the GIA, making the necessity for merging them superfluous.

4.2.2 Information Exchanged in Software Verification

We aim to define a unified artifact that is applicable in many existing concepts of cooperative software validation combining OA and UA components. Therefore, we briefly discuss which information is exchanged via (non)-standardized artifacts in different existing cooperation-based approaches and derive requirements that need to be fulfilled by a unified artifact. A more in-depth analysis can be found in [HW24].

Most existing cooperation-based approaches that aim to solve a verification task, i.e., to show the validity of safety properties from Definition 2.2, exchange information on a set of paths or information that ease the verification problem. In the former case, a component may communicate the information that a set of paths is infeasible, for example in the Ufo algorithm [AGC12] or the cooperative formalization of CEGAR (cf. Section 4.1). In approaches that follow the idea of CMC [BHKW12; CJW15; BJLW18; BJ20], artifacts encode the information that a part of the program is safe, either because all paths that would lead to a property violation are infeasible or the part of the program does not contain such paths. In the latter case, the information exchanged is often in the form of predicates, i.e., invariants, interpolants, or conditions under which a verification result is computed. For example, invariants that ease the verification task are exchanged in CoVEGI (cf. Section 3.1) or some instances of k-induction [BDW15; BD20]. In the cooperative formalization of CEGAR (cf. Section 4.1) or property-directed k-induction [BHMS20] interpolants that justify the unreachability of some paths are communicated between components. The conditions that are assumed during the verification of the complete program or a set of paths are communicated among components for example in CMC or [CMW12; CMW16].

In cooperation-based approaches that aim to generate test cases covering a set of test goals, most components exchange information on concrete program executions and the set of test goals covered by the executed paths. Examples are, among others, FuSeBMC [AABC21; Ald+23], CoVeriTest [BJ19; Jak20; JR21], or the concept proposed by Huster et al. [Hus+17]. In addition, the components may exchange information on concrete inputs that lead to the test cases or conditions that specify the part of the program that is already complete, as in CoVeriTest. Lastly, [DGH16] uses a component that reports that some of the test goals are unreachable.

Based on the analysis of existing forms of cooperation, the artifact needs to be able to encode information on:

(R1) program paths which are already known to be feasible (and may reach certain test goals or an error state),

(R2) program paths which reach an error state and are either feasible or where the feasibility is not known for sure (potential counterexample),

(R3) program paths which are already known to be safe,

(R4) program paths which are already known to be infeasible,

(R5) additional constraints on program paths like state invariants,

(R6) and additionally, an artifact needs to have context-independent semantics.

Existing concepts for cooperative software verification already use different artifacts to exchange information, e.g., witnesses, condition automata or ARGs. The detailed analysis provided in Section 4.6.4 shows that none of the artifacts used is able to encode all desired information and is usable independent of the employed tools while maintaining one semantics. Thus, we next develop an artifact for the information exchange among OA and UA verification tools, fulfilling all requirements.

4.2.3 Formalization of GIA

Our overall objective is to define an artifact with one semantics that is able to encode most information exchanged. In general, UA and OA tools either aim at showing that target nodes are reachable (for example a call to `abort` or a branch that needs to be covered) or that (a part of) the program does not reach any target node (i.e., the program is safe). The overall goal in software validation is achieved when for each target node either a path reaching it is found or the node is proven unreachable.

As stated in Section 4.2.2, the information exchanged between UA and OA tools needs to be about (1) feasible paths definitely leading to a target node (R1), (2) paths definitely not leading to a target node (either as they do not reach one or are infeasible, (R3) and (R4)), and (3) *candidate* paths potentially leading to target nodes and hence interesting to consider for the analysis, but where the definite result about it is unknown so far (R2). The latter information is used in two cases: When an UA tool has not yet covered a path, either due to resource/time limitations or because it is infeasible, and when an OA tool has discovered a path to a target node, which might be feasible. In addition, the artifact needs to be able to pass helpful information about invariants of program locations or constraints about program transitions (R5). All information needs to be encoded while maintaining one fixed, context-independent semantics (R6). Inspired by the idea of three-valued logics (e.g., for three-valued model checking [BG99]), we extend the condition automata of [BJLW18] by introducing *three* different, disjoint sets of accepting states, one for each type of exchanged information.

Definition 4.3. *A* Generalized Information Exchange Automaton (GIA) $\mathcal{A}=(\mathcal{Q}, \Sigma, \delta, q_0, F_{ut}, F_{rt}, F_{cand})$ *consists of*

- *a finite set* $\mathcal{Q} \subseteq \Omega \times \mathit{BExpr}$ *of* states *(each being a pair of a name of some set* Ω *and a boolean condition) and an* initial state $q_0 \in \mathcal{Q}$*,*
- *an* alphabet $\Sigma \subseteq 2^G \times \mathit{BExpr}$*,*

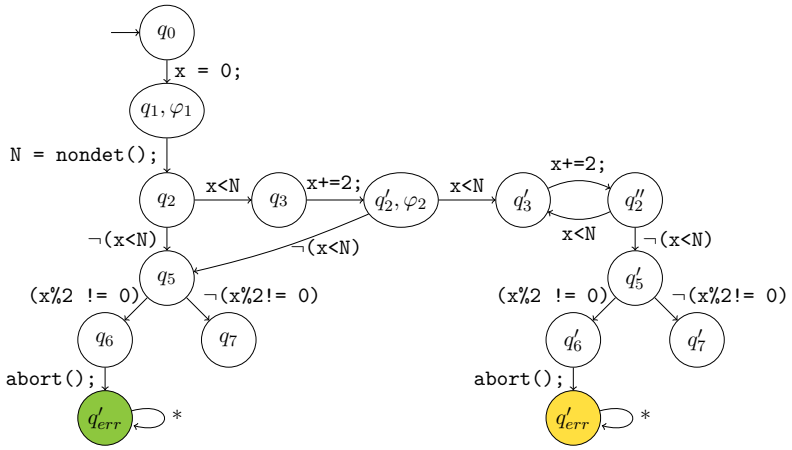


Figure 4.13: GIA A_1 generated in the third C-CEGAR iteration, containing the precision increment φ_1 generated in the first C-CEGAR iteration and φ_2 from the second iteration. In addition, it also contains the potential counterexample generated in the third C-CEGAR iteration for the example program of Figure 4.2b. States of F_{ut} are marked green and of F_{cand} yellow. We elide state invariants that are *true* and depict for transitions only the operation and non-true conditions.

- *a* transition relation $\delta \subseteq \mathcal{Q} \times \Sigma \times \mathcal{Q}$*, and*
- *three pairwise disjoint sets of* accepting states*:* F_{ut} *(for unreachable targets),* F_{rt} *(for reachable targets) and* F_{cand} *(for candidates).*

Intuitively, a GIA is an extension of a condition automaton (and thus of a protocol automaton from Definition 2.7) that has three different sets of accepting states and allows to specify state invariants.

We let $\mathcal{A}$ denote the set of all GIAs. When drawing automata, we use $*$ to denote an edge that matches any operation from Ops. We additionally require for each GIA, that (1) each state in the sets of accepting states F_{ut} and F_{rt} has no transitions to states not in F_{ut} (resp. F_{rt}) and (2) each accepting state from F_{cand} has at least a transition to itself[4]. More formally, we require that:

1. $\forall q_{ut} \in F_{ut} : \neg\exists q \in \mathcal{Q} : (q_{ut}, op, q) \in \delta \wedge q \notin F_{ut}$,
2. $\forall q_{rt} \in F_{rt} : \neg\exists q \in \mathcal{Q} : (q_{rt}, op, q) \in \delta \wedge q \notin F_{rt}$,
3. $\forall q_{cand} \in F_{cand} : (q_{cand}, *, q_{cand}) \in \delta$.

We depict in Figure 4.13 the GIA A_1 that is generated in the third C-CEGAR iteration using Craig interpolation for the example from Figure 4.2b. Based on the safety property $S = (\ell_{err}, \mathit{false})$, we use the set of target nodes $TN = \{\ell_{err}\}$. Thus, we have in Figure 4.13 $F_{ut} = \{q_{err}\}$, $F_{cand} = \{q'_{err}\}$, and $F_{rt} = \emptyset$, as the program is correct. Additionally, we depict in Appendix A.2.5.1 an example generated during cooperative test case generation where all three sets of accepting states are non-empty.

[4]This property is useful to have a single path π covering several nodes from F_{cand}.

To fulfill the requirement (R6) from Section 4.2.2, we need to define context-independent semantics. Thus, the three sets of accepting states are employed to describe three different languages of a GIA: the set of paths leading to (1) F_{ut}, (2) F_{rt}, and (3) F_{cand}. We first define what it means that a GIA covers a path, which is similar to the covering relation of condition automata and thus protocol automata. Covered semantic paths are used to establish a connection between information encoded within the GIA A and the program represented as CFA C.

Definition 4.4. *A GIA* $\mathsf{A} = (\mathcal{Q}, \Sigma, \delta, q_0, F_{ut}, F_{rt}, F_{cand})$ covers a path $\pi = \langle \ell_0, \sigma_0 \rangle \xrightarrow{g_1} \ldots \xrightarrow{g_n} \langle \ell_n, \sigma_n \rangle$ *from a CFA* C*, if there is a sequence* $\rho = (q_0, \varphi_0) \xrightarrow{(G_1,\psi_1)} \ldots \xrightarrow{(G_k,\psi_k)} (q_k, \varphi_k)$, $0 \leq k \leq n$ *in* A *(called* run*), with* $(q_{i-1}, \varphi_{i-1}) \xrightarrow{(G_i,\psi_i)} (q_i, \varphi_i) \in \delta$ *, such that*

1. $q_k \in F_{ut} \cup F_{rt} \cup F_{cand}$,
2. $g_i \in G_i$ *for all* $i \in \{1, \ldots, k\}$,
3. $\sigma_i \models \psi_i$ *for all* $i \in \{1, \ldots, k\}$,
4. $\sigma_i \models \varphi_i$ *for all* $i \in \{0, \ldots, k\}$.

We say that A X-covers i, $X \in \{ut, rt, cand\}$, *when* $q_k \in F_X$.

We allow that the run ρ has fewer states than the path π, as each state from F_{cand} has a transition to itself and states from F_{ut} resp. F_{rt} have transitions to states in F_{ut} resp. F_{rt}. Depending on the parameter value for X-cover, we define three sets of paths (languages) of a GIA A: $\mathcal{P}_{ut}(\mathsf{A}), \mathcal{P}_{rt}(\mathsf{A})$ and $\mathcal{P}_{cand}(\mathsf{A})$. These three sets are then used to establish the connection between a GIA A and a CFA C: If e.g., a path $\pi \in paths(C)$ reaches a target node ℓ_t and $\pi \in \mathcal{P}_{rt}(\mathsf{A})$, the GIA encodes the information that ℓ_t is reachable.

Using these definitions, we can exemplify the advantages of using GIA in Figure 4.13: As explained in Section 4.1.5, the precision increment A_1^{IW} that is generated in the first iteration if C-CEGAR contains the justification for the correctness of the path with no loop iteration, A_2^{IW} the one from the second iteration contains a justification for the path with a single loop iteration. The GIA A_1 from Figure 4.13 contains all information that is computed in any previous iteration of C-CEGAR, ensuring that no information computed is lost. More precisely, it contains both computed precision increments in the form of the state invariants φ_1 and φ_2 and encodes the information that the paths to the target node without and the path with a single loop iteration are *ut*-covered and thus unreachable. In addition, the counterexample generated in the third loop iteration, which contains two or more loop iterations, is also encoded.

Next, we can formally define the *correctness* of the analysis information encoded in a GIA. Thereby, we are able to later reason about the correctness of combinations of tools in a cooperative setting.

Definition 4.5. *Let* A *be a GIA,* C *a CFA and* $TN \subseteq L$ *a set of target nodes.* A *is said to be* correct wrt. C and TN *if* $\mathcal{P}_{ut}(\mathsf{A}) \subseteq \{\pi \in paths(C) \mid \pi \text{ is infeasible or } \pi \text{ is feasible and reaches no } \ell_t \in TN\}$ *and* $\mathcal{P}_{rt}(\mathsf{A}) \subseteq \{\pi \in paths(C) \mid \pi \text{ is feasible and reaches some } \ell_t \in C\}$.

Correctness thus means the automaton correctly (according to the program and the set of target nodes) marks paths as infeasible, as reaching no target or reaching some target nodes. Similarly, we can define the *soundness* of an OA or UA analysis, assuming that the target nodes TN are encoded within the program C. Soundness is also needed to reason about the correctness of combinations of tools in a cooperative setting.

Definition 4.6. *Let* tool *be an OA or UA analysis producing a GIA as output, i.e., we assume the* tool *to encode a mapping* $\mathsf{tool} : \mathcal{C} \times \mathcal{A} \to \mathcal{A}$.

If tool *is an OA analysis, it is* sound *whenever for all* $\mathsf{A}, \mathsf{A}' \in \mathcal{A}$, $C \in \mathcal{C}$ *with* $\mathsf{tool}(C, \mathsf{A}) = \mathsf{A}'$ *we have*

- $\mathcal{P}_{ut}(\mathsf{A}') \supseteq \mathcal{P}_{ut}(\mathsf{A})$ *and* $\mathcal{P}_{rt}(\mathsf{A}') = \mathcal{P}_{rt}(\mathsf{A})$, *and*
- $\forall \pi \in \mathcal{P}_{ut}(\mathsf{A}') \setminus \mathcal{P}_{ut}(\mathsf{A})$: π *is an infeasible path of* C *or is feasible but reaches no* $\ell_t \in TN$.

If tool *is an UA analysis, it is* sound *whenever for all* $\mathsf{A}, \mathsf{A}' \in \mathcal{A}$, $C \in \mathcal{C}$ *with* $\mathsf{tool}(C, \mathsf{A}) = \mathsf{A}'$ *we have*

- $\mathcal{P}_{rt}(\mathsf{A}') \supseteq \mathcal{P}_{rt}(\mathsf{A})$ *and* $\mathcal{P}_{ut}(\mathsf{A}') = \mathcal{P}_{ut}(\mathsf{A})$, *and*
- $\forall \pi \in \mathcal{P}_{rt}(\mathsf{A}') \setminus \mathcal{P}_{rt}(\mathsf{A})$: π *is a feasible path of* C *reaching some* $\ell_t \in TN$.

Consequently, a sound tool always generates a correct GIA when started with a correct one.

Finally, we can define when verification or test case generation is *completed*, namely, when a correct GIA A is generated for a CFA C such that for all target nodes ℓ_t there exists some $\pi \in \mathcal{P}_{rt}(\mathsf{A})$ such that π reaches ℓ_t, or all π reaching ℓ_t are *ut*-covered ($\forall \pi \in paths(C) : \ell_t \in \pi \Rightarrow \pi \in \mathcal{P}_{ut}(\mathsf{A})$) (all target nodes covered or all paths to a target node are unreachable).

4.2.4 GIA and Off-the-shelf Tools

GIAs are designed to be applicable in different scenarios of cooperative verification or test case generation. In the following, we have a closer look at how GIAs are used in the context of component-based CEGAR, especially when using off-the-shelf tools, as this is a core property of cooperative verification. We sketch the usage of GIAs in other scenarios like cooperative test case generation, CoVEGI or CMC in Appendix A.2.1.

To be able to employ off-the-shelf tools, for which we neither require to use GIAs as input nor to encode the information computed within a GIA, we again make use of

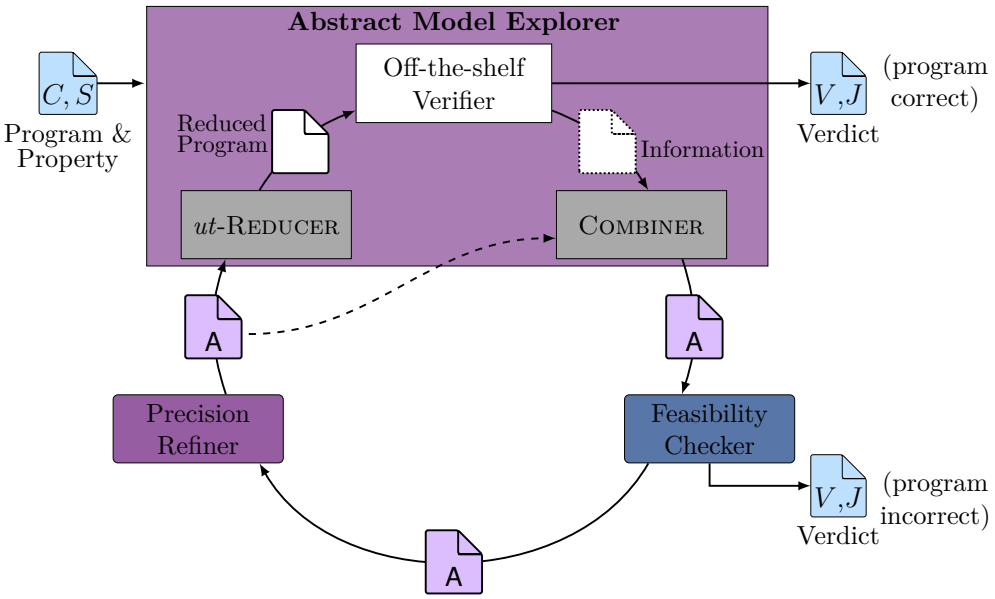


Figure 4.14: C-CEGAR using GIA, *ut*-Reducer and Combiner

additional components. The main difference to the previous constructions is that, once the so-called reducer and combiner are established, a tool can be employed in multiple different scenarios without requiring new adapters for different exchange formats. Thus, to use the existing tools in a black-box manner, we need two more operators on GIAs: (1) a way of encoding the information in the artifact into the only form of input accepted by the majority of tools, i.e., programs, and (2) a way of combining several partial results about programs as given by GIAs into one GIA to not lose any information.

We introduce the two components *reducer* for the former case and a *combiner* for the latter case that perform these operations. We depict in Figure 4.14 an instance of C-CEGAR, but in this scenario, we use an off-the-shelf verifier as an abstract model explorer and assume that feasibility checker and precision refiner can process GIAs. The GIA computed from the precision refiner may also contain information that is computed by the abstract model explorer in previous iterations, i.e., paths of the program that are proven safe. Using a reducer[5] removes all paths from the program proven safe. The reduced program is given to the off-the-shelf verifier to explore the remaining program. To be able to feed this information back into a GIA, we employ a combiner to combine the information computed by the off-the-shelf tool with the GIA generated by the precision refiner. The resulting GIA is then given to the feasibility checker and the cycle continues, whereas no information computed in earlier iterations is lost.

4.2.4.1 Reducer

For the first operation on GIAs, we use the concept of *reducers* as introduced in [CJW15; BJLW18]. A reducer reduces a program by removing some paths and thereby allows

[5]Here, we use an *ut*-Reducer, which is explained later on.

off-the-shelf tools to use the information computed by others. We define two different reducers, one removing paths that are *ut*-covered by the GIA and one removing those that are *rt*-covered.

Definition 4.7. *An* X-reducer *for* $X \in \{ut, rt\}$ *is a mapping* $red_X : \mathcal{C} \times \mathcal{A} \to \mathcal{C}$ *s.t.*

$$\forall C \in \mathcal{C}, \mathsf{A} \in \mathcal{A} : P \subseteq paths(red_X(C, \mathsf{A})) \subseteq paths(C) \ ,$$
$$where\ P = \begin{cases} paths(C) \setminus \mathcal{P}_X(\mathsf{A}) & \text{if } F_{cand} = \emptyset \text{ in } \mathsf{A} \\ \mathcal{P}_{cand}(\mathsf{A}) \setminus \mathcal{P}_X(\mathsf{A}) & \text{otherwise.} \end{cases}$$

A reducer for $X = ut$ if $F_{cand} = \emptyset$ is already existing [BJLW18]. In Algorithm 3 we provide a parameterized reducer for both values of X, building on the existing one[6].

Algorithm 3 X-Reducer

Input: CFA $C = (L, \ell_0, G)$ ▷ *CFA*
GIA $\mathsf{A} = (\mathcal{Q}, \Sigma, \delta, (q_0, \varphi_0), F_{ut}, F_{rt}, F_{cand})$ ▷ *GIA*
Output: CFA $C_r = (L_r, \ell_0^r, G_r)$ ▷ *Reduced CFA*

1: $(L_r, \ell_0^r, G_r) :=$ Reducer$(C, (\mathcal{Q}, \Sigma, \delta, (q_0, \varphi_0), F_X))$; ▷ *Call existing reducer*
2: **if** $F_{cand} \neq \emptyset$ **then**: ▷ *Over-approximate paths cand-covered*
3: $keep := \emptyset$;
4: **for each** $\ell = (\ell_i, (q_i, \varphi_i)) \in L_r$ s.t. $(q_i, \varphi_i) \in F_{cand}$ **do**
5: add all predecessors and successors w.r.t. δ of ℓ in L_r to $keep$;
6: **for each** $\ell \in L_r$ **do** ▷ *Remove paths not cand-covered*
7: **if** $\ell \notin keep$ **then**
8: Remove ℓ from L_r;
9: Remove all $(\ell, \cdot, \cdot), (\cdot, \cdot, \ell)$ from G_r;
10: **return** (L_r, ℓ_0^r, G_r);

It first calls the existing reducer and obtains a program reduced wrt. X. As $\mathcal{P}_{cand}$ contains the set of interesting paths whereon the succeeding tool should focus, X-Reducer minimizes the computed reduced CFA wrt. these paths (in lines 2 to 11). We get the following result:

Theorem 4.8. *Algorithm 3 is an* X*-reducer according to Definition 4.7.*

Proof. We first show that Algorithm 3 works correctly if $F_{cand} = \emptyset$ holds: The algorithm Reducer called in line 2 takes an automaton with one set of final states F as input. It has been shown that Reducer retains at least all paths that are not covered by the given automaton w.r.t. F and that the program generated does not contain any path that is not present in the original program [BJLW18]. We call Reducer with the GIA A only using F_X, thus it reduces the program such that at most all paths that are $X-$covered by the GIA are removed. Therefore, Reducer and thus Algorithm 3 work correctly if

[6] Algorithm 3 assumes for representation purposes that the GIA does not contain state invariants. A full construction, covering this aspect is given in the Algorithm 7 in Appendix A.2.2.

Algorithm 4 Combiner

Input: GIA $\mathsf{A}_I = (\mathcal{Q}_1, \Sigma, \delta_1, q_0, F^1_{ut}, F^1_{rt}, F^1_{cand})$ ▷ *First GIA*
GIA $\mathsf{A}'_I = (\mathcal{Q}_2, \Sigma, \delta_2, s_0, F^2_{ut}, F^2_{rt}, F^2_{cand})$ ▷ *Second GIA*
Output: GIA $\mathsf{A}_O = (\mathcal{Q}, \Sigma, \delta, p_0, F_{ut}, F_{rt}, F_{cand})$ ▷ *Combined GIA*

1: $\mathcal{Q} := \{((q_0, s_0), true)\}; p_0 := ((q_0, s_0), true)$; $\delta := \emptyset$; waitlist := $\{((q_0, s_0), true)\}$;
2: **while** waitlist $\neq \emptyset$ **do**
3: select and remove $((q_i, s_i), \varphi_i)$ from waitlist;
4: **for each** $t_1 = ((q_i, \varphi_i) \xrightarrow{g_i, \psi_i} (q_{i+1}, \varphi_{i+1})) \in \delta_1$ **do** ▷ *Merge the states*
5: **if** $\nexists((s_i, \varphi_i) \xrightarrow{g_j, \psi_j} (s_{i+1}, \varphi'_{i+1})) \in \delta_2 : g_i = g_j \vee s_i \in \{\circ, \bullet\}$ **then**
6: **if** $s_i \in \{\circ, \bullet\}$ **then** $s_{i+1} = s_i$; **else** $s_{i+1} = \circ$;
7: $\mathcal{Q} := \mathcal{Q} \cup \{(q_{i+1}, s_{i+1}), \varphi_{i+1}\}$;
8: $\delta := \delta \cup \{((q_i, s_i), \varphi_i); \xrightarrow{g_i, \psi_i} ((q_{i+1}, s_{i+1}), \varphi_{i+1})\}$
9: **if** $q_{i+1} \notin F^1_{rt} \cup F^1_{ut}$ **then**
10: waitlist := waitlist $\cup \{((q_{i+1}, s_{i+1}), \varphi_{i+1})\}$;
11: **else**
12: **for each** $t_2 = ((s_i, \varphi_i) \xrightarrow{g_j \psi_j} (s_{i+1}, \varphi'_{i+1})) \in \delta_2 : g_i = g_j$ **do**
13: waitlist, $\mathcal{Q}, \delta$:= Merge(waitlist, $\mathcal{Q}, \delta, t_1, t_2$);
14: **for each** $((s_i, \varphi_i) \xrightarrow{g_j, \psi_j} (s_{i+1}, \varphi_{i+1}) \in \delta_2)$ **do** analogously to lines 4-13
15: $F_{rt} = \{(q_i, s_i) \in \mathcal{Q} \mid q_i \in F^1_{rt} \vee s_i \in F^2_{rt}\}$;
16: $F_{ut} = \{(q_i, s_i) \in \mathcal{Q} \mid q_i \in F^1_{ut} \vee s_i \in F^2_{ut}\}$;
17: $F_{cand} = \{(q_i, s_i) \in \mathcal{Q} \mid q_i \in F^1_{cand} \cup \{\bullet\} \wedge s_i \in F^2_{cand} \cup \{\bullet\}\}$;
18: **if** $F_{rt} \cap F_{ut} \neq \emptyset$ **then return** ERROR; **end if**
19: **return** $\mathsf{A}_O = (\mathcal{Q}, \Sigma, \delta, p_0, F_{ut}, F_{rt}, F_{cand})$;

where $\circ$,$\bullet$ are replacements for a state used during splitting and are not processed.

$F_{cand} = \emptyset$. In case $F_{cand} \neq \emptyset$, the reducer has to generate a program that contains at least all paths only *cand*-covered by A. In lines 3-9 we build a set containing a superset of these paths and remove the other paths, i.e., only these that are not *cand*-covered by A. Thus, Algorithm 3 also works in this case concluding the proof. □

4.2.4.2 Combiner

When several tools compute analysis information, we have to make sure that all this information is preserved. To this end, we introduce a *combiner* for the combination of GIAs. The combiner's goal is to keep all information on $\mathcal{P}_{ut}$ and $\mathcal{P}_{rt}$ from both GIAs.

Definition 4.9. *A* combiner *is a partial mapping* $comb : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ *which is defined on* consistent *GIAs* A *and* A' *with* $\mathcal{P}_{ut}(\mathsf{A}) \cap \mathcal{P}_{rt}(\mathsf{A}') = \emptyset = \mathcal{P}_{rt}(\mathsf{A}) \cap \mathcal{P}_{ut}(\mathsf{A}')$ *such that*

$$\forall \mathsf{A}, \mathsf{A}' \in \mathcal{A} : \\ \mathcal{P}_{ut}(comb(\mathsf{A}, \mathsf{A}')) = \mathcal{P}_{ut}(\mathsf{A}) \cup \mathcal{P}_{ut}(\mathsf{A}') \wedge \mathcal{P}_{rt}(comb(\mathsf{A}, \mathsf{A}')) = \mathcal{P}_{rt}(\mathsf{A}) \cup \mathcal{P}_{rt}(\mathsf{A}').$$

An algorithm for a combiner is given in Algorithm 4, for presentation purposes assuming that each edge in δ_1, δ_2 contains only a single transition. The intuitive idea of the Combiner is to build the union of the two GIAs $\mathsf{A}_I, \mathsf{A}'_I$ and consider newly com-

puted information: For example, if there is a path π, $\pi \in \mathcal{P}_{cand}(\mathsf{A}_I)$ and $\pi \in \mathcal{P}_{ut}(\mathsf{A}'_I)$, Combiner ensures that $\pi \in \mathcal{P}_{ut}(\mathsf{A}_O)$ holds for the combined GIA A_O. To this end, Combiner builds the new GIA A_O by searching for common subpaths in the GIAs A_I and A'_I. A state in A_O is a tuple (a_1, a_2) of two states, $a_1 \in Q_1$ and $a_2 \in Q_2$, both reachable on the same path. If the paths diverge, the state is split, where the placeholders '$\circ$' and '$\bullet$' are used to replace either a_1 or a_2. We use e.g., '$\circ$' if the transitions from a_1 and from a_2 contain different CFA edges and '$\bullet$' if the successor states have different state invariants. To ensure that no information for *cand*-covered paths are lost when computing F_{cand}, the set contains also the states where one element from the state tuple is in '$\bullet$', as '$\bullet$' is used for splitting states having different state invariants and thus both paths needs to be contained. For a combination of states, Algorithm 4 applies the method `Merge`, given in Algorithm 5. Intuitively, `Merge` ensure that paths from $\mathcal{P}_{ut}$ and $\mathcal{P}_{rt}$ are preserved as well as additional conditions for paths in F_{cand}. More details about Algorithm 5 are given in Appendix A.2.3. We exemplify the use of reducer and combiner for the program of Figure 4.2b in Section 4.2.5 and provide a more complex example of an application to Combiner in Appendix A.2.5.2.

The combined GIA computed by Algorithm 4 is not guaranteed to be minimal, meaning that it may contain some paths multiple times and contain paths that do not lead to an accepting state. Despite this fact, we can prove the following:

Theorem 4.10. *Algorithm 4 is a combiner according to Definition 4.9.*

Proof. Intuitively, we have to show that for the combination A_O of two GIAs A_I and A'_I each path *rt*-covered by either A_I or A'_I is also *rt*-covered by A_O and that the reverse holds (and analogously, that both properties hold for *ut*-covered paths). We therefore inductively construct an accepting run of A_O for a path π that is *rt*-covered by either A_I or A'_I and vice versa. The full formal proof can be found in Appendix A.2.4. □

4.2.4.3 Using Reducer and Combiner

Finally, we can state that connecting tools via reducers and combiners, as depicted in Figure 4.14, does not lose any of the already computed analysis results. This property guarantees that any arbitrary combination of sound OA and UA tools using reducer and combiner achieves the same progress as if they would work directly on GIAs.

Theorem 4.11. *Let $\mathsf{A} \in \mathcal{A}$ be a correct GIA, $C \in \mathcal{C}$ a CFA, tool a sound UA or OA analysis and $X \in \{ut, rt\}$. Then for a GIA $\mathsf{A}' = comb(\mathsf{tool}(red_X(\mathsf{A}, C), \mathsf{A}), \mathsf{A})$ we get*

- $\mathcal{P}_{rt}(\mathsf{A}') = \mathcal{P}_{rt}(\mathsf{A}) \wedge \mathcal{P}_{ut}(\mathsf{A}') \supseteq \mathcal{P}_{ut}(\mathsf{A})$ *if tool is an OA, and*
- $\mathcal{P}_{ut}(\mathsf{A}') = \mathcal{P}_{ut}(\mathsf{A}) \wedge \mathcal{P}_{rt}(\mathsf{A}') \supseteq \mathcal{P}_{rt}(\mathsf{A})$ *if tool is an UA.*

Proof. As sound OA and UA tools increase the set of *ut*-covered resp. *rt*-covered paths and the reducer retains all this information, the correctness follows directly from Definition 4.6 and Theorems 4.8 and 4.10. □

Algorithm 5 Merge

Input: waitlist, $\mathcal{Q}, \delta$
$t_1 = ((q_i, \varphi_i) \xrightarrow{g_i, \psi_i} (q_{i+1}, \varphi_{i+1}))$
$t_2 = ((s_j, \varphi_j) \xrightarrow{g_j, \psi_j} (s_{j+1}, \varphi_{j+1}))$ ▷ *requirement:* $g_i = g_j$
Output: waitlist, $\mathcal{Q}, \delta$

1: **if** $\psi_i = \psi_j \wedge \varphi_i = \varphi_j \wedge ((q_{i+1} \in F^1_{ut} \cup F^1_{rt} \cup F^1_{cand}) \Leftrightarrow (s_{i+1} \in F^2_{ut} \cup F^2_{rt} \cup F^2_{cand}))$ **then**
2: $\mathcal{Q} := \mathcal{Q} \cup \{((q_{i+1}, s_{j+1}), \varphi_{i+1})\}$;
3: $\delta := \delta \cup \{((q_i, s_j), \varphi_i) \xrightarrow{g_i, \psi_i} ((q_{i+1}, s_{j+1}), \varphi_{i+1})\}$;
4: **if** $q_{i+1} \notin F^1_{rt} \cup F^1_{ut} \wedge s_{j+1} \notin F^2_{rt} \cup F^2_{ut}$ **then**
5: waitlist := waitlist $\cup\{((q_{i+1}, s_{j+1}), \varphi_{i+1})\}$;
6: **else if** $(\varphi_i = \varphi_j) \wedge (reach_{cand}(q_{i+1}) \wedge reach_{cand}(s_{j+1})) \wedge (trueCond(q_i) \vee trueCond(s_j))$ **then**
7: $\mathcal{Q} := \mathcal{Q} \cup \{((q_{i+1}, s_{j+1}), \varphi_{i+1})\}$;
8: $\delta := \delta \cup \{((q_i, s_j), \varphi_i) \xrightarrow{g_i, \psi} ((q_{i+1}, s_{j+1}), \varphi_{i+1})\}$; ▷ $\psi \in \{\psi_i, \psi_j\}, \psi \neq true$
9: **if** $q_{i+1} \notin F^1_{rt} \cup F^1_{ut} \wedge s_{j+1} \notin F^2_{rt} \cup F^2_{ut}$ **then**
10: waitlist := waitlist $\cup\{((q_{i+1}, s_{j+1}), \varphi_{i+1})\}$;
11: **else if** $(\varphi_i = \varphi_j) \wedge (reach_{ut,rt}(q_{i+1}) \wedge reach_{cand}(s_{j+1})$ **then**
12: $\mathcal{Q} := \mathcal{Q} \cup \{((q_{i+1}, \circ), \varphi_{i+1})\}$;
13: $\delta := \delta \cup \{((q_i, s_j), \varphi_i) \xrightarrow{g_i, \psi_i} (q_{i+1}, \circ), \varphi_{i+1})\}$; ▷ *Using* ψ_i
14: **if** $q_{i+1} \notin F^1_{rt} \cup F^1_{ut}$ **then** waitlist := waitlist $\cup\{((q_{i+1}, \circ), \varphi_{i+1})\}$; **end if**
15: **else if** $(\varphi_i = \varphi_j) \wedge (reach_{cand}(q_{i+1}) \wedge reach_{ut,rt}(s_{j+1})$ **then**
16: $\mathcal{Q} := \mathcal{Q} \cup \{((\circ, s_{j+1}), \varphi_{i+1})\}$;
17: $\delta := \delta \cup \{((q_i, s_j), \varphi_i) \xrightarrow{g_i, \psi_j}; (\circ, s_{j+1}), \varphi_{i+1})\}$; ▷ *Using* ψ_j
18: **if** $s_{j+1} \notin F^2_{rt} \cup F^2_{ut}$ **then** waitlist := waitlist$\cup\{((\circ, s_{j+1}), \varphi_{i+1})\}$; **end if**
19: **else**
20: $\mathcal{Q} := \mathcal{Q} \cup \{((q_{i+1}, \bullet), \varphi_{i+1}), ((\bullet, s_{j+1}), \varphi_{j+1})\}$;
21: newS := newS $\cup\{((q_{i+1}, \bullet), \varphi_{i+1}), ((\bullet, s_{j+1}), \varphi_{j+1})\}$;
22: $\delta := \delta \cup \{((q_i, s_j), \varphi_i) \xrightarrow{g_i, \psi_i} ((q_{i+1}, \bullet), \varphi_{i+1}), ((q_i, s_j), \varphi_j) \xrightarrow{g_j, \psi_j} ((\bullet, s_{j+1}), \varphi_{j+1})\}$;
23: **for each** $((q_k, s_l), \varphi_m) \in$ newS **do**
24: **if** $q_k \notin F^2_{rt} \cup F^2_{ut} \wedge s_l \notin F^2_{rt} \cup F^2_{ut}$ **then** waitlist := waitlist $\cup\{(q_k, s_l)\}$; **end if**
25: **return** waitlist, $\mathcal{Q}, \delta$;

4.2.5 Example Application of GIA in C-CEGAR

After having defined GIAs and proven that combining tools using reducer and combiner ensures that no information computed is lost, we exemplify the advantages of using GIAs next. Therefore, we revisit the scenario depicted in Figure 4.12, where we employ only GIAs within component-based CEGAR. Beneath demonstrating the advantages of having an exchange format with uniform semantics, we also exemplify reducer and combiner.

As in Section 4.1.5 we again employ an abstract example generator using predicate abstraction and a precision refiner computing Craig interpolates as well as an execution-based feasibility checker within C-CEGAR. Again, we let the configuration analyze the task shown in Figure 4.2b with the safety property $S = (\ell_{err}, false)$ for the

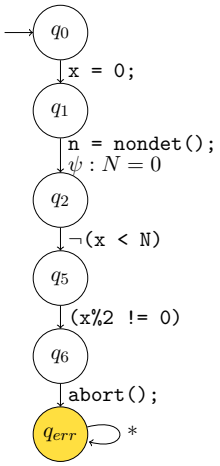


Figure 4.15: GIA A_2 containing the potential counterexample generated in the first C-CEGAR iteration

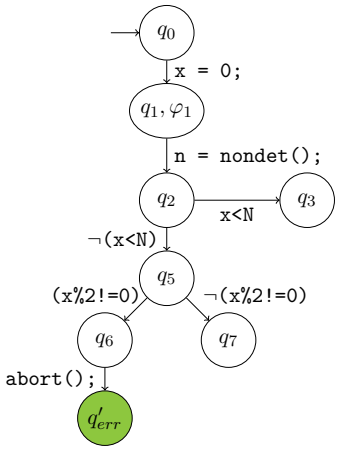


Figure 4.16: GIA A_3 containing the precision increment φ_1 and the information that the path encoded in A_2 is infeasible

CFA C_2 depicted in Figure 4.6. Thus, we use the set of target nodes $TN = \{\ell_{err}\}$ and start with the abstract model explorer with empty precision and thus no predicates.

The potential counterexample computed is encoded within the GIA A_2 from Figure 4.15, where the target node is marked as a candidate for being reached in case that N is initialized with 0. The resulting reduced program is equivalent to the path program shown in Figure 4.8. As the encoded path is infeasible, the UA feasibility checker is not able to show that there exists a path reaching the target node ℓ_{err}. Hence, the node q_{err} corresponding to ℓ_{err} is still encoded as candidate (for covering unreachable locations). As the GIA that is generated using this information is equivalent to A_2, merging the two GIAs results in an automaton equivalent to A_2.

Next, the infeasible counterexample is given to the precision refiner, and the precision increment $\varphi_1 \equiv x = 0$ is computed. In addition, as the precision refiner is an OA component, it has proven that the path to the target node without a loop iteration, encoded in A_2 is infeasible and the target node unreachable via this specific path. This information is also encoded in the GIA A_3 that is generated and shown in Figure 4.16, which is used as input for the abstract model explorer in the second iteration. As A_3 does not *ut*-cover all paths of the CFA containing the target node, the verification task is not completed and the next iteration is started. Assuming that we need to generate a reduced program as input for the abstract model explorer, we employ the algorithm Reduce using the CFA C_2 of the program and the GIA A_3 as input. We use the Reducer from Algorithm 7 and obtain the reduced program that is shown in Figure 4.17[7].

[7]More precisely, we obtain a CFA C_3 that is shown in Figure A.9 and can be transformed into the program in Figure 4.17

As already explained, the C-CEGAR cycle continues until all loop iterations are proven safe. The main advantage of using GIAs becomes more prominent in the third iteration: Therein, the two precision increments computed in the first two iterations, the information that the program is safe for no or one loop iteration, and the potential counterexample with two loop iterations are encoded within a single artifact A_1 shown in Figure 4.13. This example shows the advantages of using GIAs for the information exchange within C-CEGAR, as the uniform semantics allows for encoding all computed information within a single artifact and no information computed is lost.

```
   int prog2() {
0   unsigned int x = 0;
1   if (x == 0){
2     assert(x == 0);
3     unsigned short N =nondet();
4     if (x < N){
5       x += 2;
6 L2:   while (x < N) {
7         x += 2;
8       }
9       if (x % 2 != 0){
10        abort();
11      }else{
12        return 0;
13      }
14    }else{
15      if (x % 2 != 0){
16        return 0;
17      }else{
18        return 0;
19      }
20    }
21  }else{
22    unsigned short N =nondet();
23    GOTO L2;
24  }
25 }
```

Figure 4.17: Reduced program for A_3 and C_2

4.3 Implementation

To analyze the feasibility of the idea to decompose CEGAR, we build a decomposition of the predicate abstraction of CPAchecker implementing C-CEGAR within CoVeriTeam. In addition, we also realized GIAs based on the existing format for condition automaton, to be able to employ them within C-CEGAR. The implementation of C-CEGAR in CoVeriTeam as well as the changes to the CPAchecker and software artifacts are available and archived at Zenodo [BHLW22b; HW23].

4.3.1 Decomposing CPAchecker's Predicate Abstraction

The predicate abstraction that is implemented in CPAchecker follows the high-level description from Section 2.2.4. It is used within a CEGAR scheme realized within CPAchecker and the default configuration that we use employs lazy refinement [HJMS02], *adjustable block encoding* (ABE)[8] [BKW10], and computes interpolants using Craig interpolation. The implementation comprises two internal modules, a model explorer and a module combining a feasibility checker and precision refiner. The analysis information computed by the model explorer is stored in an ARG, which is also used by the second module for feasibility checking and precision refinement. The counterexample encoded within the ARG is validated by computing and validating the path formula of

[8] ABE is a generalization of large block-encoding allowing to specify different block sizes between one statement per block and using large block encoding.

the path of the counterexample using a solver. The newly generated precision increments in the form of predicates are directly added to the abstract states of the ARG. In addition, the precision, i.e., the set of computed predicates, is also reported using a predicate map using the SMT-2-LIB format.
CPAchecker also offers by default a configuration for validation of violation witnesses and allows to restrict the exploration of arbitrary analyses to paths contained in a violation witness.

Based on these existing implementations, we build for C-CEGAR three standalone and stateless components generating the desired output formats: The *abstract model explorer* obtains the initial precision as predicate map or invariant witness and uses predicate abstraction for building the ARG and checks if the safety property is violated. In this case, a potential counterexample path is exported as a violation witness. For the *feasibility checker*, i.e., to check a given violation witness for feasibility, we use CPAchecker's existing witness-based result validation [Bey+15], working with violation witnesses. The *precision refiner* uses the path witness to restrict the exploration of the program to only the infeasible paths encoded. It builds the path formula and uses MathSAT5's implementation of Craig interpolation for computing the precision increment. The computed interpolants are exported as predicate maps and as invariant witnesses.

As stated in Section 4.1.3, we need, among the three stateless components, a controlling unit responsible for handling the information exchange and checking, if a final answer is computed. We build this unit using CoVeriTeam. We follow the concept depicted in Figure 4.1. As all components are stateless, we need to ensure that precision increments computed in the previous iterations are also given to the abstract model explorer. We therefore store the invariant witnesses or predicate maps from the previous iterations and build a function for merging them with the newly computed precision increment within CoVeriTeam.

4.3.2 Realizing C-CEGAR using GIA

In addition to the realization of C-CEGAR, we realize GIA by extending the condition automaton from [BJLW18]. The formalization follows Definition 4.3. We extended CPAchecker, such that it can consume a GIA as input and can generate it in addition to the correctness, violation, path, and invariant witness. We included GIAs as exchange format within CoVeriTeam and integrated GIAs as exchange formats in the component-based realization of CEGAR. We built *ut*-Reducer and *rt*-Reducer described in Algorithm 7 as well as the Combiner from Algorithm 4 within CPAchecker, forming standalone-executable components, also fully integrated in CoVeriTeam.

4.4 Evaluation

The goal of our evaluation is to analyze the effect of decomposition and of using standardized exchange formats for the information exchange, especially GIAs. We analyze these two aspects separately, by first analyzing the effect of decomposition and in a second step additionally using standardized formats. Afterward, we have a closer look at whether our decomposition of CEGAR allows for building cooperative verifiers using off-the-shelf components, as components can be easily exchanged by others that implement the same interface via different concepts. The feasibility of using GIAs for the information exchange in cooperative test case generation is shown in Appendix A.2.6.

4.4.1 Evaluation Setup

To answer the research question regarding the effect of decomposition, we analyze the different effectiveness and efficiency of decomposed CEGAR instances and compare them with the default implementation. Moreover, we employ different off-the-shelf tools as feasibility checker and as precision refiner.

Configurations For analyzing the feasibility of C-CEGAR, we first evaluate three different configurations based on the decomposition of CPAchecker's predicate abstraction. The component-based version CC-Pred, uses violation witnesses, path witnesses, and the predicate map, CPAchecker's internal format to encode precision increments, to exchange information among the three components. The configuration CC-Pred-Wit uses the invariant witness instead of the predicate map (cf. Figure 4.1), the configuration CC-Pred-Gia uses GIAs for the information exchange only (cf. Figure 4.12). We call CPAchecker's default predicate abstraction using CEGAR Pred.

To also evaluate the advantages of the component-based design of C-CEGAR, we also evaluate three off-the-shelf components, employing conceptually different approaches. We employ UltimateAutomizer as a feasibility checker using its configuration for violation witness validation. Additionally, we also employ UltimateAutomizer as a precision refiner, where we chose a configuration that uses Newton refinement. Newton refinement is conceptually different from Craig interpolation which is employed by CPAchecker's precision refiner. Lastly, we also make use of FShell-witness2test as a feasibility checker. FShell-witness2test [BDLT18] is an execution-based result validator for violation witnesses that is implemented independently of any existing verification tool. Given a violation witness, a compilable test harness is extracted and executed in combination with the original program. In case the safety property is violated, the violation witness is confirmed.

Computing Resources and Benchmark Tasks We run our experiments on machines with an Intel Core i5-1230 v5, 3.40 GHz (8 cores), 33 GB of memory, and

Table 4.1: Comparison of CPAchecker's predicate abstraction and the component-based version in three variations

	correct			incorrect	
	overall	**proof**	**alarm**	**proof**	**alarm**
Pred	3 769	2 556	1 213	3	9
CC-Pred	3 524	2 450	1 074	0	3
CC-Pred-Wit	2 854	2 110	744	0	1
CC-Pred-Gia	2 641	2 068	573	1	4

Ubuntu 18.04 LTS with Linux kernel 5.4.0-96-generic. Each verification run is limited to use 15 GB of memory, 4 CPU cores, and 15 min of CPU time. The experiments are conducted on the 2022 version of SV-Benchmarks [SVB22], where we used all benchmarks regarding reachability, in total 8 347 verification tasks.

Availability The implementations of C-CEGAR and GIAs as well as all experimental data are publicly available and archived at Zenodo. In [BHLW22b], we archive C-CEGAR and the data for all the research questions except for the GIA-related evaluation, that can be found in [HW23].

4.4.2 RQ 1: How Large is the Overhead of the Component-based Design for C-CEGAR?

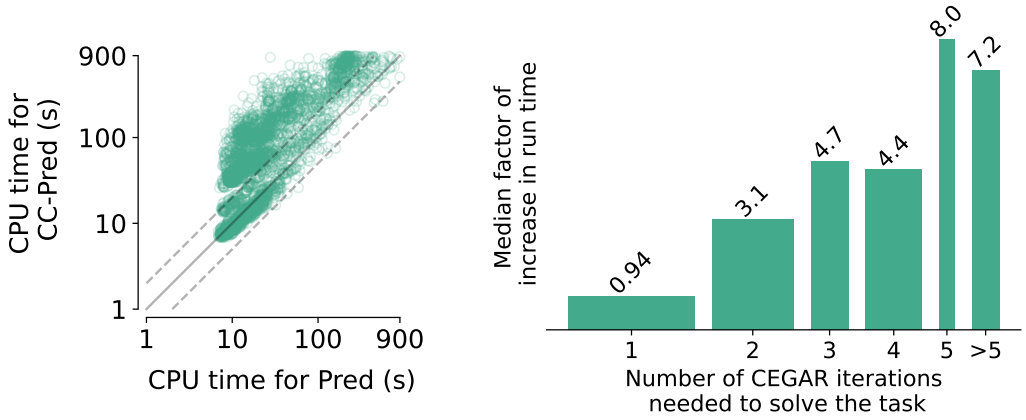
Evaluation Plan To analyze the cost of using a component-based approach, we compare the effectiveness and efficiency of Pred and our component-based version CC-Pred. To improve comparability, we configure the model explorers of both Pred and CC-Pred to start the exploration at the root of the ARG in each iteration[9].

Experimental Results For effectiveness we are interested in the number of correct answers overall, correct proofs, and alarms, as well as in the number of incorrect answers.

We present in Table 4.1 the results from our experimental evaluation. To first analyze the effect of decomposition, we compare the results of default implementation Pred and the component-based version CC-Pred. The number of tasks solved by the CC-Pred reduces from 3 769 to 3 524. There are 25 tasks that CC-Pred solves even though Pred does not, but also 270 tasks that CC-Pred fails to solve but Pred does (a 6.4 % decrease). We also observe that the number of false alarms reduces from 9 for Pred to 3 for CC-Pred, as the feasibility check used in CC-Pred is different from the one used in Pred and more precise.

The cause for the decreased effectiveness becomes visible when having a look at the efficiency of CC-Pred. As we do not add any additional parallelization by decomposing

[9]Originally, Pred uses lazy refinements to enhance performance



(a) Comparison of CPU time for Pred and CC-Pred

(b) Median factor of run time increase by CC-Pred compared to Pred. The width of the bar for i corresponds to the number of verification tasks that require exactly i CEGAR iterations.

Figure 4.18: Comparison of efficiency of Pred and CC-Pred

CEGAR, we focus on CPU time in the following. We present in Figure 4.18a a scatter plot that compares the CPU time taken by Pred on the x-axis and CC-Pred on the y-axis for tasks solved by both tools. Again, for points on the solid line, both analyses need the same CPU time, for points on the dashed lines one analysis takes twice as long as the other one. We can observe that the CPU time taken by CC-Pred is higher compared to Pred for nearly all cases, meaning that the efficiency of CC-Pred is lower. The median increase for the CPU time used by CC-Pred is 2.8.

Next, we have in Figure 4.18b a closer look at the median increases and group them with respect to the number of CEGAR iterations conducted to solve the task. First of all, we can observe that the median increase strongly correlates with the number of CEGAR iterations that are needed to solve the task. For tasks solved within the first CEGAR iteration, the CPU time consumed by CC-Pred does, in the median, not increase compared to Pred, the factor is 0.9. The overhead caused by the decomposition becomes visible for tasks solved in more than one CEGAR iteration. The width of the i-th bar corresponds to the number of tasks solved with i CEGAR iterations. As almost 95% of all tasks can be solved within 5 CEGAR iterations, we summarized the other iterations within the last bar. In short, the efficiency of CC-Pred decreases compared to Pred, where the decrease correlates to the number of conducted CEGAR iterations.

Multiple reasons could cause the decrease: As each component is stateless, there are startup times of the Java Virtual Machine (JVM) and missing caching, especially for the solvers. In addition, there is I/O overhead for reading and writing the exchange formats. Recently, Beyer et al. showed that using a microservice-based architecture for C-CEGAR leads to a comparable efficiency as Pred [BLW23]. Their results indicate that the JVM startup time and caching are the main reasons for the lower performance of CC-Pred.

Using these insights regarding efficiency, it turns out that the decreased effectiveness of CC-Pred compared to Pred is nearly exclusively caused by the fact that CC-Pred takes more time to compute a solution. When increasing the CPU time limit for CC-Pred by the factor of twelve (to 180 min), it fails only on 60 tasks, a 1.7 % decrease compared to Pred. The remaining 60 unsolved tasks are caused by the feasibility checker used by CC-Pred. In contrast to Pred, it (1) rejects more counterexamples because it is more precise than the internal check of Pred, (2) explores paths with unsupported program features that Pred does not visit, or (3) triggers SMT errors because different interpolation sequences are queried. All three issues are not related to the decomposition but rather to inconsistencies between the feasibility checker used internally in Pred and the one used by CC-Pred. We conclude that decomposing Pred to CC-Pred does not influence the ability to solve verification tasks, except for the three issues mentioned above and taking the lower efficiency of CC-Pred into account.

Results

Decomposing an existing CEGAR implementation into components has (almost) no negative effects on the effectiveness of the approach. The efficiency of CC-Pred decreases only by a constant factor (median smaller than three). The decomposed instance can have a higher precision because better components can be used.

4.4.3 RQ 2: What Are the Costs for Using Standardized Formats?

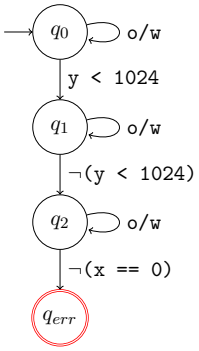
Evaluation Plan After having analyzed the effect of decomposition in isolation by using the predicate map within the information exchange, we are next interested in analyzing the effect of using standardized formats only. Hence, we compare the effectiveness and efficiency of CC-Pred-Wit, the configuration using violation, path, and invariant witnesses to exchange information (as depicted in Figure 4.1) and CC-Pred-Gia, a configuration employing GIAs only (as shown in Figure 4.12) with CC-Pred.

Experimental Results For effectiveness, we also present the results of CC-Pred-Wit and CC-Pred-Gia in Table 4.1. CC-Pred-Wit solves in total 2 854 tasks, computing 2 110 correct proofs and 744 correct alarms. Compared to CC-Pred, the effectiveness reduces by 670 tasks, a decrease of around 20 %. CC-Pred-Gia solves in total 2 641 tasks, computing 2 068 correct proofs and 573 correct alarms. Compared to CC-Pred, the effectiveness reduces by 883 tasks, a decrease of around 25%. Comparing CC-Pred-Gia and CC-Pred-Wit, we observe that CC-Pred-Gia solves in total 213 fewer tasks than CC-Pred-Wit. Important to notice that there are 125 tasks that are solved by CC-Pred-Gia but not by CC-Pred-Wit.

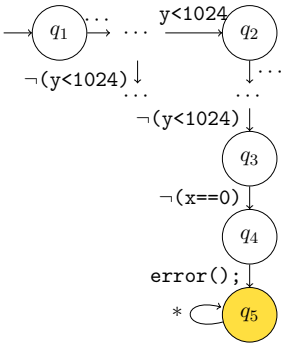
The decreased effectiveness of CC-Pred-Wit compared to CC-Pred reasons mostly in the fact that the precision refiner used within CC-Pred-Wit does not add the computed precision increment to the invariant witnesses. Due to the fact that in-

```
   int prog3(void) {
0    unsigned int x = 1;
1    unsigned int y = 0;
2
3    while (y < 1024) {
4      x = 0;
5      y++;
6    }
7    if (x == 0) {}
8    else
9      error();
10 }
```

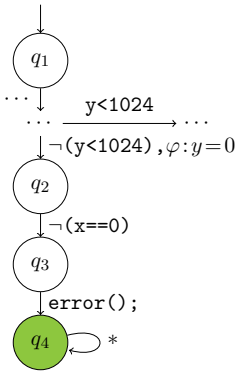
(a) Program prog3, where $x = 0$ is not a valid invariant at the loop head



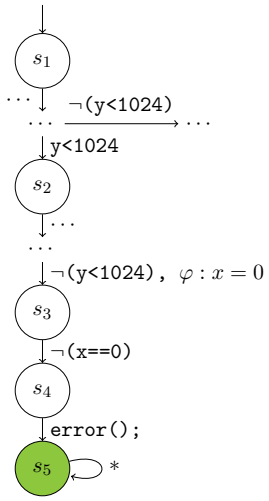
(b) Counterexample A_3^{VW} computed by model explorer



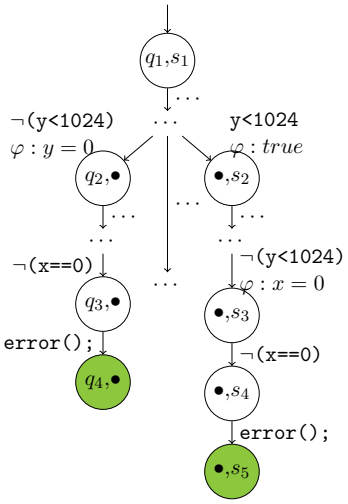
(c) Counterexample A_4 computed by model explorer



(d) Precision increment A_5 computed in first iteration



(e) Precision increment A_6 computed in second iteration

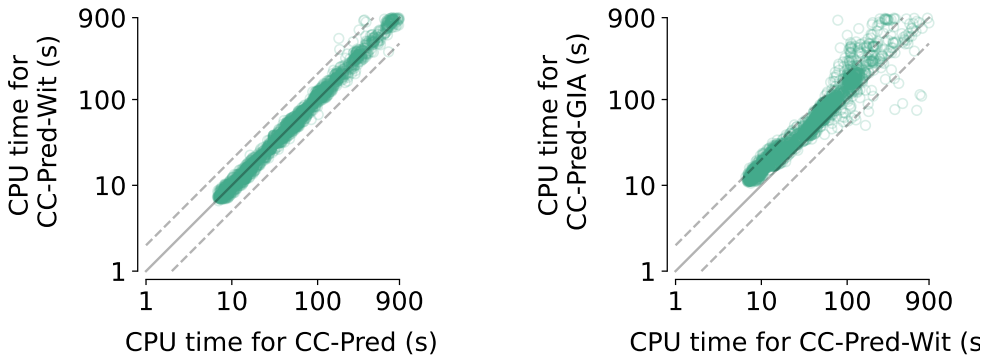


(f) Combined precision increment

Figure 4.19: Example showing difference between witness and GIA as exchange formats

variant witnesses are based on correctness witnesses and correctness witnesses are not primarily designed for exchanging information of precision increments, we regularly observe that CC-Pred-Wit gets stuck in an endless loop, as the same infeasible counterexample is computed by the model explorer over and over again until it is eventually aborted.

We exemplify this observation and present in Figure 4.19a a program from the evaluation. In the program, the value of x is 1 before the loop is executed for the first time and 0 after each loop iteration. In the experiments, we observe that the abstract model explorer used generates in the first iteration a counterexample containing no loop iteration that is rejected by the feasibility checker. The precision increment computed is $y = 0$, which is valid before line 2 and rules the counterexample out. Using this predicate, the model explorer now generates a counterexample that is shown as violation witness A_3^{VW} in Figure 4.19b and as GIA A_4 in Figure 4.19c. The counterexample



(a) Comparison of CPU time for CC-Pred and CC-Pred-Wit

(b) Comparison of CPU time for CC-Pred-Wit and CC-Pred-Gia

Figure 4.20: Scatter plots comparing the CPU time of C-CEGAR instances using standardized exchange formats

contains exactly one loop iteration. The new precision increment computed is $x = 0$ valid after the first loop iteration. Note that $x = 0$ is not a valid loop invariant, as it does not satisfy establishment (cf. (2.4)). Although the invariant witnesses format can conceptually be used to express loop unrollings and thus can contain the new predicate, none of the precision refiners used encodes these or comparable predicates in an invariant witness, they generate an invariant witness without any predicate. In contrast, using GIAs as an exchange format allows the precision refiner to build the GIA depicted in Figure 4.19e, precisely encoding the spurious counterexample as a path leading to $s_5 \in F_{ut}$, where the new predicate is present as an assumption at the edge to s_3 (after the first loop iteration). To not lose any information, the GIAs from the first two iterations A_5 in Figure 4.19d and A_6 in Figure 4.19e are combined into the GIA depicted in Figure 4.19f[10]. It contains two paths, one with the precision increment $x = 0$, and one with the precision increment $y = 0$. In the third iteration, these two predicates (and their negation) are sufficiently precise, such that the model explorer proves all paths leading to the target node unreachable.

In this example, using GIAs for the information exchange is preferable compared to invariant witnesses, as tools can use GIAs uniform semantics to safely encode all information computed. In the experiments, we see this effect on the 125 tasks for that CC-Pred-Gia computes a solution, in contrast to CC-Pred-Wit. Nevertheless, CC-Pred-Wit has overall a higher effectiveness compared to CC-Pred-Gia. The difference originates mostly in the fact that CC-Pred-Gia is not able to compute a solution in the given time limit for 299 tasks, for which CC-Pred-Wit computes a solution within 900 seconds.

[10]In case that reducer and combiner are applied. Otherwise, there is no need to merge GIAs.

Table 4.2: C-CEGAR using different off-the-shelf components

Feasibility	**correct**					**incorrect**	
Checker	**overall**	**proof**	unique	**alarm**	unique	**proof**	**alarm**
CPAchecker	2 854	2 110	494	744	441	0	1
FShell-witness2test	1 223	1 126	0	97	64	0	0
UltimateAutomizer	1 941	1 614	4	327	29	0	1

Precision	**correct**					**incorrect**	
Refiner	**overall**	**proof**	unique	**alarm**	unique	**proof**	**alarm**
CPAchecker	2 854	2 110	709	744	436	0	1
UltimateAutomizer	1 739	1 430	29	309	1	0	1

Finally, we have a look at the efficiency of CC-Pred-Wit and CC-Pred-Gia compared to CC-Pred. We compare in Figure 4.20a the CPU time consumed to compute a correct solution for CC-Pred on the x-axis and CC-Pred-Wit on the y-axis. It turns out that, except for a few outliers, both configurations need in the median the same time to solve a task and thus have comparable efficiency. For CC-Pred-Gia, we see that CC-Pred-Gia takes in the median the 1.4-fold CPU time compared to CC-Pred. As CC-Pred and CC-Pred-Wit have the same efficiency, CC-Pred-Gia has also a lower efficiency compared to CC-Pred-Wit, cf. Figure 4.20b. In CC-Pred-Wit, information from correctness witnesses are joined using a *syntactic* approach, which is fast and, as it is only applied within this setting, expresses the precision increment in a way optimized for C-CEGAR. In contrast, CC-Pred-Gia employs the Combiner, which takes the *semantics* of the two GIAs that are combined into account to guarantee that no information is lost. The resulting GIA is significantly larger (contains more states and edges) and not optimized for C-CEGAR, which is the reason most likely causing the lower efficiency and the increased number of timeouts.

Results

The effectiveness of C-CEGAR reduces by 20% when using witnesses, and the efficiency is not influenced. GIAs can also be applied in C-CEGAR, whereas the effectiveness reduced by 25%, caused by a lower efficiency compared to CC-Pred.

4.4.4 RQ 3: Does Using Off-the-shelf Components Pays Off?

Evaluation Plan Our objective is to show the most important advantage of C-CEGAR, namely that using complementary techniques can lead to increased effectiveness through uniquely solved tasks. First, we analyze how using different feasibility checkers influences the effectiveness. Therefore, we replace within CC-Pred-Wit the feasibility checker using CPAchecker with two off-the-shelf tools, namely FShell-witness2test and UltimateAutomizer. Thereafter, we exchange the precision refiner

within CC-Pred-Wit that computes Craig interpolants for refinement by an instance of UltimateAutomizer that uses Newton refinement to compute precision increments.

Experimental Results We present in the upper half of Table 4.2 the experimental results for using different feasibility checkers. It contains the number of overall correct answers, correct proofs, and correct alarms. In addition, we included the number of uniquely computed proofs and alarms, i.e., these tasks that are only solved using either FShell-witness2test or UltimateAutomizer for the feasibility checking but not using CPAchecker's violation witness validation. Moreover, we include the number of incorrect proofs and alarms. The table shows that C-CEGAR using CPAchecker as feasibility checker produces the best results. For using UltimateAutomizer and FShell-witness2test we observe 29 respectively 64 tasks solved only using the new feasibility checkers.

Finally, we have a look at using different precision refiners that employ conceptually different techniques, namely Craig interpolation in CPAchecker and Newton refinement in UltimateAutomizer. In the lower half of Table 4.2, we present the obtained experimental results. CC-Pred-Wit with UltimateAutomizer as precision refiner is able to find 1 430 proofs (vs. 2 110) and 309 alarms (vs. 744). Although the effectiveness is lower compared to using CPAchecker's precision refiner, the instance is still able to find 29 proofs and 1 alarm that are not found by CC-Pred-Wit with CPAchecker. These results indicate the different precision refiners using conceptually different techniques have different strengths and weaknesses, showing that the easy replacement that is possible due to C-CEGAR can be beneficial. Such an example where a replacement is beneficial is already presented as motivation in Figure 4.2b. Having a closer look at the generated precision increments by UltimateAutomizer, it turns out that it often contains correct but complex predicates, for which the model explorer runs into a timeout.

Results

C-CEGAR allows a simple exchange of feasibility checkers and precision refiners. The use of conceptually different off-the-shelf feasibility checkers or precision refiners can increase the effectiveness of C-CEGAR through uniquely solved tasks.

4.5 Discussion

Our experimental evaluation demonstrates that decomposing the existing CEGAR scheme is possible to make it applicable in a cooperative setting. Moreover, we have shown that we can use our novel exchange format GIA for the information exchange in cooperative settings, especially in C-CEGAR. We can again see that combining different tools in the cooperative approaches combines their strengths and enables these approaches to solve tasks that the non-cooperative tool alone could not solve. Nev-

ertheless, neither a decomposition nor using an exchange format applicable in many scenarios comes for free, i.e., the efficiency reduces, and due to the higher runtime of the decomposed instances or due to information not being exported some tasks can not be solved anymore. We first discuss the validity of our experimental evaluation before discussing the approach itself:

We used for the evaluation 8 347 tasks from the SV-Benchmarks from 2022. Although this dataset is widely used and accepted for benchmarking, our findings may not completely carry over to real-world C programs or other programming languages like Java. As the results from CC-Pred and Pred show a high agreement in the results and CC-Pred does not cause any additional incorrect answers, we are confident that the implementation does not suffer from bugs. Anyhow, such bugs would influence the effectiveness only negatively and our findings would remain valid.

We have additionally shown that GIAs can be used for exchanging information within C-CEGAR. For other cooperative approaches using standardized exchange formats, such as test case generation or CMC, we explained how GIAs can be employed within these concepts and evaluated that GIAs are applicable in cooperative test case generation (cf. Appendices A.2.1 and A.2.6). Thus, the findings from our evaluation will most likely carry over to these other forms of cooperation, meaning that GIAs can be applied in different scenarios as well.

Nevertheless, there are two underlying assumptions when using GIAs for exchanging information: First, we assume that each tool that either takes a GIA as input or produces a GIA works with the original C program or the reduced program generated by the reducer. In case a tool is working on a different representation (e.g., LLVM or Boogie), it has to be ensured that the information generated by the tool is mapped back to the original C program. The concept of mapper and encoder proposed in Section 3.1.4.1 can then be used to map the information back to the level of the C program. Second, we assume that the task solved by the cooperation of tools can be expressed in terms of reachability and thus the information communicated can be expressed in terms of reachability, e.g., the (non)-reachability of certain locations or conditions and state invariants that hold on some or all paths to a certain location or a function. Although test case generation and the verification of certain correctness criteria can be expressed in terms of reachability, there might be other correctness criteria or properties that cannot be expressed, meaning that GIAs might not be applicable as an exchange format. In addition, GIAs allow to express additional information in terms of predicates. Hence, concrete (input) values needed for executing a specific path or information on variable values gathered during analysis, e.g., interval values, need to be transformed into predicates. For example, concrete variable values can be expressed using assignments, and the information $x \in [1, 4]$ is translated to the formula $1 \leq x \leq 4$. In case a combination of analyses is exchanging analysis information that is not representable using predicates (e.g. tainted or unused variables), it is likely that the information cannot be encoded within a GIA (or any other instance of a protocol automaton).

Although there are two assumptions, that might limit the use of GIAs in certain, special cases, all in all, we think that GIAs can encode the information that is typically exchanged between OA and UA tools.

During the evaluation of GIAswe have shown that GIAs can be employed in C-CEGAR. We also observed that GIAs are larger compared to invariant witnesses tailored to the use within C-CEGAR. Due to the theoretical guarantees given and the fact that we can use GIAs in many different scenarios, we observe the decreased efficiency as a downside. This trade-off has to be kept in mind when choosing the formats for information exchange during the design of new forms of cooperative verification.

Information exchange among components is key to success in cooperative verification. Beneath having suitable formats for the information exchange such as witnesses or GIAs, the information computed by the individual components has to be exported. For the abstract model explorer and feasibility checker, it may suffice to only report the potential error paths, although including more precise information in the form of assumptions on variable values may ease the task of the next tool. In contrast, the precision refiner has to encode the precision increment within the invariant witnesses or GIAs. We have already seen in the evaluation that missing information is the main reason for the reduced effectiveness of CC-Pred-Wit compared to CC-Pred. We selected among CPAchecker only the tool UltimateAutomizer as precision refiner, as it is, to the best of our knowledge in 2021 when conducting the experiments, the only formal-verification tool that is able to process violation witnesses as additional input and that also outputs invariant witnesses. In theory, any off-the-shelf verifier can be transformed into a precision refiner using the construction depicted in Figure 4.5. In practice and based on data from SV-COMP'21, no other tool than CPAchecker and UltimateAutomizer was able in 2021 to produce meaningful invariant witnesses. Fortunately, tool developers have improved their tools' output since then, such that the variety of available precision refiners has increased. For example, Symbiotic is now able to also produce non-trivial invariant witnesses [Cha+22]. This opens up the possibility to build more configurations employing more tools that make use of conceptual different techniques within C-CEGAR. These new configurations have the potential to (partially) overcome known limitations and to increase the effectiveness of C-CEGAR.

4.6 Related Work

In this chapter we focused on building a cyclic cooperative verification scheme by decomposing CEGAR and additionally develop and employ a novel verification artifact called GIA, that is applicable in many scenarios combining over-approximating and under-approximating verification and testing tools. In the following, we discuss other (non-cooperative) cyclic software verification approaches. Thereafter, we provide an overview of other existing verification artifacts used in (non-cooperative) approaches.

4.6.1 Cyclic Combinations

In the following, we focus on concepts that combine the strengths of different approaches and exchange information between them in a cyclic manner, either cooperatively or using conceptual integration.

4.6.2 Conceptual Integrations

The idea of a cyclic combination of tools is used in many scenarios, especially to combine the different strengths or general capabilities of tools [GNRT10; Gul+06; NRTT09; BNRS08; AGC12; GHMW16; GD17; Chr+21; NKP18]. In Smash [GNRT10], a predicate analysis is combined with dynamic test generation, wherein both tools compute information in an alternating way. The Smash algorithm maintains two sets of function summaries in the form of predicates and implications. One set contains witnesses for concrete execution paths within the function, whereas the other summaries express certain properties (postconditions) that hold for all executions of the function satisfying certain preconditions. Synergy [Gul+06] (with its implementation in the tool Yogi [NRTT09]) and Dash [BNRS08] share the idea of combining predicate abstraction with a testing approach. Both maintain two separate data structures, an over-approximation of the state space and a tree of concrete program executions. The core idea is to steer testing along potential counterexamples and use information obtained by testing to guide the refinement process. The Ufo algorithm follows a similar idea but stores all information within a single ARG [AGC12]. In [GD17] over- and under-approximative tools are combined to characterize program failures as precisely as possible by providing formulas bounding the input space causing the program failures. In [Chr+21], a neuro-aware program analyzer is presented, that iteratively combines an abstract interpretation tool and an analyzer for neural networks, allowing an analysis of programs that contain calls to neural networks. In [NKP18] a symbolic execution engine and a fuzzer are iteratively combined to discover program paths whose execution is resource-intensive.

The idea of concolic testing is to enrich a testing tool with concrete test inputs that may lead to unexplored parts of the program [SMA05; SA06; TH08; BDMP17; MS07; LEMR20b; LEMR20a; MSSA22; CDE08]. The concrete inputs are computed using a symbolic execution. Daca et al. [DGH16] use a concolic execution engine in combination with predicate abstraction. The predicate abstraction guides the search of the concolic tester by identifying unreachable program parts. Beneath this information, the concolic tester communicates the test goals already covered. Information is exchanged using an ARG. In [ARCB14], dynamic symbolic execution and static symbolic execution are combined in an alternating way, aiming to increase the effectiveness of detecting program violations. In [GHMW16], a concolic tester is combined with BMC, where BMC is used on loop- and recursion-free parts of the program to reduce the number of paths that need to be explored by concolic execution.

All the presented cyclic approaches combine several concepts within a single tool as white-box integration. In contrast, we aim to build a cooperative cyclic approach. Nevertheless, each is a candidate for being decomposed and thereby made ready for being used as a scheme for cooperative verification. For example, we demonstrate in Appendix A.2.1 how to use GIAs for cooperative test case generation, following and generalizing the idea of [DGH16].

4.6.3 Cooperative Approaches and Decomposition

CoVeriTest [BJ19; Jak20; JR21] generalizes the idea of [DGH16] by combining arbitrary verifiers for test case generation. Initially, the set of test goals that need to be covered is computed. Then, each verifier tries to reduce the set of open test goals and generates a condition describing the explored state space, such that other tools can safely ignore it. Hence, the condition is exchanged among the components for cooperation. A similar approach only employing testing tools is presented in [BL19a]. For cooperative test case generation, we explain in Appendix A.2.1 how to realize these concepts using GIAs for the information exchange, allowing to reuse of components built for other purposes, e.g., for C-CEGAR.

The concept of property-directed k-induction [GI17; JD16] is formalized in a cooperative way in [BHMS20]. The formalization comprises two kinds of components. The first component, the induction-checking engine, steers the verification and searches for a k-inductive invariant to show that the program is safe. Within the cooperation, it makes use of the second component, the finite reachability engine, which checks if a certain formula is satisfied in any state reachable in a given, finite number of steps. As an answer, either a concrete trace or an invariant is provided and exchanged. The exchange is realized using no standardized formats. Although the information is exchanged using no standardized formats, we could also employ GIAs, especially for communicating the concrete paths that reach the property or the invariants.

In [Hel+20], Helm et al. present Opal, a framework for building cooperative analyses based on the blackboard approach. Therein, an analysis can request specific analysis information from the blackboard, that is either already computed by another analysis or computed on-demand. The blackboard is orchestrating the collaborating analyses. The analysis information computed is exchanged within Opal using lattice elements. For example, a three-address-code based intermediate representation of Java bytecode is computed in [Rei+20], comprising an abstract interpretation in combination with analyses computing precise information on return types and field types. Opal is providing the infrastructure to build collaborative analyses and can thus be seen as comparable to CoVeriTeam. The collaborative program analyses implemented in Opal are orthogonal to C-CEGAR, as they either decompose existing analyses or build new analyses on existing components. The information exchanged is mostly analysis-dependent and is represented using lattices. Compared to GIAs, where information on the reachability

of program locations or invariants are exchanged, the shared analysis results exchanged in Opal are on a more fine-grain level, like the return type of a method.

4.6.4 Existing Artifacts

Many existing approaches already exchange information using different artifacts. We have a closer look at these existing artifacts used, summarize their characteristics, and discuss to which extent they are suited as general formats for the information exchange between OA and UA tools, based on the requirements stated in Section 4.2.2.

In conceptual integrations, white-box combinations of multiple components, information are exchanged not via clearly defined artifacts but rather using internal formats, method calls, or accessing shared data structures. These approaches may exchange information on concrete program executions and the resulting goals covered, or the unreachability or safety of certain parts (under some boolean conditions) of the program. Combinations exchanging information only on already covered goals are for example FuSeBMC [AABC21; Ald+23], DyTa [GTXT11] or the concept proposed by Huster et al. [Hus+17]. These approaches do not use standardized formats. Examples of tools that combine an OA and an UA component and exchange both types of information are for example Smash [GNRT10], Synergy [Gul+06] (with its implementation in the tool Yogi [NRTT09]), Dash [BNRS08], the Ufo algorithm [AGC12], or the approach presented by Daca et al. [DGH16]. Within some of these approaches [DGH16; AGC12] an *ARG*, cf. Definition 2.6, is used for information exchange. An ARG represents the abstract state space containing the analysis results computed as a graph. In general, the ARG can be used to represent all desired information that should be exchanged. Due to the analysis-dependent information, ARG states generated by different analyses (e.g., by interval analysis, live variable analysis, or predicate abstraction) may, however, have different shapes, which makes an exchange of ARGs between different analyses in a general setting impossible.

In contrast to conceptual integration, cooperative approaches use components as black-boxes, and information is exchanged only using clearly defined verification artifacts. Conditions under which the program is verified are exchanged using condition automata in CMC [BHKW12]. In order to use off-the-shelf tools in CMC, the condition automaton can be transformed into a reduced program [CJW15; BJLW18; BJ20]. To generate test cases cooperatively, CoVeriTest [BJ19; Jak20; JR21] combines arbitrary verifiers, where each tries to reduce the set of open test goals and generates a condition describing the explored state space, such that other tools can safely ignore it. The condition, again represented as a condition automaton, is then used for cooperation. A similar approach only employing testing tools is presented in [BL19a], where in contrast to CoVeriTest conditions are encoded as a set of CFA-edges. A *condition automaton*,

cf. Definition 2.11, states which semantic paths of the program are already successfully verified and under which condition. Although condition automata can mark certain regions as safe, paths (potentially) leading to a node from TN cannot be encoded. In addition, condition automata do not allow adding state invariants. Hence, (R2) and (R5) are, in contrast to GIAs, not fulfilled.

Besides being used as justification for the computed verdicts, witnesses are also used for information exchange in cooperative verification approaches. C-CEGAR exchanges information on potential counterexamples using violation witnesses, on infeasible paths using path witnesses, and on precision increments in the form of new predicates using invariant witnesses. Invariant witnesses are also used in CoVEGI to exchange loop invariants. A *violation witness*, cf. Definition 2.10, encodes a set of feasible program paths where some lead to a property violation. By design, the violation witness does not allow the use of state invariants. Thus, its semantics does neither allow encoding that a path does not reach a node from TN (i.e., is safe) or is infeasible or some justification of this in the form of state invariants. Hence, (R3), (R4), and (R5) are not fulfilled. A *correctness witness*, cf. Definition 2.8, is used to encode that a program is safe (no node from TN is reachable). They do not allow to specify the reachability of nodes from TN nor to encode partial results. Therefore, encoding paths to nodes from TN as well as marking that only certain paths of the program (and not the whole program) are safe is impossible. Hence, (R1) and (R2) are not fulfilled. An *invariant witness* allows for encoding that a part of the program satisfies the specification (R1), but it still does not allow for encoding violating paths (R2). As the Yaml-based format for correctness and violation witnesses does not increase the expressive power of the formats, the same line of argument also holds for these two formats.
Condition automaton, as well as version 1.0 of violation and correctness witnesses, are defined as protocol automaton [BW20]. The *protocol automaton*, cf. Definition 2.7, describes a set of paths, and its semantics is context-depended. Consequently, in contrast to GIAs, it is impossible to mark within one protocol automaton both, a path to a node from TN as unreachable and state that another path reaches a different node from TN. Hence, (R6) and either (R1) or (R3) are not fulfilled.

A completely different form of cooperation is employed in [BSU22], where an automatic verifier is cooperating with an interactive verifier[11]. It allows for an interactive verification using automatic verifiers. The interactive verifier, in this scenario Frama-C, uses inputs by human experts in the form of ANSI/ISO C Specification Language (ACSL) annotations, that are written directly into the program code. Christakis et al. propose concepts to explicitly state the conditions or assumptions under which the program is verified [CMW12; CMW16]. These conditions or assumptions are also added as anno-

[11] An interactive verifier relies on user inputs, e.g., (loop) invariants. Automatic verifiers are called *verifiers* in this thesis, as we do not consider interactive verifiers.

tations directly in the program code. One common option for annotating C programs with behavioral properties like global or loop invariants, function contracts, or (additional) assertions is ACSL [Bau+24]. The idea of ACSL-annotations is to ease the verification by providing helpful information, thus these annotations can be seen as a similar concept to correctness witnesses. The ACSL annotations are not designed for encoding property violations and do not fulfill (R2).

5 Parallel Cooperation

Having discussed methods for building sequential and cyclic cooperative approaches, we next focus on building a cooperative concept that allows for using different actors in parallel to cooperatively solve the verification task. The easiest way of combining tools in parallel is using a parallel portfolio, that is regularly used and has proven to be successful [TFNM11; BL19a; Gro+12; Hol+16; HJG08; YDLW19; CH23; BKR22; Luc+16]. Inspired by economics, the core idea of using a parallel portfolio is to increase the chance of computing the correct solution by employing multiple conceptually different tools or configurations in parallel [HLH97]. More precisely, all selected tools are executed in parallel, where each tries to solve the verification task, and the first result that is computed is returned. A parallel portfolio is easily realizable, as it does not necessitate a concept for splitting the verification task into subtasks. As a downside, it causes a lot of redundant computations, as each tool solves the same verification task. In addition, a parallel portfolio is not cooperative, as information computed by one tool is not shared with the others.

For building a cooperative parallel verification scheme that avoids unnecessary recomputations, we aim to make use of a core principle in computer science, namely the divide-and-conquer approach [CLRS09]. The underlying idea of this approach is to split the task into multiple smaller subtasks, such that each can be solved independently in parallel. The partial results computed for each of the smaller problems are then combined into a solution for the overall task. To be able to employ the divide-and-conquer approach for software verification, we need a way to split the verification task into subtasks in advance. The splitting has to guarantee that subtasks can be solved in parallel and that no parts of the program are left out. As we focus on the (non-)reachability of (error) locations, which is defined based on a single path (cf. Definition 2.2), we need to ensure that each feasible path of a program is contained in at least one subtask.

There exist several ways to split the paths of a program, e.g., using path ranges [SK12], input ranges [Mis+07], or path-prefixes [FSK12; SK20; SK21]. Instead of developing a novel concept as in Chapter 3 or decomposing an existing scheme as in Chapter 4, we generalize an existing concept that is currently applicable for a single technique only, such that we can employ arbitrary off-the-shelf tools. We base our cooperative approach for parallel program verification on the idea of *Ranged Symbolic Execution*, introduced by Siddiqui and Khurshid [SK12]. It enables multiple instances of symbolic execution to solve the verification problem in parallel, where each instance is assigned a *path range* and analyzes only those paths within the range. A path range describes a set of ordered paths using a lower and an upper bound and makes use of an ordering relation for program paths. To ensure that the instances of symbolic execution work on the given interval only, ranged symbolic execution uses the branching conditions internally generated by symbolic execution. Clearly, this concept is not directly applicable to arbitrary off-the-shelf analyses.

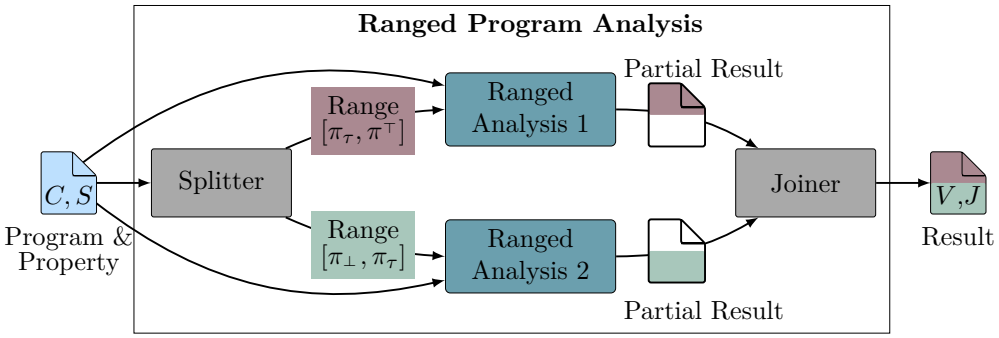


Figure 5.1: Ranged program analysis using two ranged analyses

In this chapter, we aim to generalize the idea of ranged symbolic execution, such that arbitrary off-the-shelf verifiers can be used as actors, cooperatively solving the verification task in parallel. The main challenge thereby is to restrict the analyses to a given path range. Our concept of *ranged program analysis*, depicted in Figure 5.1, generalizes ranged symbolic execution and allows to use off-the-shelf tools. It comprises three steps: First, given the program and the property, a *splitter* is used to generate the path ranges. Each range is given to an actor called *ranged analysis*, which computes a result for the given range, and thus a partial result for the overall verification task. Internally, each ranged analysis utilizes an off-the-shelf verifier for solving the verification task on the given range. To restrict the verifiers to the given range, we develop two different methods, one based on the composition of Configurable Program Analyses, called *range reduction*, and the other using program instrumentation. As each range can be analyzed separately, the ranged analyses are executed in parallel. Finally, the partial results computed for each range are aggregated by a *joiner* to a final result for the overall verification task. The joiner does not only generate a verdict but also computes an aggregated justification in the form of a violation or a correctness witness for the verification task.

After motivating ranged program analysis by exemplifying the advantages in Section 5.1, we formally define path ranges and an ordering on paths in Section 5.2. We explain splitter and joiner as well as both methods for building ranged analyses in Section 5.3. In addition, we introduce work stealing in ranged program analysis, a concept to increase its effectiveness. Next, we describe the implementation of ranged program analysis in Section 5.4 and evaluate the proposed concept in Section 5.5. We conclude the chapter by discussing the results in Section 5.6 and present related work on parallel cooperative approaches in Section 5.7.

5.1 Motivating Example

Each verification approach has individual strengths and weaknesses. Some are good at finding property violations, and others compute precise loop invariants to prove that the specification is not violated. Especially for large tasks or those containing complex program constructs, dividing the verification task into easier solvable or smaller subtasks can increase overall performance.

Sharing the work for a program containing a loop unrolled hundreds of thousands of times when using BMC can be beneficial, as the path formulas computed for each path range may be smaller and allow the underlying solver to compute a solution faster. In addition, the ranged program analysis using a divide-and-conquer approach allows for combining the different strengths of analyses on one program: When analyzing the running example shown in Figure 2.3, employing a combination of BMC, that is good and fast at detecting property violations and predicate abstraction, that can prove programs correct, can increase the verifier's performance. In case the property would be violated, BMC working only on a bounded part of the program may find a counterexample fast. In contrast, in (the actual) case that the program is correct, BMC shows that the bounded part it works on is free of errors and predicate abstraction can compute a proof for the remaining part using its abstraction technique.

5.2 Exchange Formats in Ranged Program Analysis

The information exchanged within the default configuration of ranged program analysis, as depicted in Figure 5.1, comprises the path ranges and the partial results[1]. For the latter, we can reuse verdicts and witnesses. In case the range contains a property violation, a violation witness is computed. Otherwise, an invariant witness showing the correctness of all program paths in the range and thus of a part of the program is generated.

Our goal is to define path ranges as a set of "consecutive" paths, such that it can be guaranteed that all program paths are analyzed. To compare two paths, we need an

[1]Work stealing, introduced in Section 5.3.6 allows for reusing the analysis precision or previously computed invariants.

ordering $\leq$ on program paths based on their edges. Intuitively, the edge representing the *true*-evaluation of an assume statement is smaller than the edge representing the *false*-evaluation.

Definition 5.1. *Given two program paths* $\pi = \langle \ell_0, \sigma_0 \rangle \xrightarrow{g_1} \langle \ell_1, \sigma_1 \rangle \xrightarrow{g_2} \ldots \xrightarrow{g_n} \langle \ell_n, \sigma_n \rangle \in paths(C)$ *and* $\pi' = \langle \ell_0, \sigma'_0 \rangle \xrightarrow{g'_1} \langle \ell'_1, \sigma'_1 \rangle \xrightarrow{g'_2} \ldots \xrightarrow{g'_m} \langle \ell'_m, \sigma'_m \rangle \in paths(C)$. *We define the path ordering* $\leq$ *as follows:*

$$\pi \leq \pi' \Leftrightarrow \exists\ 0 \leq k \leq n : \forall\ 1 \leq i \leq k : g_i = g'_i \wedge \\ ((n = k \wedge m \geq n) \vee (m > k \wedge n > k \wedge B_C(g_{k+1}) = T \wedge B_C(g'_{k+1}) = F)).$$

Note that the path order $\leq$ is a total ordering, as each feasible execution path of a CFA C starts in ℓ_0. Based on the ordering $\leq$, we can now specify path ranges:

Definition 5.2. *A* path range *(or in short* range*)* $[\pi, \pi']$ *is the ordered set of paths* $\{\pi_r \in paths(C) \mid \pi \leq \pi_r \leq \pi' \ \vee\ \pi_r$ *is a prefix of* π *or* $\pi'\}$.

We also define $\pi_\bot, \pi^\top \notin paths(C)$ as the lowest and greatest path to easily describe ranges that are not bound on the left or right. More formally, $(\pi \leq \pi^\top) \wedge (\pi^\top \not\leq \pi) \wedge (\pi_\bot \leq \pi) \wedge (\pi \not\leq \pi_\bot)$ for all $\pi \in paths(C)$. Consequently, $[\pi_\bot, \pi^\top] = paths(C)$.

Instead of describing paths directly as a sequence of pairs of program state and location, we use the following two, more compact representations, namely via test cases and sequences of branching decisions. We define a *test case* τ, $\tau : Var \rightharpoonup \mathbb{Z}$, that maps each input variable to a concrete value. As formalized in Figure 2.2, the programs analyzed are deterministic except for the inputs. Thus, τ describes exactly a single path, we say that τ *induces* π and write this path as π_τ[2]. For example, $\tau_1 = \{input \mapsto -2\}$ induces π_{τ_1} in the running example C_1 from Figure 2.3:

$$\begin{aligned}
\pi_{\tau_1} = (\ell_0, \{input \mapsto -2\}) &\xrightarrow{\texttt{input<0}} && (\ell_1,\ \{input \mapsto -2\}) \\
&\xrightarrow{\texttt{input=-input}} && (\ell_3,\ \{input \mapsto 2\}) \\
&\xrightarrow{\texttt{rem=input}} && (\ell_4,\ \{input \mapsto 2, rem \mapsto 2\}) \\
&\xrightarrow{\texttt{res=0}} && (\ell_5,\ \{input \mapsto 2, rem \mapsto 2, res \mapsto 0\}) \\
&\xrightarrow{\texttt{rem>1}} && (\ell_6,\ \{input \mapsto 2, rem \mapsto 2, res \mapsto 0\}) \\
&\xrightarrow{\texttt{rem-=2}} && (\ell_7,\ \{input \mapsto 2, rem \mapsto 0, res \mapsto 0\}) \\
&\xrightarrow{\texttt{res++}} && (\ell_5,\ \{input \mapsto 2, rem \mapsto 0, res \mapsto 1\}) \\
&\xrightarrow{\texttt{!(rem>1)}} && (\ell_9,\ \{input \mapsto 2, rem \mapsto 0, res \mapsto 1\}) \\
&\xrightarrow{\texttt{!(input!=2*res+rem)}} && (\ell_{11},\ \{input \mapsto 2, rem \mapsto 0, res \mapsto 1\}) \\
&\xrightarrow{\texttt{return res}} && (\ell_{12},\ \{input \mapsto 2, rem \mapsto 0, res \mapsto 1\})
\end{aligned}$$

[2]More concretely, a test case τ describes a single maximal path and all its prefixes. The case when τ defines only a part of the input is discussed later.

As a consequence, given two test cases τ_1 and τ_2, we can define a range by their induced paths $[\pi_{\tau_1}, \pi_{\tau_2}]$. In case that $\pi_{\tau_1} \not\leq \pi_{\tau_2}$ holds, the range is empty. For the running example in Figure 2.3 and the two test cases $\tau_1 = \{input \mapsto -2\}$ and $\tau_2 = \{input \mapsto 2\}$, the range $[\pi_{\tau_1}, \pi_{\tau_2}]$ contains five (feasible) maximal program paths, namely those where the input variable *input* is in the interval $[-2, 2]$ and that contain at most one loop iteration.

Test cases offer an extremely compact way of representing paths. As a downside, computing the induced path for a test case requires either a program execution or a semantic analysis of the program. A second way to represent program paths to ease computing locations contained on a path is using a *sequence of branching decisions* s_τ. For an induced path π_τ, s_τ contains the branching decisions taken on the path. We can compute s_τ using the recursive function $\mathcal{T}_C : paths(C) \to \{T, F\}^*$:

$$\mathcal{T}_C(\ell_i \xrightarrow{g_i} \ell_j \xrightarrow{g_j} \cdots) = \begin{cases} () & \text{if } \ell_j \text{ has no successor} \\ x \circ \mathcal{T}_C(\ell_j \xrightarrow{g_j} \cdots) & \text{if } x = B_C(g_i) \in \{T, F\} \\ \mathcal{T}_C(\ell_j \xrightarrow{g_j} \cdots) & \text{otherwise} \end{cases}$$

For example, for π_{τ_1} and π_{τ_2} induced in C_1 by $\tau_1 = \{input \mapsto -2\}$ and by $\tau_2 = \{input \mapsto 2\}$, we have $\mathcal{T}_{C_1}(\pi_{\tau_1}) = s_{\tau_1} = (T, T, F, F)$ and $\mathcal{T}_{C_1}(\pi_{\tau_2}) = s_{\tau_2} = (F, T, F, F)$.

Using the defined ordering $\leq$ on the paths, we can build an *execution tree*. It represents for each path of the program the sequence of branching decisions taken. Thus, its nodes are labeled with the boolean conditions inside assume statements present in the program and its leafs with `abort`- and `return`- statements. A solid edge represents the *true*-evaluation of the condition ($B_C(b) = T$), a dashed edge the *false*-evaluation ($B_C(b) = F$) for a condition b. We depict in Figure 5.2 a (partial) execution tree for the running example shown in Figure 2.3. Note that an execution tree does not take the program semantics into account, hence it (can) contain infeasible program paths.

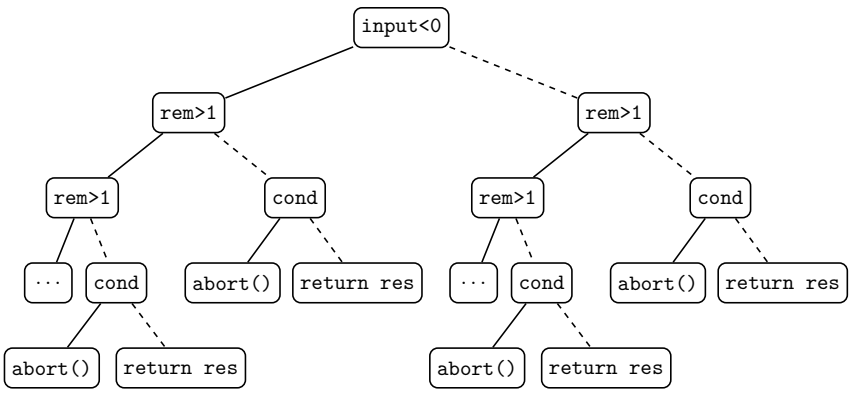


Figure 5.2: Partial program execution tree for the running example C_1 from Figure 2.3. We define `cond`≡`input!=2*res+rem`.

5.3 Components in Ranged Program Analysis

The goal of ranged program analysis is now to divide the program into ranges along program paths, where each range is analyzed by a ranged analysis, as depicted in Figure 5.1. When using two ranged analyses, the splitter generates one path π that forms the two ranges $[\pi_\bot, \pi]$ and $[\pi, \pi^\top]$. The number of ranges needed depends thereby on the number of ranged analyses used. For example, if three ranged analyses are used, two paths π_1, π_2 are generated, forming the ranges $[\pi_\bot, \pi_1]$, $[\pi_1, \pi_2]$, and $[\pi_2, \pi^\top]$. To exemplify the concrete paths that need to be analyzed by a ranged analysis, we illustrate in Figure 5.3 the range $[\pi_1, \pi_2]$ for the running example. All elements that are not in the range are depicted in gray, the two paths π_1 and π_2 are highlighted in green resp. red. We assume that π_1 is induced by $\tau_1 = \{input \mapsto -2\}$ $(\pi_1 = \pi_{\tau_1})$ and π_2 is induced by $\tau_2 = \{input \mapsto 2\}$ $(\pi_2 = \pi_{\tau_2})$.

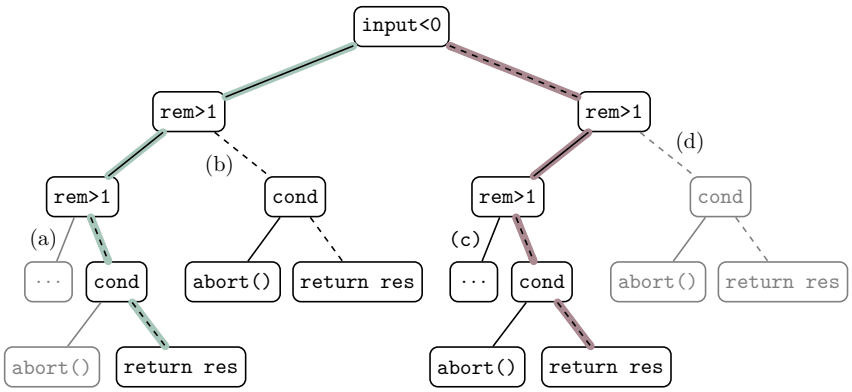


Figure 5.3: Program execution tree for running example from Figure 2.3 with the interval $[\pi_{\tau_1}, \pi_{\tau_2}]$, where π_{τ_1} is highlighted in green and π_{τ_2} in red. Elements that are not in the range are depicted in gray. We define `cond`≡`input!=2*res+rem`.

The ranged analysis that is working on the interval $[\pi_1, \pi_2]$ has to analyze all paths that are executed when the input is positive and that contain at least one loop iteration and the two paths for negative inputs with at most one loop iteration. Next, we introduce all components present in ranged program analysis, starting with the splitter.

5.3.1 Splitter

Within ranged analysis, ranges can be defined either using test cases or sequences of branching decisions. We develop four different splitting strategies, two are based on the number of loop unrollings, and two use a random path exploration. Each strategy builds the program execution tree demand-driven, i.e., explores only these paths that are returned. Then, for each path either the sequence of branching decisions or a test case that induces the path is computed.

Splitter Based on Loop Bounds The goal of these splitters is to generate a range that contains paths with a finite number of loop iterations. We propose two splitters,

Lb3 and Lb10. More precisely, Lb3 computes the left-most path in the execution tree that contains exactly three unrollings of each loop, analogously for Lb10 with ten loop iterations. If the program contains nested loops, each nested loop is unrolled three times in each iteration of the outer loop. To generate the test case for the computed path, we (1) build its path formula using the strongest postcondition operator (cf. Definition 2.3), (2) use an SMT-solver to check the formula for satisfiability and (3) in case it is satisfiable, use the evaluation of the input variables provided by the SMT-solver in the path formula as one test case. In case the path formula is unsatisfiable, we shorten the path by iteratively removing the last statement from it, until we get a satisfying path formula. For loop-free programs, we cannot bind the number of loop unrollings. Hence, Lb3 and Lb10 fail and we generate a single range $[\pi_\bot, \pi^\top]$.

Splitter Based on Random Exploration The other two strategies Rdm and Rdm9 select paths randomly by traversing the CFA and deciding at each assume statement to either follow the *true*- or the *false*-branch. The probabilities of Rdm for selecting the *true*- or *false*-branch are 50%, whereas Rdm9 selects the *true*-branch with a 90% probability. As execution trees of programs with loops are often not balanced but rather grow to the left, Rdm9 likely generates longer paths.

5.3.2 Ranged Analyses for Off-the-shelf Analyses in General

The task of the actor *ranged analysis* is to analyze at least all paths that are contained in a given range. The result computed for the range is thereby a partial result for the overall task. We depicted the inputs and outputs of a ranged analysis in Figure 5.4. As input, it is given a program and the specification as well as the range in the form of two test cases or as two sequences of branching decisions. It then returns a verdict and a justification, that are valid for the given range.

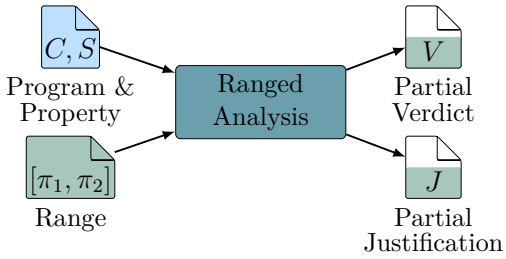


Figure 5.4: Conceptual view of a ranged analysis. Verdict and justification are colored half to indicate that they are partial results for the given range.

As we aim to employ off-the-shelf analyses within ranged program analysis, we need a way to guide them to only analyze paths within the given range, as by default no off-the-shelf analysis supports ranges as input. We present two different methods for achieving the goal, the first working for CPA-based analyses using a so-called range reduction, the other using program instrumentation, usable for arbitrary analyses. Both

methods make use of the same underlying idea. The idea is based on the following three observations, that we make for the range $[\pi_1, \pi_2]$.

Observation 1. The range is a closed interval, thus the two paths π_1 and π_2 are also included. Hence, each prefix of either π_1 or π_2 may be included in the interval. Thus, *we need to track if the current paths is a prefix of the bounds.*

Observation 2. The ordering of two elements can be decided based on local decisions. Assume we have a path $\pi = \langle \ell_0, \sigma_0 \rangle \xrightarrow{g_1} \ldots \xrightarrow{g_i} \langle \ell_i, \sigma_i \rangle$, where ℓ_i represents an assume statement, meaning that there are two successors (ℓ_i, g_j, ℓ_j) and (ℓ_i, g_k, ℓ_k), where $B(g_j) = T$ and $B(g_k) = F$. If we now construct the two paths $\pi_j = \pi \xrightarrow{g_j} \langle \ell_j, \sigma_j \rangle$ and $\pi_k = \pi \xrightarrow{g_k} \langle \ell_k, \sigma_k \rangle$, we have $\pi_j \leq \pi_k$. More importantly, the ordering is the same for all continuations of π_j and π_k.

Observation 3. As only local branching decisions matter, *inclusion in the range for a path π is decidable early.* In case that $\pi_1 \not\leq \pi$ or $\pi \not\leq \pi_2$ holds, we can stop the exploration of π, as π and any continuation is not included in $[\pi_1, \pi_2]$. Moreover, we know that if $\pi \not\leq \pi_1$ and $\pi_2 \not\leq \pi$ holds, any continuation of π is included in the interval $[\pi_1, \pi_2]$.

Within the ranged analysis, we restrict the exploration of the off-the-shelf analyses either using a range reduction or program instrumentation. We now exemplify the underlying idea using Figure 5.3. Due to Observation 1, both techniques monitor if the current branch that is analyzed by the off-the-shelf analysis is a prefix of either the lower bound π_1 or the upper bound π_2. In case the exploration would "leave π_1 to the left" (the continuation is strictly smaller with respect to $\leq$, e.g., the edge marked (a)) or "leave π_2 to the right" (the continuation is strictly greater with respect to $\leq$, e.g., the edge marked (d)), the exploration can safely be aborted (Observation 3). If the explored path is strictly greater than π_1 and strictly smaller than π_2, all continuations are in the interval and thus the monitoring is not necessary anymore. In Figure 5.3 two edges leading to such a situation are marked with (b) and (c).

5.3.3 CPA-based Ranged Analyses

The first method on how to build ranged analyses is using a so-called *range reduction* and is based on the composition of CPAs. Thus, it works for analyses that are defined as CPAs, for example, all analyses realized within CPAchecker.

The idea, depicted in Figure 5.5 is to make use of the three observations presented in Section 5.3.2 within the range reduction analysis $\mathbb{R}$ and then compose it with the actual analysis $\mathbb{A}$. The range reduction then ensures that only those paths within the range are analyzed and stops the exploration of the composed analysis for all other paths. It takes as input two paths, i.e., two test cases that induce the lower and upper bound of the interval. Before explaining how the range reduction works, we next present the necessary fundamentals on CPAs, based on [BHT07].

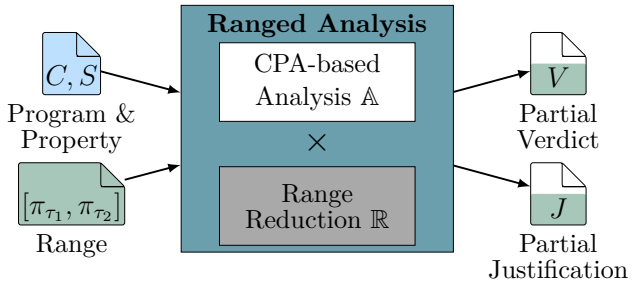


Figure 5.5: Conceptual overview of ranged analysis using range reduction

5.3.3.1 CPA-based Analyses in General

The CPA-framework allows to define analyses based on abstract interpretation, abstract domains, and transfer function. The analysis information is computed using the CPA-algorithm, presented in [BHT07]. The algorithm requires in addition a function for combining analysis information and a function indicating when to stop the exploration. Formally, a *CPA* $\mathbb{A} = (D, \rightsquigarrow, \mathit{merge}, \mathit{stop})$ consists of

- the *abstract domain* $D = (\mathit{Loc} \times \Sigma, (E, \top, \sqsubseteq, \sqcup), [\![\cdot]\!])$, composed of a set $\mathit{Loc} \times \Sigma$ of program states, a join semilattice on the abstract states E as well as a concretization function $[\![\cdot]\!]$ fulfilling

$$\forall e, e' \in E : [\![e]\!] \cup [\![e']\!] \subseteq [\![e \sqcup e']\!] \text{ and } [\![\top]\!] = \mathit{Loc} \times \Sigma$$

- the *transfer relation* $\rightsquigarrow \subseteq E \times G \times E$ defining the abstract semantics that safely over-approximates the program semantics, i.e.,

$$\forall e \in E,\ g \in \mathit{Loc} \times \mathit{Ops} \times \mathit{Loc} :$$
$$\{\langle \ell', \sigma' \rangle \mid \exists \text{ valid execution step } \langle \ell, \sigma \rangle \xrightarrow{g} \langle \ell', \sigma' \rangle : \langle \ell, \sigma \rangle \in [\![e]\!]\} \subseteq \bigcup_{(e,g,e') \in \rightsquigarrow} [\![e']\!]$$

- the *merge operator* $\mathit{merge} : E \times E \to E$ used to combine information satisfying

$$\forall e, e' \in E : e' \sqsubseteq \mathit{merge}(e, e')$$

- the *termination check* $\mathit{stop} : E \times 2^E \to \mathbb{B}$ deciding whether the exploration of an abstract state can be omitted and fulfilling

$$\forall e \in E, E_{\text{sub}} \subseteq E : \mathit{stop}(e, E_{\text{sub}}) \implies [\![e]\!] \subseteq \bigcup_{e' \in E_{\text{sub}}} [\![e']\!]$$

In addition, an initial value $e_{init} \in E$ the analysis is started with is needed.

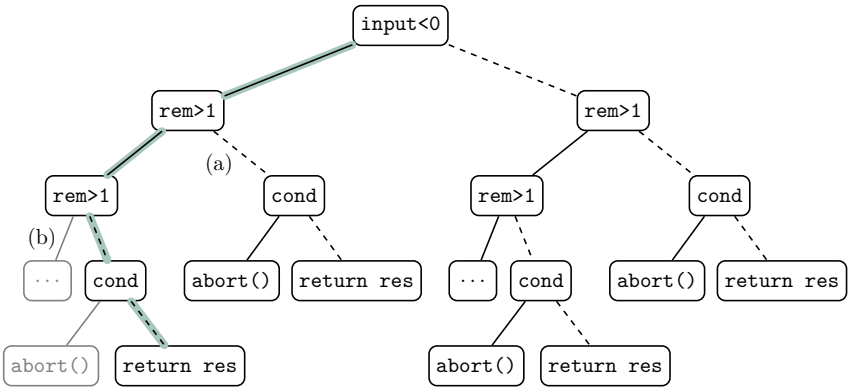
Value Analysis as CPA The range reduction that we develop gets two test cases as input and tracks the induced paths directly. Therefore, we make use of components already defined for the value analysis presented in Section 2.2.2. As we work with CPA-based analyses, we next formalize the CPA for value analysis $\mathbb{V}$ [BHT08]: An abstract state v of the value analysis maps each variable to either a concrete value of its domain or $\top$, representing any value. The partial order $\sqsubseteq_{\mathbb{V}}$ and the join operator $\sqcup_{\mathbb{V}}$ are defined variable-wise while ensuring that $v \sqsubseteq_{\mathbb{V}} v' \Leftrightarrow \forall x \in \mathit{Var} : v(x) = v'(x) \vee v'(x) = \top$[3] and $(v \sqcup_{\mathbb{V}} v')(x) = v(x)$ if $v(x) = v'(x)$ and otherwise $(v \sqcup_{\mathbb{V}} v')(x) = \top$. The concretization of abstract state v contains all program states that agree on the concrete variable values, i.e., $[\![v]\!]_{\mathbb{V}} := \{\langle \ell, \sigma \rangle \in \mathit{Loc} \times \Sigma \mid \forall x \in \mathit{Var} : v(x) = \top \vee v(x) = \sigma(x)\}$. If the values for all relevant variables are known, the transfer relation $\rightsquigarrow_{\mathbb{V}}$ will behave like the operational semantics defined in Figure 2.4. Otherwise, an over-approximation may be computed using $\top$ for some variables, in case a concrete value cannot be determined. If the successor state of an assume statement is unreachable, the transfer relation returns $\bot_{\mathbb{V}}$. The merge operator $\mathit{merge}_{\mathbb{V}}$ combines the elements using $\sqcup_{\mathbb{V}}$ and $\mathit{stop}_{\mathbb{V}}(e, E_{\text{sub}}) = \exists e' \in E_{\text{sub}} : e \sqsubseteq_{\mathbb{V}} e'$.

Composite CPA The main advantage of CPAs is the possibility of composing them to a new analysis [BHT07]. We will also make use of a composition and thus formalize the *composite CPA* next. A composite CPA $\mathbb{A}_\times = (D_\times, \rightsquigarrow_\times, \mathit{merge}_\times, \mathit{stop}_\times)$ of CPA $\mathbb{A}_1 = ((\mathit{Loc} \times \Sigma, (E_1, \top_1, \sqsubseteq_1, \sqcup_1), [\![\cdot]\!]_1), \rightsquigarrow_1, \mathit{merge}_1, \mathit{stop}_1)$ and CPA $\mathbb{A}_2 = ((\mathit{Loc} \times \Sigma, (E_2, \top_2, \sqsubseteq_2, \sqcup_2), [\![\cdot]\!]_2), \rightsquigarrow_2, \mathit{merge}_2, \mathit{stop}_2)$ is working on the product domain $D_\times = (\mathit{Loc} \times \Sigma, (E_1 \times E_2, (\top_1, \top_2), \sqsubseteq_\times, \sqcup_\times), [\![\cdot]\!]_\times)$, that consists of pairs of abstract states. It defines $(e_1, e_2) \sqsubseteq_\times (e_1', e_2')$ if $e_1 \sqsubseteq_1 e_1'$ and $e_2 \sqsubseteq_2 e_2'$, $(e_1, e_1) \sqcup_\times (e_1', e_2') = (e_1 \sqcup_1 e_1', e_2 \sqcup e_2')$, and $[\![(e_1, e_2)]\!] = [\![e_1]\!]_1 \cap [\![e_2]\!]_2$. The transfer relation can be composed using $\rightsquigarrow_1$ and $\rightsquigarrow_2$. Similarly, we define $\mathit{merge}_\times$ and $\mathit{stop}_\times$ element-wise, i.e., $\mathit{merge}_\times((e_1, e_2), (e_1', e_2')) = (\mathit{merge}_1(e_1, e_1'), \mathit{merge}_2(e_2, e_2'))$ and $\mathit{stop}_\times(e, E_{\text{sub}}) = \exists e' \in E_{\text{sub}} : e \sqsubseteq_\times e'$.

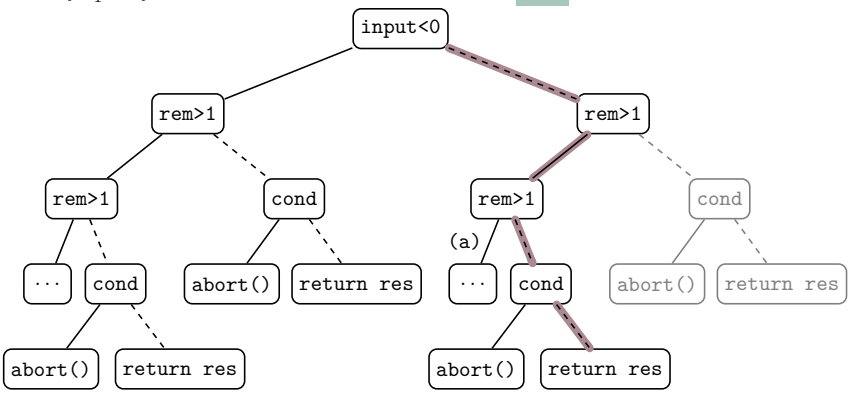
5.3.3.2 Ranged Analysis using Range Reduction

To ensure that an arbitrary program analysis $\mathbb{A}$ formalized as CPA only explores all paths in a given range, we compose it with the newly defined *range reduction* $\mathbb{R}$. Following from the three observations from Section 5.3.2, the range reduction has to track for an interval $[\pi_{\tau_1}, \pi_{\tau_2}]$ whether the path that is currently explored is a prefix of either π_{τ_1}or π_{τ_2}. As the handling of paths that are strictly smaller or greater than the bound is different for the lower and the upper bound, we decompose the range reduction for $[\pi_{\tau_1}, \pi_{\tau_2}]$ into a composition of two specialized CPA analyses, namely the *lower bound CPA* $\mathbb{R}_{[\pi_{\tau_1}, \pi^\top]}$ and the *upper bound CPA* $\mathbb{R}_{[\pi_\bot, \pi_{\tau_2}]}$. Each of them decides whether a path is in the range $[\pi_{\tau_1}, \pi^\top]$ and $[\pi_\bot, \pi_{\tau_2}]$, respectively. Due to the ordering function

[3] Consequently, $\forall x \in \mathit{Var} : \top_{\mathbb{V}}(x) = \top$.



(a) $\mathbb{R}_{[\pi_{\tau_1},\pi^\top]}$ for $\tau_1 = \{input \mapsto -2\}$, where π_{τ_1} is highlighted in green



(b) $\mathbb{R}_{[\pi_\perp,\pi_{\tau_2}]}$ for $\tau_2 = \{input \mapsto 2\}$, where π_{τ_2} is highlighted in red

Figure 5.6: Visualization of $\mathbb{R}_{[\pi_\perp,\pi_{\tau_2}]}$ and $\mathbb{R}_{[\pi_{\tau_1},\pi^\top]}$ for the example of Figure 2.3. The intersection of both ranges is explored by $\mathbb{R}$. We define `cond`≡`input!=2*res+rem`.

$\leq$, we know that $[\pi_{\tau_1}, \pi_{\tau_2}] = [\pi_{\tau_1}, \pi^\top] \cap [\pi_\perp, \pi_{\tau_2}]$ holds. As the composition $\mathbb{R}_{[\pi_{\tau_1},\pi_{\tau_2}]}$ of $\mathbb{R}_{[\pi_\perp,\pi_{\tau_2}]}$ and $\mathbb{R}_{[\pi_{\tau_1},\pi^\top]}$ ($\mathbb{R}_{[\pi_{\tau_1},\pi_{\tau_2}]} = \mathbb{R}_{[\pi_\perp,\pi_{\tau_2}]} \times \mathbb{R}_{[\pi_{\tau_1},\pi^\top]}$) stops the exploration of a path if one of the two composed analyses returns $\perp$ (i.e., denotes a state as unreachable), $\mathbb{R}_{[\pi_{\tau_1},\pi_{\tau_2}]}$ only explores paths that are included in both ranges and thus only paths in $[\pi_{\tau_1}, \pi_{\tau_2}]$.

We visualize in Figure 5.6 how the two range reductions work on the example. We depict the paths that are explored by $\mathbb{R}_{[\pi_\perp,\pi_{\tau_2}]}$ in Figure 5.6b and in Figure 5.6a the paths explored by $\mathbb{R}_{[\pi_{\tau_1},\pi^\top]}$. The composition of both analyses thus explores these paths that are explored by both analyses, i.e. the paths shown in Figure 5.3.
Formally, we construct the composition for a given range $[\pi_{\tau_1}, \pi_{\tau_2}]$ using range reduction to transform an arbitrary program analysis $\mathbb{A}$ into a ranged analysis, as follows:

$$\mathbb{R}_{[\pi_{\tau_1},\pi^\top]} \times \mathbb{R}_{[\pi_\perp,\pi_{\tau_2}]} \times \mathbb{A}$$

For the composition, we define $stop_\times$ and $merge_\times$ component-wise for the individual merge operators as explained in Section 5.3.3.1.

Both $\mathbb{R}_{[\pi_{\tau_1},\pi^\top]}$ and $\mathbb{R}_{[\pi_\perp,\pi_{\tau_2}]}$ reuse components of the value analysis $\mathbb{V}$, introduced in Section 5.3.3.1. In case all variable values are constant, the program behavior is

deterministic and the value analysis explores the only feasible path. We exploit this behavior for the range reduction analyses by initializing the variable values using the given test case:

$$e_{init} = \begin{cases} v(x) = \tau(x) & \text{if } x \in dom(\tau), x \in Var \\ v(x) = \top & \text{otherwise} \end{cases}$$

As the program behavior is completely determined by the test case, the value analysis follows π_{τ_1} respectively π_{τ_2}. We handle cases where functions returning random values (i.e. `nondet()`) or user inputs are used to model randomness analogously, by extracting the values on demand from the test case[4]. As we are interested in not only exploring the path induced by the test case but rather all paths from the intervals $[\pi_{\tau_1}, \pi^\top]$ respectively $[\pi_\bot, \pi_{\tau_2}]$, we next modify the transfer function of the value analysis for $\mathbb{R}_{[\pi_{\tau_1}, \pi^\top]}$ and $\mathbb{R}_{[\pi_\bot, \pi_{\tau_2}]}$ and define the two analyses formally:

Lower Bound CPA For the CPA range reduction $\mathbb{R}_{[\pi_{\tau_1}, \pi^\top]}$, we borrow all components of the value analysis except for the transfer relation $\rightsquigarrow_{\tau_1}$. The transfer relation $\rightsquigarrow_{\tau_1}$ is defined as follows:

$$(v, g, v') \in \rightsquigarrow_{\tau_1} \text{ iff } \begin{cases} v = \top \wedge v' = \top, \text{or} & (5.1) \\ v \neq \top \wedge \ v' = \top \wedge B_C(g) = F \wedge (v, g, \bot) \in \rightsquigarrow_{\mathbb{V}}, \text{or} & (5.2) \\ v \neq \top \wedge (v' \neq \bot \vee B_C(g) \neq F) \wedge (v, g, v') \in \rightsquigarrow_{\mathbb{V}} & (5.3) \end{cases}$$

Note that we use $\top$ for states on a path that is definitely included in the range and $\bot$ represents an unreachable state in the value analysis, which stops the exploration of the path. Hence, (5.2) ensures that $\mathbb{R}_{[\pi_{\tau_1}, \pi^\top]}$ also visits the *false*-branch of a condition when the path induced by τ_1 follows the *true*-branch, e.g. at the edge in Figure 5.6a labeled with (a). Thus, $\mathbb{R}_{[\pi_{\tau_1}, \pi^\top]}$ visits all paths π with $\pi_{\tau_1} \leq \pi$. Note that in case that $\rightsquigarrow_{\mathbb{V}}$ computes $\bot$ as a successor state for an assumption g with $B_C(g) = T$, the exploration of the path is stopped, as π_{τ_1} follows the *false*-branch. This is contained in (5.3) and can be observed in Figure 5.6a at the edge labeled with (b).

Upper Bound CPA For the CPA range reduction $\mathbb{R}_{[\pi_\bot, \pi_{\tau_2}]}$, we again borrow all components of the value analysis except for the transfer relation $\rightsquigarrow_{\tau_2}$. The transfer relation $\rightsquigarrow_{\tau_2}$ is defined as follows:

$$(v, g, v') \in \rightsquigarrow_{\tau_2} \text{ iff } \begin{cases} v = \top \wedge v' = \top & (5.4) \\ v \neq \top \wedge v' = \top \wedge B_C(g) = T \wedge (v, g, \bot) \in \rightsquigarrow_{\mathbb{V}} & (5.5) \\ v \neq \top \wedge (v' \neq \bot \vee B_C(g) \neq T) \wedge (v, g, v') \in \rightsquigarrow_{\mathbb{V}} & (5.6) \end{cases}$$

(5.5) now ensures that $\mathbb{R}_{[\pi_\bot, \pi_{\tau_2}]}$ also visits the *true*-branch of a condition when π_{τ_2}

[4]We assume that the test case contains the necessary number of return values to define a single program path. We discuss underspecified test cases, i.e. test cases that contain too few values in Section 5.3.3.3.

follows the *false*-branch, cf. for example the edge labeled (a) in Figure 5.6b. (5.5) is also symmetric to (5.3) and (5.4) identical to (5.1).

5.3.3.3 Handling Underspecified Test Cases

So far, we have assumed that test cases are fully specified, i.e. contain values for all input variables, and the behavior of the program is deterministic such that executing a test case τ follows a single (maximal) execution path π_τ. However, in practice, we observe that test cases can be underspecified such that a test case τ does not provide concrete values for all input variables. We denote by P_τ the *set* of all paths that are then induced by τ. In this case, we define:

$$[\pi_\bot, P_\tau] = \{\pi \mid \forall \pi' \in P_\tau : \pi \leq \pi'\} = \{\pi \mid \pi \leq \min(P_\tau)\}$$

and

$$[P_\tau, \pi^\top] = \{\pi \mid \exists \pi' \in P_\tau : \pi' \leq \pi\} = \{\pi \mid \min(P_\tau) \leq \pi\}$$

By defining $\pi_\tau = \min(P_\tau)$ for an underspecified test case τ we can handle the range as if π_τ would be fully specified.

5.3.4 Instrumentation-Based Ranged Analysis

Next, we propose a method for encoding the ranges directly into the program using instrumentation and obtaining a so-called *range program* instead of using a range reduction analysis. The main advantage of using range programs is that they are valid C programs, hence any off-the-shelf verifier can analyze them directly. Thus, instrumentation overcomes the limitation of the range reduction analysis, which works only in combination with other CPA-based analyses. The core idea of program instrumentation is to exclude the paths that are out of the given range. Therefore, we add additional constraints, such that these paths become infeasible. We again make use of the three assumptions stated in Section 5.3.2 to instrument relevant assume statements (such as loops and branches) in the program by adding additional range constraints. Instead of test cases we use sequences of branching decisions for instrumenting the program. Sequences of branching decisions only require a syntactic processing of the program compared to test cases, which require taking the program semantics into account. Following the stated observations, it is sufficient to decide whether the current execution path performs the same branching decisions as the path for the lower or upper bound. We depict the general construction of ranged program analysis using instrumentation and off-the-shelf tools in Figure 5.7. Given two sequences of branching decisions, the range instrumentation computes the *range program*. As it is a valid program, it is given directly to the off-the-shelf verifier. As paths not contained in the given range are unreachable in the range program, the computed results, i.e. the verdict and justification, are only valid for the interval and thus partial results. To achieve that those paths not contained in the range are unreachable in the program, we instrument the code

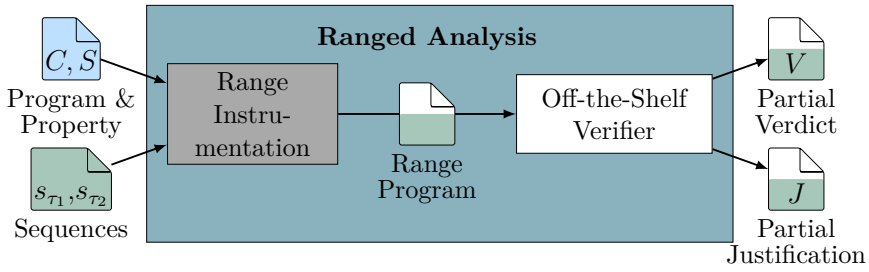


Figure 5.7: Construction of a ranged analysis from an off-the-shelf program analysis for a range $[\pi_{\tau_1}, \pi_{\tau_2}]$ defined by two sequences of branching decisions s_{τ_1} and s_{τ_2}

by adding additional assumptions and return statements. More precisely, we instrument all assume statements[5] represented by two edges $\ell_{i-1} \xrightarrow{g_i} \ell_i$ and $\ell_{i-1} \xrightarrow{g_j} \ell_j$ with $B_C(g_i) = T$ and $B_C(g_j) = F$. The instrumentation itself is conducted independently for both the upper and lower bound. Again, it suffices to check if the program paths currently explored take the same branching decisions as the lower or the upper bound. We therefore introduce a readout function and additional global variables for tracking, if the current branching decision taken leads to paths not contained in the interval.

5.3.4.1 Readout Function

The *readout function* R_{s_τ} for a sequence of branching decisions $s_\tau = (b_0, b_1, \ldots, b_n)$ with $b_i \in \{T, F\}$ returns the i-th branching decision. For $x \in \mathbb{N}$, we define

$$R_{s_\tau}(x) = x > n \vee \bigvee_{b_i = T} (x = i),$$

which is *true* if the predicate in the function evaluates to *true* and *false* otherwise. For the instrumentation, we use the readout function R_{s_τ} to read the branching decision in s_τ at the current assume statement (e.g., loop heads or if-branches). Therefore, we introduce counter variables `lcntr` and `ucntr` that keep track of the current assume statement for the lower respectively upper bound. The counters are thus incremented at each branching decision in the program. To also account for underspecified test cases, the readout function always returns *true* if $x > n$ holds, i.e., the number of branching decisions contained in the sequence is exceeded. We explain the handling of underspecified test cases in Section 5.3.4.3.

5.3.4.2 Tracking the Bounds

To keep track of whether the current path is a prefix of the lower or upper bound, we introduce two global boolean variables `on_lpath` and `on_upath` for the lower respectively upper bound. Both variables are initialized with *true* and are set to *false*, when the current execution path is no longer a prefix of the bounds. We use the same idea

[5] In our implementation, we only instrument assume statements that occur on the paths induced by the lower bound or upper bound.

as for the range reduction, i.e., we define the instrumentation separately for lower and upper bound for the intervals $[\pi_{\tau_1}, \pi^\top]$ and $[\pi_\bot, \pi_{\tau_2}]$, whereas the combination yields an instrumentation for the complete interval.

Instrumentation for the Lower Bound We aim to ensure that only paths in the interval $[\pi_{\tau_1}, \pi^\top]$ are feasible. When an execution path diverges with the lower bound, we handle the following two cases: (1) the execution path "leaves π_{τ_1} to the left" ($B_P(g_i) = T$ and $R_{s_\tau}(lcntr) = F$) and (2) the execution path "leaves π_{τ_1} to the right" ($B_P(g_j) = F$ and $R_{s_\tau}(lcntr) = T$). In the former case, we add the following instrumentation directly after the branching decision g_i:

```
if(on_lpath) { if([[ R_{s_τ1}(lcntr) = F ]]) {return;} lcntr++; }
```

Thereby, we stop the exploration, as any continuation of the current execution path is not contained in the interval. In the latter case, all continuations of the current paths are contained in the interval $[\pi_{\tau_1}, \pi^\top]$, thus we can safely disable checks added during instrumentation for the lower bound. As we guard the code added using `on_lpath` and `on_upath`, i.e., it is only executed when `on_lpath` or `on_upath` are *true*, setting `on_lpath = 0` disables the instrumentation.

Instrumentation for the Upper Bound Now, we aim to ensure that only paths in the interval $[\pi_\bot, \pi_{\tau_2}]$ are feasible. Thus, we stop the execution if the current execution path "leaves π_{τ_2} to the right" ($B_P(g_j) = F$ and $R_{s_{\tau_2}}(ucntr) = T$). For this, we add the following instrumentation after g_j:

```
if(on_upath) { if([[ R_{s_τ2}(ucntr) = T ]]) {return;} ucntr++; }
```

In the other case where the execution path "leaves the upper path with π_{τ_2} to the left" ($B_P(g_i) = T$ and $R_{s_{\tau_2}}(\texttt{ucntr}) = F$), all continuations of the execution path are smaller than the upper path and thus contained in $[\pi_\bot, \pi_{\tau_2}]$. Therefore, we can disable the code added during instrumentation for the upper bound by setting `on_upath = 0`.

5.3.4.3 Handling Underspecified Sequences of Branching Decisions

For sequences of branching decisions that are underspecified, i.e., when the sequences induce a set of paths, we use the same idea as in range reduction. Therefore, we store n, the number of branching decisions contained in the sequence. If the next branching decision is not specified anymore, i.e., the index is greater or equal to n, we disable the instrumentation for the lower bounds and stop the exploration for the upper bound. We exemplify the instrumentation and the handling of underspecified sequences next.

5.3.4.4 Example of the Instrumentation

To exemplify the instrumentation, we exemplarily employ the lower bound and upper bound instrumentation on the running example C_1, where we use the interval $[\pi_{\tau_1}, \pi_{\tau_2}]$, as depicted in Figure 5.3. We use the two sequences $\mathcal{T}_{C_1}(\pi_{\tau_1}) = s_{\tau_1} = (T, T, F, F)$ and

$\mathcal{T}_{C_1}(\pi_{\tau_2}) = s_{\tau_2} = (F, T, F, F)$. To increase the readability, we depict in Figure 5.8a the instrumentation using s_{τ_1} as lower bound and in Figure 5.8b the instrumentation using s_{τ_2} as upper bound. We highlight the code added during instrumentation in green and red. The resulting program where we apply the instrumentation for the lower and upper bound is given in Appendix A.3.1. Recap that the goal of the instrumentation is to ensure, that all program paths that are not in the given range are infeasible, such that verifiers do not have to analyze these paths. We achieve this by adding return statements to all infeasible paths. Each instrumentation adds seven new code blocks, one for the global variable definition and two blocks for each of the three assume statements in the program. As explained in Section 5.3.4.3, we added in lines 9, 24, and 37 of Figures 5.8a and 5.8b the code for handling underspecified sequences.

5.3.5 Joiner

The last component within ranged program analysis is the *joiner*, that's task is to combine the partial results computed by the ranged analyses for each range to a result for the complete program. Thereby, it has to combine two different verification artifacts, namely partial verdicts and partial witnesses.

5.3.5.1 Joining Partial Verdicts

The combination of different verdicts is straightforward: When one ranged analysis has raised an alarm, i.e., it computes a path within the assigned interval that contains a property violation, the path is also feasible in the full program. Thus, the overall verdict is *false* and the other running ranged analyses could be aborted. In case all ranged analyses have computed the verdict *true*, the overall program is correct, as all feasible program paths are contained in at least one interval. Thus, the overall verdict returned is *true*. If at least one ranged analysis has computed no result for a range (e.g., returns *unknown*) and all others have proven their assigned range safe, the overall verdict return is also *unknown*, as the range that is not solved may contain a path violating the property or is also safe.

5.3.5.2 Joining Partial Witnesses for Range Reduction

Beneath a verdict, the ranged analyses (at best) also compute a partial correctness or violation witness. We first discuss how to join witnesses when the ranged analyses are built using range reduction. By using a range reduction in composition with the off-the-shelf analyses, the program analyzed is not modified and thus the paths contained in the witnesses can also be found in the original program. As for the verdicts, a violation witness computed for a range contains a path that is present in the original program, thus each partial violation witness is also valid for the full program[6].

[6]If more than one ranged program analysis computes a violation witness, reporting all paths violating the property may be beneficial for users and developers, as one path might be shorter or simpler than

```
0  unsigned int on_lpath = 1;
1  unsigned int lcntr = 0;
2  int lredout( int pos)
     {return pos>=4||!(pos==2||pos==3);}
3
4  int div2WithReminderAbs(short input){
5    if (input < 0){
6      if(on_lpath){
7        if(!lredout(lcntr)){return;}
8        lcntr++;
9        if(on_lpath&&lcntr>=4)on_lpath=0;
10     }
11     input = -input;
12   }else{
13     if(on_lpath){
14       on_lpath = !lredout(lcntr);
15       lcntr++;
16     }
17   }
18   int rem = input;
19   int res = 0;
20   while (rem > 1){
21     if(on_lpath){
22       if(!lredout(lcntr)){return;}
23       lcntr++;
24       if(on_lpath&&lcntr>=4)on_lpath=0;
25     }
26     rem -= 2;
27     res++;
28   }
29   if(on_lpath){
30     on_lpath = !lredout(lcntr);
31     lcntr++;
32   }
33   if (input != 2 * res + rem){
34     if(on_lpath){
35       if(!lredout(lcntr)){return;}
36       lcntr++;
37       if(on_lpath&&lcntr>=4)on_lpath=0;
38     }
39     abort();
40   }else{
41     if(on_lpath){
42       on_lpath = !lredout(lcntr);
43       lcntr++;
44     }
45   }
46   return res;
47 }
```

(a) A lower bound instrumentation using s_{τ_1}

```
0  unsigned int on_upath = 1;
1  unsigned int rcntr = 0;
2  int rredout( int pos)
     {return pos >= 4 || pos == 1;}
3
4  int div2WithReminderAbs(short input){
5    if (input < 0){
6      if(on_upath){
7        on_upath = rredout(rcntr);
8        rcntr++;
9        if(on_upath && rcntr>=4){return;}
10     }
11     input = -input;
12   }else{
13     if(on_upath){
14       if(rredout(rcntr)){return;}
15       rcntr++;
16     }
17   }
18   int rem = input;
19   int res = 0;
20   while (rem > 1){
21     if(on_upath){
22       on_upath = rredout(rcntr);
23       rcntr++;
24       if(on_upath && rcntr>=4){return;}
25     }
26     rem -= 2;
27     res++;
28   }
29   if(on_upath){
30     if(rredout(rcntr)){return;}
31    rcntr++;
32   }
33   if (input != 2 * res + rem){
34     if(on_upath){
35       on_upath = rredout(rcntr);
36       rcntr++;
37       if(on_upath && rcntr>=4){return;}
38     }
39     abort();
40   }else{
41     if(on_upath){
42       if(rredout(rcntr)){return;}
43       rcntr++;
44     }
45   }
46   return res;
47 }
```

(b) An upper bound instrumentation using s_{τ_2} called `rangeProg1`

Figure 5.8: Range programs generated using instrumentation for the running example

Next, we have a look at the case where all ranged analyses compute partial correctness witnesses and assume, that we only use two ranged analyses in parallel. As both

the other. We plan to also be able to handle multiple violation witnesses in future work.

ranged analyses have proven that the program is correct, we could simply generate a trivial correctness witness, i.e., a witness containing each CFA location once and no invariants. Nevertheless, we lose all information computed to justify the correctness, namely the (partial) invariants contained in the partial correctness witnesses. We propose in Algorithm 6 an algorithm to join two correctness witnesses[7], that explores both witnesses in parallel, guaranteeing that computed invariants are retained.

Algorithm 6 Joining two correctness witnesses

Input: CFA $C = (L, \ell_0, G)$ ▷ *CFA*
Witnesses $A_1^{\text{CW}} = (Q, \Sigma, \delta, (q_0, \varphi_0), Q)$, ▷ *First witness*
$A_2^{\text{CW}} = (Q', \Sigma, \delta', (q_0', \varphi_0'), Q')$ ▷ *Second witness*
Output: A^{CW} ▷ *Joined correctness witness*

1: $waitlist = N = \{(\ell_0, (q_0, \varphi_0), (q_0', \varphi_0'))\}$;
2: **while** $(waitlist \neq \emptyset)$ **do**
3: pop $(\ell, (q_i, \varphi_i), (q_i', \varphi_i'))$ from $waitlist$
4: **for each** each $(\ell, op, \ell_s) \in G$ **do**
5: **for each** each $((q_i, \varphi_i), (G_i, \psi_i), (q_j, \varphi_j)) \in \delta$ with $(\ell, op, \ell_s) \in G_i$ **do**
6: **for each** each $((q_i', \varphi_i'), (G_i', \psi_i')', (q_j', \varphi_j')) \in \delta'$ with $(\ell, op, \ell_s) \in G_i'$ **do**
7: **if** $\psi_i \not\equiv false \wedge \psi_i' \not\equiv false \wedge (\ell_s, (q_j, \varphi_j), (q_j', \varphi_j')) \notin N$ **then**
8: $N = N \cup \{(\ell_s, (q_j, \varphi_j), (q_j', \varphi_j'))\}$;
9: $waitlist = waitlist \cup \{(\ell_s, (q_j, \varphi_j), (q_j', \varphi_j'))\}$;
10: $Q_{\text{join}} = \{(\ell, inv) | \ell \in L, inv = false \bigvee_{(\ell,(q,\varphi),(q',\varphi')) \in N} \varphi \vee \varphi'\}$;
11: $\delta_{\text{join}} = \{(\ell, (\{(\ell, op, \ell')\}, true), \ell') \mid (\ell, op, \ell') \in G\}$
$\cup \{(\ell, (\{(\ell_p, op, \ell_s) \in G \mid \ell_p \neq \ell\}, true), \ell) \mid \ell \in L\}$;
12: **return** $A^{\text{CW}} = (L, \Sigma, \delta_{\text{join}}, \ell_0, Q_{\text{join}})$;

The parallel exploration (lines 1–9) of the two witnesses also takes the program given as CFA into account, to generate more compact and potentially easier verification witnesses. It starts at the initial nodes of the CFA and the two correctness witnesses and explores the successor of each node once. As the goal is to explain the behavior of the program we only combine successors that adhere to the control flow of the program, As the correctness witnesses only contain trivial assumptions ψ, we can ignore them. We continue this exploration until all successors are computed. Thereafter (lines 10 and 11), we compute the states of the joined correctness witness, and combine all invariants from both witnesses computed for each location of the CFA. We choose to combine the information via disjunction, as the different nodes most likely represent different behaviors of the program ranges. Thus, we may lose information in case both witnesses consider the full behavior when computing their invariant information. However, disjunction ensures that we remain sound in case different behaviors are considered. Note that we include *false* in the disjunction to account for syntactically unreachable locations. Thereafter, we compute the transfer relation for the joined in-

[7]For more correctness witnesses, one could iteratively combine these by first joining the first two witnesses with Algorithm 6 and then successively using Algorithm 6 to combine the remaining witnesses.

variant witnesses using the transfer relation of the CFA, where we use true assumptions only. Additionally, we add self-loops to each state in Q that cover all remaining edges, which do not adhere to the control flow.

Finally, the question arises, which state invariants should be used by the ranged analysis for locations that are out of range. Basically, there are two options: *true* or *false*. In contrast to *false*, *true* is always a correct invariant with respect to the complete program. However, correctness witnesses of ranged analyses that only analyze a subset of the program behavior already do not guarantee correct invariants for the complete program for locations they visited during their analysis, especially if they do not analyze all paths to that location. They only encode information on behavior that they have seen. In addition, providing *true* results in losing all correctness witness information provided by a different ranged analysis, in case invariants for the same location are combined using a disjunction. The invariants need to be combined using a disjunction to compute a sound solution, as the invariants may be generated for different program paths. Thus, *false* better fits the information encoded at other seen locations. Since correctness witnesses were not designed for incomplete analysis and do not provide any requirements on how to handle program locations that are not reached, it depends on the verifier how it handles locations not analyzed.

5.3.5.3 Joining Witnesses for Instrumentation

Lastly, we briefly discuss how to join correctness witnesses in case ranged analyses are built using instrumentation. By instrumenting the code individually for each range, additional program statements are added. Thus, correctness and violation witnesses may contain these additional statements not present in the original program. In both cases, the instrumentation has to be reverted within the artifacts, meaning that added lines of code are removed and additional information contained in the witnesses, as line numbers of statements, are mapped back to the original program. Thereafter, the methods discussed in Section 5.3.5.2 are applicable. To revert the instrumentation it would suffice to mark each line added, e.g., using comments or an extra file, remove them afterward from the witnesses, and ensure, that no variables added during instrumentation are present in the witness.

5.3.6 Work Stealing

When employing ranged program analysis with different analyses for the ranges, one has to decide which analysis should work on which range. As all program analyses have different strengths and weaknesses, the assignment of ranges can affect the performance of the ranged program analysis. Especially, there exist tasks that certain analyses can solve very fast. Thus, when using such an analysis in combination with another analysis, that can not solve the assigned range, the ranged program analysis is likely to fail.

As the assignment of program analyses and ranges is fixed for all tasks, we propose

using *work stealing* to avoid such situations, where the ranged program analysis fails at solving a task, although at least one of the used ranged analyses can solve all ranges of the task. The idea of work stealing is simple but efficient: The ranged analysis that first finishes proving its assigned range safe is restarted on the other range unless it already found a violation and, thus, already determined the analysis result. Thereby, the other range is analyzed by both ranged analyses in parallel, until one of them computes a result for the range.

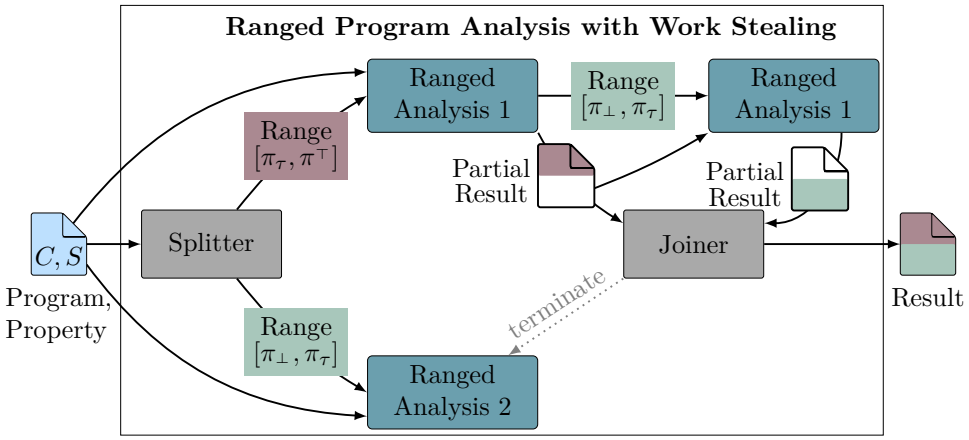


Figure 5.9: Ranged analysis with work stealing, where Ranged Analysis 1 completes the verification of both ranges

We depict work stealing in case the analysis of the range $[\pi_\tau, \pi^\top]$ is completed first in Figure 5.9 and Figure 5.10. In both scenarios, *Ranged Analysis 1* is started for a second time and analyzes the range $[\pi_\perp, \pi_\tau]$. In Figure 5.9, we depict the situation that *Ranged Analysis 1* also finishes for the range $[\pi_\perp, \pi_\tau]$, i.e., it successfully analyzes both ranges and *Ranged Analysis 2* is terminated. Figure 5.10 depicts the opposite situation, where *Ranged Analysis 2* finishes the analysis of the range $[\pi_\perp, \pi_\tau]$ before *Ranged Analysis 1* and *Ranged Analysis 1* is terminated.

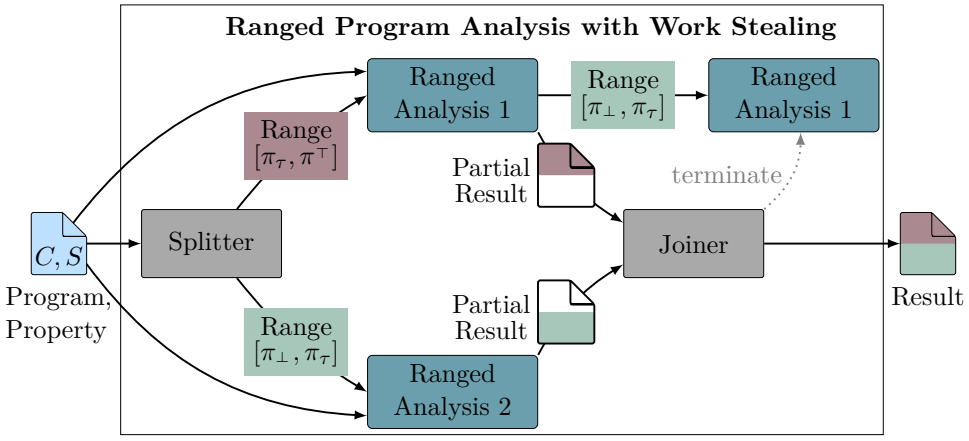


Figure 5.10: Ranged analysis with work stealing, where both Ranged Analysis 1 and Ranged Analysis 2 complete the verification of a range

Note that work stealing offers a simple but efficient way to reduce the risk of the

ranged program analysis failing, especially in cases where one ranged analysis used can solve the task. Conceptually, work stealing also allows for precision reuse, as in [Bey+13], or reusing information encoded in the partial results, as in CoVEGI, when an analysis is restarted. We depict this precision reuse in Figure 5.9, where *Ranged Analysis 1* also uses the partial result computed for the upper range as additional input. Work stealing can increase the resource consumption, especially the CPU time, as there may be three analyses in total (but at most two in parallel).

5.3.7 Example Application of Ranged Program Analysis

After having explained the concept of ranged program analysis and the different methods for building ranged analyses, we revisit the running example from Figure 2.3. We use ranged program analysis with two ranged analyses, a predicate abstraction and a tool employing BMC. Both ranged analyses are constructed using instrumentation to ensure that only paths within the bounds are analyzed[8]. We configure ranged program analysis to let the predicate abstraction analyze the first range and the tool employing BMC the second range.

As we combine two analyses, the splitter generates a single path π_τ, which is induced by the test case $\tau = \{input \mapsto 2\}$ and forms the sequence $s_\tau = (F, T, F, F)$. Thus, the program is divided into the two ranges $[\pi_\bot, \pi_\tau]$ and $[\pi_\tau, \pi^\top]$. The second range contains only two feasible program paths, namely the one that contains one loop iteration, and the one that has no loop iteration. The first range contains all paths with at least two loop iterations for positive inputs and all paths traversed with negative input.

Now, we use the instrumentor to construct two range programs, the first one called `rangeProg1` is depicted in Figure 5.8b, the second one called `rangeProg2` is shown in Appendix A.3.1 in Figure A.12. As configured, `rangeProg1` is given to the predicate abstraction and `rangeProg2` is analyzed by the BMC tool. The predicate abstraction computes the invariant $inv \equiv input = rem + 2 * res$, which suffices to prove that the assigned range is safe. In contrast, BMC does not employ any abstraction techniques, but encodes the full program up to the bound n using the strongest postcondition. In the first iteration with $n = 0$, only the path without a loop iteration is checked and proven correct. Increasing n, e.g. to 2, enables BMC to encode the full range, such that the range is proven safe. As both ranges are proven safe, the joined results are returned.

To demonstrate the advantage of work stealing, we change the configuration such that the BMC tool is working on the first range and predicate abstraction on the second one. Thus, `rangeProg1` is given to the BMC tool and predicate abstraction is analyzing `rangeProg2`. Again, the predicate abstraction computes the invariant inv for `rangeProg2` and completes the assigned task. In contrast, BMC has to unroll the

[8] We could also combine ranged analyses using instrumentation and range reduction within a single ranged program analysis. Then, the splitter generates a test case and we transform the test case in a sequence of branching decisions as explained in Section 5.2.

loop to a bound of 16 384 to complete the analysis of the range. Clearly, this number of unrollings is significantly larger than in the former example, and BMC may exceed the given time limitations. Then, the ranged program analysis fails, as no solution for the range $[\pi_\perp, \pi_\tau]$ is known. When employing work stealing, the predicate abstraction is assigned `rangeProg1` after it finishes the analysis of `rangeProg2`, where the computed invariants are reused. Hence, given the invariant *inv*, predicate abstraction succeeds fast in proving `rangeProg1` safe. Both partial results computed by predicate abstraction are joined and the final results are returned.

5.4 Implementation

To analyze the feasibility of ranged program analysis for CPA-based and non-CPA-based off-the-shelf analyses, we implemented ranged program analysis and all necessary components. More precisely, we realized the concept of ranged program analysis in CoVeriTeam, implemented range reduction and program instrumentation as well as four different splitters and a joiner for verdicts and witnesses.

Realizing Ranged Program Analysis The composition of ranged program analyses is implemented within CoVeriTeam. To be able to use work stealing, i.e., to keep track of which program range is already successfully analyzed, we build a novel *ranged program analysis component* within CoVeriTeam. Its task is to (1) orchestrate the composition of ranged analyses, especially for work stealing, and (2) aggregate the partial results. The algorithm follows the description in Figure 5.4 and Section 5.3.6 and receives as inputs the two ranged analyses *Ranged Analysis 1* and *Ranged Analysis 2*, and a splitter. It first uses the splitter to generate the ranges. If the splitter fails, e.g., Lb3 cannot compute a test case, when the program does not contain a loop, we execute *Ranged Analysis 1* on the interval $[\pi_\perp, \pi^\top]$. We extended CoVeriTeam by adding ranged analyses as a new type of verifier. Thus, it executes the two ranged analyses in parallel using the execution mechanism of CoVeriTeam. The algorithm collects the verdicts and witnesses computed by the ranged analyses and employs, if configured, work stealing. Moreover, the witnesses are joined. Therefore, the partial verification witnesses generated by *Ranged Analysis 1* and *Ranged Analysis 2* are collected first. Then, the algorithm for joining witnesses is used to compute the joined witness.

Generating Ranges For the evaluation, we build our different splitters introduced in Section 5.3.1 as standalone components within CPAchecker. Each of them generates either test cases in the standardized XML-based TEST-Comp test case format[9] or sequences of branching decisions. For the latter, we implemented the transformation from test cases to sequences explained in Section 5.2 within the splitter to generate the

[9] `https://gitlab.com/sosy-lab/test-comp/test-format/blob/testcomp23/doc/Format.md`

correct input format needed for the instrumentation. The two splitters based on loop-unrollings Lb3 and Lb10 build the program execution tree demand-driven, meaning that only the path containing three respectively ten loop unrollings is generated. Using the path formula, an SMT-solver is called to find satisfiable assignments for the program input variables. These assignments are then used within the test case that is returned. The two splitters based on a random selection of branching decisions, namely Rdm and Rdm9 are also realized as standalone components within CPAchecker.

Ranged Analyses within CPAchecker At first, we implement the composition of range reduction from Section 5.3.3 with an arbitrary Configurable Program Analysis in CPAchecker, making use of the existing composite pattern. More precisely, we build a novel range reduction analysis within CPAchecker, which combines the transfer relations of lower bound CPA and upper bound CPA as formalized in Section 5.3.3.2. Hence, it reuses elements from the value analysis, especially from the transfer relation, which are already implemented within CPAchecker.

Ranged Analyses for Off-the-shelf-tools We implement the instrumentation as a standalone component in Python[10]. First, we use an AST parser[11] to identify all branching points in the program. Then, we instrument the program as defined in Section 5.3.4, using the sequences generated by the splitter. Our implementation supports the instrumentation of (GNU) C programs except for `switch` statements.

Witness Joining We implemented Algorithm 6 for the joining of witnesses presented above in CPAchecker[12]. For the parallel exploration, we compose a CPA tracking the control flow with one CPA per witness that tracks the transitions of that witness. We then use CPAchecker to compute the joined witness with combined information and use CPAchecker's witness generation for computing witnesses similar to Algorithm 6.

5.5 Evaluation

The goal of our evaluation is to analyze the effect of using ranged program analysis for CPA-based and off-the-shelf analyses not based on the CPA algorithm. Siddiqui and Khurshid [SK12] already showed that the idea of using ranges for splitting work among instances of symbolic execution analyses increases the overall efficiency. In the following, we are interested in a more systematic analysis of the effect of using ranged program analysis. The overall goal of the evaluation is to investigate, to which extent using ranged program analysis can increase the performance (i.e., effectiveness and efficiency) of off-the-shelf analyses and how it compares to a parallel portfolio.

[10]The implementation was led by Cedric Richter as part of the work for [HJRW23c].

[11]`https://tree-sitter.github.io/tree-sitter/`

[12]The conceptual work and implementation was led by Marie-Christine Jakobs as part of the work for [HJRW24b].

Hence, we compare ranged program analysis with the basic analysis employed. We first start with using symbolic execution in ranged program analysis to analyze the feasibility of ranged program analysis and evaluate the performance of the different splitting strategies. Thereafter, we use seven verifiers and evaluate different configurations of ranged program analysis to analyze the overall effect of using parallel execution based on ranges, combining different analyses, and using work stealing. Next, we compare the best-performing instance with a parallel portfolio. Lastly, we experimentally demonstrate the feasibility of the algorithm for joining witnesses.

5.5.1 Evaluation Setup

To answer the research questions regarding the effect of using ranges for arbitrary off-the-shelf analyses, we analyze the effectiveness and efficiency of different configurations of ranged program analysis compared to the analyses running standalone, which are called *basic analyses*. We use CPAchecker with the configurations for symbolic execution, predicate abstraction, value analysis, and BMC, as well as Klee, Symbiotic, and UltimateAutomizer.

Configurations For the evaluation, we build different configurations of ranged program analysis using the seven basic analyses. We use the abbreviations Value for value analysis, Se for symbolic execution, Pred for predicate abstraction, Symb for Symbiotic, and UA for UltimateAutomizer. For configurations using UltimateAutomizer, Symbiotic, or Klee, we employ instrumentation to build the ranged analyses, for the others we use the range reduction. A ranged program analysis that uses value analysis for the range $[\pi_\bot, \pi_\tau]$ and BMC for $[\pi_\tau, \pi^\top]$ for some computed test input τ is denoted by Ra-Value-Bmc. In case we use the same analysis for both intervals, we denote this by e.g., Ra-Value. To achieve a fair comparison of basic analyses and ranged program analyses, we also executed the basic analyses in CoVeriTeam, where we build a simple configuration that directly calls them.

Computing Resources and Benchmark Tasks All experiments were run on machines with an Intel Xeon E3-1230 v5 @ 3.40 GHz (8 cores), 33 GB of memory, and Ubuntu 22.04 LTS with Linux kernel 5.15.0. We used in total 10 229 C tasks from all sub-categories of the SV-COMP dealing with the reachability of error labels [SVB23], whereof 6 791 tasks fulfill the property and the other 3.438 contain a property violation. In a verification run a tool is given one of the 10 229 C tasks, containing a program and a specification, and is asked to either compute a proof (in case the program fulfills the specification) or to raise an alarm (if it violates the specification). We limit the available resources to a total of 15 min CPU time on 4 CPU cores and 15 GB of memory. To ensure a comparison on equal ground, we use the same resource limitations for each tool. This means that the basic analyses have the same resources available as the

Table 5.1: The number of correct and incorrect verdicts reported by symbolic execution employing different splitting strategies. The column *par. only*(parallel only) contains the number of tasks solved by the ranged program analysis but not by symbolic execution.

	correct				incorrect	
	overall	**proof**	**alarm**	**par. only**	**proof**	**alarm**
Se	1 593	584	1 009	-	5	27
Ra-Se-Lb3	1 639	583	1 056	113	5	57
Ra-Se-Lb10	1 638	582	1 056	117	5	58
Ra-Se-Rdm	1 578	583	995	70	4	39
Ra-Se-Rdm9	1 539	582	957	31	4	57

ranged program analyses. As we employ two ranged analyses within a ranged program analysis, the available resources are split between the two ranged analyses[13].

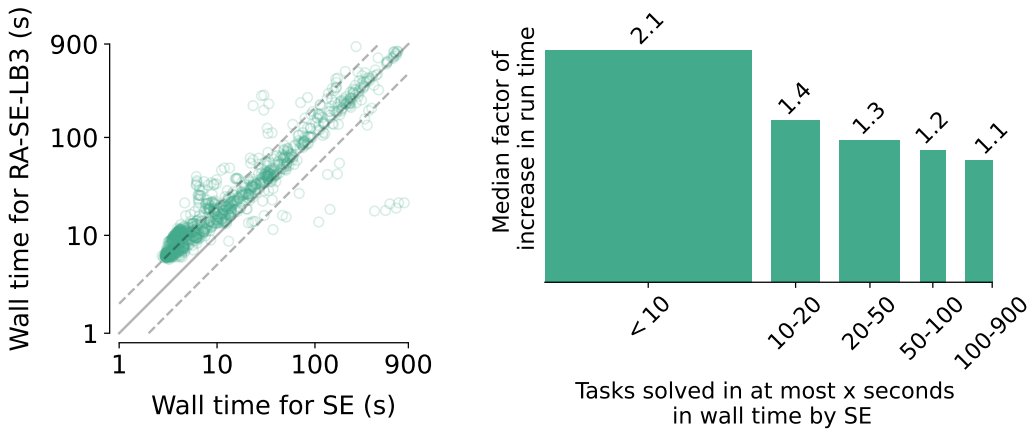
Availability The implementations of the ranged program analysis and all components as well as all experimental data are publicly available and archived at Zenodo in the unified artifact [HJRW24a].

5.5.2 RQ 1: Does Ranged Analysis and the Different Splitting Strategies Work for Symbolic Execution?

Evaluation Plan We aim to analyze the performance of symbolic execution in a composition of ranged analyses. Therefore, we compare the effectiveness and efficiency of the composition of ranged program analyses with two ranged analyses each using a symbolic execution with symbolic execution running standalone. Therein, we use one of the four splitters from Section 5.3.1. For efficiency, we focus on the (real) time taken to solve the task (*wall time*). Thereby, we take the advantages of the parallelization employed within the ranged program analysis into account.

Experimental Results We report the experimental results regarding effectiveness in Table 5.1. Each row contains the number of overall correctly solved tasks, the number of correctly computed proofs and alarms, and the number of tasks correctly solved only by parallel combinations but not by symbolic execution.
Therein, we observe that Ra-Se using the splitter based on loop-bounds (Lb3 and Lb10) correctly solves 1 639 respectively 1 638 tasks and symbolic execution 1 593 tasks, meaning that Ra-Se-Lb3 and Ra-Se-Lb10 can solve 46 and 45 tasks more than symbolic execution, an increase of 3%. In addition, there are 113 tasks for Ra-Se-Lb3 and 117 tasks for Ra-Se-Lb10 that are not solved by the basic analysis. In all these cases, the ranged program analysis has raised valid alarms. For computing an alarm, it suffices to find a single path that violates the specification. Thus, using two symbolic execution

[13]We evaluated configurations using three ranged analyses in parallel in [HJRW23b].



(a) Comparison of Wall time for Ra-Se-Lb3 and symbolic execution

(b) Median factor of run-time increase by Ra-Se-Lb3 compared to symbolic execution. The width of the bar for i corresponds to the number of verification tasks that are solved by symbolic execution in the interval.

Figure 5.11: Comparison of efficiency of Ra-Se-Lb3 and symbolic execution

analyses in parallel, working on different parts of the program increases the chance of finding such a violating path. The number of reported proofs is nearly constant, as Se, Ra-Se-Lb3, and Ra-Se-Lb10 have to check all paths in the program leading to a property violation for infeasibility. The splitting strategies that randomly select a path for generating the ranges do not perform as well as the loop-based splitting strategies. Nevertheless, both allow the ranged program analyses to raise alarms not found by the basic analysis. All configurations of ranged analyses compute a few more false alarms. For these tasks, symbolic execution runs into a timeout and would also compute a false alarm, if its time limit would be increased.

For comparing the efficiency of compositions of ranged analyses, we compare the wall time taken to compute a correct solution by Se and the configuration Ra-Se-Lb3, as we have already seen that Lb3 leads to the best effectiveness. We excluded all tasks where the generation of the ranges fails, as symbolic execution and the Ra-Se-Lb3 behave equally in these cases. The scatter plot in Figure 5.11a visualizes the wall time consumed to compute a result in a log-scale by Se (on the x-axis) and by Ra-Se-Lb3 (on the y-axis), for tasks solved correctly by both analyses. It indicates that for tasks solved quickly, Ra-Se-Lb3 requires more time than symbolic execution, as the points are in most cases above the diagonal, and that the difference gets smaller the longer the analyses run.

We present a more detailed analysis of the efficiency in Figure 5.11b. Each of the bar plots represents the median factor of the increase in the run time for tasks that are solved by Se within the time interval that is given on the x-axis. If for example Se solves all tasks in five seconds and Ra-Se-Lb3 in six seconds, the factor would be 1.2, if symbolic execution takes five seconds and Ra-Se-Lb3 only three, the factor is

0.6. The width of the bars corresponds to the number of tasks within the interval. For Ra-Se-Lb3, the median increase is 1.7 for all tasks. Taking a closer look, in the median it takes twice as long to solve tasks that are solved by Se within at most ten seconds. Most importantly, the impact of the influence of the additional overhead gets smaller the longer the analyses need to compute the result (the factor is decreasing). For the more complex tasks requiring more time, ranged program analysis and symbolic execution are equally fast.

To analyze why the ranged program analysis needs more overall time to compute a solution compared to symbolic execution, especially for tasks that can be solved fast, we have a more detailed look at the time needed to compute the ranges. Therefore, we measured the wall time taken by Lb3 to compute a test case separately for all tasks and depicted the results as a boxplot in Figure 5.12. In the median, Lb3 needs overall 4.6 seconds to compute the bounds, visualized using the orange line. The boxes contain the median values for 25% resp. 75% of the tasks. Hence, 75% of all bounds are computed within 6.9 seconds of wall time. The whisker contains 99% of all tasks, hence there are only very few cases where Lb3 needs more than 38 seconds.

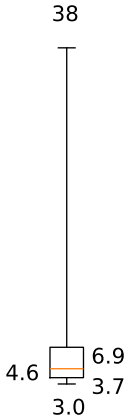


Figure 5.12: Wall time in seconds to compute a test case using Lb3

As computing the test cases takes in the median 4.6 seconds wall time, this additional computational effort is a main factor for the time increase. As the time taken to compute a range is approximately the same for most of the tasks, it influences the overall wall time for complex tasks only to a small extent. For these tasks, the advantages of ranged program analysis, namely analyzing different parts of the program in parallel, become visible. Unfortunately, the additional overhead of ranged program analysis, e.g., caused by the range reduction, exceeds the advantages of parallelization, as the ranged program analysis is nearly as fast as the basic analysis for complex tasks, but not faster.

Results

The use of ranged program analysis for symbolic execution increases its effectiveness for finding violations of the specification, whereas using Lb3 yields the best results. The overall time consumed to compute the result does not increase for large or complex tasks due to the parallelization employed. Nevertheless, the initial overhead caused by generating the ranges is observable especially for simple tasks.

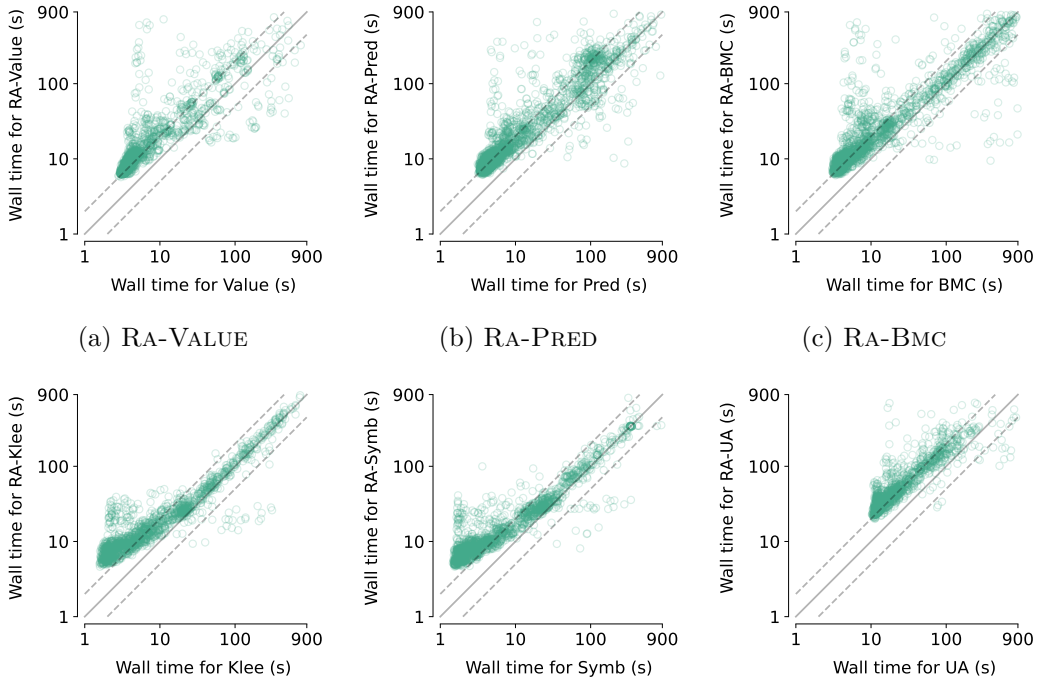
5.5.3 RQ 2: Can Ranged Program Analysis Increase the Efficiency and Effectiveness of Off-the-shelf Analyses?

Evaluation Plan To analyze the performance of using the seven basic analyses within ranged program analysis, we compare the effectiveness and efficiency of the ranged program analysis with the basic analysis running standalone. For efficiency, we again focus on the wall time.

Table 5.2: Number of correct and incorrect verdicts reported by the seven basic analyses and the combination of ranged program analysis using Lb3. The column *par. only* (parallel only) contains the number of tasks that are correctly solved by a ranged program analysis (Ra, Ws) but not by the basic analysis employed within the configuration.

	correct				incorrect	
	overall	**proof**	**alarm**	**par.only**	**proof**	**alarm**
Se	1 593	584	1 009	-	5	27
Value	3 231	2 324	907	-	4	20
Pred	3 741	2 354	1 387	-	10	40
Bmc	3 282	1 376	1 906	-	5	63
Klee	2 982	1 294	1 688	-	77	3
Symbiotic	3 918	2 232	1 686	-	77	1
UA	4 240	3 096	1 144	-	23	0
Ra-Se	1 639	583	1 056	113	5	57
Ra-Value	2 972	1 972	1 000	218	15	48
Ra-Pred	3 560	2 298	1 262	67	6	50
Ra-Bmc	3 217	1 363	1 854	117	5	63
Ra-Klee	2 968	1 293	1 675	7	77	2
Ra-Symb	3 881	2 185	1 696	78	95	1
Ra-UA	3 964	2 925	1 039	24	22	0
Ra-Se-Pred	2 515	1 248	1 267	33	5	40
Ra-Value-Pred	3 250	2 134	1 116	32	5	52
Ra-Bmc-Pred	3 394	1 725	1 669	29	5	63
Ra-Symb-UA	3 981	2 728	1 253	25	13	0
Ra-Klee-UA	3 912	2 657	1 255	24	13	0
Ws-Se-Pred	3 460	1 996	1 464	33	6	40
Ws-Value-Pred	3 930	2 593	1 337	39	6	52
Ws-Bmc-Pred	4 240	2 493	1 747	34	6	63
Ws-Symb-UA	4 939	3 223	1 716	62	85	0
Ws-Klee-UA	4 924	3 185	1 739	21	74	0

Experimental Results The results of our evaluation are shown in the first two segments of Table 5.2, where we report the number of overall solved tasks, the correct proofs and alarms, as well as incorrectly computed proofs and raised alarms. In addition, we report in column *par.only* the number of tasks that are only solved using ranged program analysis but not by the basic analysis running standalone.



(a) Ra-Value (b) Ra-Pred (c) Ra-Bmc

(d) Ra-Klee (e) Ra-Symb (f) Ra-UA

Figure 5.13: Scatter plots comparing the wall time of the basic analyses and ranged program analysis using two instances of the same analysis

Having a look at the total numbers of correctly solved tasks, we first observe, as in Section 5.5.2, that symbolic execution benefits from being used within ranged program analysis. For the other six basic analyses, the total number of tasks correctly solved by the ranged program analyses does not increase.

For Klee, the second symbolic execution, we observe a comparable effectiveness, as it solves 2 968 tasks, that are 14 fewer than Klee standalone, solving 2 982 tasks. The tasks that are only solved by Klee standalone could not be solved by Ra-Klee within the given resource limits. If we double these limits, all of them could be solved. There are 7 tasks uniquely solved by Ra-Klee. In all cases, Ra-Klee detects the property violation within the given resource limits, as using a ranged program analysis allows it to search in different parts of the program in parallel. Value analysis is also a path-based analysis, but in contrast to the former two, the effectiveness decreases slightly, as Ra-Value solves 2 972 tasks compared to 3 231 tasks that are solved by the basic analysis. Having a look at the 259 tasks, we notice that Ra-Value-Pred either exceeds the memory or the CPU time limits given. When value analysis is used within ranged program analysis, 218 task can be solved that could not be solved before. The vast majority of them are correctly raised alarms. There are also a few cases, where sharing the work among two instances simplifies the task, such that Ra-Value successfully proves the task correct.

Next, we have a look at BMC, predicate abstraction, Symbiotic, and UA, all four being non-path-based analyses, where the latter three aim to find abstractions. The

corresponding instances of ranged program analyses that use two instances of these analyses can solve 67 tasks fewer for Ra-Pred, 117 for Ra-Bmc, 37 for Ra-Symb and 276 for Ra-UA. This decrease is most likely caused by the fact that the analyses consider multiple paths of the program at once, either through abstraction or full program encoding. Nevertheless, the instances of ranged program analysis can compute correct proofs and alarms that are not computed by the basic analysis running standalone, namely 67 for Ra-Pred, 117 for Ra-Bmc, 78 for Ra-Symb and 24 for Ra-UA.

Lastly, we discuss the number of incorrectly computed results. We observe that especially the CPA-based combinations Ra-Se, Ra-Value, and Ra-Pred raise more incorrect alarms compared to the basic analyses and Ra-Symb computes more incorrect proofs. To validate that the additional incorrect alarms are not caused by an error in the implementation of the ranged program analysis, especially in the range reduction, we analyzed a set of randomly selected tasks and restricted the basic analyses to explore only the path encoded in the reported violation. For all tasks, the basic analyses now also raised the alarms instead of running into a timeout. In addition, we randomly selected and analyzed tasks where Ra-Symb computes additional incorrect proofs manually, validated that the range program contains a property violation, and cross-verified them using UA and Klee.

After having discussed the effectiveness, we have a closer look at the efficiency of ranged program analysis. We depict the scatter plots comparing the wall time of the basic analyses (on the x-axis) and the ranged program analysis (on the y-axis) in Figure 5.13. We obtain similar results as in RQ 1: The ranged program analysis needs approximately twice as long for simple tasks that can be solved within less than ten seconds, showing the overhead of the generation of the ranges. For complex tasks, the ranged program analysis has a comparable overall run time.

Results

All seven combinations of analyses can benefit from being used within ranged program analysis, as each solves tasks not solved by the respective standalone analysis. Again, path-based analyses benefit the most. The overhead caused by ranged analysis reduces for more complex tasks.

5.5.4 RQ3: Does Combining Different Off-the-shelf Analyses Pays Off?

Evaluation Plan To analyze the performance of combining different basic analyses within ranged program analysis, we build in total five different combinations: For the CPA-based analyses, we combine predicate abstraction, a technique that can effectively compute proofs, with symbolic execution, value analysis, and BMC, which are good at detecting property violations. Thereby, we get the three configurations Ra-

Se-Pred, Ra-Value-Pred, and Ra-Bmc-Pred. For the off-the-shelf tools, we combine UltimateAutomizer, the tool having the best performance overall in SV-COMP' 23 with Klee and Symbiotic, yielding the two configurations Ra-Symb-UA and Ra-Klee-UA. As we are moreover interested in evaluating the effect of work stealing (cf. Section 5.3.6), we also evaluate each of the five configurations with work stealing, indicated by using Ws instead of Ra as a prefix. Again, we compare the effectiveness and efficiency.

Experimental Results At first, we analyze the results for the instances of ranged program analysis without work stealing and compare the first and third blocks of Table 5.2. The column *par.only* now contains the tasks that neither of the two basic analyses can solve but the ranged program analysis.

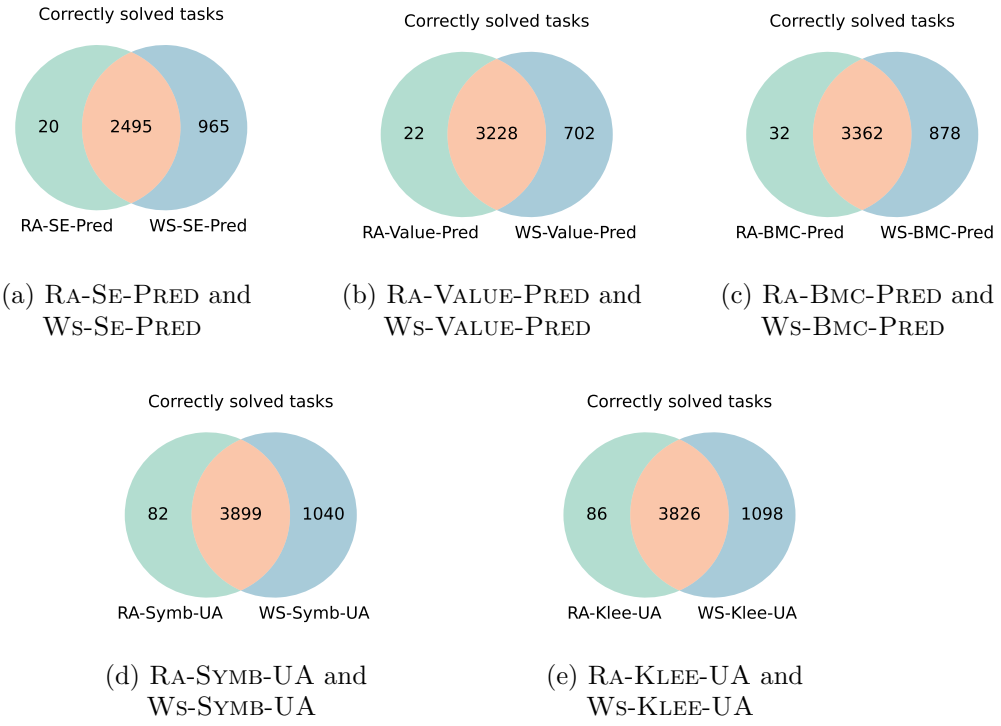
For all five combinations, we make the same observation when comparing the total number of correctly solved tasks with the two basic analyses. In each of the combinations, one basic analysis has a higher effectiveness than the other one, in our configurations these basic analyses are predicate abstraction and UltimateAutomizer. Each of the instances of ranged program analysis can now solve more tasks than the basic analysis with lower effectiveness (symbolic execution, BMC or value analysis for Ra-Se-Pred, Ra-Value-Pred, and Ra-Bmc-Pred and Klee or Symbiotic for Ra-Klee-UA and Ra-Symb-UA). Compared to predicate abstraction or UltimateAutomizer, the ranged program analyses have lower effectiveness, as they compute fewer correct solutions. For many of these tasks, either predicate abstraction or UltimateAutomizer successfully verifies the assigned range, but the other analysis does not succeed. Then, the ranged program analysis does not return a result, as one range is not successfully verified[14].

Encouragingly, we can again observe the positive effect of combining two conceptually different off-the-shelf analyses, as all five configurations solve tasks for which none of the two employed basic analyses can compute a solution: Ra-Se-Pred solves 33 tasks not solved by both basic analyses, Ra-Value-Pred 32 tasks, Ra-Bmc-Pred 29 tasks, Ra-Symb-UA 25 task and Ra-Klee-UA 24 tasks, again showing the advantages of cooperative software verification.

The Effect Of Work Stealing Motivated by the fact that the combination of different analyses within ranged program analyses has overall no positive effect on the effectiveness, we next employ work stealing in the five configurations. We compare the results of the configurations using work stealing given in the last block of Table 5.2 to the basic analyses (the first block) and to ranged program analyses without work stealing (the third block).

First and foremost, we observe that work stealing has a huge positive effect. In

[14]There are also cases where we observe that predicate abstraction or UltimateAutomizer do not finish the assigned range, but the other analysis does.



(a) RA-SE-PRED and WS-SE-PRED

(b) RA-VALUE-PRED and WS-VALUE-PRED

(c) RA-BMC-PRED and WS-BMC-PRED

(d) RA-SYMB-UA and WS-SYMB-UA

(e) RA-KLEE-UA and WS-KLEE-UA

Figure 5.14: Venn diagrams comparing the total number of correctly solved tasks using ranged program analysis with and without work stealing

comparison to the basic analyses, four of the five combinations of ranged program analysis now outperform both employed basic analyses. Only the combination WS-SE-PRED can solve in total 3 460 tasks, that are 281 tasks fewer than predicate abstraction (3 741), but in total 1 867 tasks more than symbolic execution.

For the other four combinations, each employed basic analysis benefits significantly from being combined with another analysis: WS-BMC-PRED solves in total 4 240 tasks, that are 958 tasks more than BMC (3 282) and 499 more than predicate abstraction (3 741), increasing the effectiveness by 29% respectively 13%. For WS-VALUE-PRED, that solves overall 3 930 tasks, we notice an increase by 699 respectively 189 tasks compared to value analysis and predicate abstraction, an increase by 22% respectively 5%. For the configuration WS-KLEE-UA, that solves in total 4 924 tasks using two basic analyses from off-the-shelf tools, we observe an increased effectiveness by 1 942 tasks (65%) compared to KLEE (2 982) and 684 tasks (16%) compared to ULTIMATEAUTOMIZER (4 240). A similar situation is noticed for WS-SYMB-UA, which combines the strengths of SYMBIOTIC and ULTIMATEAUTOMIZER to increase the effectiveness by 1 021 tasks (26%) and 699 tasks (16%) compared to SYMBIOTIC respectively ULTIMATEAUTOMIZER.

Although work stealing can increase the overall performance, there might also be some negative effects. As we execute three ranged analyses instead of two when employing work stealing, the resources additionally consumed by the ranged analysis that tries to steal the work may cause the complete ranged program analysis to run into a

timeout or increase the wall time taken for computing a solution. To analyze them, we now compare our five instances once with work stealing enabled and once without.

We present in Figure 5.14 Venn diagrams comparing the numbers of correct solved tasks for the five configurations. Each Venn diagram shown depicts on the left (in green) the number of tasks solved exclusively when using ranged program analysis without work stealing, on the right (in blue) the number of tasks solved exclusively when using work stealing and the intersection is labeled with the number of tasks solved by both. The positive effect of work stealing becomes again visible, as the instances using work stealing have a significantly higher effectiveness.

To also analyze, if work stealing leads to any negative effects, we have a look at the tasks that are solved only when work stealing is not employed. Then, the additional overhead of work stealing, i.e., of the third analysis instance running, has caused the ranged program analysis to exceed their memory or time limitations. Having a look at Figure 5.14, we observe that instances not using work stealing do only solve between 20 and 86 tasks, that are not solved when using work stealing. Compared to the increase of 702 to 1 098 tasks, when using work stealing, the few tasks not solved are negligible.

Finally, we summarize the results of the detailed analysis of the efficiency of ranged program analyses using conceptual different analyses given in Appendix A.3.2: In some cases, using work stealing increases the efficiency compared to one of the basic analyses employed, if the other one is significantly faster, as the faster analysis completes the analysis of both ranges faster than the slower basic analysis. For simple tasks, the overhead of ranged program analysis is observable but the effect decreases for more complex tasks. The two configurations Ws-Symb-UA and Ws-Klee-UA are for complex tasks faster than both employed basic analyses. Thus, enabling work stealing does not influence the efficiency of ranged program analysis negatively, but rather slightly increases it for some configurations.

Results

Combining two conceptually different analyses within a ranged program analysis pays off, in case that work stealing is used. With respect to effectiveness, we see that the combination of two best-performing basic analyses within ranged program analysis now outperforms the two basic analyses by 26% respectively 16%. Again, all five configurations of ranged program analysis can solve tasks that are not solved by both basic analyses. Moreover, work stealing has no significant negative effect in terms of efficiency compared to default ranged program analysis.

Table 5.3: Number of correct and incorrect verdicts reported by Ws-Symb-UA and the parallel portfolio Portf(Symb,UA) in comparison to the two basic analyses. The column *par. only* (parallel only) contains the number of tasks that are correctly solved by a Ws-Symb-UA and Portf(Symb,UA) but not by the basic analysis employed.

	correct				**incorrect**	
	overall	**proof**	**alarm**	**par. only**	**proof**	**alarm**
Symbiotic	3 918	2 232	1 686	-	77	1
UA	4 240	3 096	1 144	-	23	0
Ws-Symb-UA	4 939	3 223	1 716	62	85	0
Portf(Symb,UA)	5 724	3 789	1 935	0	77	1

5.5.5 RQ 4: How Does Ranged Program Analysis Compare to a Parallel Portfolio?

Evaluation Plan Next, we compare ranged program analysis with a parallel portfolio. Therefore, we build in CoVeriTeam a parallel portfolio running Symbiotic and UltimateAutomizer in parallel, called Portf(Symb,UA), and compare the effectiveness and efficiency with Ws-Symb-UA, the best-performing instance of ranged program analysis using work stealing.

Experimental Results We present in Table 5.3 the results of our experimental evaluation. When comparing the best-performing configuration of ranged program analysis and the parallel portfolio, we observe that the parallel portfolio solves 5 724 tasks, an increase of 785 tasks compared to Ws-Symb-UA (4 939). For 601 of these 785 tasks no range was generated and Ws-Symb-UA uses UltimateAutomizer as default analysis, which solves only 4 of these tasks. For the 184 tasks a range is generated, there are 81 tasks neither Ra-UA nor Ra-Symb solve, thus we do not expect Ws-Symb-UA to solve these tasks.

Although the overall effectiveness of Ws-Symb-UA is 16% lower than the parallel portfolio, one advantage of ranged program analysis becomes visible when comparing the number of tasks that are only solved by Ws-Symb-UA or the portfolio but not by any of the basic analysis. There are 62 tasks that are only solved by the ranged program analysis but not by the basic analyses. As the parallel portfolio executes both basic analyses in parallel, it solves no tasks not solved by one of the basic analysis. This result emphasizes the main difference between the two approaches: Ranged program analysis is a technique that uses cooperation to actually enable a combination of tools to solve tasks, that none can solve standalone. In contrast, the portfolio cannot solve any additional tasks, as the tools run side-by-side rather than working in cooperation.

To analyze the efficiency of the parallel portfolio, we present in Figure 5.15 a scatter plot comparing the wall time of Ws-Symb-UA and the parallel portfolio. Clearly, the parallel portfolio, returning the first answer computed is faster than the ranged program analysis and the additional overhead caused by using ranged program analysis is again observable. For small tasks solved by the portfolio in less than ten seconds, Ws-Symb-UA takes in the median the 2.6-fold time, for tasks solved in more than 20 seconds wall time the 1.4-fold time.

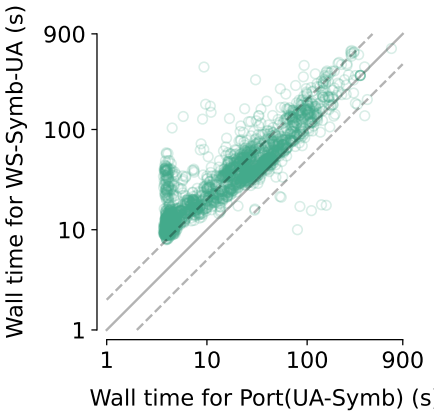


Figure 5.15: Scatter plot comparing the wall time of Ws-Symb-UA and the parallel portfolio

Results

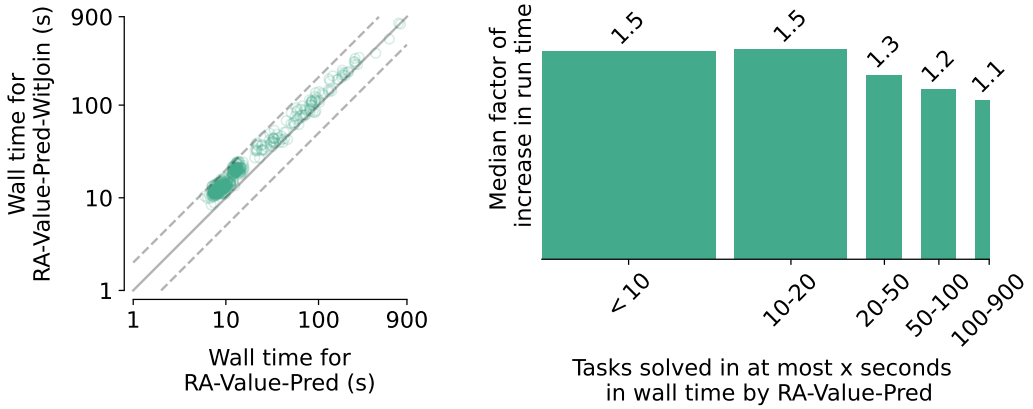
The parallel portfolio is on the benchmark even more effective than ranged program analysis. Nevertheless, the parallel portfolio does not solve any tasks not solved by the basic analyses, in contrast to the ranged program analysis.

5.5.6 RQ 5: Does Joining Witnesses Work?

Evaluation Plan Finally, we want to analyze whether the novel algorithm for witness joining generates correctness witnesses that can be validated. For witness validation, we follow the schema used in the SV-COMP and call a witness *validated*, if there is at least one validator accepting the witness. We employed the two best-performing correctness witness validators in SV-COMP'23, namely UA and CPAchecker. We are again interested in effectiveness and efficiency. As Algorithm 6 only joins correctness witnesses (violation witnesses do not need to be merged), we selected all correct tasks from the category `ReachSafety` for which the splitter Lb3 generates a range. From the remaining tasks, we select these tasks where both value analysis and predicate abstraction generate a correctness witness that is validated, yielding in total 469 tasks.

Experimental Results The ranged program analysis using value and predicate abstraction solve 463 tasks, the remaining 6 tasks are not solved by Ra-Value-Pred. We validated all generated witnesses using CPAchecker and UA, whereby 460 (99.4%) of the joined correctness witnesses where validated. The remaining 3 tasks are not validated, as the validator reaches the memory limitations during the validation. A manual inspection of the witnesses shows that these witnesses are also valid.

When comparing the effectiveness of Ws-Value-Pred to predicate abstraction and value analysis (cf. Table 5.2), we notice that there are 32 tasks only solved by the ranged program analysis, whereas 5 are additional proofs. Out of these 5 additional



(a) The scatter plot

(b) Median wall time increase of RA-Value-Pred with witness joining

Figure 5.16: Comparison of RA-Value-Pred with and without witness joining

proofs 4 are validated. One additional proof is not validated by the validators, as they reach the memory limitations during validation.[15]

Beneath the quality of the joined verification witnesses, we are also interested in the additional overhead caused by the application of the witness join. We compare in Figure 5.16a the overall time taken by RA-Value-Pred, where we depict on the x-axis the time of the analysis when the joining of the witnesses is disabled and on the y-axis with enabled witness joining. As expected, we observe that computing the joined correctness witness increases the overall time. For a more detailed analysis, we group the tasks by the time taken by the analysis without witness join and compute for each group the median increase in the run time, visualized in Figure 5.16b. We notice that the additional overhead caused by joining the witnesses decreases for tasks that need more time to solve. For tasks that take more than 100 seconds to solve, the additional time needed to compute a solution is only around 10% of the overall computation time.

Results

The algorithm for witness joining proposed in Algorithm 6 works, as it generates witnesses that are in more than 99% of the cases successfully validated.

5.6 Discussion

Our experimental evaluation of ranged program analysis demonstrates that dividing the verification task using ranges is possible for arbitrary off-the-shelf verifiers, thereby allowing them to cooperate in solving the verification task. Using conceptually different approaches within ranged program analysis allows for solving tasks, none of the basic analysis can solve standalone. Especially the use of work stealing within ranged

[15] Due to the size of the generated witness, a manual inspection is infeasible.

program analysis allows for building cooperative analyses that have a higher effectiveness compared to the basic analyses employed. Next, we first discuss the validity of our experimental evaluation before discussing the concept of ranged program analysis itself.

We have conducted the experiments on the SV-Benchmarks from 2023 [SVB23]. Although it is widely used, especially in the SV-COMP, our findings may not completely carry over to other real-world C programs or to other programming languages. Currently, the instrumentation does not cover concurrent programs. Moreover, we do not support external functions, as the source code is needed for instrumentation.

It is unlikely that the implementation suffer from bugs. We randomly selected a subset of the tasks for which the ranged program analyses compute incorrect results, that are not computed by the basic analyses employed. For all tasks, we confirmed that the incorrect result was not caused by the ranged program analysis. In addition, we evaluated in [HJRW23c] symbolic execution in ranged program analysis using instrumentation and range reduction, obtaining comparable results.

As observed in Section 5.5.2, the splitting strategy used can have a significant effect on the performance of ranged program analysis. We did not evaluate each of the four splitting strategies for all configurations of ranged program analysis. In addition, we did not evaluate the effect of a different assignment of the ranges for instances of ranged program analysis using two different analyses. It might be the case that certain combinations of ranged program analyses perform even better when a different splitting strategy is used. In that case, the findings from our evaluation and conclusions drawn still remain valid.

The Algorithm 6 for joining witnesses is currently only applicable in combination with range reduction. As the justification generated by verifiers used in combination with instrumentation refers to program location in the instrumented program, joining them directly is currently not supported. Combining correctness witnesses when using instrumentation is planned as future work.

The comparison of ranged program analysis and the parallel portfolio has shown that the parallel portfolio is currently more effective. Nevertheless, ranged program analysis opens up new possibilities compared to a parallel portfolio, as it does not only execute several ranged analyses in parallel but let them cooperate. In fact, ranged program analysis can be seen as a novel form of cooperation. First, it may ease the verification of tasks, such that tasks can be solved that are not solvable by the basic analyses. Second, it allows for using different splitting strategies and allows for parallelization of analyses that do not offer such a mechanism by default. Siddiqui and Khurshid successfully scaled symbolic execution by using up to 20 instances in parallel and not limiting the overall consumed CPU time [SK12]. We used in our evaluation the same resource limitations for each tool. In case the available resources are not limited, ranged program analysis allows for using ten, 20, or even more instances in parallel. Investigating to which extent such a heavy parallelization positively influences the ef-

fectiveness when using other off-the-shelf analysis techniques is another interesting line of future work.

In general, the experimental evaluation shows that using two instances of the same analysis within a ranged program analysis is only beneficial for analysis techniques that work path-based. For analyses that employ abstraction, the effectiveness decreases when using two instances in parallel. An in-depth analysis to identify causes for the reduced effectiveness may allow circumventing such situations, e.g., investigating to which extent the ranged analyses are influenced by the instrumentation.

In some cases we observe that one range generated by Lb3 is easy or even trivial to solve, hence one analysis is significantly faster than the other one. Work stealing can reduce the effect of trivial ranges but does not solve the underlying problem. Hence, defining a splitter that can generate more balanced ranges may further increase the positive effect of using ranged program analysis. To be able to employ multiple ranged analyses in parallel, novel splitting strategies allowing to generate multiple bounds are needed. Developing and evaluating these novel splitting strategies is another line of future work: Currently, the splitter does not use additional information obtained when analyzing the program. For example, generating ranges using constants from the program or generating ranges that contain certain features allows for using more specialized analysis techniques. Moreover, using algorithm selection to select the most promising ranged analysis per range is another possibility for increasing the performance of ranged program analysis.

5.7 Related Work

In this chapter, we focus on generalizing the idea of ranged symbolic execution as a general cooperative verification approach. In the following, we discuss other (non-cooperative) parallel software verification approaches and have a look at methods for partitioning the search space of analyses, load balancing, aggregation of analysis results, and program instrumentation.

5.7.1 Parallel Combinations

Next, we discuss concepts that combine the strengths of different approaches and exchange information between them in parallel, either cooperatively or using conceptual integration.

5.7.1.1 Conceptual Integrations

Many approaches [TFNM11; BL19a; Gro+12; Hol+16; HJG08; YDLW19; CH23; BKR22; Luc+16] employ a parallel portfolio, the easiest way of combing different techniques. Therein, all tools work on the same task in parallel without cooperation and the available resources are shared among them. The portfolio usually returns the first

answer computed. In [MH21], four different strategies for portfolio-based SMT-solving are presented, e.g., using the fastest result, majority vote, or conditioned verdict validation.

More advanced concepts let the components work together by exchanging and using information. Sage [GLM08] uses a similar idea as ranged symbolic execution, but instead of using random paths, work is shared among the instances of a dynamic symbolic execution engine dynamically. Starting with an initial test input, the program is executed and the conditions on the path are collected. Next, the conditions are systematically negated and all resulting test cases are processed in parallel. The dynamic symbolic execution instances share the unprocessed inputs, where each defines an unexplored program path. A combination of dynamic symbolic execution and fuzzing can be found in tools like Driller [Ste+16] or HyDiff [Nol+20], where both tools are executed in parallel and exchange information on computed inputs periodically. Parallel execution of multiple analyses, where each analysis can access and use the information computed by other analyses is theoretically proposed by Cousot and Cousot [CC79], and realized in different verification tools as CPAchecker [BHT07], especially its k-induction [BDW15], or Astrèe [Cou+06]. In contrast to these parallel approaches, ranged program analysis uses the ranges to avoid program paths being analyzed by several analyses. Additionally, using program instrumentation, arbitrary off-the-shelf tools can be used within the ranged program analysis.

5.7.1.2 Cooperative Approaches

Inspired by the idea of ranged program analysis, Chalupa and Richter propose Bubaak-SpLit [CR24], a tool that dynamically splits the program. Instead of computing bounds in advance, Bubaak-SpLit splits the program on demand, i.e., in case Klee, a lightweight and fast analysis that is employed first, fails to solve the verification (sub-)task in a given time limit. Thereafter, the program is split at the first branching node into two parts and two range programs are generated using instrumentation. The lightweight analysis again tries to verify the two range programs. In case it fails, the programs are split again. This process is repeated until a fixed number of programs that are not solved by Klee and thus claimed to be hard to verify are generated. These programs are then analyzed by Bubaak and Slowbeast in parallel. Compared to ranged program analysis, the dynamic splitting strategy of Bubaak-SpLit works without a splitter and aims at generating ranges that are hard to verify, avoiding trivial ranges.

Orthogonal to the idea of range program analysis and based on CMC, Franke proposes parallel conditional model checking with multiple ranged conditions [Fra23]. Instead of executing several conditional model checkers sequentially, the condition automata are generated based on paths, allowing for an analysis of the program with all condition automata in parallel. Compared to ranged program analysis, the condition automata have the same function as test cases or sequences of branching decisions.

5.7.2 Search Space Partitioning

Partitioning the search space allows for a parallel execution of multiple tools exploring different parts of the program and thereby avoids redundant computations. To be able to partition the search space, it has to be dividable into a finite number of partitions, e.g., if the search space has finitely many elements or a total order. Then, a partition can either be computed dynamically during the analysis based on intermediate results or statically in advance. In the former case, the search space is usually partitioned based on intermediate results into an already processed and a remaining part that needs to be analyzed. To be able to dynamically define the remaining parts, a characterization of the search space is needed. Among others, test goals [BL19a; AABC21], open proof obligations [Hus+17], program paths [BJ20; BJ21; BJLW18; CMW16; CMW12; CJW15; DGH16; FWCD17; GD19; JVSJ06; CR24], or the already explored symbolic execution tree [BUZC11; Cio+09; Qiu+18; SK12; ZGQH13; Wei+23] can be used for partitioning the space. In contrast to the dynamic splitting strategies, ranged program analysis uses a splitter for computing static splits of the program.

Static partitioning aims to divide the overall task in advance, such that each element can (potentially) be solved in parallel. Such a partitioning can be realized by defining subtasks [PW19; YDW15; BL19a], according to special analysis capabilities [PW19], or use a static functions for state partitioning [BF18; SD97; LS99; BBC03]. Splitting the program paths is another way of statically partitioning a program. Therefore, an ordering or characterization of a set of paths is needed. Conditional static analysis [SD18] uses the order of executed program branches, some approaches that aim for scaling symbolic execution define path prefixes [SK20; SK21; FSK12]. The approach proposed by Staats and Pasareanu [SP10] uses an initial run of symbolic execution to collect path constraints, that are next used to define input constraints. Each symbolic execution running in parallel uses a specific input constraint. Similarly, Korat uses input ranges defined via predicates for partitioning, where several instances of Korat process the input ranges in parallel in order to generate test inputs using a systematic search [Mis+07]. Ranged symbolic execution employs path ranges for defining the ranges. In contrast to path ranges, input ranges are not guaranteed to induce a set of ordered paths. Hence the range reduction may not work. In contrast to path prefixes, path ranges are more precise and can express ranges not expressible using a path prefix. Path ranges are also employed by different approaches employing ranged symbolic execution, either generated using an initial, shallow symbolic execution [SK20; SK21], random selection [SK12] or tests [Qiu+18; Yan+19]. In contrast, ranged program analysis introduces a splitter as an individual component, allowing it to realize different strategies as limiting the number of loop unrollings. Moreover, we also allow using sequences of branching decisions to describe ranges.

5.7.3 Load Balancing

Having an even distribution of work among the analyses running in parallel allows for a faster computation of the final result. One way is using a static partitioning [Mis+07; SD97; LS99; GMS01; BBC03]. As estimating the workload in advance is a challenging task, several approaches [SK20; SK12; SK21; Yan+19] assign new tasks to idle workers, just as our work stealing. Dynamic range refinement [Yan+19] computes the range that is not explored and splits it into two ranges, where one of these ranges is reassigned to an idle worker. Other approaches do not simply reassign a predefined task as work stealing does but rather divide assigned tasks into multiple subtasks and reassign some subtasks that are not completed. For example, these approaches redistribute the analysis of one subtree [SK20; SK12; SK21], several subtrees [BUZC11; Cio+09] or several states [KM04] that need to be analyzed. All of the approaches that reassign work make use of a white-box integration and employ only one specific verification technique, mostly symbolic execution. Thereby, they are able to access the internal state of a worker and can identify parts of the state space that are assigned to the worker but not analyzed so far. This insight allows for a redistribution of unexplored program parts. In contrast, ranged program analysis goes beyond these concepts, as it is able to use arbitrary off-the-shelf verifiers and distribute the ranges among them. As a downside, we are not able to access their internal states and know how the employed tools work, prohibiting a dynamic load balancing that is more fine grain than the predefined ranges.

5.7.4 Result Aggregation

Whenever verification approaches jointly work on a verification task, the (partial) results need to be aggregated. Most important and rather simple is the aggregation of verdicts. In case the combination only employs techniques that under-approximate the behavior of the program, as in [Hol+16; ARCB14; Ngu+17; IT20], the only result typically reported are property violations found by one of the employed analyses. The counterexample reported is also typically forwarded from the analysis that raises the alarm, as this requires no aggregation. In contrast, combinations employing under- and over-approximating tools [BNRS08; Gul+06; MPV15; AGC12] report the counterexamples computed by the under-approximating components and the proofs generated by the over-approximating once. In case parallel portfolios are employed [CH23; BKR22; Luc+16; BL19a; Gro+12; HJG08], the first computed result is returned. Whenever the search space is partitioned among different analyses [BHKW12; BJLW18; CMW16; CMW12; CJW15; DGH16; FWCD17; GD19; HJRW23c], as we also do in ranged program analysis, found errors are returned directly, while the program is reported correct only in case that all analyses have succeeded. In contrast, Scar [YDLW19] outputs a quantitative value that describes the (in)correctness for each of the computed results.

Instead of only combing computed verdicts and forwarding computed counterexam-

ples, combining partial justifications of the correctness of a program is discussed rarely. Jakobs proposes a method for combining partial analysis results in the form of partial ARGs [Jak17]. The combined ARG can be used for generating a correctness witness. Garavel et al. propose a method for combining partial labeled transition systems, that are obtained by distributive explicit state space exploration [GMS01]. In contrast to these two concepts, the witness join presented in Algorithm 6 allows for combining correctness witnesses generated for program ranges. It is independent of the ranges and can thus be employed for the combination of arbitrary partial witnesses, especially for tools that do not generate ARGs.

5.7.5 Program Instrumentation

Program instrumentation is used in several areas of software verification or testing. In testing and test case generation, instrumentation allows for collecting information on executed parts of a program when running test cases to compute their test coverage. The collected information and measured coverage are then used for guiding the execution, e.g., in gray-box fuzzers as AFL [Zal13; FMMB23], libFuzzer [Pro24], or Fizzer [JSTU24a; JSTU24b], or to rate the quality of the set of test cases, e.g., in competitions as TEST-Comp [BL19b; Bey22a].

In contrast to testing, instrumentation allows in the area of software verification not only to guide certain components but is also employed to ease the verification task, e.g., by Symbiotic. Symbiotic uses the idea of instrumenting the code with runs of state machines that represent the safety properties to retain only the statements affecting the state machine using program slicing [SST12]. By reducing the number of statements or paths in the program, thus the verification task is simplified. The sliced program is then analyzed by Klee [SST13] or a composition of different LLVM-based tools [CS19]. In contrast, ranged program analysis employs instrumentation to split the work among different analyses, rather than removing non-relevant code fragments. In [CMW16], program instrumentation is used for guiding dynamic symbolic execution, e.g., to abort the exploration of program parts that are already verified or to guide it to not fully verified parts. The information employed for instrumenting and thus guiding the dynamic symbolic execution is inferred from partial verification results computed by a verifier, which is then annotated in the program code. In comparison to ranged program analysis, this approach uses partial results for guiding tools rather than sharing the work among different verifiers working in parallel.

6 Conclusion

The goal of this thesis was to conduct a systematic analysis of the forms of cooperative software verification and their potential benefits. Therefore, we developed three different concepts for cooperative verification, one that sequentially combines the actors, one that cyclically combines them, and one that combines them in parallel. In the following, we summarize the results in Section 6.1 of this thesis, discuss them, and provide an outlook for future work in Section 6.2, and present a resume in Section 6.3.

6.1 Summary

The first contribution of this thesis is CoVEGI, a concept for sequentially combining actors in a cooperative manner. More precisely, it enables a main verifier to request and receive invariants generated by helper invariant generators. The invariants are exchanged using invariant witnesses. In order to use arbitrary off-the-shelf invariant generators, CoVEGI foresees the use of a mapper and an encoder to transform the invariants given in tool-dependent formats into invariant witnesses. The experimental evaluation demonstrated that the use of CoVEGI can increase the effectiveness of the two main verifiers, resulting in an increased number of correctly solved tasks. Moreover, it does not significantly negatively influence efficiency. Furthermore, we presented MIGml, a modular framework that allows for generating loop invariants through machine learning, and experimentally demonstrated its feasibility.

As a second contribution, we have developed a modular version of the verification schema CEGAR. The decomposed version C-CEGAR comprises three standalone components that are cooperatively solving the verification task and communicating via the verification artifacts invariant witness, violation witness, and path witness. For the experimental evaluation, we decomposed an existing CEGAR instance into three standalone components. This demonstrated that the decomposition is indeed possible. The evaluation showed that the effectiveness of the decomposed version and the white-box

instance are comparable, in case formats without loss of information are used. The decomposition entails additional overhead, which is found to be bounded by a constant factor. The effectiveness of the decomposed version decreases by around 20% in case standardized verification artifacts are used, as not all information computed is encoded within the artifacts. More importantly, we demonstrated that C-CEGAR makes the integration of novel, off-the-shelf components a configuration task, allowing us to easily integrate novel ideas and thereby solve tasks that could not be solved before. Furthermore, we developed and evaluated a novel verification artifact called GIA, which is applicable in many different scenarios in cooperative software verification and testing while maintaining one fixed semantics.

The third contribution of this thesis is ranged program analysis, an approach for a parallel cooperation of several verifiers. The approach is based on the concept of divide-and-conquer and allows for dividing the verification task along path ranges into independent subtasks, each solvable in parallel by ranged analyses. We proposed two different techniques for restricting an off-the-shelf analysis to a given range. The first uses range reduction for CPA-based analyses, the other program instrumentation. The experimental evaluation showed that using two instances of the same verifier allows for solving tasks that could not be solved before and increases the overall effectiveness of symbolic execution. Combinations of two conceptually different analyses in parallel, especially when using work stealing, leads to an increased effectiveness. The best-performing instance yields an increase of 26% respectively 16% compared to the two verifiers running standalone.

In summary, we conclude that cooperative software verification can employ sequential, cyclic, and parallel forms of cooperation. Moreover, we see combining different approaches with individual strengths in a cooperative way is beneficial, as all three concepts can solve tasks that cannot be solved without cooperation.

6.2 Discussion and Outlook

As shown by using the three defined concepts, the use of cooperative software verification is beneficial, especially as it allows to solve tasks that cannot be solved without cooperation. Finally, we discuss the concepts on a high level, highlight directions for future work, and point out open challenges.

GraphML-based and Yaml-based Exchange Formats As shown within this thesis, the use of cooperative software verification is beneficial. In order to successfully establish cooperation, communication and information exchange is the key to success. Missing information or the inability of a participant to process the information is therefore a major obstacle. Hence, the use of well-suited exchange formats plays a crucial role in the design of cooperative software verification concepts.

CoVEGI and C-CEGAR use GraphML-based correctness witnesses and violation witnesses for the information exchange. The witnesses and GIAs establish the connection between the information encoded in the artifacts and the program based on the CFA. Although the GraphML format has been used in SV-COMP since 2015, there is no standardized transformation from a C program to a CFA, and thus the semantics of the format have some ambiguities [Aya+24; Str22]. To avoid these ambiguities, the recently published Yaml-based witness format 2.0 is no longer graph-based, its semantics are formulated directly using terms and concepts from the programming language. It associates invariants directly with program location and its semantics require that these invariants hold on all executions [Aya+24]. In particular, in C-CEGAR, we do not use correctness witnesses, but invariant witnesses, which have different semantics than correctness witnesses. An invariant witness does not necessarily have to contain a proof that the complete program is correct, it suffices to prove the infeasibility of a single path. For example, such a path can contain multiple loop iterations and the computed precision increment is valid only in the second iteration (e.g., as in Figure 4.19a). Using the Yaml-based witness format is hardly possible in this specific situation. The same is true for GIAs, where encoding information about the safety of a single path is a key property. This is the main reason why we decided to use graph-based exchange formats in this thesis. Note that the semantics of no versions of correctness and violation witnesses allow encoding partial results and that we use modified semantics for invariant and path witnesses. Thus, the Yaml-based format may also be used in cooperative verification if formats with modified semantics are designed.

Using Artifacts with Off-the-shelf Tools Another major challenge in cooperative software verification is the use of off-the-shelf tools. Most of the artifacts used are graph-based and establish the connection to the program via the CFA, such as the GraphML-based correctness and violation witnesses, condition automata, or GIAs. In contrast, off-the-shelf tools can (usually) only work with C programs as inputs. Thus, there are several concepts, e.g., reducer as in Section 4.2.4 and [BJLW18], program instrumentation as in Section 5.3.4 and [CR24], or MetaVal [BS20] to encode the information from verification artifacts into the program. The results of these transformations are residual programs, reduced programs, path programs, or range programs. Unlike the graph-based formats, the Yaml-based witness format "attaches" information to the program and Christakis et al. store the assumptions directly in the program code [CMW12]. Chalupa and Richter exchange range programs that only contain the remaining task [CR24]. In our point of view, it is worth studying whether it is possible and beneficial to encode all computed information directly into the program (or "attach" it, as in the Yaml-based witnesses) instead of designing additional artifacts. Then, each component working in cooperative verification returns an (annotated) program that contains the remaining task as output along with verdict and justification. Such a significant reformulation of the concept of cooperative verification leads to new

challenges. One challenge is to ensure that all transformations performed during cooperative verification, such as instrumentation, are reverted, in order to compute a justification that is valid for the original program.

Combination of Artifacts In general, most actors in cooperative verification approaches communicate partial results, e.g., a loop invariant, the successful verification of a range, or a precision increment for a specific path. To be able to provide a proper justification for the cooperatively computed verdict, the information computed by each actor needs to be collected and combined. As new concepts for cooperative verification may necessitate joining any artifact used, it is useful to design methods for combining them, especially as such methods are specific to the format used and independent of the use case. In this thesis, we already proposed methods for joining GIAs and correctness witnesses in the GraphML format, in order to generate justifications in ranged program analysis when using range reduction. In case the program is instrumented, joining correctness witnesses is only described conceptually in Section 5.3.5. For other formats, there currently exists no method for joining them. In general, artifacts associating information directly to source code locations, like Yaml-based correctness witnesses, are well-suited for being joined, as joining them does not require computing a combination of two automata. As the Yaml-based witness format has been included in SV-COMP since 2024 and there are already eight participants who generate these witnesses, a method for joining the Yaml-based witness format can be developed in the future. Additionally, the combination of correctness witnesses generated for instrumented programs in ranged program analysis is another direction of future work.

Information Exchange in General Beneath the format used for exchanging information, another crucial point in cooperative verification is the exchanged information itself. Clearly, if tools do not produce meaningful information as output, they are not well-suited for being used in cooperative approaches. For example, invariant generators that produce only trivial invariants or precision refiners that do not include the computed precision increment in the invariant witness hamper cooperative verification. Thus, exchange formats that provide an easy way to encode all computed information can lead to higher-quality artifacts. In the future, competition may also motivate tool developers to enhance the output of their tools, e.g., by giving higher scores to more meaningful outputs. Recently, Beyer et al. showed that a too precise description of a counterexample can impede the validation of violation witnesses [BKL24b]. Thus, it is worth investigating whether these findings also carry over to cooperative verification approaches, i.e., to analyze which level of precision of computed information in verification artifacts leads to the best results.

Alternatives to Loop Invariants in CoVEGI In CoVEGI, invariants are exchanged among the main verifier and helper invariant generator. As helpful (loop)

invariants ease the verification task, they are widely used and ideal for showing the program's correctness [BS22]. Thus, they are well-suited in CoVEGI. Nevertheless, we could also design CoVEGI to exchange different analysis information as arbitrary location invariants that are not related to loops, as in the Yaml-based witness format [Aya+24] or summaries of loops or functions [SM12; SFS11; BS19; BKL24a]. Such summaries do not take the current program state when entering the loop or a function into account, but describe the effect of their execution on the variables more generally. Thus, loop or function summaries can be used for all occurrences of a loop or function call in the program, whereas loop invariants have to be computed for each loop. As a downside, there is currently no standardized format for exchanging loop or function summaries, and expressing them in terms of C expressions is more difficult than expressing invariants. Nevertheless, it is interesting to analyze, if CoVEGI can be updated to also use externally generated loop or function summaries.

Range Generation within Ranged Program Analysis The idea of ranged program analysis employs static slitting to generate the ranges. This allows us to generate the ranges upfront and solve them in parallel.

Dynamic splitting strategies identify parts of the program that are hard to solve based on the internal progress of a verifier. Using such strategies in a cooperative setting is not possible for two reasons. First, all components are used as black-box, hence accessing their internal state is not possible. Second, they typically do not output intermediate results. Based on the idea of ranged program analysis, Chalupa and Richter use a different concept for dynamic splitting with program instrumentation [CR24]. Instead of splitting the task based on the internal states of the verifier, they split the program in advance and let a verifier try to solve the task for a short time. If no result is computed, the task is considered hard and is split again. By repeating this process until a fixed number of hard tasks is generated, several off-the-shelf verifiers can be used in parallel afterward. The main advantage of their approach is that no trivial ranges are generated. As a disadvantage, the splitting strategy employed is simple, namely splitting the program at the first branching point. In ranged program analysis, the number of ranges that need to be generated by the splitter is known in advance. Hence, we can define a splitter that generates ranges containing a specific number of loop iterations.

Other Safety Properties Throughout this thesis, we have focused on the verification of the (non)-reachability of error locations. There are also other safety properties that can be verified. For example, the SV-COMP defines for C programs a total of six different properties, namely (1) unreachability of error functions, (2) memory safety (no invalid deallocations, pointer dereferences or untracked allocated memory), (3) memory cleanup (allocated memory is deallocated), (4) no overflow, (5) no data-race and (6) termination. The termination and no overflow properties can be translated to an

equivalent reachability task [SB05; Bai+24]. In our opinion, analyses aiming to show memory-related properties can also be used within ranged program analysis, as long as the program itself is sequential, as these properties are checked per path.

For parallel programs, e.g. for detecting data-race, there are some open challenges: CoVEGI and C-CEGAR use invariant witnesses to exchange invariants or precision increments. Beyer et al. argued that a concurrent CFA can be used for multi-threaded programs [Bey+22]. To address these open problems, it can be analyzed whether invariant generators and precision refiners compute invariants for concurrent programs, encode them correctly in the invariant witness, and whether the main verifier and model explorer can use this information. Using ranged program analysis for parallel programs is more challenging. The program execution tree for defining the ranges does currently not take the execution of multiple threads in parallel into account. In addition, the ordering relation $\leq$ is also defined for programs with a non-parallel CFA. Investigating, whether ranged program analysis can be modified to work with parallel programs, e.g., modifying $\leq$ to work with concurrent CFAs, is another challenge for the future. Ideally, ranged program analysis allows the analysis of different interleavings of multiple threads by different off-the-shelf verifiers.

Use of LLMs in Cooperative Software Verification LLMs like ChatGPT have made enormous progress in the last years and are used in many domains, e.g., for the generation of invariants [JRW24; Cha+23; Kam+23]. As these approaches show promising results, it is an interesting topic for future work to investigate to which extent these invariant generators are also applicable in CoVEGI. In addition, there are also other tasks for LLMs that may be useful apart from invariant generation. For example, ChatGPT is able to explain the program code given as input. Thus, certain LLMs may also be able to explain why a location of a program is unreachable. Thus, it is worth investigating, to which extent LLMs can be used as a precision refiner within C-CEGAR or to generate ranges for ranged program analysis.

6.3 Resume

In this thesis, we presented a systematic analysis of potential forms of cooperative verification and the advantages of using them. We developed three different concepts for cooperative software verification: The sequential concept CoVEGI, the cyclic concept C-CEGAR, and the parallel concept of ranged program analysis. The experimental evaluation demonstrates that the combination of actors in a black-box manner allows for solving tasks that could not be solved without cooperation. Hence, the results of this thesis provide a fundamental contribution towards cooperative software verification. Open challenges and directions for future work identified may lead to novel forms of cooperative verification or further increase the effectiveness and efficiency of existing approaches.

Bibliography

[AABC21] Kaled M. Alshmrany, Mohannad Aldughaim, Ahmed Bhayat, and Lucas C. Cordeiro. "FuSeBMC: An Energy-Efficient Test Generator for Finding Security Vulnerabilities in C Programs". In: *Proc. TAP*. Vol. 12740. 2021, pp. 85–105. DOI: 10.1007/978-3-030-79379-1_6.

[Afz+19] Mohammad Afzal, A. Asia, Avriti Chauhan, Bharti Chimdyalwar, Priyanka Darke, Advaita Datar, Shrawan Kumar, and R. Venkatesh. "VeriAbs : Verification by Abstraction and Test Generation". In: *Proc. ASE*. 2019, pp. 1138–1141. DOI: 10.1109/ASE.2019.00121.

[AGC12] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. "From Under-Approximations to Over-Approximations and Back". In: *Proc. TACAS*. Vol. 7214. 2012, pp. 157–172. DOI: 10.1007/978-3-642-28756-5_12.

[Ald+23] Mohannad Aldughaim, Kaled M. Alshmrany, Mikhail R. Gadelha, Rosiane de Freitas, and Lucas C. Cordeiro. "FuSeBMC_IA: Interval Analysis and Methods for Test Case Generation - (Competition Contribution)". In: *Proc. FASE*. Vol. 13991. 2023, pp. 324–329. DOI: 10.1007/978-3-031-30826-0_18.

[Alu+13] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. "Syntax-guided synthesis". In: *Proc. FMCAD*. 2013, pp. 1–8. DOI: 10.1109/FMCAD.2013.6679385.

[AMK18] Pavel Andrianov, Vadim Mutilin, and Alexey Khoroshilov. "Predicate Abstraction Based Configurable Method for Data Race Detection in Linux Kernel". In: *Proc. TMPA*. 2018. DOI: 10.1007/978-3-319-71734-0_2.

[Ang87] Dana Angluin. "Learning Regular Sets from Queries and Counterexamples". In: *Inf. Comput.* 75.2 (1987), pp. 87–106. DOI: 10.1016/0890-5401(87)90052-6.

[ARCB14] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. "Enhancing Symbolic Execution with Veritesting". In: *Proc. ICSE*. 2014, pp. 1083–1094. DOI: 10.1145/2568225.2568293.

[AS23] Paulína Ayaziová and Jan Strejček. “Symbiotic-Witch 2: More Efficient Algorithm and Witness Refutation - (Competition Contribution)”. In: *Proc. TACAS*. Vol. 13994. 2023, pp. 523–528. DOI: 10.1007/978-3-031-30820-8_30.

[AS24] Paulína Ayaziová and Jan Strejček. “Witch 3: Validation of Violation Witnesses in the Witness Format 2.0 - (Competition Contribution)”. In: *Proc. TACAS*. Vol. 14572. 2024, pp. 341–346. DOI: 10.1007/978-3-031-57256-2_18.

[Ass20] Association for Computing Machinery (ACM). *Artifact Review and Badging Version 1.1*. 2020. URL: https://www.acm.org/publications/policies/artifact-review-and-badging-current (visited on 09/15/2021).

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Aya+24] Paulína Ayaziová, Dirk Beyer, Marian Lingsch-Rosenfeld, Martin Spiessl, and Jan Strejček. “Software Verification Witnesses 2.0”. In: *Proc. SPIN*. 2024.

[Bai+24] Daniel Baier, Dirk Beyer, Po-Chun Chien, Marek Jankola, Matthias Kettl, Nian-Ze Lee, Thomas Lemberger, Marian Lingsch Rosenfeld, Martin Spiessl, Henrik Wachowitz, and Philipp Wendler. “CPAchecker 2.3 with Strategy Selection - (Competition Contribution)”. In: *Proc. TACAS*. Vol. 14572. 2024, pp. 359–364. DOI: 10.1007/978-3-031-57256-2_21.

[Bal+18] Roberto Baldoni, Emilio Coppa, Daniele Cono D’Elia, Camil Demetrescu, and Irene Finocchi. “A Survey of Symbolic Execution Techniques”. In: *ACM Comput. Surv.* 51.3 (2018), 50:1–50:39. DOI: 10.1145/3182657.

[Bau+21] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. “The dogged pursuit of bug-free C programs: the Frama-C software analysis platform”. In: *Commun. ACM* 64.8 (2021), pp. 56–68. DOI: 10.1145/3470569.

[Bau+24] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*. 2024. URL: http://frama-c.com/download/acsl.pdf.

[BBC03] Jiří Barnat, Luboš Brim, and Jakub Chaloupka. “Parallel Breadth-First Search LTL Model-Checking”. In: *Proc. ASE*. 2003, pp. 106–115. DOI: 10.1109/ASE.2003.1240299.

[BBF21] Daniel Baier, Dirk Beyer, and Karlheinz Friedberger. “JavaSMT 3: Interacting with SMT Solvers in Java”. In: *Proc. CAV*. Vol. 12760. 2021, pp. 195–208. DOI: 10.1007/978-3-030-81688-9_9.

[BD20] Dirk Beyer and Matthias Dangl. "Software Verification with PDR: An Implementation of the State of the Art". In: *Proc. TACAS*. Vol. 12078. 2020, pp. 3–21. DOI: 10.1007/978-3-030-45190-5_1.

[BDDH16] Dirk Beyer, Matthias Dangl, Daniel Dietsch, and Matthias Heizmann. "Correctness witnesses: exchanging verification results between verifiers". In: *Proc. FSE*. 2016, pp. 326–337. DOI: 10.1145/2950290.2950351.

[BDFW08] Ingo Brückner, Klaus Dräger, Bernd Finkbeiner, and Heike Wehrheim. "Slicing Abstractions". In: *Fundam. Inform.* 89.4 (2008), pp. 369–392.

[BDLT18] Dirk Beyer, Matthias Dangl, Thomas Lemberger, and Michael Tautschnig. "Tests from Witnesses - Execution-Based Validation of Verification Results". In: *Proc. TAP*. Vol. 10889. 2018, pp. 3–23. DOI: 10.1007/978-3-319-92994-1_1.

[BDMP17] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. "Combining symbolic execution and search-based testing for programs with complex heap inputs". In: *Proc. ISSTA*. 2017, pp. 90–101. DOI: 10.1145/3092703.3092715.

[BDW15] Dirk Beyer, Matthias Dangl, and Philipp Wendler. "Boosting k-Induction with Continuously-Refined Invariants". In: *Proc. CAV*. Vol. 9206. 2015, pp. 622–640. DOI: 10.1007/978-3-319-21690-4_42.

[BDW18] Dirk Beyer, Matthias Dangl, and Philipp Wendler. "A Unifying View on SMT-Based Software Verification". In: *J. Autom. Reason.* 60.3 (2018), pp. 299–335. DOI: 10.1007/S10817-017-9432-6.

[Bey+13] Dirk Beyer, Stefan Löwe, Evgeny Novikov, Andreas Stahlbauer, and Philipp Wendler. "Precision reuse for efficient regression verification". In: *Proc. ESEC/FSE*. 2013, pp. 389–399. DOI: 10.1145/2491411.2491429.

[Bey+15] Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, and Andreas Stahlbauer. "Witness validation and stepwise testification across software verifiers". In: *Proc. ESEC/FSE*. 2015, pp. 721–733. DOI: 10.1145/2786805.2786867.

[Bey+22] Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, Thomas Lemberger, and Michael Tautschnig. "Verification Witnesses". In: *ACM Trans. Softw. Eng. Methodol.* 31.4 (2022), 57:1–57:69. DOI: 10.1145/3477579.

[Bey20] Dirk Beyer. "Advances in Automatic Software Verification: SV-COMP 2020". In: *Proc. TACAS*. Vol. 12079. 2020, pp. 347–367. DOI: 10.1007/978-3-030-45237-7_21.

[Bey22a] Dirk Beyer. "Advances in Automatic Software Testing: Test-Comp 2022". In: *Proc. FASE*. Vol. 13241. 2022, pp. 321–335. DOI: 10.1007/978-3-030-99429-7_18.

[Bey22b] Dirk Beyer. "Progress on Software Verification: SV-COMP 2022". In: *Proc. TACAS*. Vol. 13244. 2022, pp. 375–402. DOI: 10.1007/978-3-030-99527-0_20.

[Bey23] Dirk Beyer. "Competition on Software Verification and Witness Validation: SV-COMP 2023". In: *Proc. TACAS*. 2023, pp. 495–522. DOI: 10.1007/978-3-031-30820-8_29.

[Bey24] Dirk Beyer. "State of the Art in Software Verification and Witness Validation: SV-COMP 2024". In: *Proc. TACAS*. Vol. 14572. 2024, pp. 299–329. DOI: 10.1007/978-3-031-57256-2_15.

[BF18] Dirk Beyer and Karlheinz Friedberger. "Domain-independent multi-threaded software model checking". In: *Proc. ASE*. 2018, pp. 634–644. DOI: 10.1145/3238147.3238195.

[BFOS84] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984.

[BG99] Glenn Bruns and Patrice Godefroid. "Model Checking Partial State Spaces with 3-Valued Temporal Logics". In: *Proc. CAV*. Vol. 1633. 1999, pp. 274–287. DOI: 10.1007/3-540-48683-6_25.

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. "The software model checker Blast". In: *Int. J. Softw. Tools Technol. Transf.* 9.5-6 (2007), pp. 505–525. DOI: 10.1007/s10009-007-0044-z.

[BHKW12] Dirk Beyer, Thomas A. Henzinger, M. Erkan Keremoglu, and Philipp Wendler. "Conditional model checking: a technique to pass information between verifiers". In: *Proc. FSE*. 2012, p. 57. DOI: 10.1145/2393596.2393664.

[BHLW22a] Dirk Beyer, Jan Haltermann, Thomas Lemberger, and Heike Wehrheim. "Decomposing Software Verification into Off-the-Shelf Components: An Application to CEGAR". In: *Proc. ICSE*. 2022, pp. 536–548. DOI: 10.1145/3510003.3510064.

[BHLW22b] Dirk Beyer, Jan Haltermann, Thomas Lemberger, and Heike Wehrheim. *Reproduction Package (VM Version) for the article 'Decomposing Software Verification into Off-the-Shelf Components: An Application to CEGAR'*. 2022. DOI: 10.5281/zenodo.5301636.

[BHMS20] Martin Blicha, Antti E. J. Hyvärinen, Matteo Marescotti, and Natasha Sharygina. "A Cooperative Parallelization Approach for Property-Directed k-Induction". In: *Proc. VMCAI*. Vol. 11990. 2020, pp. 270–292. DOI: 10.1007/978-3-030-39322-9_13.

[BHT07] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. "Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis". In: *Proc. CAV*. Vol. 4590. 2007, pp. 504–518. DOI: 10.1007/978-3-540-73368-3_51.

[BHT08] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. "Program Analysis with Dynamic Precision Adjustment". In: *Proc. ASE*. 2008, pp. 29–38. DOI: 10.1109/ASE.2008.13.

[Bie+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. "Bounded model checking". In: *Advances in Computers* 58 (2003), pp. 117–148. DOI: 10.1016/S0065-2458(03)58003-2.

[Bie+99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Masahiro Fujita, and Yunshan Zhu. "Symbolic Model Checking Using SAT Procedures instead of BDDs". In: *Proc DAC*. 1999, pp. 317–320. DOI: 10.1145/309847.309942.

[Bis07] Christopher M. Bishop. *Pattern recognition and machine learning, 5th Edition*. Springer, 2007.

[BJ19] Dirk Beyer and Marie-Christine Jakobs. "CoVeriTest: Cooperative Verifier-Based Testing". In: *Proc. FASE*. Vol. 11424. 2019, pp. 389–408. DOI: 10.1007/978-3-030-16722-6_23.

[BJ20] Dirk Beyer and Marie-Christine Jakobs. "FRed: Conditional Model Checking via Reducers and Folders". In: *Proc. SEFM*. Vol. 12310. 2020, pp. 113–132. DOI: 10.1007/978-3-030-58768-0_7.

[BJ21] Dirk Beyer and Marie-Christine Jakobs. "Cooperative verifier-based testing with CoVeriTest". In: *Int. J. Softw. Tools Technol. Transf.* 23.3 (2021), pp. 313–333. DOI: 10.1007/S10009-020-00587-8.

[BJKS15] Martin Brain, Saurabh Joshi, Daniel Kroening, and Peter Schrammel. "Safety Verification and Refutation by k-Invariants and k-Induction". In: *Proc. SAS*. Vol. 9291. 2015, pp. 145–161. DOI: 10.1007/978-3-662-48288-9_9.

[BJLW18] Dirk Beyer, Marie-Christine Jakobs, Thomas Lemberger, and Heike Wehrheim. "Reducer-based construction of conditional verifiers". In: *Proc. ICSE*. 2018, pp. 1182–1193. DOI: 10.1145/3180155.3180259.

[BK11] Dirk Beyer and M. Erkan Keremoglu. "CPAchecker: A Tool for Configurable Software Verification". In: *Proc. CAV*. Vol. 6806. 2011, pp. 184–190. DOI: 10.1007/978-3-642-22110-1_16.

[BK22] Dirk Beyer and Sudeep Kanav. "CoVeriTeam: On-Demand Composition of Cooperative Verification Systems". In: *Proc. TACAS*. Vol. 13243. 2022, pp. 561–579. DOI: 10.1007/978-3-030-99524-9_31.

[BKL24a] Dirk Beyer, Matthias Kettl, and Thomas Lemberger. "Decomposing Software Verification Using Distributed Summary Synthesis". In: *Proceedings of the ACM on Software Engineering* (2024).

[BKL24b] Dirk Beyer, Matthias Kettl, and Thomas Lemberger. "Fault Localization on Verification Witnesses". In: *Proc. SPIN*. 2024, to appear.

[BKR22] Dirk Beyer, Sudeep Kanav, and Cedric Richter. "Construction of Verifier Combinations Based on Off-the-Shelf Verifiers". In: *Proc. FASE*. Vol. 13241. 2022, pp. 49–70. DOI: 10.1007/978-3-030-99429-7_3.

[BKW10] Dirk Beyer, M. Erkan Keremoglu, and Philipp Wendler. "Predicate abstraction with adjustable-block encoding". In: *Proc. FMCAD*. 2010, pp. 189–197.

[BL13] Dirk Beyer and Stefan Löwe. "Explicit-State Software Model Checking Based on CEGAR and Interpolation". In: *Proc. FASE*. Vol. 7793. 2013, pp. 146–162. DOI: 10.1007/978-3-642-37057-1_11.

[BL17] Dirk Beyer and Thomas Lemberger. "Software Verification: Testing vs. Model Checking - A Comparative Evaluation of the State of the Art". In: *HVC*. Vol. 10629. 2017, pp. 99–114. DOI: 10.1007/978-3-319-70389-3_7.

[BL18] Dirk Beyer and Thomas Lemberger. "CPA-SymExec: efficient symbolic execution in CPAchecker". In: *Proc. ASE*. 2018, pp. 900–903. DOI: 10.1145/3238147.3240478.

[BL19a] Dirk Beyer and Thomas Lemberger. "Conditional Testing: Off-the-Shelf Combination of Test-Case Generators". In: *Proc. ATVA*. Vol. 11781. 2019, pp. 189–208. DOI: 10.1007/978-3-030-31784-3_11.

[BL19b] Dirk Beyer and Thomas Lemberger. "TestCov: Robust Test-Suite Execution and Coverage Measurement". In: *Proc. ASE*. 2019, pp. 1074–1077. DOI: 10.1109/ASE.2019.00105.

[BLW15a] Dirk Beyer, Stefan Löwe, and Philipp Wendler. "Refinement Selection". In: *Proc. SPIN*. Vol. 9232. 2015, pp. 20–38. DOI: 10.1007/978-3-319-23404-5_3.

[BLW15b] Dirk Beyer, Stefan Löwe, and Philipp Wendler. "Sliced Path Prefixes: An Effective Method to Enable Refinement Selection". In: *Proc. FORTE*. Vol. 9039. 2015, pp. 228–243. DOI: 10.1007/978-3-319-19195-9_15.

[BLW19] Dirk Beyer, Stefan Löwe, and Philipp Wendler. "Reliable benchmarking: requirements and solutions". In: *Int. J. Softw. Tools Technol. Transf.* 21.1 (2019), pp. 1–29. DOI: 10.1007/s10009-017-0469-y.

[BLW23] Dirk Beyer, Thomas Lemberger, and Henrik Wachowitz. *Reproduction Package for TACAS 2024 Submission 'CPA-Daemon: Mitigating Tool Restarts for Java- Based Verifiers'*. 2023. DOI: 10.5281/zenodo.8383787.

[BNRS08] Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, and Robert J. Simmons. "Proofs from tests". In: *Proc. ISSTA*. 2008, pp. 3–14. DOI: 10.1145/1390630.1390634.

[BP22] Dirk Beyer and Andreas Podelski. "Software Model Checking: 20 Years and Beyond". In: *Principles of Systems Design - Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday*. Vol. 13660. 2022, pp. 554–582. DOI: 10.1007/978-3-031-22337-2_27.

[BR02a] Thomas Ball and Sriram K. Rajamani. *Generating abstract explanations of spurious counterexamples in C programs*. Tech. rep. MSR-TR-2002-09. Microsoft Research, 2002.

[BR02b] Thomas Ball and Sriram K. Rajamani. "The SLAM project: debugging system software via static analysis". In: *Proc. POPL*. 2002, pp. 1–3. DOI: 10.1145/503272.503274.

[Bra11] Aaron R. Bradley. "SAT-Based Model Checking without Unrolling". In: *Proc. VMCAI*. Vol. 6538. 2011, pp. 70–87. DOI: 10.1007/978-3-642-18275-4_7.

[BS08] Jacob Burnim and Koushik Sen. "Heuristics for Scalable Dynamic Test Generation". In: *Proc. ASE*. 2008, pp. 443–446. DOI: 10.1109/ASE.2008.69.

[BS19] Marko Kleine Büning and Carsten Sinz. "Automatic Modularization of Large Programs for Bounded Model Checking". In: *Proc. ICFEM*. Vol. 11852. 2019, pp. 186–202. DOI: 10.1007/978-3-030-32409-4_12.

[BS20] Dirk Beyer and Martin Spiessl. "MetaVal: Witness Validation via Verification". In: *Proc. CAV*. Vol. 12225. 2020, pp. 165–177. DOI: 10.1007/978-3-030-53291-8_10.

[BS22] Dirk Beyer and Jan Strejček. "Case Study on Verification-Witness Validators: Where We Are and Where We Go". In: *Proc. SAS*. Vol. 13790. 2022, pp. 160–174. DOI: 10.1007/978-3-031-22308-2_8.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.0*. Available at `www.SMT-LIB.org`. 2010.

[BSU22] Dirk Beyer, Martin Spiessl, and Sven Umbricht. "Cooperation Between Automatic and Interactive Software Verifiers". In: *Proc. SEFM*. Vol. 13550. 2022, pp. 111–128. DOI: `10.1007/978-3-031-17108-6_7`.

[Büc62] J Richard Büchi. "On a Decision Method in Restricted Second Order Arithmetic". In: *Proc. Int. Congress on Logic, Method, and Philosophy of Science* (1962). DOI: `10.1007/978-1-4613-8928-6_23`.

[BUZC11] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. "Parallel Symbolic Execution for Automated Real-World Software Testing". In: *Proc. EuroSys*. 2011, pp. 183–198. DOI: `10.1145/1966445.1966463`.

[BW20] Dirk Beyer and Heike Wehrheim. "Verification Artifacts in Cooperative Verification: Survey and Unifying Component Framework". In: *Proc. ISoLA*. Vol. 12476. 2020, pp. 143–167. DOI: `10.1007/978-3-030-61362-4_8`.

[Cal+15] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. "Moving Fast with Software Verification". In: *Proc. NFM*. Vol. 9058. 2015, pp. 3–11. DOI: `10.1007/978-3-319-17524-9_1`.

[Cas+17] Franck Cassez, Anthony M. Sloane, Matthew Roberts, Matthew Pigram, Pongsak Suvanpong, and Pablo González de Aledo Marugán. "Skink: Static Analysis of Programs in LLVM Intermediate Representation - (Competition Contribution)". In: *Proc. TACAS*. Vol. 10206. 2017, pp. 380–384. DOI: `10.1007/978-3-662-54580-5_27`.

[CC77] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Proc. POPL*. 1977, pp. 238–252. DOI: `10.1145/512950.512973`.

[CC79] Patrick Cousot and Radhia Cousot. "Systematic Design of Program Analysis Frameworks". In: *Proc. POPL*. 1979, pp. 269–282. DOI: `10.1145/567752.567778`.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *Proc. OSDI*. 2008, pp. 209–224.

[CE82] Edmund M. Clarke and E. Allen Emerson. "Design and synthesis of synchronization skeletons using branching time temporal logic". In: *Logics of Programs*. 1982, pp. 52–71. DOI: `10.1007/BFb0025774`.

[CFS09] Satish Chandra, Stephen J. Fink, and Manu Sridharan. "Snugglebug: a powerful approach to weakest preconditions". In: *Proc. PLDI*. 2009, pp. 363–374. DOI: 10.1145/1542476.1542517.

[CH23] Marek Chalupa and Thomas A. Henzinger. "Bubaak: Runtime Monitoring of Program Verifiers - (Competition Contribution)". In: *Proc. TACAS*. Vol. 13994. 2023, pp. 535–540. DOI: 10.1007/978-3-031-30820-8_32.

[CH78] Patrick Cousot and Nicolas Halbwachs. "Automatic Discovery of Linear Restraints Among Variables of a Program". In: *Proc. POPL*. 1978, pp. 84–96. DOI: 10.1145/512760.512770.

[Cha+22] Marek Chalupa, Vincent Mihalkovič, Anna Řechtáčková, Lukáš Zaoral, and Jan Strejček. "Symbiotic 9: String Analysis and Backward Symbolic Execution with Loop Folding - (Competition Contribution)". In: *Proc. TACAS*. Vol. 13244. 2022, pp. 462–467. DOI: 10.1007/978-3-030-99527-0_32.

[Cha+23] Saikat Chakraborty, Shuvendu K. Lahiri, Sarah Fakhoury, Akash Lal, Madanlal Musuvathi, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. "Ranking LLM-Generated Loop Invariants for Program Verification". In: *Findings of EMNLP*. 2023, pp. 9164–9175.

[Chr+21] Maria Christakis, Hasan Ferit Eniser, Holger Hermanns, Jörg Hoffmann, Yugesh Kothari, Jianlin Li, Jorge A. Navas, and Valentin Wüstholz. "Automated Safety Verification of Programs Invoking Neural Networks". In: *Proc. CAV*. Vol. 12759. 2021, pp. 201–224. DOI: 10.1007/978-3-030-81685-8_9.

[CHV18] Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith. "Introduction to Model Checking". In: *Handbook of Model Checking*. 2018, pp. 1–26. DOI: 10.1007/978-3-319-10575-8_1.

[CHVB18] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, eds. *Handbook of Model Checking*. Springer, 2018. DOI: 10.1007/978-3-319-10575-8.

[Cio+09] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. "Cloud9: A Software Testing Service". In: *OSR* 43.4 (2009), pp. 5–10. DOI: 10.1145/1713254.1713257.

[CJW15] Mike Czech, Marie-Christine Jakobs, and Heike Wehrheim. "Just Test What You Cannot Verify!" In: *Proc. FASE*. Vol. 9033. 2015, pp. 100–114. DOI: 10.1007/978-3-662-46675-9_7.

[Cla+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-Guided Abstraction Refinement". In: *Proc.CAV*. Vol. 1855. 2000, pp. 154–169. DOI: 10.1007/10722167_15.

[Cla+03] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-guided abstraction refinement for symbolic model checking". In: *J. ACM* 50.5 (2003), pp. 752–794. DOI: 10.1145/876638.876643.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[CMW12] Maria Christakis, Peter Müller, and Valentin Wüstholz. "Collaborative Verification and Testing with Explicit Assumptions". In: *Proc. FM*. Vol. 7436. 2012, pp. 132–146. DOI: 10.1007/978-3-642-32759-9_13.

[CMW16] Maria Christakis, Peter Müller, and Valentin Wüstholz. "Guiding dynamic symbolic execution toward unverified program executions". In: *Proc. ICSE*. 2016, pp. 144–155. DOI: 10.1145/2884781.2884843.

[CN20] Cristian Cadar and Martin Nowack. "KLEE symbolic execution engine in 2019". In: *International Journal on Software Tools for Technology Transfer* (2020), pp. 1–4.

[Coo18] Byron Cook. "Formal Reasoning About the Security of Amazon Web Services". In: *Proc. CAV*. Vol. 10981. 2018, pp. 38–47. DOI: 10.1007/978-3-319-96145-3_3.

[Cou+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "The ASTREÉ Analyzer". In: *Proc. ESOP*. Vol. 3444. 2005, pp. 21–30. DOI: 10.1007/978-3-540-31987-0_3.

[Cou+06] Patrick Cousot, Radhia Cousot, J. Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "Combination of Abstractions in the ASTRÉE Static Analyzer". In: *Proc. ASIAN*. Vol. 4435. 2006, pp. 272–300. DOI: 10.1007/978-3-540-77505-8_23.

[CR24] Marek Chalupa and Cedric Richter. "Bubaak-SpLit: Split what you cannot verify (Competition contribution)". In: *Proc. TACAS*. Vol. 14572. 2024, pp. 353–358. DOI: 10.1007/978-3-031-57256-2_20.

[Cra57a] William Craig. "Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem". In: *J. Symb. Log.* 22.3 (1957), pp. 250–268. DOI: 10.2307/2963593.

[Cra57b] William Craig. "Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory". In: *J. Symb. Log.* 22.3 (1957), pp. 269–285. DOI: 10.2307/2963594.

[CS05] Christoph Csallner and Yannis Smaragdakis. "Check 'n' Crash: combining static checking and testing". In: *Proc. ICSE*. 2005, pp. 422–431. DOI: 10.1145/1062455.1062533.

[CS13] Cristian Cadar and Koushik Sen. “Symbolic execution for software testing: three decades later”. In: *Commun. ACM* 56.2 (2013), pp. 82–90. DOI: 10.1145/2408776.2408795.

[CS19] Marek Chalupa and Jan Strejček. “Evaluation of Program Slicing in Software Verification”. In: *Proc. IFM*. Vol. 11918. 2019, pp. 101–119. DOI: 10.1007/978-3-030-34968-4_6.

[CS21] Marek Chalupa and Jan Strejček. “Backward Symbolic Execution with Loop Folding”. In: *Proc. SAS*. Vol. 12913. 2021, pp. 49–76. DOI: 10.1007/978-3-030-88806-0_3.

[CSV20] Marek Chalupa, Jan Strejček, and Martina Vitovská. “Joint forces for memory safety checking revisited”. In: *Int. J. Softw. Tools Technol. Transf.* 22.2 (2020), pp. 115–133. DOI: 10.1007/s10009-019-00526-2.

[CSX08] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. “DSD-Crasher: A hybrid analysis tool for bug finding”. In: *TOSEM* 17.2 (2008), 8:1–8:37. DOI: 10.1145/1348250.1348254.

[Cyt+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. “Efficiently Computing Static Single Assignment Form and the Control Dependence Graph”. In: *ACM Trans. Program. Lang. Syst.* 13.4 (1991), pp. 451–490. DOI: 10.1145/115372.115320.

[DAV21] Priyanka Darke, Sakshi Agrawal, and R. Venkatesh. “VeriAbs: A Tool for Scalable Verification by Abstraction (Competition Contribution)”. In: *Proc. TACAS*. Vol. 12652. 2021, pp. 458–462. DOI: 10.1007/978-3-030-72013-1_32.

[DDLM13] Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. “Inductive invariant generation via abductive inference”. In: *Proc. OOPSLA*. 2013, pp. 443–456. DOI: 10.1145/2509136.2509511.

[DGH16] Przemyslaw Daca, Ashutosh Gupta, and Thomas A. Henzinger. “Abstraction-Driven Concolic Testing”. In: *Proc. VMCAI*. Vol. 9583. 2016, pp. 328–347. DOI: 10.1007/978-3-662-49122-5_16.

[DHKR11] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. “Software Verification Using k-Induction”. In: *Proc. SAS*. Vol. 6887. 2011, pp. 351–368. DOI: 10.1007/978-3-642-23702-7_26.

[Die+17] Daniel Dietsch, Matthias Heizmann, Betim Musa, Alexander Nutz, and Andreas Podelski. “Craig vs. Newton in software model checking”. In: *Proc. ESEC/FSE*. 2017, pp. 487–497. DOI: 10.1145/3106237.3106307.

[Dij72] Edsger W. Dijkstra. “The Humble Programmer”. In: *Commun. ACM* 15.10 (1972), pp. 859–866. DOI: 10.1145/355604.361591.

[DKW08] Vijay Victor D'Silva, Daniel Kroening, and Georg Weissenbacher. "A Survey of Automated Techniques for Formal Software Verification". In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 27.7 (2008), pp. 1165–1178. DOI: 10.1109/TCAD.2008.923410.

[DLW15] Matthias Dangl, Stefan Löwe, and Philipp Wendler. "CPAchecker with Support for Recursive Programs and Floating-Point Arithmetic - (Competition Contribution)". In: *Proc. TACS.* Vol. 9035. 2015, pp. 423–425. DOI: 10.1007/978-3-662-46681-0_34.

[DS90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics.* Springer, 1990. DOI: 10.1007/978-1-4612-3228-5.

[Dut14] Bruno Dutertre. "Yices 2.2". In: *VSL.* Vol. 8559. 2014, pp. 737–744. DOI: 10.1007/978-3-319-08867-9_49.

[ECGN99] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. "Dynamically Discovering Likely Program Invariants to Support Program Evolution". In: *Proc. ICSE.* 1999, pp. 213–224. DOI: 10.1145/302405.302467.

[EHMU19] Gidon Ernst, Marieke Huisman, Wojciech Mostowski, and Mattias Ulbrich. "VerifyThis - Verification Competition with a Human Factor". In: *Proc. TACAS.* Vol. 11429. 2019, pp. 176–195. DOI: 10.1007/978-3-030-17502-3_12.

[Ern+07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. "The Daikon system for dynamic detection of likely invariants". In: *Sci. Comput. Program.* 69.1-3 (2007), pp. 35–45. DOI: 10.1016/j.scico.2007.01.015.

[Ern20] Gidon Ernst. "A Complete Approach to Loop Verification with Invariants and Summaries". In: *CoRR* abs/2010.05812 (2020).

[Ern23] Gidon Ernst. "Korn - Software Verification with Horn Clauses (Competition Contribution)". In: *TACAS.* Vol. 13994. 2023, pp. 559–564. DOI: 10.1007/978-3-031-30820-8_36.

[Ezu+18] P. Ezudheen, Daniel Neider, Deepak D'Souza, Pranav Garg, and P. Madhusudan. "Horn-ICE learning for synthesizing invariants and contracts". In: *Proc. ACM Program. Lang.* 2.Proc. OOPSLA (2018), 131:1–131:25. DOI: 10.1145/3276501.

[FDA24] FDA. 2024. URL: https://www.fda.gov/medical-devices/medical-device-recalls/tandem-diabetes-care-inc-recalls-version-27-apple-ios-tconnect-mobile-app-used-conjunction-tslim-x2 (visited on 05/14/2024).

[FMMB23] Andrea Fioraldi, Alessandro Mantovani, Dominik Christian Maier, and Davide Balzarotti. "Dissecting American Fuzzy Lop: A FuzzBench Evaluation". In: *ACM Trans. Softw. Eng. Methodol.* 32.2 (2023), 52:1–52:26. DOI: 10.1145/3580596.

[Fra23] Zora Franke. "Realizing Parallel Conditional Model Checking with Multiple Ranged Conditions". MA thesis. Technical University of Darmstad, Germany, 2023.

[FSK12] Diego Funes, Junaid Haroon Siddiqui, and Sarfraz Khurshid. "Ranged Model Checking". In: *ACM SIGSOFT Softw. Eng. Notes* 37.6 (2012), pp. 1–5. DOI: 10.1145/2382756.2382799.

[FWCD17] Kostas Ferles, Valentin Wüstholz, Maria Christakis, and Isil Dillig. "Failure-directed Program Trimming". In: *Proc. FSE*. 2017, pp. 174–185. DOI: 10.1145/3106237.3106249.

[GD17] Mitchell J. Gerrard and Matthew B. Dwyer. "Comprehensive failure characterization". In: *Proc. ASE*. 2017, pp. 365–376. DOI: 10.1109/ASE.2017.8115649.

[GD19] Mitchell J. Gerrard and Matthew B. Dwyer. "ALPACA: A Large Portfolio-based Alternating Conditional Analysis". In: *Proc. ICSE*. 2019, pp. 35–38. DOI: 10.1109/ICSE-Companion.2019.00032.

[GDP17] Marius Greitschus, Daniel Dietsch, and Andreas Podelski. "Loop Invariants from Counterexamples". In: *Proc. SAS*. 2017, pp. 128–147. DOI: 10.1007/978-3-319-66706-5_7.

[GHMW16] Min Gao, Lei He, Rupak Majumdar, and Zilong Wang. "LLSPLAT: Improving Concolic Testing by Bounded Model Checking". In: *Proc. SCAM*. 2016, pp. 127–136. DOI: 10.1109/SCAM.2016.26.

[GI17] Arie Gurfinkel and Alexander Ivrii. "K-induction without unrolling". In: *FMCAD*. 2017, pp. 148–155. DOI: 10.23919/FMCAD.2017.8102253.

[GKKN15] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. "The SeaHorn Verification Framework". In: *Proc. CAV*. Vol. 9206. 2015, pp. 343–361. DOI: 10.1007/978-3-319-21690-4_20.

[GLM08] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. "Automated Whitebox Fuzz Testing". In: *Proc. NDSS*. 2008.

[GLMN14] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. "ICE: A Robust Framework for Learning Invariants". In: *Proc. CAV*. Vol. 8559. 2014, pp. 69–87. DOI: 10.1007/978-3-319-08867-9_5.

[GMKC13] Alwyn Goodloe, César A. Muñoz, Florent Kirchner, and Loïc Correnson. "Verification of Numerical Programs: From Real Numbers to Floating Point Numbers". In: *Proc. NFM*. Vol. 7871. 2013, pp. 441–446. DOI: 10.1007/978-3-642-38088-4_31.

[GMS01] Hubert Garavel, Radu Mateescu, and Irina M. Smarandache. "Parallel State Space Construction for Model-Checking". In: *Proc. SPIN*. 2001, pp. 217–234. DOI: 10.1007/3-540-45139-0_14.

[GNMR16] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. "Learning invariants using decision trees and implication counterexamples". In: *Proc. POPL*. 2016, pp. 499–512. DOI: 10.1145/2837614.2837664.

[GNRT10] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and SaiDeep Tetali. "Compositional may-must program analysis: unleashing the power of alternation". In: *Proc. POPL*. 2010, pp. 43–56. DOI: 10.1145/1706299.1706307.

[GR09] Ashutosh Gupta and Andrey Rybalchenko. "InvGen: An Efficient Invariant Generator". In: *Proc. CAV*. Vol. 5643. 2009, pp. 634–640. DOI: 10.1007/978-3-642-02658-4_48.

[Gro+12] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr. "Swarm Testing". In: *Proc. ISSTA*. 2012, pp. 78–88. DOI: 10.1145/2338965.2336763.

[GS18] Patrice Godefroid and Koushik Sen. "Combining Model Checking and Testing". In: *Handbook of Model Checking*. 2018, pp. 613–649. DOI: 10.1007/978-3-319-10575-8_19.

[GS20] Aman Goel and Karem A. Sakallah. "AVR: Abstractly Verifying Reachability". In: *Proc. TACAS*. Vol. 12078. 2020, pp. 413–422. DOI: 10.1007/978-3-030-45190-5_23.

[GS97] Susanne Graf and Hassen Saïdi. "Construction of Abstract State Graphs with PVS". In: *Proc. CAV*. Vol. 1254. 1997, pp. 72–83. DOI: 10.1007/3-540-63166-6_10.

[GTXT11] Xi Ge, Kunal Taneja, Tao Xie, and Nikolai Tillmann. "DyTa: dynamic symbolic execution guided with static verification results". In: *Proc. ICSE*. 2011, pp. 992–994. DOI: 10.1145/1985793.1985971.

[Gul+06] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. "SYNERGY: a new algorithm for property checking". In: *Proc. FSE*. 2006, pp. 117–127. DOI: 10.1145/1181775.1181790.

[Gur22] Arie Gurfinkel. "Program Verification with Constrained Horn Clauses (Invited Paper)". In: *Proc. CAV*. Vol. 13371. 2022, pp. 19–29. DOI: 10.1007/978-3-031-13185-1_2.

[Hei+17] Matthias Heizmann, Yu-Wen Chen, Daniel Dietsch, Marius Greitschus, Alexander Nutz, Betim Musa, Claus Schätzle, Christian Schilling, Frank Schüssele, and Andreas Podelski. "Ultimate Automizer with an On-Demand Construction of Floyd-Hoare Automata - (Competition Contribution)". In: *Proc. TACAS*. Vol. 10206. 2017, pp. 394–398. DOI: 10.1007/978-3-662-54580-5_30.

[Hei+18] Matthias Heizmann, Yu-Fang Chen, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler, and Andreas Podelski. "Ultimate Automizer and the Search for Perfect Interpolants - (Competition Contribution)". In: *Proc. TACAS*. Vol. 10806. 2018, pp. 447–451. DOI: 10.1007/978-3-319-89963-3_30.

[Hei+23] Matthias Heizmann, Max Barth, Daniel Dietsch, Leonard Fichtner, Jochen Hoenicke, Dominik Klumpp, Mehdi Naouar, Tanja Schindler, Frank Schüssele, and Andreas Podelski. "Ultimate Automizer and the CommuHash Normal Form - (Competition Contribution)". In: *Proc. TACAS*. 2023, pp. 577–581.

[Hel+20] Dominik Helm, Florian Kübler, Michael Reif, Michael Eichberg, and Mira Mezini. "Modular Collaborative Program Analysis in OPAL". In: *Proc. FSE*. 2020, pp. 184–196. DOI: 10.1145/3368089.3409765.

[HHP13] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. "Software Model Checking for People Who Love Automata". In: *Proc. CAV*. Vol. 8044. 2013, pp. 36–52. DOI: 10.1007/978-3-642-39799-8_2.

[HJG08] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. "Swarm Verification". In: *Proc. ASE*. 2008, pp. 1–6. DOI: 10.1109/ASE.2008.9.

[HJMM04] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. "Abstractions from proofs". In: *Proc. POPL*. 2004, pp. 232–244. DOI: 10.1145/964001.964021.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. "Lazy abstraction". In: *Proc. POPL*. 2002, pp. 58–70. DOI: 10.1145/503272.503279.

[HJRW23a] Jan Haltermann, Marie-Christine Jakobs, Cedric Richter, and Heike Wehrheim. *Replication package for article 'Parallel Program Analysis via Range Splitting'*. 2023. DOI: 10.5281/zenodo.7547541.

[HJRW23b] Jan Haltermann, Marie-Christine Jakobs, Cedric Richter, and Heike Wehrheim. "Parallel Program Analysis via Range Splitting". In: *Proc. FASE*. Vol. 13991. 2023, pp. 195–219. DOI: 10.1007/978-3-031-30826-0_11.

[HJRW23c] Jan Haltermann, Marie-Christine Jakobs, Cedric Richter, and Heike Wehrheim. "Ranged Program Analysis via Instrumentation". In: *Proc. SEFM*. Vol. 14323. 2023, pp. 145–164. DOI: 10.1007/978-3-031-47115-5_9.

[HJRW23d] Jan Haltermann, Marie-Christine Jakobs, Cedric Richter, and Heike Wehrheim. *Replication package for article 'Ranged Program Analysis via Instrumentation'*. 2023. DOI: 10.5281/zenodo.8096028.

[HJRW24a] Jan Haltermann, Marie-Christine Jakobs, Cedric Richter, and Heike Wehrheim. *Artifact for research regarding Ranged Program Analysis*. Version 1.0. 2024. DOI: 10.5281/zenodo.12532813.

[HJRW24b] Jan Haltermann, Marie-Christine Jakobs, Cedric Richter, and Heike Wehrheim. "Parallel program analysis on path ranges". In: *Science of Computer Programming* 238 (2024). DOI: 10.1016/j.scico.2024.103154.

[HJRW24c] Jan Haltermann, Marie-Christine Jakobs, Cedric Richter, and Heike Wehrheim. *Replication package for article 'Parallel Program Analysis on Path Ranges'*. Version 2.0. 2024. DOI: 10.5281/zenodo.10854172.

[HLH97] Bernardo Huberman, Rajan Lukose, and Tad Hogg. "An economics approach to hard computational problems". In: *Science* 275.5296 (1997), pp. 51–54.

[HM20] Ákos Hajdu and Zoltán Micskei. "Efficient Strategies for CEGAR-Based Model Checking". In: *J. Autom. Reason.* 64.6 (2020), pp. 1051–1091. DOI: 10.1007/S10817-019-09535-X.

[HM22] Falk Howar and Malte Mues. "GWIT: A Witness Validator for Java based on GraalVM (Competition Contribution)". In: *Proc. TACAS*. Vol. 13244. 2022, pp. 446–450. DOI: 10.1007/978-3-030-99527-0_29.

[Hol+16] Lukáš Holík, Michal Kotoun, Petr Peringer, Veronika Šoková, Marek Trtík, and Tomáš Vojnar. "Predator Shape Analysis Tool Suite". In: *Proc. HVC*. Vol. 10028. 2016, pp. 202–209. DOI: 10.1007/978-3-319-49052-6_13.

[Hol+17] Lukáš Holík, Martin Hruška, Ondřej Lengál, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. "Forester: From Heap Shapes to Automata Predicates - (Competition Contribution)". In: *Proc. TACAS*. Vol. 10206. 2017, pp. 365–369. DOI: 10.1007/978-3-662-54580-5_24.

[HSW13] Frédéric Herbreteau, B. Srivathsan, and Igor Walukiewicz. "Lazy Abstractions for Timed Automata". In: *Proc. CAV*. Vol. 8044. 2013, pp. 990–1005. DOI: 10.1007/978-3-642-39799-8_71.

[Hus+17] Stefan Huster, Jonas Ströbele, Jürgen Ruf, Thomas Kropf, and Wolfgang Rosenstiel. "Using Robustness Testing to Handle Incomplete Verification Results When Combining Verification and Testing Techniques". In: *Proc. ICTSS*. Vol. 10533. 2017, pp. 54–70. DOI: 10.1007/978-3-319-67549-7_4.

[HW20] Jan Haltermann and Heike Wehrheim. "Cooperative Verification via Collective Invariant Generation". In: *CoRR* abs/2008.04551 (2020).

[HW21a] Jan Haltermann and Heike Wehrheim. "CoVEGI: Cooperative Verification via Externally Generated Invariants". In: *Proc. FASE*. Vol. 12649. 2021, pp. 108–129. DOI: 10.1007/978-3-030-71500-7_6.

[HW21b] Jan Haltermann and Heike Wehrheim. *Replication Package for article 'CoVEGI: Cooperative Verification via Externally Generated Invariants'*. Version 1.0. 2021. DOI: 10.5281/zenodo.5956798.

[HW21c] Jan Haltermann and Heike Wehrheim. *Replication package for article 'Machine Learning based Invariant Generation: A Framework and Reproducibility Study'*. 2021. DOI: 10.5281/zenodo.5534068.

[HW22a] Jan Haltermann and Heike Wehrheim. "Information Exchange Between Over- and Underapproximating Software Analyses". In: *SEFM*. Vol. 13550. 2022, pp. 37–54. DOI: 10.1007/978-3-031-17108-6_3.

[HW22b] Jan Haltermann and Heike Wehrheim. "Machine Learning Based Invariant Generation: A Framework and Reproducibility Study". In: *Proc. ICST*. 2022, pp. 12–23. DOI: 10.1109/ICST53961.2022.00012.

[HW23] Jan Haltermann and Heike Wehrheim. *Replication Package for article 'Information Exchange between Over- and Underapproximating Software Analyses'*. Version 1.1. 2023. DOI: 10.5281/zenodo.6749669.

[HW24] Jan Haltermann and Heike Wehrheim. "Exchanging information in cooperative software validation". In: *Software and Systems Modeling* 23 (2024), pp. 695–719. DOI: 10.1007/s10270-024-01155-3.

[HWZ08] Holger Hermanns, Björn Wachter, and Lijun Zhang. "Probabilistic CEGAR". In: *Proc. CAV*. Vol. 5123. 2008, pp. 162–175. DOI: 10.1007/978-3-540-70545-1_16.

[IT20] Omar Inverso and Catia Trubiani. "Parallel and distributed bounded model checking of multi-threaded programs". In: *Proc. PPoPP*. 2020, pp. 202–216. DOI: 10.1145/3332466.3374529.

[Jak17] Marie-Christine Jakobs. "$PART_{PW}$: From Partial Analysis Results to a Proof Witness". In: *Proc. SEFM*. 2017, pp. 120–135. DOI: 10.1007/978-3-319-66197-1_8.

[Jak20] Marie-Christine Jakobs. "CoVeriTest with Dynamic Partitioning of the Iteration Time Limit (Competition Contribution)". In: *Proc. FASE*. Vol. 12076. 2020, pp. 540–544. DOI: 10.1007/978-3-030-45234-6_30.

[JD16] Dejan Jovanovic and Bruno Dutertre. "Property-directed k-induction". In: *FMCAD*. 2016, pp. 85–92. DOI: 10.1109/FMCAD.2016.7886665.

[JKSC08] Himanshu Jain, Daniel Kroening, Natasha Sharygina, and Edmund M. Clarke. "Word-Level Predicate-Abstraction and Refinement Techniques for Verifying RTL Verilog". In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 27.2 (2008), pp. 366–379. DOI: 10.1109/TCAD.2007.907270.

[JM09] Ranjit Jhala and Rupak Majumdar. "Software Model Checking". In: *ACM Comput. Surv.* 41.4 (2009). DOI: 10.1145/1592434.1592438.

[JR21] Marie-Christine Jakobs and Cedric Richter. "CoVeriTest with Adaptive Time Scheduling (Competition Contribution)". In: *Proc. FASE*. Vol. 12649. 2021, pp. 358–362. DOI: 10.1007/978-3-030-71500-7_18.

[JRW24] Christian Janßen, Cedric Richter, and Heike Wehrheim. "Can ChatGPT support software verification?" In: *Proc. FASE*. Vol. 14573. 2024, pp. 266–279. DOI: 10.1007/978-3-031-57259-3_13.

[JSTU24a] Martin Jonáš, Jan Strejček, Marek Trtík, and Lukáš Urba. "Fizzer: New Gray-Box Fuzzer - (Competition Contribution)". In: *Proc. FASE*. Vol. 14573. 2024, pp. 309–313. DOI: 10.1007/978-3-031-57259-3_17.

[JSTU24b] Martin Jonáš, Jan Strejček, Marek Trtík, and Lukáš Urban. "Gray-Box Fuzzing via Gradient Descent and Boolean Expression Coverage". In: *Proc. TACAS*. Vol. 14572. 2024, pp. 90–109. DOI: 10.1007/978-3-031-57256-2_5.

[JVSJ06] Pankaj Jalote, Vipindeep Vangala, Taranbir Singh, and Prateek Jain. "Program Partitioning: A Framework for Combining Static and Dynamic Analysis". In: *Proc. WODA*. 2006, pp. 11–16. DOI: 10.1145/1138912.1138916.

[Kam+23] Adharsh Kamath, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K. Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. "Finding Inductive Loop Invariants using Large Language Models". In: *CoRR* abs/2311.07948 (2023). DOI: 10.48550/ARXIV.2311.07948.

[KFB16] Egor George Karpenkov, Karlheinz Friedberger, and Dirk Beyer. "JavaSMT: A Unified Interface for SMT Solvers in Java". In: *Proc. VSTTE*. 2016, pp. 139–148. DOI: 10.1007/978-3-319-48869-1_11.

[KGC14] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. "SMT-Based Model Checking for Recursive Programs". In: *Proc. CAV*. Vol. 8559. 2014, pp. 17–34. DOI: 10.1007/978-3-319-08867-9_2.

[Kil73] Gary A. Kildall. "A Unified Approach to Global Program Optimization". In: *Proc. POPL*. 1973, pp. 194–206. DOI: 10.1145/512927.512945.

[Kin76] James C. King. "Symbolic Execution and Program Testing". In: *Commun. ACM* 19.7 (1976), pp. 385–394. DOI: 10.1145/360248.360252.

[Kir+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. "Frama-C: A software analysis perspective". In: *Formal Asp. Comput.* 27.3 (2015), pp. 573–609. DOI: 10.1007/s00165-014-0326-7.

[KM04] Rahul Kumar and Eric G. Mercer. "Load Balancing Parallel Explicit State Model Checking". In: *Proc. PDMC*. 3. 2004, pp. 19–34. DOI: 10.1016/j.entcs.2004.10.016.

[KMPZ09] Alexey V. Khoroshilov, Vadim S. Mutilin, Alexander K. Petrenko, and Vladimir Zakharov. "Establishing Linux Driver Verification Process". In: *Proc. PSI*. Vol. 5947. 2009, pp. 165–176. DOI: 10.1007/978-3-642-11486-1_14.

[KPW15] Siddharth Krishna, Christian Puhrsch, and Thomas Wies. "Learning Invariants using Decision Trees". In: *CoRR* abs/1501.04725 (2015).

[Kri59] Saul Kripke. "A Completeness Theorem in Modal Logic". In: *J. Symb. Log.* 24.1 (1959), pp. 1–14. DOI: 10.2307/2964568.

[KRSS16] Temesghen Kahsai, Philipp Rümmer, Huascar Sanchez, and Martin Schäf. "JayHorn: A Framework for Verifying Java programs". In: *Proc. CAV*. Vol. 9779. 2016, pp. 352–358. DOI: 10.1007/978-3-319-41528-4_19.

[Lam77] Leslie Lamport. "Proving the Correctness of Multiprocess Programs". In: *IEEE Trans. Software Eng.* 3.2 (1977), pp. 125–143. DOI: 10.1109/TSE.1977.229904.

[LD22] Will Leeson and Matthew B. Dwyer. "Graves-CPA: A Graph-Attention Verifier Selector (Competition Contribution)". In: *Proc. TACAS*. Vol. 13244. 2022, pp. 440–445. DOI: 10.1007/978-3-030-99527-0_28.

[LEMR20a] Dongge Liu, Gidon Ernst, Toby Murray, and Benjamin I. P. Rubinstein. "LEGION: Best-First Concolic Testing". In: *Proc. ASE*. 2020, pp. 54–65. DOI: 10.1145/3324884.3416629.

[LEMR20b] Dongge Liu, Gidon Ernst, Toby Murray, and Benjamin I. P. Rubinstein. "Legion: Best-First Concolic Testing (Competition Contribution)". In: *Proc. TACAS*. Vol. 12076. 2020, pp. 545–549. DOI: 10.1007/978-3-030-45234-6_31.

[Li+17] Jiaying Li, Jun Sun, Li Li, Quang Loc Le, and Shang-Wei Lin. "Automatic loop-invariant generation and refinement through selective sampling". In: *Proc. ASE*. 2017, pp. 782–792. DOI: 10.1109/ASE.2017.8115689.

[Lio+96] Jacques-Louis Lions, Lennart Lübeck, Jean-Luc Fauquembergue, Gilles Kahn, Wolfgang Kubbat, Stefan Levedag, Leonardo Mazzini, Didier Merle, and Colin O'Halloran. *Ariane 501 Inquiry Board Report*. 1996.

[LLV23] LLVM-Project. *LLVM Language Reference Manual*. 2023. URL: https://llvm.org/docs/LangRef.html (visited on 12/12/2023).

[LS99] Flavio Lerda and Riccardo Sisto. "Distributed-Memory Model Checking with SPIN". In: *Proc. SPIN*. Vol. 1680. 1999, pp. 22–39. DOI: 10.1007/3-540-48234-2_3.

[LT93] Nancy G. Leveson and Clark S. Turner. "An investigation of the Therac-25 accidents". In: *Computer* 26.7 (1993), pp. 18–41. DOI: 10.1109/MC.1993.274940.

[Luc+16] Kasper Soe Luckow, Marko Dimjasevic, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamaric, and Vishwanath Raman. "JDart: A Dynamic Symbolic Analysis Framework". In: *Proc. TACAS*. Vol. 9636. 2016, pp. 442–459. DOI: 10.1007/978-3-662-49674-9_26.

[Man+12] Mikhail U. Mandrykin, Vadim S. Mutilin, Evgeny Novikov, Alexey V. Khoroshilov, and Pavel Shved. "Using linux device drivers for static verification tools benchmarking". In: *Program. Comput. Softw.* 38.5 (2012), pp. 245–256. DOI: 10.1134/S0361768812050039.

[McM03] Kenneth McMillan. "Interpolation and SAT-Based Model Checking". In: *Proc. CAV*. Vol. 2725. 2003, pp. 1–13. DOI: 10.1007/978-3-540-45069-6_1.

[McM06] Kenneth McMillan. "Lazy Abstraction with Interpolants". In: *Proc. CAV*. Vol. 4144. 2006, pp. 123–136. DOI: 10.1007/11817963_14.

[MH21] Malte Mues and Falk Howar. "Data-Driven Design and Evaluation of SMT Meta-Solving Strategies: Balancing Performance, Accuracy, and Cost". In: *Proc. ASE*. 2021, pp. 179–190. DOI: 10.1109/ASE51524.2021.9678881.

[Min06] Antoine Miné. "The octagon abstract domain". In: *High. Order Symb. Comput.* 19.1 (2006), pp. 31–100. DOI: 10.1007/S10990-006-8609-1.

[Mis+07] Sasa Misailovic, Aleksandar Milicevic, Nemanja Petrovic, Sarfraz Khurshid, and Darko Marino. "Parallel Test Generation and Execution with Korat". In: *Proc. FSE*. 2007, pp. 135–144. DOI: 10.1145/1287624.1287645.

[MPV15] Petr Müller, Petr Peringer, and Tomáš Vojnar. “Predator Hunting Party (Competition Contribution)”. In: *Proc. TACAS*. Vol. 9035. 2015, pp. 443–446. DOI: 10.1007/978-3-662-46681-0_40.

[MS07] Rupak Majumdar and Koushik Sen. “Hybrid Concolic Testing”. In: *Proc. ICSE*. 2007, pp. 416–426. DOI: 10.1109/ICSE.2007.41.

[MSSA22] Filipe Marques, José Fragoso Santos, Nuno Santos, and Pedro Adão. “Concolic Execution for WebAssembly”. In: *Proc. ECOOP*. Vol. 222. 2022, 11:1–11:29. DOI: 10.4230/LIPIcs.ECOOP.2022.11.

[NARH17] ThanhVu Nguyen, Timos Antonopoulos, Andrew Ruef, and Michael Hicks. “Counterexample-guided approach to finding numerical invariants”. In: *Proc. ESEC/FSE*. 2017, pp. 605–615. DOI: 10.1145/3106237.3106281.

[Ngu+17] Truc L. Nguyen, Peter Schrammel, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. “Parallel bug-finding in concurrent programs via reduced interleaving instances”. In: *Proc ASE*. 2017, pp. 753–764. DOI: 10.1109/ASE.2017.8115686.

[NKP18] Yannic Noller, Rody Kersten, and Corina S. Păsăreanu. “Badger: Complexity Analysis with Fuzzing and Symbolic Execution”. In: *Proc. ISSTA*. 2018, pp. 322–332. DOI: 10.1145/3213846.3213868.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999. DOI: 10.1007/978-3-662-03811-6.

[Nol+20] Yannic Noller, Corina S. Păsăreanu, Marcel Böhme, Youcheng Sun, Hoang Lam Nguyen, and Lars Grunske. “HyDiff: Hybrid Differential Software Analysis”. In: *Proc. ICSE*. 2020, pp. 1273–1285. DOI: 10.1145/3377811.3380363.

[NRTT09] Aditya V. Nori, Sriram K. Rajamani, SaiDeep Tetali, and Aditya V. Thakur. “The YogiProject: Software Property Checking via Static Analysis and Testing”. In: *Proc. TACAS*. Vol. 5505. 2009, pp. 178–181. DOI: 10.1007/978-3-642-00768-2_17.

[OHe18] Peter W. O’Hearn. “Continuous Reasoning: Scaling the impact of formal methods”. In: *Proc. LICS*. 2018, pp. 13–25. DOI: 10.1145/3209108.3209109.

[Pau23] Felix Pauck. “Cooperative Android App analysis”. PhD thesis. University of Paderborn, Germany, 2023.

[Pei+23] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. “Can Large Language Models Reason about Program Invariants?” In: *Proc. ICML*. Vol. 202. 2023, pp. 27496–27520.

[PM17] Saswat Padhi and Todd D. Millstein. “Data-Driven Loop Invariant Inference with Automatic Feature Synthesis”. In: *CoRR* abs/1707.02029 (2017).

[PNRR15] Van-Thuan Pham, Wei Boon Ng, Konstantin Rubinov, and Abhik Roychoudhury. “Hercules: Reproducing Crashes in Real-World Application Binaries”. In: *Proc. ICSE*. 2015, pp. 891–901. DOI: 10.1109/ICSE.2015.99.

[Pro24] LLVM Project. *libFuzzer – a library for coverage-guided fuzz testing*. 2024. URL: https://llvm.org/docs/LibFuzzer.html (visited on 05/03/2024).

[PSM16] Saswat Padhi, Rahul Sharma, and Todd D. Millstein. “Data-driven precondition inference with learned features”. In: *Proc. PLDI*. 2016, pp. 42–56. DOI: 10.1145/2908080.2908099.

[PW19] Felix Pauck and Heike Wehrheim. “Together strong: cooperative Android app analysis”. In: *Proc. ASE*. 2019, pp. 374–384. DOI: 10.1145/3338906.3338915.

[Qiu+18] Rui Qiu, Sarfraz Khurshid, Corina S. Pasareanu, Junye Wen, and Guowei Yang. “Using Test Ranges to Improve Symbolic Execution”. In: *Proc. NFM*. Vol. 10811. 2018, pp. 416–434. DOI: 10.1007/978-3-319-77935-5_28.

[Qui93] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[RE14] Zvonimir Rakamaric and Michael Emmi. “SMACK: Decoupling Source Language Details from Verifier Implementations”. In: *Proc. CAV*. Vol. 8559. 2014, pp. 106–113. DOI: 10.1007/978-3-319-08867-9_7.

[Rei+20] Michael Reif, Florian Kübler, Dominik Helm, Ben Hermann, Michael Eichberg, and Mira Mezini. “TACAI: an intermediate representation based on abstract interpretation”. In: *Proc. SOAP*. 2020, pp. 2–7. DOI: 10.1145/3394451.3397204.

[RHJW20] Cedric Richter, Eyke Hüllermeier, Marie-Christine Jakobs, and Heike Wehrheim. “Algorithm Selection for Software Validation Based on Graph Kernels”. In: *JASE* 27.1 (2020), pp. 153–186. DOI: 10.1007/s10515-020-00270-x.

[Ric53] Henry Gordon Rice. “Classes of recursively enumerable sets and their decision problems”. In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366. DOI: 1:10.1090/s0002-9947-1953-0053041-6.

[Roc+17] Williame Rocha, Herbert Rocha, Hussama Ismail, Lucas C. Cordeiro, and Bernd Fischer. "DepthK: A k-Induction Verifier Based on Invariant Inference for C Programs - (Competition Contribution)". In: *Proc. TACAS*. Vol. 10206. 2017, pp. 360–364. DOI: 10.1007/978-3-662-54580-5_23.

[RU17] Anton R. Volkov and Mikhail U. Mandrykin. "Predicate Abstractions Memory Modeling Method with Separation into Disjoint Regions". In: *Proc. ISPRAS* 29 (4 2017), pp. 203–216. DOI: 10.15514/ISPRAS-2017-29(4)-13.

[RW19] Cedric Richter and Heike Wehrheim. "PeSCo: Predicting Sequential Combinations of Verifiers (Competition Contribution)". In: *Proc. TACAS*. 2019, pp. 229–233. DOI: 10.1007/978-3-030-17502-3_19.

[Rya+20] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. "CLN2INV: Learning Loop Invariants with Continuous Logic Networks". In: *Proc. Int. Conf. on Learning Representations*. 2020.

[SA06] Koushik Sen and Gul Agha. "CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools". In: *CAV*. Vol. 4144. 2006, pp. 419–423. DOI: 10.1007/11817963_38.

[SA14] Rahul Sharma and Alex Aiken. "From Invariant Checking to Invariant Inference Using Randomized Search". In: *Proc. CAV*. Vol. 8559. 2014, pp. 88–105. DOI: 10.1007/978-3-319-08867-9_6.

[Saa+24] Simmo Saan, Michael Schwarz, Julian Erhard, Helmut Seidl, Sarah Tilscher, and Vesal Vojdani. "Correctness Witness Validation by Abstract Interpretation". In: *Proc. VMCAI*. Vol. 14499. 2024, pp. 74–97. DOI: 10.1007/978-3-031-50524-9_4.

[Sad+18] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. "Lessons from building static analysis tools at Google". In: *Commun. ACM* 61.4 (2018), pp. 58–66. DOI: 10.1145/3188720.

[SB05] Viktor Schuppan and Armin Biere. "Liveness Checking as Safety Checking for Infinite State Spaces". In: *Proc. INFINITY*. Vol. 149. 1. 2005, pp. 79–96. DOI: 10.1016/J.ENTCS.2005.11.018.

[Sci20] Scikit-learn. *scikit-learn - Machine Learning in Python*. 2020. URL: https://scikit-learn.org/stable/ (visited on 09/15/2021).

[SD18] Elena Sherman and Matthew B. Dwyer. "Structurally Defined Conditional Data-Flow Static Analysis". In: *Proc. TACAS*. Vol. 10806. 2018, pp. 249–265. DOI: 10.1007/978-3-319-89963-3_15.

[SD97] Ulrich Stern and David L. Dill. "Parallelizing the Mur*phi* Verifier". In: *Proc. CAV*. Vol. 1254. 1997, pp. 256–278. DOI: 10.1007/3-540-63166-6_26.

[SFS11] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. "Interpolation-Based Function Summaries in Bounded Model Checking". In: *Proc. HVC*. Vol. 7261. 2011, pp. 160–175. DOI: 10.1007/978-3-642-34188-5_15.

[Sha+13] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. "A Data Driven Approach for Algebraic Loop Invariants". In: *Proc. ESOP*. Vol. 7792. 2013, pp. 574–592. DOI: 10.1007/978-3-642-37036-6_31.

[SHB19] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. "PhASAR: An Inter-procedural Static Analysis Framework for C/C++". In: *Proc. TACAS*. Vol. 11428. 2019, pp. 393–410. DOI: 10.1007/978-3-030-17465-1_22.

[Si+18] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. "Learning Loop Invariants for Program Verification". In: *Proc. NeurIPS 2018*. 2018, pp. 7762–7773.

[Si+20] Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. "Code2Inv: A Deep Learning Framework for Program Verification". In: *Proc. CAV*. Vol. 12225. 2020, pp. 151–164. DOI: 10.1007/978-3-030-53291-8_9.

[SK12] Junaid Haroon Siddiqui and Sarfraz Khurshid. "Scaling symbolic execution using ranged analysis". In: *Proc. OOPSLA*. 2012, pp. 523–536. DOI: 10.1145/2384616.2384654.

[SK20] Shikhar Singh and Sarfraz Khurshid. "Parallel Chopped Symbolic Execution". In: *Proc. ICFEM*. 2020, pp. 107–125. DOI: 10.1007/978-3-030-63406-3_7.

[SK21] Shikhar Singh and Sarfraz Khurshid. "Distributed Symbolic Execution using Test-Depth Partitioning". In: *CoRR* abs/2106.02179 (2021).

[SM12] Jan Strejček and Zhendong Su editor = Mats Per Erik HeGreitschimdahl and Marek Trtík. "Abstracting path conditions". In: *Proc. ISSTA*. 2012, pp. 155–165. DOI: 10.1145/2338965.2336772.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: a concolic unit testing engine for C". In: *Proc. ESES/FSE*. 2005, pp. 263–272. DOI: 10.1145/1081706.1081750.

[SNA12] Rahul Sharma, Aditya V. Nori, and Alex Aiken. "Interpolants as Classifiers". In: *Proc. CAV*. Vol. 7358. 2012, pp. 71–87. DOI: 10.1007/978-3-642-31424-7_11.

[SP10] Matt Staats and Corina S. Pasareanu. "Parallel symbolic execution for structural test generation". In: *Proc. ISSTA*. 2010, pp. 183–194. DOI: 10.1145/1831708.1831732.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. "Checking Safety Properties Using Induction and a SAT-Solver". In: *Proc. FMCAD*. Vol. 1954. 2000, pp. 108–125. DOI: 10.1007/3-540-40922-X_8.

[SST12] Jiri Slaby, Jan Strejček, and Marek Trtík. "Checking Properties Described by State Machines: On Synergy of Instrumentation, Slicing, and Symbolic Execution". In: *Proc. FMICS*. Vol. 7437. 2012, pp. 207–221. DOI: 10.1007/978-3-642-32469-7_14.

[SST13] Jiri Slaby, Jan Strejček, and Marek Trtík. "Symbiotic: Synergy of Instrumentation, Slicing, and Symbolic Execution - (Competition Contribution)". In: *Proc. TACAS*. Vol. 7795. 2013, pp. 630–632. DOI: 10.1007/978-3-642-36742-7_50.

[Ste+16] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution". In: *Proc. NDSS*. 2016.

[Str22] Jan Strejček. 2022. URL: https://gitlab.com/sosy-lab/benchmarking/sv-witnesses/-/blob/main/GraphML_witness_format_issues.pdf (visited on 05/17/2024).

[SVB20] SVBenchmarks-Community. *SV-Benchmarks*. 2020. URL: https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/svcomp20.

[SVB22] SVBenchmarks-Community. *SV-Benchmarks*. 2022. URL: https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/svcomp22.

[SVB23] SVBenchmarks-Community. *SV-Benchmarks*. 2023. URL: https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/svcomp23.

[SVB24] SVBenchmarks-Community. *SV-Benchmarks*. 2024. URL: https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/svcomp24.

[TFNM11] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. "Usable Verification of Object-Oriented Programs by Combining Static and Dynamic Techniques". In: *Proc. SEFM*. Vol. 7041. 2011, pp. 382–398. DOI: 10.1007/978-3-642-24690-6_26.

[TH08] Nikolai Tillmann and Jonathan de Halleux. "Pex-White Box Test Generation for .NET". In: *Proc. TAP*. Vol. 4966. 2008, pp. 134–153. DOI: 10.1007/978-3-540-79124-9_10.

[THW23] Nicola Thoben, Jan Haltermann, and Heike Wehrheim. "Timeout Prediction for Software Analyses". In: *Proc. SEFM*. Vol. 14323. 2023, pp. 340–358. DOI: 10.1007/978-3-031-47115-5_19.

[TK09] Sergios Theodoridis and Konstantinos Koutroumbas. *Pattern recognition*. eng. 4th ed. Academic Press, 2009.

[Wah13] Thomas Wahl. *The k-induction principle*. 2013. URL: https://www.khoury.northeastern.edu/home/wahl/Publications/k-induction.pdf (visited on 11/20/2023).

[Wan+16] Dexi Wang, Chao Zhang, Guang Chen, Ming Gu, and Jiaguang Sun. "C Code Verification based on the Extended Labeled Transition System Model". In: *Proc. MoDELS*. Vol. 1725. 2016, pp. 48–55.

[Wei+23] Guannan Wei, Songlin Jia, Ruiqi Gao, Haotian Deng, Shangyin Tan, Oliver Bracevac, and Tiark Rompf. "Compiling Parallel Symbolic Execution with Continuations". In: *Proc. ICSE*. 2023, pp. 1316–1328. DOI: 10.1109/ICSE48619.2023.00116.

[Wei84] Mark D. Weiser. "Program Slicing". In: *IEEE Trans. Software Eng.* 10.4 (1984), pp. 352–357. DOI: 10.1109/TSE.1984.5010248.

[Wen13] Philipp Wendler. "CPAchecker with Sequential Combination of Explicit-State Analysis and Predicate Analysis - (Competition Contribution)". In: *Proc. TACAS*. Vol. 7795. 2013, pp. 613–615. DOI: 10.1007/978-3-642-36742-7_45.

[WSC22] Tong Wu, Peter Schrammel, and Lucas C. Cordeiro. "Wit4Java: A Violation-Witness Validator for Java Verifiers (Competition Contribution)". In: *Proc. TACAS*. Vol. 13244. 2022, pp. 484–489. DOI: 10.1007/978-3-030-99527-0_36.

[WW22] Jingbo Wang and Chao Wang. "Learning to Synthesize Relational Invariants". In: *Proc. ASE*. 2022, 65:1–65:12. DOI: 10.1145/3551349.3556942.

[Yan+19] Guowei Yang, Rui Qiu, Sarfraz Khurshid, Corina S. Pasareanu, and Junye Wen. "A Synergistic Approach to Improving Symbolic Execution Using Test Ranges". In: *Innov. Syst. Softw. Eng.* 15.3-4 (2019), pp. 325–342. DOI: 10.1007/s11334-019-00331-9.

[Yao+20] Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. "Learning nonlinear loop invariants with gated continuous logic networks". In: *Proc. PLDI*. 2020, pp. 106–120. DOI: 10.1145/3385412.3385986.

[YDLW18] Liangze Yin, Wei Dong, Wanwei Liu, and Ji Wang. "On Scheduling Constraint Abstraction for Multi-Threaded Program Verification". In: *IEEE Trans. Software Eng.* 46.5 (2018), pp. 549–565. DOI: 10.1109/TSE.2018.2864122.

[YDLW19] Liangze Yin, Wei Dong, Wanwei Liu, and Ji Wang. "Parallel Refinement for Multi-Threaded Program Verification". In: *Proc. ICSE*. 2019, pp. 643–653. DOI: 10.1109/ICSE.2019.00074.

[YDW15] Guowei Yang, Quan Chau Dong Do, and Junye Wen. "Distributed Assertion Checking Using Symbolic Execution". In: *ACM SIGSOFT Softw. Eng. Notes* 40.6 (2015), pp. 1–5. DOI: 10.1145/2830719.2830729.

[YWW23] Shiwen Yu, Ting Wang, and Ji Wang. "Loop Invariant Inference through SMT Solving Enhanced Reinforcement Learning". In: *Proc ISSTA*. 2023, pp. 175–187. DOI: 10.1145/3597926.3598047.

[Zal13] Michał Zalewski. *American fuzzy lop*. 2013. URL: https://lcamtuf.coredump.cx/afl/ (visited on 05/03/2024).

[ZGNK18] Chenguang Zhu, Arie Gurfinkel, Jorge Navas, and Temesghen Kahsai. *LinearArbitrary - GitHub Repository*. 2018. URL: https://github.com/GaloisInc/LinearArbitrary-SeaHorn (visited on 09/15/2021).

[ZGQH13] Lin Zhou, Shuitao Gan, Xiaojun Qin, and Wenbao Han. "SECloud: Binary Analyzing Using Symbolic Execution in the Cloud". In: *Proc. CBD*. 2013, pp. 58–63. DOI: 10.1109/CBD.2013.31.

[ZMJ18] He Zhu, Stephen Magill, and Suresh Jagannathan. "A data-driven CHC solver". In: *Proc. PLDI*. 2018, pp. 707–721. DOI: 10.1145/3192366.3192416.

A Appendix

A.1 Appendix for CoVEGI and MIGml

A.1.1 Encoder for LLVM-based Helper Invariant Generators

Our encoder follows the general construction depicted in Figure A.1. The LLVM-IR is a low-level, SSA-based representation of programs [LLV23]. A program consists of functions and basic blocks inside the functions. A basic block is a code fragment having a single entry location (the first) and does not contain branching, i.e. the control flow is linear. The last statement of a basic block is called the terminator and can contain branching or function returns. Loops are realized in LLVM using multiple basic blocks, whereas the loop condition is checked in the terminator of each basic block leading to the first block of the loop body[1]. Thus, tools often associate invariants to LLVM basic blocks, i.e. the beginning of the first basic block corresponding to the loop body.

Our encoder follows the general construction depicted in Figure A.1.

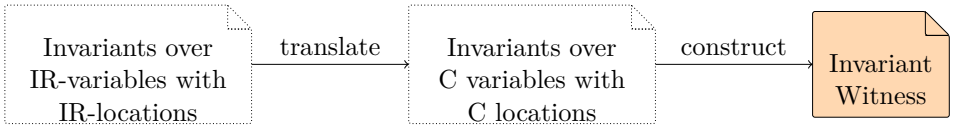


Figure A.1: Workflow of an encoder for a helper working on an IR

To construct an invariant witness, we need to translate the invariants and find the matching C code location for the basic block. For this, we generate the LLVM-IR equipped with debug information, obtained when using the compiler with launch parameter `-g`. We exemplify the process on the running example from Figure 2.3. First, we generate the LLVM-IR fragment, shown in simplified form in Figure A.2, containing the most important debug information as comments. Note that we used the function

[1] https://llvm.org/docs/LoopTerminology.html

```
    entry:
0   %_0 = bitcast i16 (...)* @__VERIFIER_nondet_short to i16 ()*
1   %_1 = tail call signext i16 %_0() #4, !dbg !8
2   %_2 = icmp slt i16 %_1, 0, !dbg !21
3   %_3 = sub i16 0, %_1, !dbg !23
4   %..i = select i1 %_2, i16 %_3, i16 %_1, !dbg !25
5   %_4 = sext i16 %..i to i32, !dbg !26                          ▷ input
6   %_5 = icmp sgt i16 %..i, 1, !dbg !29
7   br i1 %_5, label %_bb, label %error, !dbg !30
8
9   _bb:       ; preds = %.lr.ph, %_bb
10  %.0.i2 = phi i32 [ 0, %.lr.ph ], [ %_8, %_bb ]               ▷ res
11  %.02.i1 = phi i32 [ %_4, %.lr.ph ], [ %_7, %_bb ]            ▷ rem
12  %_7 = add nsw i32 %.02.i1, -2, !dbg !31
13  %_8 = add nsw i32 %.0.i2, 1, !dbg !33
14  %_9 = icmp sgt i32 %.02.i1, 3, !dbg !29
15  br i1 %_9, label %_bb, label %error.loopexit, !dbg !30, !llvm.loop !34
16
17  error.loopexit:      ; preds = %_bb
18  %.lcssa8 = phi i32 [ %_7, %_bb ]
19  %.lcssa = phi i32 [ %_8, %_bb ]
20  %phitmp = mul i32 %.lcssa, 2, !dbg !36
21  br label %error, !dbg !36
22
23  error:       ; preds = %error.loopexit, %entry
24  %.02.i.lcssa = phi i32 [ %_4, %entry ], [ %.lcssa8, %error.loopexit ]
25  %.0.i.lcssa = phi i32 [ 0, %entry ], [ %phitmp, %error.loopexit ]
26  %_10 = add nsw i32 %.0.i.lcssa, %.02.i.lcssa, !dbg !36
27  %_11 = icmp eq i32 %_10, %_4, !dbg !36
28  [...]
```

Figure A.2: Part of the LLVM-IR code for the running example

`__VERIFIER_nondet_short` to model a random input of type `short`. The example contains four basic blocks, `entry`, `_bb`, `verifier.error.loopexit` and `verifier.error`.

The helper invariant generator used computes the invariant $inv_{LLVM} \equiv \%\_4 - 2 * \%.0.i2 - 1 * \%.02.i1 = 0$ for the example and associates it with the basic block `_bb`. As explained in Figure A.1, we first need to find the relation between variables defined in the scope of LLVM-IR and C variables, i.e. we need the relation for all variables present in the generated invariant inv_{LLVM}. In our example, the debug information can directly be used for establishing the relation. As annotated within the comments, the relation contains the information (`%_4`$\mapsto$`input`), (`%.0.i2`$\mapsto$`res`) and (`%.02.i1`$\mapsto$`rem`). Thus, the translated invariant generated by the helper is $inv_C \equiv input - 2 * res - rem = 0$. In general, a more sophisticated procedure is needed, as LLVM-IR uses a three-address code. Therein, complex expressions (e.g. the if-condition in line 9 of the running example) are split into several statements using intermediate variables, that first need to be resolved to C expressions.

Afterward, the transformed invariant needs to be associated with the correct lo-

cation in the C code. We analyze the LLVM IR program structure to map the basic blocks back to C locations. For this, we employed some basic functions provided by Phaser [SHB19] in our encoder. As the program in Figure 2.3 contains mostly simple arithmetic operations and assignments, the relation is easily human-readable: The basic block `entry` contains the if-branch in lines 0 to 2 and a check, whether the loop body needs to be entered, the block `_bb` contains the statements from the loop body, and the blocks `error.loopexit` and `error` the evaluation of the condition of the if-branch in line 9, stored in `%_11`. The code blocks corresponding to the last program statements are left out for representation purposes.

Finally, we construct an invariant witness for the invariants and the locations. Therefore, we build a protocol automaton using the CFA of the C program and store the invariants at the corresponding nodes. The final result is identical to the correctness witness depicted in Figure 2.9, where $\varphi_5 \equiv inv_C$ holds.

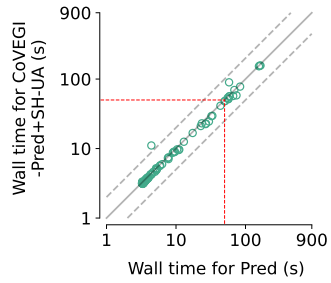
A.1.2 Additional Results for CoVEGI

We present in Figure A.3 the scatter plots to analyze the effectiveness of CoVEGI when using two helper invariant generators in parallel. With respect to efficiency, it turns out that using two helpers in parallel yields a higher CPU time for correctly solved tasks, whereas the overall consumed time does not change when using one or two helpers in parallel.

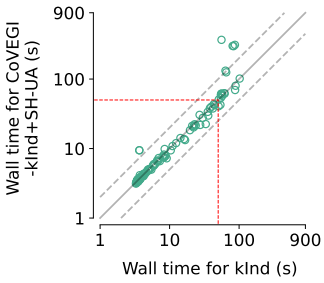
A.1.3 Existing Approaches for Invariant Generation within MIGml

The modular structure of MIGml allows its instantiation with existing concepts for machine learning-based invariant generation. Thereby, it facilitates a conceptual comparison of different approaches as well as an experimental evaluation of them on equal grounds. We selected four existing and conceptually different approaches, published by Sharma, Nori and Aiken (SNA12) [SNA12], Garg, Neider, Madhusudan and Roth (GNMR16) [GNMR16], Krishna, Puhrsch and Wies (KPW15) [KPW15] and Zhu, Magill and Jagannathan (ZMJ18) [ZMJ18]. The four approaches have slightly different objectives and employ different techniques for example generation (execution-based vs. logic-based), model validation (logic-based vs. none), predicate generation (template-based vs. classification-based), and classification (SVM vs. DT learner). Due to the conceptual differences, we consider our selection to be reasonable w.r.t. the variety of existing machine learning approaches for invariant generation. Next, we shortly introduce each of the four approaches. We summarize the results in Table 3.1.

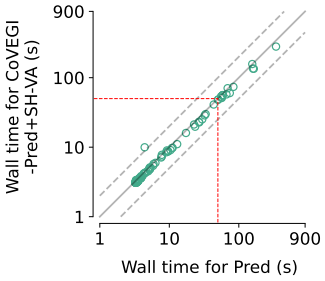
SNA12 Sharma, Nori, and Aiken proposed in 2012 one of the earliest ideas on learning interpolants and likely invariants using ML. Therefore, two formulae are generated for programs with a single loop using the strongest postcondition semantics, assuming that the programs are correct. The first formula, representing establishment (cf. (2.4))



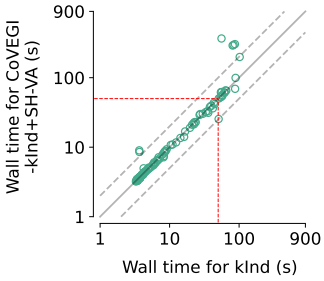
(a) for Pred-SH-UA



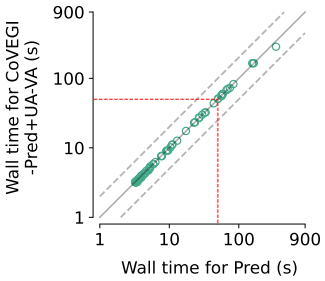
(b) for kind-SH-UA



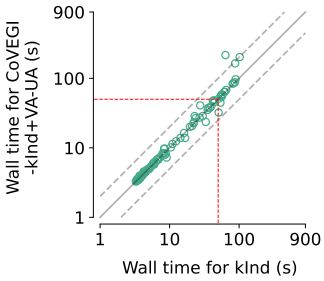
(c) for Pred-SH-VA



(d) for kind-SH-VA



(e) for Pred-VA-UA



(f) for kind-VA-UA

Figure A.3: Scatter plots comparing the wall time of CoVEGI instances using two helper invariant generators with the main verifier running standalone

and preservation (cf. (2.5)), summarizes the path from the program entry with one loop iteration. The second one encodes the paths from the loop head to the assertion at the program's end, used for generating helpful loop invariants (cf. (2.7)). These formulae are used for model validation and example generation. For the initial example generation, a logic-based example generator is used: It samples ten data points, using satisfying assignments of each of the two formulas. Data points generated using the first formula are labeled positively, the others negatively. For model validation, a candidate invariant is checked for being a valid interpolant [Cra57b] for the two formulae. In case it is not a valid interpolant, new data points are generated using satisfying assignments of the violated condition. Within the Learner, an SVM is used as classifier, learning predicates from the Polyhedra domain [CH78; DKW08]. In case the training data is not linearly separable, a combination of SVMs is used, as explained in Section 3.2.1.

The approach does not use additional predicates as input to the SVM.

GNMR16 In 2014, Garg et al. proposed the idea of *ICE learning* [GLMN14], extended in 2016 [GNMR16], aiming at generating loop invariants that help in proving the program correct. Thus, this approach also assumes that the program is correct. As introduced in Section 3.2, it comprises a Learner and a Teacher. For model validation and example generation three formulas representing establishment (2.4), preservation (2.5), and check (2.7) are generated. Initially, an empty set of training data is used. In the candidate generated by the Learner violates establishment respectively check a new positively respectively negatively labeled data point is generated. The main novelty of the ICE approach is their handling of the preservation condition realized by also including implication data points into the training data. Whenever an ML-model violates the preservation condition, the dataset is extended with a new implication point. On the Learner side, the approach employs a decision tree learner. As the use of implication points is by default not supported by a decision tree learner, an enhanced learning algorithm that is able to handle implications points is presented in [GNMR16]. The classifier is also able to handle predicates as additional input. In this approach, all predicates from the octagon domain [Min06] are generated using a template.

KPW15 The idea presented by Krishna et al. employs a similar Learner instantiation as in the ICE learning approach: It also applies a decision tree as classifier and generates predicates from the octagon domain. In contrast, the Teacher is realized differently than in all other approaches: The main idea of this approach is to generate a single, rich set of training data and use the decision tree learner only once, not iteratively as in the other approaches. In more detail, the example generator is execution based, meaning that it executes the program with variable values from a predefined interval $[-L, L]$. Program states observed at the loop head during program execution with a bounded number of loop iterations are labeled positively. As the approach assumes that the program is correct, negative points are generated by mutating the positive once and checking, if executing the program with the mutated state leads to a property violation. If the model validator confirms that an ML-model is a valid loop invariant, an invariant witness is generated. Otherwise, the process aborts without a result.

ZMJ18 In contrast to the former three approaches, Zhu et al. aim for program verification, by solving CHCs using machine learning. The CHCs that are generated for the program, as explained in Section 2.4.6, contains uninterpreted functions. The machine learning approach is asked to learn a predicate for each uninterpreted function symbol. For programs with loops, the CHCs introduce an uninterpreted function for each loop, modeling the loop invariant and contains conditions similar to establishment and preservation using this function. Hence, it also learns loop invariants.

Initially, an empty set of training data is used. In the validation step, the learned interpretations for the uninterpreted function in the CHCs are checked. A Counterexample may be generated in two situations: If the loop invariant learned is too restrictive, e.g. if it violates the initialization condition (2.4), additional positive data points are generated and all negative data points are removed. Otherwise, if a CHC representing establishment (2.5) or check (2.7) is violated with the current interpretation and it cannot be decided using the set of training data if the new data point is observable at the loop head, it is labeled negatively. A Counterexample violating check (2.7) that is observable at the loop head means that the program violates the specification. For the Learner, a decision tree is used as classifier. Predicates are generated using an octagon-based template and an SVM. At first, an SVM is asked for a classification of the training data, where the predicates from the ML-model are also used[2]. Using a combination of two concepts for generating predicates may lead to more complex predicates that depend on the training data.

Conceptual comparison Realizing the four existing approaches as instances of MIGml allow us to easily compare them on a conceptual level: Although the approaches ZMJ18 and GNMR16 pursue different objectives and look completely different at first, our instantiation shows that they differ only in two points: First, they employ different techniques for representing the program and generating new elements for the training data within the Teacher, and second, ZMJ18 uses an SVM as second, additional predicate generator. A comparison of GNMR16 and KPW15 shows that both approaches use the same conceptual components within the Learner and the main difference between them lies in the Teacher. Especially in the non-iterative manner of KPW15 and the facts, that GNMR16 employs training data including implication data points but initially do not generate any training data.

In addition, we state in Table 3.1, if the approach is able to detect property violations or is assuming that the program given is correct. In general, loop invariants help to prove the correctness of a program, hence assuming that the program the invariant is generated for adheres to the specification is reasonable. Nevertheless, in case the program contains a property violation, data-driven approaches like KPW15 may generate a set of training data that is *inconsistent*, i.e. that contains duplicated data points, once labeled positively and once negatively. As tasks violating the safety property are not foreseen in some approaches, they do not provide a mechanism for detecting inconsistent training data. As the classifier is not able to correctly classify all data points, the approaches may run into problems, as the learning process may either fail or produce unreliable results. The approach presented by ZMJ18 is the only one that is also able to detect property violations by design.

[2]The authors told us that the template-based generation is also employed, which is only mentioned implicitly in their paper.

Artifact Availability of the Approaches As we aim to use existing approaches off the shelf, we are interested in the availability and reusability of the provided software artifacts. Hence, we provide a summary next: For ZMJ18, the artifact is available at GitHub [ZGNK18], but we were not able to execute the tool due to issues while building it. Both artifacts of KPW15 and GNMR16 are not publicly available anymore, but the authors provided them and we included them in our own artifact [HW21c]. The tool developed by Krishna et al. is only applicable to the benchmark used in their evaluation, as it requires some manual transformation. The implementation of Garg et al. is applicable to Boogie programs only and the tool is only executable on Windows-32Bit. For SNA12, no artifact is available anymore. Our re-implementations do not suffer from any of these restrictions.

A.1.4 Implementation Details of MIGml

Utility Functions The utility functions are mainly used to generate information on the variables present in the program and their values at loop heads. All functions are realized using CPAchecker [BK11]. The *Default variable collector* collects the variable names as well as information on the variable (like domain, or if value is constant). The *Interval-based program executor* is given an integer interval. It executes the program, setting the values of input variables to values from the interval, and collects all variable values at loop heads. The novel *Symbolic execution based program executor* uses the symbolic execution tool Klee [CDE08; CN20] for data generation. The last function is the *Injectable value analysis*. It determines whether a given state observed at a loop head may result in a property violation. It extends the Value Analysis of CPAchecker, which is introduced in Section 2.2.2.

The utility functions are mainly used to generate information on the variables present in the program and their values at loop heads. All functions are realized using CPAchecker [BK11]. The *Default variable collector* collects the variable names as well as information on the variable (like domain, or if value is constant). The *Interval-based program executor* is given an integer interval. It executes the program, setting the values of input variables to values from the interval, and collects all variable values at loop heads. The novel *Symbolic execution based program executor* uses the symbolic execution tool Klee [CDE08; CN20] for data generation. The last function is the *Injectable value analysis*. It determines whether a given state observed at a loop head may result in a property violation. It extends the Value Analysis of CPAchecker, which is introduced in Section 2.2.2.

A.1.4.1 Learner

In many areas where machine learning techniques are applied, the training data may be noisy and generally does not perfectly represent all data. Therefore, the techniques are configured to *not* classify all training data correctly to avoid overfitting. In contrast,

in this setting the models have to correctly classify all data points that are present in the training data, which we require for all used techniques. Moreover, a precise transformation of the learned model to a boolean representation is needed. In MIGml, we currently provide two different classification algorithms, namely SVMs and decision tree learner. Both learned models are relatively easily transformable into boolean formulae. In addition, both can be configured to generate a classification which *overfits* to the training data, i.e. instead of trying to generalize the training data the classifiers should at best generate a model correctly classifying all training data. In the following, we explain for both how to generate a boolean representation for the ML-Model learned, i.e. the conditions that need to be fulfilled by a point to be classified positively. This condition is used as the potential loop invariant.

SVM We have already exemplified in Section 3.2.1, how to represent a model learned by an SVM using boolean formulae. Due to the learning algorithm employed by the SVM, the coefficients present in the learned model are real values, where the variables present in the program are (in most cases) integers. Consequently, an invariant containing integer coefficients like 2 or reals like 0.5 is more likely correct than an invariant containing 1.999974 or 0.5192. The resulting problem of *rounding coefficients* during transformation is often addressed with too little detail in the publication, although being a major influencing factor. Using the information present in the articles as well as existing available implementations, we implement two different rounding approaches, both using the SVM implementation from the scikit-learn library [Sci20]. The *close rounding* technique, denoted by Svm-C, is used by SNA12 and only rounds the real values that are close to an integer to the next integer. In MIGml, close is defined by a configurable parameter denoting the maximal distance to the nearest integer, by default 0.1. For example, 1.999974 is rounded to 2, but 1.48 is not rounded. An enhancement of the close rounding called *scaled rounding* (Svm-S), that is used by ZMJ18. It searches for a scaling factor of the form $\frac{n+1}{n}, n \in \mathbb{N}$, s.t. all coefficients in the scaled model are close to the next integer. For example, $0.497 * rem + res$ is scaled by 2 to $0,994 * rem + 2 * res$ and rounded to $rem + 2 * res$. As the SVM's model contains only a single linear equation, we apply the algorithm proposed by ZMJ18 for Svm-S to split the dataset during learning, to be able to learn an arbitrary boolean combination of predicates and the simpler algorithm applied by SNA12 for Svm-C. A SVM offers a set of hyper-parameters, containing among others a *C-value*. The higher the C-value, the more data points are tried to be classified correctly by the learned model. For Svm-C, we use a C-value of 1 000 "forcing" the SVM to classify all training data correctly, whereas Svm-S may learn a more general model (C-value of 10).

Decision Tree Learner The decision trees are generated by decision tree learners from the scikit-learn library, called Dt-Skl. The boolean formula generated out of a tree is the disjunction of the formulae for every path from the root to a leaf which them-

selves are conjunctions of the boolean conditions on the nodes. Additional predicates are integrated using the transformation of the training data described in [KPW15]: The concrete value for an additional predicate is computed for each data point and added as an additional column to the training data by introducing a temporary variable t_i for a predicate p_i. When a decision node contains t_i, it is replaced by p_i when generating the boolean formula for the Decision Tree. To be able to employ implication points, we integrated the existing decision tree learner (called Dt-Ice) proposed in 2016 by Garg et al. [GNMR16], that makes use of the latest freely available version of the C5.0 algorithm [Qui93].

Predicate Generator To generate predicates for the classifier, MIGml offers the option to either use the predicates generated by any ML-classifier or generate the predicates using a template. Currently, a template-based predicate generator for predicates from the octagon domain (Pg-Oct) is implemented.

A.1.4.2 Teacher

The teacher's main task is to guide the learner towards finding an invariant. It comprises two components, model validator and example generator.

Model Validator MIGml provides three different implementations of Model Validators, each employing a different technique for abstracting the program: Inspired by SNA12, Val-Ip generates two formulae for validation, one abstracting the path from the program entry to the loop head with one loop iteration, and the second from the loop head to the program exit, using the strongest postcondition operator. The Val-SP, used by GNMR16, generates the establishment, preservation, and check conditions (see (2.4), (2.5) and (2.7)) using the strongest postcondition operator. It is tailored to find models with small values, iteratively adding the condition that all variable values are smaller or equal to 2, 5, 10, ∞. Both, Val-Ip and Val-SP are realized using CPAchecker. Val-CHC is based on CHCs, which is used by ZMJ18. The CHCs are generated using the tool Korn [Ern20; Ern23], being, to the best of our knowledge, the only tool generating CHCs for C programs directly and exporting them as SMT-2-LIB-code. Korn generates additional predicates for each if-branch, increasing the complexity of finding suitable interpretations for the predicates. We applied a simple but efficient heuristic to resolve the additional branching predicates. Indeed, there are few cases where we are not able to resolve the predicates, meaning that the validation fails. All three model validators additionally classify, if the generated model was generated while validating the positive, negative, or implication conditions, information useful for the example generator. For Val-SP and Val-Ip, this information is already present. For Val-CHC, we use the algorithm presented by ZMJ18. We employ the Java-SMT library [KFB16; BBF21], providing support for widely used solvers like Z3 or MathSAT5, where we currently use Z3.

Table A.1: Comparison of the two main verifiers running standalone, using two helper invariant generators with and without a restart of the main verifier

		restart			inject		
k-induction	**alone**	**SH-UA**	**SH-VA**	**UA-VA**	**SH-UA**	**SH-VA**	**UA-VA**
cor. overall	146	153	156	163	156	155	161
cor. proof	102	109	112	119	112	111	118
add. proof	-	7	11	19	10	9	16
cor. alarm	44	44	44	44	44	44	43
add. alarm	-	0	0	0	0	0	0
incor. overall	1	1	1	1	1	1	1

		restart			inject		
pred	**alone**	**SH-UA**	**SH-VA**	**UA-VA**	**SH-UA**	**SH-VA**	**UA-VA**
cor. overall	116	130	130	136	132	132	136
cor. proof	78	92	92	98	94	94	98
add. proof	-	9	9	15	11	11	15
cor. alarm	38	38	38	38	38	38	38
add. alarm	-	0	0	0	0	0	0
incor. overall	1	1	1	1	1	1	1

Example Generator For the initial example generation, MIGml contains a logic-based (EX-Log) and execution-based example generator (EX-Exec). The logic-based example generator asks the model validator to generate a fixed number of satisfying assignments for each formula and extracts the data points. In contrast, EX-Exec first calls either the interval-based (EX-I) or symbolic execution-based program executor (EX-S) to generate a set of positive data points. Thereafter, these points are mutated by changing some variable values, resulting in a set of candidate negative points. For each candidate, the injectable value analysis is used to check whether the postcondition is violated. This approach was proposed by KPW15. Note that EX-I is not guaranteed to be sound when bounding the number of loop executions. Thereby, the training data may contain points labeled negatively which could also be labeled positively using a greater bound. As EX-S does not limit the loop executions, this problem does not occur.

A.1.5 Analysis Results for Restarted Main Verifier

In addition to the result presented in Sec. 3.1, we additionally evaluated the effectiveness of the two main verifiers using the option `restartMain` to *false*. We present the results in Table A.1. The last three columns contain the results obtained when injecting the computed witnesses into the running main verifier. When comparing the overall number of correct results, we observe that restarting the main verifier has nearly no effect. In case predicate analysis is used, the two instances of CoVEGIusing SeaHorn can solve in total two tasks more than the combination that are restarted and in case that

Table A.2: Results for the Reproduction using MIGml

	Benchmark-Tasks		Results			
	reported	**used**	**helpful**	**valid**	**invalid**	**other**
SNA12	10	3	0	0	3	0
GNMR16	58	35	15	1	0	19
KPW15	22	13	0	0	11	2
ZMJ18	199	143	26	3	1	113

UltimateAutomizer and VeriAbs are used, no difference is found. For k-induction, we see that injecting the witnesses is not beneficial when using the best performing combination of UltimateAutomizer and VeriAbs compared to a restart of the main verifier.

A.1.6 Reproducing experiments using MIGml

After having build a framework that allows for being instantiated with existing approaches, namely those presented in [SNA12; KPW15; ZMJ18; PSM16], we want to use it to reproduce the experiments to confirm the authors findings.

Evaluation Plan: To inspect the reproducibility of the reported results, we evaluate our re-implementations on the tasks used in the respective publications. As MIGml uses the same configuration for all tasks in the benchmark, we always employ the configurations described in Tab. 3.3.

Benchmark Tasks. A comparison of tools based on the existing evaluations was difficult due to only partially overlapping sets of verification tasks. We collected the tasks which are still available and published this set as *unified benchmark* in our artifact [HW21c]. As not all tool artifacts are online, the sets are unfortunately incomplete. In summary, 147 tasks from the four approaches presented are available and usable in MIGml. Therein, 3 are used by SNA12, 35 by GNMR16, 13 by KPW15 and 109 by ZMJ18, where tasks from GNMR16 and KPW15 are also used by ZMJ18.

Computing Resources. The evaluation is conducted on virtual machines, having an Intel Xeon E5-2695 v4 CPU with eight cores, a frequency of 2.10 GHz and 16GB memory, running an Ubuntu 20.04 LTS with Linux Kernel 5.4. Each run is limited to 5 minutes of CPU time on 8 cores, having 15GB of memory available, which is the largest timeout used within the four evaluations.

Experimental Results Table A.2 summarizes the results of the reproduction. It contains the number of benchmark-cases used in the original paper and used in the reproduction study. Moreover, the result of the experiments are given, reporting the number of task for which at least one helpful, valid or invalid invariant is generated. The column "other" summarizes timeouts, exceptions and the outcome "unknown".

For SNA12, MIGml is not able to generate valid or helpful invariants for the three tasks reconstructed, whereas the paper reports that the approach can compute such invariants. The invariants generated are invalid, as they violate the preservation condition. As this configuration employs Svm-C, some predicates present in the invariants contain non-rounded real coefficients. In addition, Val-Ip encodes only the first loop iteration, which may also lead to learning invalid invariants. The implementation used is not available anymore, hence we are not able to analyze the difference to the re-implementation.

For GNMR16, the re-implementation is able to generate 15 helpful invariants, where one invariant is helpful for falsification. The publication reports that helpful invariants are generated for all tasks. When analyzing the runs with different results, it turns out that the MIGml Teacher adds different datapoints for the same ML-model learned, compared to the original implementation. Thereby, the Learner generates different models. This diverging behavior is caused by different ways to compute the formulae for validation (weakest precondition in GNMR16 and strongest postcondition in MIGml) and due to different solvers used to generate the models, also confirmed by the authors as likely explanation.

For KPW15, MIGml is also not able to generate valid or helpful invariants, although reported in the paper. In most of the cases, the invariant generated is not valid, as it also violates the preservation condition. In two cases, the initial training data computation takes longer than five minutes, resulting in a timeout. The low performance is mainly caused by the choice of the configurable parameter and additional tunings per task not applied. MIGml is able to learn for some tasks at least valid invariants, if started with the values for L, I and M used in the original evaluation. Four tasks can only be solved using predicates not from the octagon domain, e.g. modulo operations, stated by the authors. These and other task dependent optimizations are not applied by MIGml.

For ZMJ18, the re-implementation is able to solve 26 tasks, where one invariant for falsification and in three cases a counterexample are generated. The authors reported that their tool is able to compute a solution for more than 95% of the tasks. For the vast majority of the tasks (93), MIGml is not able to compute an answer within the time limit. These differences reside most likely in the fact that (1) the original implementation calls the Svm-S only in each n-th iteration and (2) a different tool for generating the CHCs is used. As different values for n are used for difference subsets of tasks, MIGml calls the Svm-S in each iteration ($n = 1$). Although the original implementations source code is available, we were not able to rerun the experiments. Therefore, there might be additional reasons also causing the different results obtained.

Results

In summary, MIGml can reproduce the experiments and is able to partially confirm the reported results.

Threads to validity Our reproducibility study has shown that we were only partially able to reproduce the results. This does not imply that the reported results in the original papers are invalid. The most plausible explanations for the deviation for each approach given in RQ1.

A.2 Appendix for component-based CEGAR and GIA

A.2.1 Using GIAs in Cooperative Scenarios

The basic idea of cooperation is to store analysis results computed by one tool in an artifact and let another tool start its work *using this additional information*. Next, We briefly summarize existing approaches of cooperative validation and explain how they could make use of GIAs. Note that not all forms of cooperation make use of OA and UA components, some may combine only OA or only UA tools. In these cases, we are still able to use GIAs as an exchange format within the cooperation.

A.2.1.1 Cooperative Test Case Generation

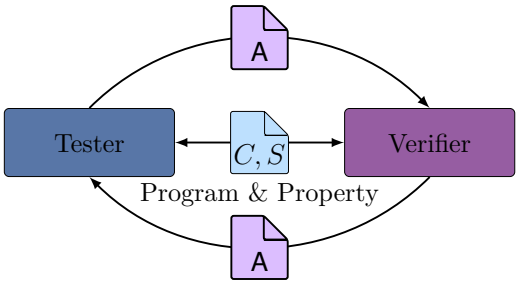


Figure A.4: Cooperative Test Case Generation using GIA as exchange formats

The goal of test case generation is the computation of a test suite leading to paths covering all target nodes. This can be implemented as a cooperation of an UA analysis *Tester* (e.g. concolic execution) with an OA analysis *Verifier* (e.g. bounded model checking) as depicted in Figure A.4. *Tester* is responsible for generating the test suite and *Verifier* for identifying unreachable target nodes. Hence, *Tester* reports in a GIA within $\mathcal{P}_{rt}$ the set of already found paths to targets, where the concrete variable values used for following this path are added as assumptions, and in $\mathcal{P}_{cand}$ the set of not yet covered target paths; *Verifier* tries to show infeasibility of paths in $\mathcal{P}_{cand}$ and if it succeeds, moves these into $\mathcal{P}_{ut}$. Next, *Tester* continues on the remaining targets, and this cycle continues until all target nodes are covered by the test suite. In addition, *Verifier* might add *assumptions* on program transitions to guide *Tester* to uncovered targets.

This form of analysis has been proposed by Daca et al. [DGH16] as a conceptual integration using an ARG for information exchange, and can be realized using GIA in

a cooperative setting. There exist other cooperative approaches for test case generation, namely CoVeriTest [BJ19; Jak20; JR21] and conditional testing [BL19a]. In contrast to the approach depicted in Figure A.4, conditional testing runs two different UA approaches either cyclic or in a sequence. Although it is strictly speaking not a combination of OA and UA approaches, we can also realize the cooperation using GIA as an exchange format following the same idea and encoding all found test cases in $\mathcal{P}_{rt}$. In CoVeriTest, two OA analyses are combined for test case generation, each of them reporting the candidate test cases in $\mathcal{P}_{cand}$ and explored paths in $\mathcal{P}_{ut}$. Each of them is equipped with a UA tool that validates all candidates and stores them in $\mathcal{P}_{rt}$.

A.2.1.2 Cooperative Verification via Conditional Model Checking

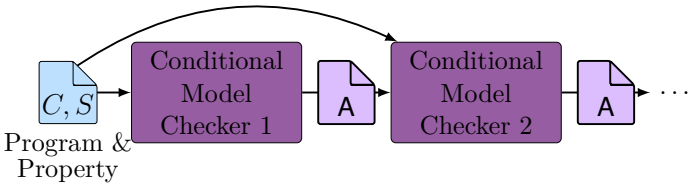


Figure A.5: CMC using GIA as exchange formats

In contrast to the approaches discussed earlier, the concept of Conditional Model Checking [BJLW18; BHKW12] foresees a sequential combination of multiple conditional model checkers, where each of them is an OA tool. Information is exchanged using condition automata. Although the original combination consists of OA tools only, we can realize CMC using GIA, as depicted in Figure A.5. Each conditional verifier reports the partial verification result within $\mathcal{P}_{ut}$ using conditions. The next one continues working on the remaining target nodes. In [CJW15], the second conditional verifier is replaced by a testing tool, yielding a cooperation between OA and UA tools.

A.2.1.3 Cooperation on Invariant Generation

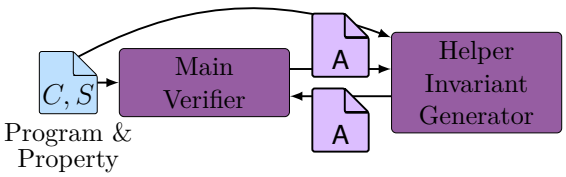


Figure A.6: CoVEGI using GIA as exchange formats

The concept of CoVEGI, presented in Section 3.1 can also be realized using only GIAs. Summarizing the core idea, an OA analysis (the *Main Verifier*) is supported by a *Helper Invariant Generator*, as depicted in Figure A.6. The task of the *Helper Invariant Generator* is to compute loop invariants for specific locations. As a loop invariant is an over-approximation of the concrete loop executions, the *Helper Invariant Generator*

is also an OA component. The *Main Verifier* generates a GIA, where it reports $\mathcal{P}_{cand}$ all paths from the program entry via the loop, for which the invariant is requested. Thereby, these paths are marked as a candidate for leading to a target node. The *Helper Invariant Generator* is now asked to compute predicates, showing that the paths are in fact infeasible. These invariants are encoded as state invariants for the head of the loop. By encoding the task of invariant generation in this way, we see that a *Helper Invariant Generator* solves the same task as a *Precision Refiner* in CEGAR.

A.2.1.4 Using GIAs in Other Forms of Cooperation

For using GIAs to either decompose an existing conceptual integration or to build a novel form of cooperation, a component-wise procedure is advisable. First, each component needs to be classified either being OA or UA. In general, each component within the cooperation solves a certain task, e.g. proving that certain paths are infeasible, finding a concrete execution or a concrete path to a specific location, or generating a new abstraction in the form of predicates for a set of paths. For using GIAs as an exchange format between the components, (1) the tasks that should be solved need to be encoded within a GIA with respect to reachability and (2) the computed answer has to be stored within the GIA. For the former, one can use a set of paths within $\mathcal{P}_{cand}$, either by using all target states or only a specific one, if the component should focus on a specific path while completing the task. For the latter, the component can either move (some paths) to $\mathcal{P}_{ut}$ respectively $\mathcal{P}_{rt}$, depending on whether it is OA or UA, or it does not change the paths and only adds additional information in form of path constraints or state invariants to it.

A.2.2 Full Algorithm for the X-Reducer for GIAs

The full construction of X-Reducer is given in Algorithm 7, extending the construction given in [BJLW18]. Intuitively, we have to ensure that all paths that are X-covered are not present in the reduced program. We therefore adapt the existing reducer presented in [BJLW18]. As in their work, we compute all paths of the CFA C X-covered by the GIA A. We therefore iteratively build a new CFA C_r which locations are pairs of locations from C and states from A. We then iteratively construct all pairs reachable from the element $(\ell_0, (q_0, \varphi_0))$ using a waitlist. For the current element retrieved from the waitlist, we stop the exploration of its successors if all paths that start in the successor are X-covered (see lines 9-10 and 15-16). As we now may have non-trivial state invariants, we also need to account for the case that a path does not meet a state invariant. We thus split the path into two sub-paths for non-trivial state invariants in lines 8-13 using an if-then-else like structure: We create one that is taken only if the state invariant is met, assumes the state invariant and that may lead to a state in F_{rt}, the other in case that the state invariant does not hold, leading to the temporary node $(q_t, true)$, for which it is guaranteed that it will never be removed and that contains the

Algorithm 7 X-Reducer (extended)

Input: CFA $C = (L, \ell_0, G)$ ▷ *CFA*
GIA $\mathsf{A} = (\mathcal{Q}, \Sigma, \delta, (q_0, \varphi_0), F_{ut}, F_{rt}, F_{cand})$ ▷ *GIA*
$q_t \notin \mathcal{Q}$ ▷ *additional state*
Output: CFA $C_r = (L_r, \ell_0^r, G_r)$ ▷ *reduced CFA*

1: $L_r := \{(\ell_0, (q_0, \varphi_0))\}$; $\ell_0^r := (\ell_0, (q_0, \varphi_0))$; $G_r := \emptyset$;
2: waitlist := L_r;
3: **while** waitlist $\neq \emptyset$ **do**
4: choose and remove $(\ell_i, (q_i, \varphi_i))$ from waitlist;
5: **for each** $g = (\ell_i, op, \ell_{i+1}) \in G$ **do**
6: **if** $(q_i, \varphi_i) \in \mathcal{Q} \wedge \exists((q_i, \varphi_i), (G_i, true), (q_{i+1}, \varphi_{i+1})) \in \delta$ *s.t.* $g \in G_i$ **then**
7: **for each** $((q_i, \varphi_i), (G_i, true), (q_{i+1}, \varphi_{i+1})) \in \delta$ *s.t.* $g \in G_i$ **do**
8: **if** $\varphi_{i+1} \neq true$ **then**
9: **if** $\neg allReach_X(q_{i+1}) \wedge (\ell_{i+1}, (q_{i+1}, \varphi_i)) \notin L$ **then**
10: waitlist := waitlist $\cup(\ell_{i+1}, (q_{i+1}, \varphi_i))$;
11: **if** $(\ell_{i+1}, (q_t, true)) \notin L$ **then** waitlist:=waitlist$\cup(\ell_{i+1}, (q_t, true))$; **end if**
12: $L_r := L_r \cup \{(\ell_{i+1}, (q_i', \varphi_i)), (\ell_{i+1}, (q_{i+1}, \varphi_{i+1}), (\ell_{i+1}, (q_t, true))\}$;
// Create intermediate node and realize state invariant using if-else-construct
13: $G_r := G_r \cup \{((\ell_i, (q_i, \varphi_i)),\ op\ , (\ell_{i+1}, (q_i', \varphi_i))),$
$(((\ell_{i+1}, (q_i', \varphi_i)),\ \varphi_{i+1}\ , (\ell_{i+1}, (q_{i+1}, \varphi_{i+1}))),$
$(((\ell_{i+1}, (q_i', \varphi_i)),\ \neg\varphi_{i+1}\ , (\ell_{i+1}, (q_t, true)))\}$;
14: **else**
15: **if** $\neg allReach_X(q_{i+1})) \wedge (\ell_{i+1}, (q_{i+1}, \varphi_{i+1})) \notin L$ **then**
16: waitlist := waitlist $\cup\{(\ell_{i+1}, (q_{i+1}, \varphi_{i+1}))\}$;
17: $L_r := L_r \cup \{(\ell_{i+1}, (q_{i+1}, \varphi_{i+1}))\}$;
18: $G_r := G_r \cup \{(((\ell_i, (q_i, \varphi_i)),\ op\ , ((\ell_{i+1}, (q_{i+1}, \varphi_{i+1})))\}$;
19: **else**
20: **if** $(\ell_{i+1}, (q_t, true)) \notin L$ **then** waitlist := waitlist $\cup\{(\ell_{i+1}, (q_t, true))\}$ **end if**
21: $L_r := L_r \cup \{(\ell_{i+1}, (q_t, true))\}$;
22: $G_r := G_r \cup \{(((\ell_i, (q_i, \varphi_i)), op, (\ell_{i+1}, (q_t, true)))\}$;
23: **if** $F_{cand} \neq \emptyset$ **then**: ▷ *Over-approximate paths cand-covered*
24: $toKeep := \emptyset$;
25: **for each** $\ell = (\ell_i, (q_i, \varphi_i)) \in L_r$ s.t. $(q_i, \varphi_i) \in F_{cand}$ **do**
26: add all predecessors and successors of ℓ in L_r to $toKeep$;
27: **for each** $\ell \in L_r$ **do** ▷ *Remove paths not cand-covered*
28: **if** $\ell \notin toKeep$ **then**
29: Remove ℓ from L_r;
30: Remove all $(\ell, \cdot, \cdot), (\cdot, \cdot, \ell)$ from G_r;
31: **return** (L_r, ℓ_0^r, G_r);

where $allReach_X(q_i) = true$ if all paths starting in q_i leads to F_X, $X \in \{ut, rt\}$

remaining parts of the CFA. Taking this line of argument into account, we can conclude that X-Reducer is in fact a reducer, following the proof structure from [BJLW18]. Again, the resulting CFA is deterministic.

A.2.3 Algorithm for Merge for GIAs

Algorithm 5 consists of five different cases to ensure that paths from $\mathcal{P}_{ut}$ and $\mathcal{P}_{rt}$ are preserved and additional conditions for paths in F_{cand} are also preserved by splitting paths. It makes use of additional functions:

- $\circ, \bullet$ are placeholder that are not processed,
- $trueCond(q_i) = true$ if all pah starting in q_i only contain true conditions,
- $reach_{cand}(q_i) = true$ if no path starting in q_i leads to $F_{ut} \cup F_{rt}$ and at least one to F_{cand},
- $reach_{ut,rt}(q_i) = true$ if at least one path starting in q_i leads to $F_{ut} \cup F_{rt}$.

Next, we briefly summarize the five cases:

(a) Case line 1: State invariants and conditions of t_1 and t_2 are equal $\Rightarrow$ Path is not split.

(b) Case line 6: Both paths lead only to states from F_{cand}, one contains conditions, the other one does not $\Rightarrow$ Keep the non-true assumption

(c) Case line 11: At least one path starting in p_{i+1} eventually leads to $F^1_{ut} \cup F^1_{rt}$ and none starting in s_{j+1} reach $F^2_{ut} \cup F^2_{rt}$ $\Rightarrow$ Ignore s_{j+1}.

(d) Case line 15: At least one path starting in s_{j+1} eventually leads to $F^2_{ut} \cup F^2_{rt}$ and none starting in p_{i+1} reach $F^1_{ut} \cup F^1_{rt}$ $\Rightarrow$ Ignore p_{i+1}.

(e) Case line 19: Otherwise split the path into two paths.

A.2.4 Proof of Theorem 4.10

Recall the Theorem 4.10: Algorithm 4 is a combiner according to Definition 4.9.

Proof. Intuitively, we have to show that for the combination A_O of two GIAs A_I and A'_I each path rt-covered by either A_I or A'_I is also rt-covered by A_O and that the reverse holds (and analogously, that both properties hold for ut-covered paths). We therefore inductively construct an accepting run of A_O for a path π that is rt-covered by either A_I or A'_I and vice versa. We will prove this property by induction on the length of the run of the automaton A_O respectively A_I and A'_I.

We assume wlog. that $\mathcal{P}_{ut}(\mathsf{A}_I) \cup \mathcal{P}_{rt}(\mathsf{A}'_I) = \emptyset = \mathcal{P}_{rt}(\mathsf{A}_I) \cup \mathcal{P}_{ut}(\mathsf{A}'_I)$. As Algorithm 4 and Algorithm 5 works in the same way for states from F_{ut} and F_{rt} and a GIA requires that neither states in F_{ut} nor F_{rt} can be left, it suffices to show that for two arbitrary GIA $\mathsf{A}_I, \mathsf{A}'_I$ and $\mathsf{A}_O = comb(\mathsf{A}_I, \mathsf{A}'_I)$ it holds that:

$$\mathcal{P}_{ut}(\mathsf{A}_I) \cup \mathcal{P}_{ut}(\mathsf{A}'_I) \subseteq \mathcal{P}_{ut}(\mathsf{A}_O) \tag{A.1}$$

$$\text{and } \mathcal{P}_{ut}(\mathsf{A}_I) \cup \mathcal{P}_{ut}(\mathsf{A}'_I) \supseteq \mathcal{P}_{ut}(\mathsf{A}_O) \tag{A.2}$$

Let π^i denote the prefix of length i of π for a path or run. We say that a run $\rho = (q_0, \varphi_0) \xrightarrow{(G_1,\psi_1)} \dots \xrightarrow{(G_k,\psi_k)} (q_k, \varphi_k)$, of a GIA $\mathsf{A}_O = (\mathcal{Q}, \Sigma, \delta, q_0, F_{ut}, F_{rt}, F_{cand})$ *follows a path* $\pi = \langle \ell, \sigma \rangle \xrightarrow{g_1} \dots \xrightarrow{g_n} \langle \ell_n, \sigma_n \rangle$ if

1. $\forall i, 1 \leq i \leq k : g_i \in G_i$,
2. $\forall i, 1 \leq i \leq k : \sigma_i \models \varphi_i$,
3. $\forall i, 0 \leq i \leq k : \sigma_i \models \psi_i$.

For (A.1), given a path $\pi = \langle \ell_0, \sigma_0 \rangle \xrightarrow{g_1} \dots \xrightarrow{g_n} \langle \ell_n, \sigma_n \rangle$ assume wlog. $\pi \in \mathcal{P}_{ut}(\mathsf{A}_I)$. Hence, there is a run $\rho = (q_0, \varphi_0) \xrightarrow{g_1,\psi_1} \dots \xrightarrow{g_k,\psi_k} (q_k, \varphi_k)$ of A_I. Note that the transitions of ρ could contain more than one edge, which we can ignore in the following and thus directly write g_i. We inductively construct a run $\tau = (p_0, \varphi_0) \xrightarrow{g_1,\psi_1} \dots \xrightarrow{g_k,\psi_k} (p_k, \varphi_k)$ of A_O accepting π, where $p_i = (q_i, s_i)$.
Induction start: $(q_0, true) \in \rho$, $((q_0, s_0), true) \in \tau^0$, hence τ^0 follows π^0.
Induction step:
Given $i \in \mathbb{N}$, s.t. $0 \leq i \leq k$. We know by induction hypothesis that τ^i follows π^i. The next transition of ρ is $\xrightarrow{g_{i+1},\psi_{i+1}} (q_{i+1}, \varphi_{i+1})$. We distinguish, if δ_2 also contains a transition $((s_i, \varphi_i) \xrightarrow{g_{i+1},\psi'_{i+1}} (s_{i+1}, \varphi'_{i+1}))$:
If there is no $((s_i, \varphi_i) \xrightarrow{g_{i+1},\psi'_{i+1}} (s_{i+1}, \varphi'_{i+1})) \in \delta_2$ or $s_i \in \{\circ, \bullet\}$ then τ^i is extended by $\xrightarrow{g_{i+1}\psi_{i+1}} ((q_{i+1}, \circ), \varphi_{i+1})$ (by line 5-9 of Algorithm 4) and accepts π^{i+1}.
Otherwise, if there is $((s_i, \varphi_i) \xrightarrow{g_{i+1},\psi'_{i+1}} (s_{i+1}, \varphi'_{i+1})) \in \delta_2$, then τ is extended either by $\xrightarrow{g_{i+1},\psi_{i+1}} ((q_{i+1}, s_{i+1}), \varphi_{i+1})$ in line 1-4 of Algorithm 5 by $\xrightarrow{g_{i+1},\psi_{i+1}} ((q_{i+1}, \circ), \varphi_{i+1})$ in line 9-12 of Algorithm 5 or by $\xrightarrow{g_{i+1},\psi_{i+1}} ((q_{i+1}, \bullet), \varphi_{i+1})$ and by $\xrightarrow{g_{i+1},\psi'_{i+1}} ((\bullet, s_{i+1}), \varphi'_{i+1})$ in line 18-20 of Algorithm 5. In all cases, τ^{i+1} covers π^{i+1}.

As $(q_k, \varphi_k) \in F^1_{ut}$ and $((q_k, s_k), \varphi_k) \in F_{ut}$, we know that $\tau \in \mathsf{A}_O$ and A_O covers π. Thus we can conclude that $\mathcal{P}_{ut}(\mathsf{A}_I) \cup \mathcal{P}_{ut}(\mathsf{A}'_I) \subseteq \mathcal{P}_{ut}(\mathsf{A}_O)$ holds.

For (A.2), given a path $\pi = \langle \ell_0, \sigma_0 \rangle \xrightarrow{g_1} \dots \xrightarrow{g_n} \langle \ell_n, \sigma_n \rangle$ from $\mathcal{P}_{ut}(\mathsf{A}_O)$, accepted by a run $\tau = ((q_0, s_0), \varphi_0) \xrightarrow{g_1,\psi_1} \dots \xrightarrow{g_k,\psi_k} ((q_k, s_k), \varphi_k)$. We inductively construct a run $\rho = (p_0, \varphi_0) \xrightarrow{g_1,\psi_1} \dots \xrightarrow{g_k,\psi_k} (p_k, \varphi_k) \in \mathsf{A}_I$ or A'_I accepting π. As both A_I and A'_I may contain such a run, we start with constructing two runs $\rho_1 \in \mathsf{A}_I$ and $\rho_2 \in \mathsf{A}'_I$ and show that at least one of them accepts π.
Induction start: $((q_0, s_0), true) \in \tau^0$, $(q_0, true) \in \rho^0_1$ and $(s_0, true) \in \rho^0_2$. Hence, both ρ^0_1 and ρ^0_2 follow π^0.
Induction step: Let $0 \leq i \leq k$ be arbitrary but fixed. τ has the transition $((q_i, s_i), \varphi_i) \xrightarrow{g_{i+1},\psi_{i+1}} ((q_{i+1}, s_{i+1}), \varphi_{i+1})$. We now distinguish, if both runs are still under construction (case*2.1*) or not (case*2.2*).

Case 2.1: By induction hypothesis, we know that there are two runs $\rho^i_1 \in \mathsf{A}_I$ and $\rho^i_2 \in \mathsf{A}'_I$ following π^i. The next transition of τ, $t = \xrightarrow{g_{i+1},\psi_{i+1}} ((q_{i+1}, s_{i+1}), \varphi_{i+1})$ is

either added by line 5 to 8 (2.1.1) of Algorithm 4, by line 14 (2.1.2) of Algorithm 4 or by Algorithm 5 (2.1.3).

Case 2.1.1: There is a transition $((q_i, \varphi_i) \xrightarrow{g_{i+1}, \psi_{i+1}} (q_{i+1}, \varphi_{i+1})) \in \delta_1$, but no transition $((s_i, \varphi_i) \xrightarrow{g_{i+1}, \psi'_{i+1}} (s_{i+1}, \varphi'_{i+1})) \in \delta_2$ (or $s_i \in \{\circ, \bullet\}$). Hence, the construction of ρ_2 stops and $s_{i+1} \in \{\circ, \bullet\}$. ρ_1^i can be extended by $\xrightarrow{g_i, \psi_i} (q_{i+1}, \varphi_{i+1}))$, and follows π^{i+1}.

Case 2.1.2: works analogously to Case 2.1.1.

Case 2.1.3: If t is added by line 1-4, ρ_1^i and ρ_2^i can be extended using t, and both follow π^{i+1}. We know that t cannot be added using line 5-8, as either $reach_{cand}(q_{i+1})$ or $reach_{cand}(s_{i+1})$ is false. If t is added by line 9-12, then ρ_1^i can be extended using t, and follows π^{i+1}, whereas ρ_2 cannot be extended, as $s_{i+1} = \circ$. If t is added by line 13-16, then ρ_2^i can be extended using t, and follows π^{i+1}, whereas ρ_1 cannot be extended, as $q_{i+1} = \circ$. Otherwise, τ^i is extended in line 18-30. There are $t_1 = ((q_i, \varphi_i) \xrightarrow{g_{i+1}, \psi'_{i+1}} (q_{i+1}, \varphi'_{i+1})) \in \delta_1$ and $t_2 = ((s_i, \varphi_i) \xrightarrow{g_{i+1}, \psi''_{i+1}} (s_{i+1}, \varphi''_{i+1})) \in \delta_2$. As τ is constructed using t_1 or t_2, ρ_1^i or ρ_2^i can be extended, depending whether $\varphi_{i+1} = \varphi'_{i+1} \wedge \psi_{i+1} = \psi'_{i+1}$ (τ is in A_I) or $\varphi_{i+1} = \varphi''_{i+1} \wedge \psi_{i+1} = \psi''_{i+1}$ (τ is in A'_I). As paths accepted by a run with true condition are also accepted with any condition, we know that the extended run accepts π^{i+1}.

Case 2.2: Wlog. assume that ρ_1 is still under construction. As the construction of ρ_2 has stopped, $s_i \in \{\circ, \bullet\}$ (follows from Case 2.1.1). Thus, the next transition $\xrightarrow{g_{i+1}, \psi_{i+1}} ((q_{i+1}, s_i), \varphi_{i+1}) \in \tau$ must be added by line 5 to 9 of Algorithm 4. This works analogously to Case 2.1.1 ✓

As $((q_k, s_k), \varphi_k) \in F_{ut}$, $(q_k, \varphi_k) \in F_{ut}^1$ for ρ_1 or $(s_k, \varphi_k) \in F_{ut}^2$ for ρ_2, at least one of them accepts π, concluding the proof. □

A.2.5 Additional GIAs for the Example Application

A.2.5.1 Additional Example of a GIA

Figure A.7 depicts an example of a GIA for the program of Figure 2.3b with target nodes $TN = \{\ell_1, \ell_2, \ell_6, \ell_9, \ell_{10}, \ell_{11}\}$, that is generated during test case generation with the criterion branch coverage.

We have $F_{rt} = \{q_1\}$, $F_{ut} = \{q_{10}\}$ and $F_{cand} = \{q_3, q_6, q_9, q_{11}\}$ and $\varphi_5 \equiv input == 2 * res + rem$. It contains the information that ℓ_1 is reachable when the condition $input = -1$ holds, ℓ_{10} is unreachable and that ℓ_3, ℓ_6, ℓ_9 and ℓ_{11} are candidates for being reached when the condition $input = 2$ holds. Additionally, to justify the unreachability of ℓ_{10}, it contains the state invariant $\varphi_5 \equiv input == 2 * res + rem$.

A.2.5.2 Example of Combining two GIAs

Next, we exemplify the application of Combiner on the two GIAs A_7, depicted in Figure A.8a, and A_8, depicted in Figure A.8b generated for the running example from Figure 2.3 with target nodes $TN = \{\ell_1, \ell_2, \ell_6, \ell_9, \ell_{10}, \ell_{11}\}$. The result is depicted in

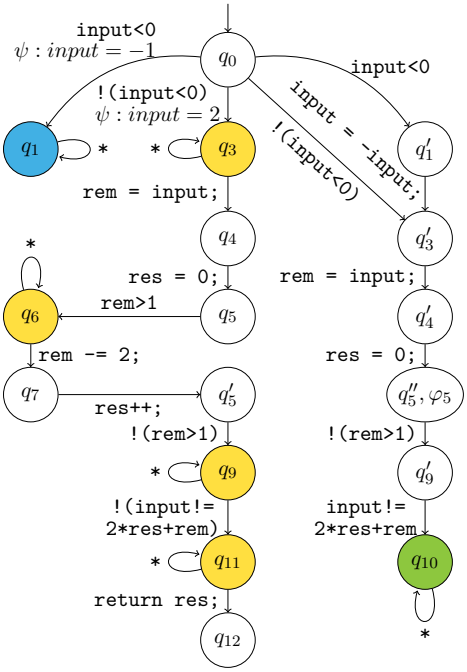
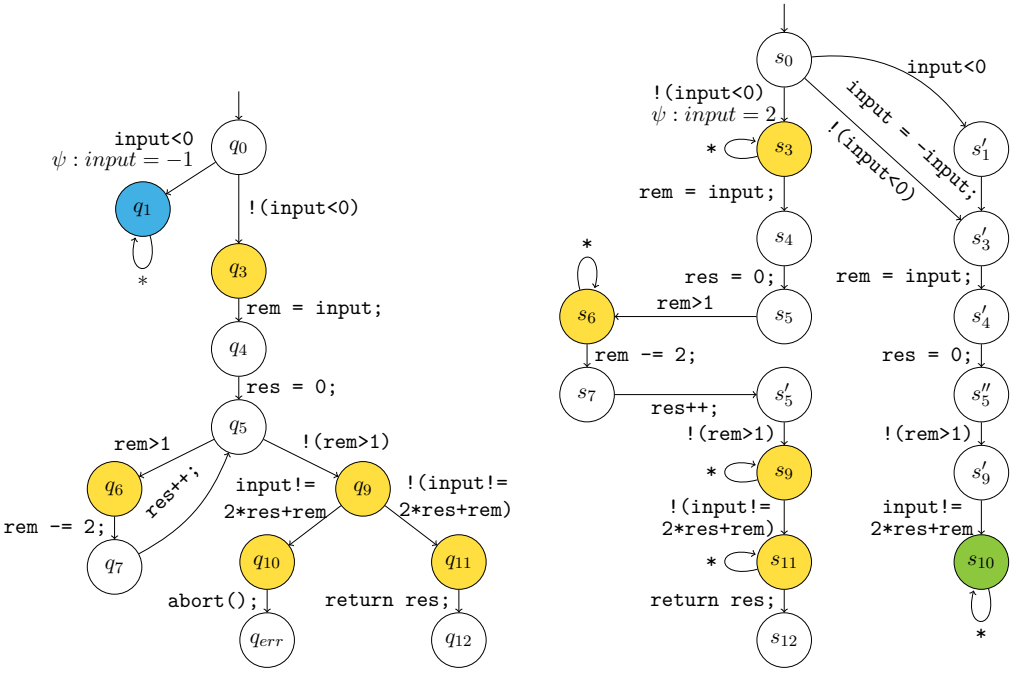


Figure A.7: A GIA generated during cooperative test case generation for the example program of Fig. 2.3 with states of F_{ut} marked green, of F_{rt} blue and of F_{cand} yellow. We elide state invariants that are *true* and depict for transitions only the operation and non-true conditions.

Figure A.8c. Both GIAs used as input could be generated during a cooperative test case generation(cf. Appendix A.2.1.1), where A_7 is generated by the tester (UA) and A_8 by the Verifier (OA). A_7 encodes the information that ℓ_1 is reachable with the condition $input = -1$. A_8 is produced by an OA tool, that marks the target node $\ell_1 0$ as unreachable. In addition, it contains the information that if the program is started with $input = 2$.

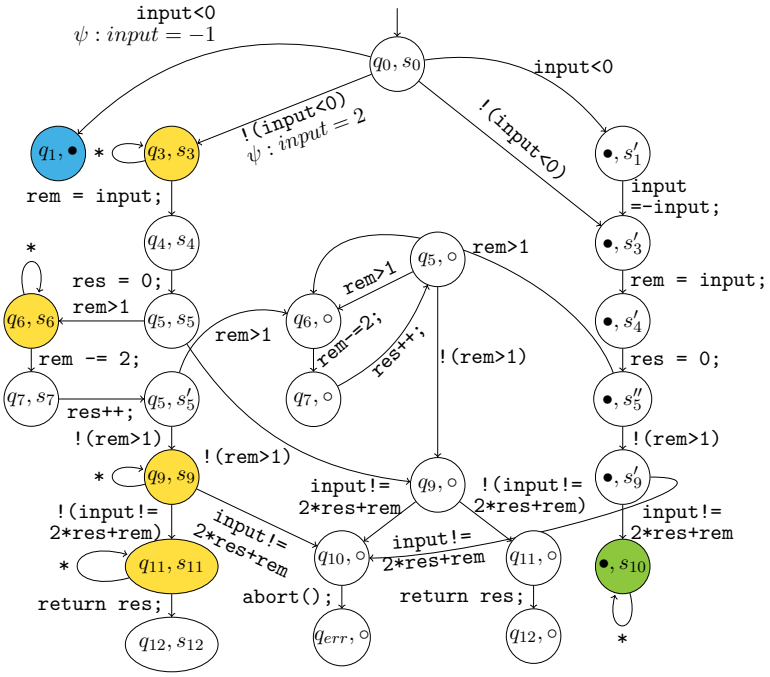
As both, A_7 and A_8, contain a path to $\ell_1 0$, but $q_{10} \in F_{cand}$ in A_7 and $s_{10} \in F_{ut}$ in A_8, the combiner generated a successor using '$\bullet$' instead of q_{10} for A_9 to maintain the information that ℓ_{10} is unreachable. In contrast, one successor of the state (q_9,s_9) is $(q_{10}, \circ)$, as q_9 has a successor q_{10} but s_9 does not. Using '$\circ$' instead of '$\bullet$' ensures that $(q_{10}, \circ)$ is not in F_{cand} in A_9 (cf. line 18 in Alg. 4), because A_8 contains the information that this node is unreachable. Thereby, it is ensured that all paths of the CFA that contain ℓ_{10} are *ut*-covered.

Additionally, Combiner maintains more precise information on paths from $\mathcal{P}_{cand}$: If a path π is present in $\mathcal{P}_{cand}(\mathsf{A}_7)$ and $\mathcal{P}_{cand}(\mathsf{A}_8)$, once with and once without condition, the condition is also present on the path in the combined GIA. In our example, both A_7 and A_8 contain a path covering among others ℓ_3. A_7 has a path $(q_0, q_3, \ldots)$ with the condition *true* and A_8 a path $(s_0, s_3, \ldots)$ labeled with $input = 2$. As a result A_9 contains the non-*true* condition $input = 2$ as for the transition from (q_0, s_0) to (q_3, s_3) and thus maintains the more precise information.



(a) First input A_7

(b) Second input A_8



(c) Result A_9

Figure A.8: Application of Combine($\mathsf{A}_7, \mathsf{A}_8$) = A_9 for the program from Figure 2.3

A.2.5.3 CFA generated by Reducer

Figure A.9 contains the CFA obtained when starting Algorithm 7 using the CFA from Figure 4.6 and the GIA A_3 from Figure 4.16 as input.

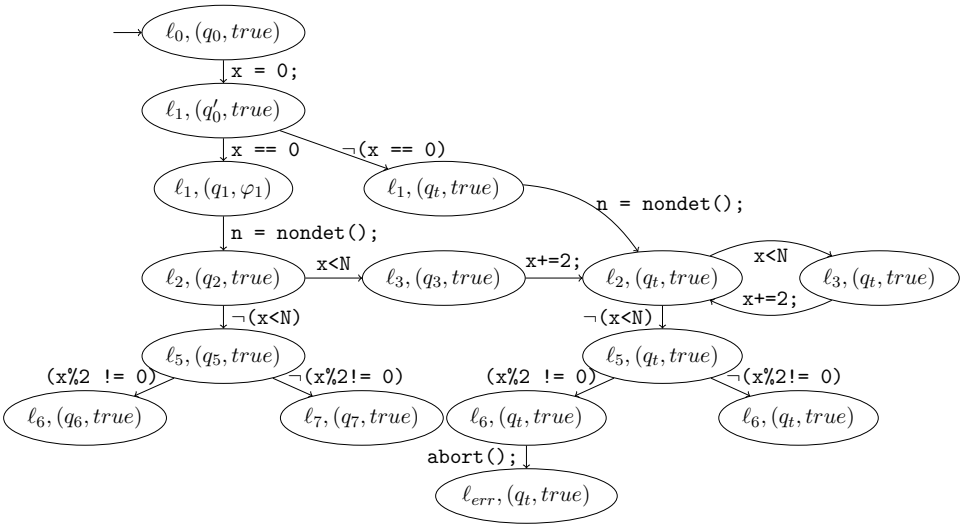


Figure A.9: The CFA C_3 obtained when starting Algorithm 7 using the CFA C_2 from Figure 4.6 and the GIA A_3 from Figure 4.16 as input

A.2.6 Using GIAs in Cooperative Test Case Generation

We also realized cooperative test case generation using GIAs for the information exchange, as depicted in Figure A.4. We follow the tooling used by Daca et al. [DGH16], employing the concolic tester Crest [BS08] as UA tool and CPAchecker's predicate abstraction using CEGAR as OA analysis.

A.2.6.1 Implementation

Crest [BS08] is a concolic tester. Hence, test cases used as inputs are generated by computing solutions to the path conditions that use symbolic inputs and solved by an SMT-solver, in this case Yices [Dut14]. As Crest is computing an under-approximation of the state space, it will only report violations of the safety property. By default, Crest first generates up to 100 000 different inputs. Hence, Crest will eventually generate test inputs covering all reachable branches. As Crest is a testing tool under-approximating the state space, it is not able to identify paths of the program as unreachable. We therefore combine it with a predicate abstraction from CPAchecker. The predicate abstraction over-approximates the reachable state space and can thus mark target nodes as unreachable. Due to the precisely defined and uniformly applicable semantics of the GIA, we reuse the modules in CPAchecker that we built for CC-Pred-Gia, hence we do not need to employ combiner or reducer in this setting.

We use Crest as off-the-shelf tool, hence it can neither process nor produce GIAs, hence we build and employed combiner and reducer, resulting in the cooperative test case generator depicted in Figure A.10.

For running Crest, we employed parts of tbf [BL17] to let Crest generate test

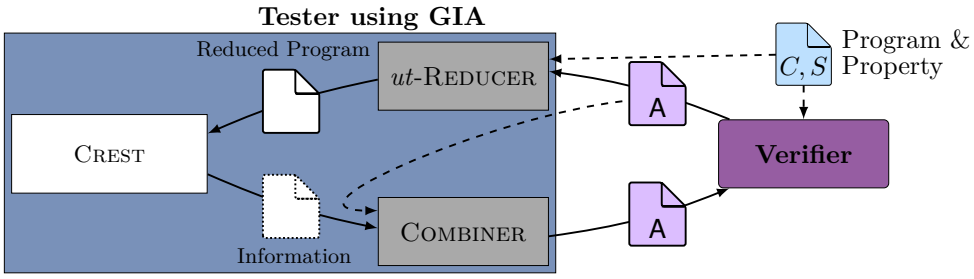


Figure A.10: Cooperative Test Case Generation using GIA as exchange formats

inputs in the TEST-Comp test case format[3] and generated a GIA for them. We additionally optimized the resulting GIA by removing duplicate paths, i.e. paths traversing the same nodes but are labeled with different assumptions. Within each iteration, Crest is started and generates at most 100 test inputs, before the predicate abstraction is called to identify unreachable target nodes. The computation is complete, if all target nodes are *ut*-covered or *rt*-covered by the generated GIA. In the last step, we extract a test suite from that GIA by traversing its path leading to F_{rt} and collecting all assumptions on the return values from `nondet()`. The resulting cooperative test case generation approach is called CoTest.

We additionally used Crest standalone for comparison with CoTest. In the first step, Crest is used in the default configuration, generating at most 100 000 test cases in its internal format. Afterwards, TBF is used to remove duplicate tests and transform the test cases into a test suite in the TEST-Comp format, that is needed to measure the coverage of the generated test suite. To ensure that the test suite is generated, we stopped Crest after 80% of the available time and start the transformation.

A.2.6.2 Evaluation

Test case generation aims at finding program inputs, such that either a certain statement (statement coverage) or all branching points (branch coverage) are visited at least once when executing the program with the given inputs. As we want to evaluate the cooperation of a tester and an OA analysis technique, we focus on branch coverage, as this yields in general several target nodes for a program. We compare the branch coverage of the test cases generated for our cooperative test case generation approach (called CoTest) with Crest as a standalone tool.

Evaluation Setup We used the same evaluation machines as in RQ 1, but limited the time for test case generation to 5 minutes. We evaluated both approaches on a small subset of the SV-Benchmarks, in the version used for the TEST-Comp'22[4]. As we are interested in exemplarily showing the usefulness of GIAs in the cooperative test case generation setting, we selected a subset of the tasks from the `ControlFlow` category of

[3]https://gitlab.com/sosy-lab/test-comp/test-format/blob/testcomp22/doc/Format.md
[4]https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/testcomp22

Table A.3: Results of test case generation for CoTest and Crest. Note that TestCov was not able to analyze the full test suite generated by Crest in the given time limit.

	CoTest			Crest		
Task	#tests	coverage	time(s)	#tests	coverage	time(s)
a1	2	75.0 %	32	33 335	75.0%	237
a2	5	87.5 %	63	18 183	37.5%	225
l5	7	84.9 %	50	50 447	84.9%	251
l6	8	84.6 %	50	50 266	84.6%	253
l7	19	84.4 %	120	50 180	84.4%	256
l8	10	84.3 %	56	50 072	84.3%	251
l9	11	84.2 %	55	50 036	84.2%	266
l10	12	84.1 %	57	50 035	81.0%	266
l11	13	84.1 %	67	50 022	84.1%	265
l12	14	84.0 %	72	50 020	84.0%	254
l13	15	84.0 %	76	50 019	84.0%	254
l14.1	16	83.9 %	78	50 015	83.9%	268
l14.2	-	-	T.O.	50 015	70.1%	259
l15.1	-	-	T.O.	50 019	69.9%	268
l15.2	17	83.9 %	81	50 019	83.9%	271

TEST-Comp (tasks l5-l15.2) and used the running example of [HW24] (task a1) and an extended version, also shown in [HW24] (task a2). The selected tasks work with simple integer variables and do not make use of arrays or pointers. The tasks selected from the SV-Benchmarks contain an infinite loop, where all variables have the same value at the start of each iteration. Thus, the loop does not affect the reachability of the target nodes. As our reducer implementation works best for loop-free programs and to avoid infinite computation for the coverage measurement, we removed the loop.

To compute the coverage of the generated test suites, we used TestCov [BL19b], the tool also used in the TEST-Comp. We use TestCov with a 30 minutes timeout, in contrast to TEST-Comp, where only a five-minute timeout is used.

Evaluation Results Table A.3 contains the experimental results for Crest and our cooperative test case generation approach CoTest. It contains the size of the test suite generated in the column #tests, the coverage achieved with each test suite and the CPU time taken to compute the test suite for each task.

We generally observe that both tools generate test suites with nearly the same code coverage, especially the size of the test suite generated by CoTest is significantly smaller than for Crest. On average, the test suite generated by CoTest has only 0.024% of the size of the test suite generated by Crest, the largest difference is 0.006% and the smallest difference is 0.038%. In other words, the test suite that is generated by Crest within the given time limits contains on average 4 100 times more test cases than the test suite generated by CoTest.

Another significant difference between the test suites generated by Crest and those

generated by CoTest is their size, i.e. the number of generated test cases in the suite. Each GIA contains the information which target nodes are reached for each test input. Hence, detecting test cases following the same path within the CFA and thus leading to the same target nodes is easy. This allows us to reduce the number of paths within the GIA and thereby the test suite extracted in the end. Although Crest used within CoTest generates up to 100 test cases per iteration, the evaluation indicates that using GIA allows for a reduction of the test suite by at most 80% on the benchmark set used, as for test cases following the same path within the CFA and hence covering exactly the same target nodes only one per path is exported. We also observe the advantage of the significantly smaller test suite, as TestCov is not able to process the full test suite generated by Crest within the time limit of five minutes.
For two tasks (`a2`, `l10`) CoTest can cover more branching points than Crest. Due to the size of the test suites generated by Crest, TestCov can only analyze around 10% of it before reaching the given time limit, which is most likely the reason for the lower coverage measured. For two tasks (`l14.2` and `l15.1`), CoTest was not able to cover all target nodes within the given time restrictions. As we transform the GIA into a test suite only if all target nodes are covered, no test suite is generated in these two cases.

When comparing the CPU time consumed to generate the test suite, we observe that CoTest can complete the test case generation task faster than Crest. In the median, CoTest can finish the computation in only 28% of the time that is taken by Crest. As GIAs allow to precisely encode information on reachable and non reachable target nodes in a single artifact, predicate abstraction can mark all unreachable target nodes as such and Crest can report all paths to target nodes within the same GIA. Thereby, the computation can be stopped in case all target nodes are either *ut*-covered or *rt*-covered. In contrast, Crest running standalone cannot detect that all target nodes are covered, in case some of them are unreachable. Thus, it continues the test case generation, until it reaches its internal timeout of 240 seconds.

In summary, GIAs are also suited as an exchange format for cooperative test case generation, allowing encoding information of reachable and unreachable target nodes within a single artifact. Due to the precisely defined semantics, it can be easily detected whether the task is already completed. Thereby, cooperative test case generation also benefits from using GIAs as an exchange format.

A.3 Appendix for Ranged Program Analysis

A.3.1 Additional Examples of Instrumented Programs

```
0 unsigned int on_lpath = 1;
1 unsigned int on_rpath = 1;
2 unsigned int rcntr = 0;
3 unsigned int lcntr = 0;
4
5 int rredout(int pos){
6   return pos >= 4 || pos == 1;}
7 int lredout(int pos){
8   return pos >= 4 || !(pos==2||pos==3);}
9
10 int div2WithReminderAbs(short input){
11   if (input < 0){
12     if (on_lpath){
13       if (!lredout(lcntr))
14         return;
15       lcntr++;
16       if (on_lpath && lcntr >= 4)
17         on_lpath = 0;
18     }
19     if (on_rpath){
20       on_rpath = rredout(rcntr);
21       rcntr++;
22       if (on_rpath && rcntr >= 4)
23         return;
24     }
25     input = -input;
26   }else{
27     if (on_lpath){
28       on_lpath = !lredout(lcntr);
29       lcntr++;
30     }
31     if (on_rpath){
32       if (rredout(rcntr))
33         return;
34       rcntr++;
35     }
36   }
37   int rem = input;
38   int res = 0;
39   while (rem > 1){
40     if (on_lpath){
41       if (!lredout(lcntr))
42         return;
43       lcntr++;
44       if (on_lpath && lcntr >= 4)
45         on_lpath = 0;
46     }
47     if (on_rpath){
48       on_rpath = rredout(rcntr);
49       rcntr++;
50       if (on_rpath && rcntr >= 4)
51         return;
52     }
53     rem -= 2;
54     res++;
55   }
56   if (on_lpath){
57     on_lpath = !lredout(lcntr);
58     lcntr++;
59   }
60   if (on_rpath){
61     if (rredout(rcntr))
62       return;
63     rcntr++;
64   }
65   if (input != 2 * res + rem){
66     if (on_lpath){
67       if (!lredout(lcntr))
68         return;
69       lcntr++;
70       if (on_lpath && lcntr >= 4)
71         on_lpath = 0;
72     }
73     if (on_rpath){
74       on_rpath = rredout(rcntr);
75       rcntr++;
76       if (on_rpath && rcntr >= 4)
77         return;
78     }
79     abort();
80   }else{
81     if (on_lpath){
82       on_lpath = !lredout(lcntr);
83       lcntr++;
84     }
85     if (on_rpath){
86       if (rredout(rcntr))
87         return;
88       rcntr++;
89     }
90   }
91   return res;
92 }
```

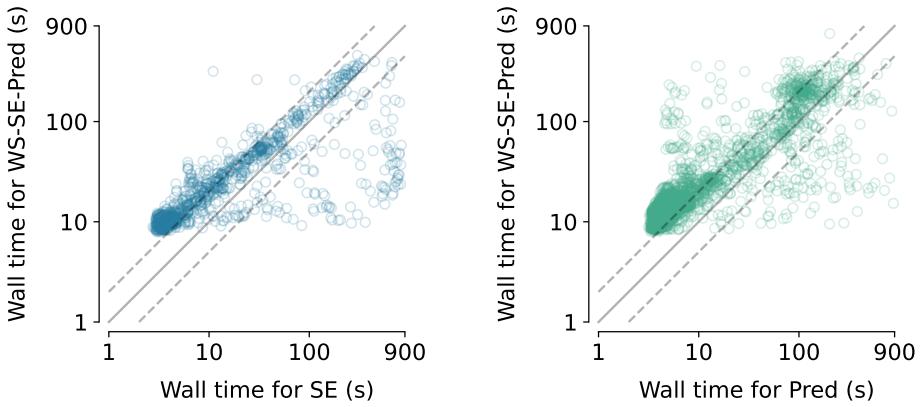
Figure A.11: Range programs generated using instrumentation for the lower bound and upper bound and the running example C_1

```
0 unsigned int on_lpath = 1;
1 unsigned int lcntr = 0;
2 int lredout(int pos) {return pos >= 4 || pos == 1; }
3
4 int div2WithReminderAbs(short input){
5   if (input < 0){
6     if (on_lpath){
7       if (!lredout(lcntr)){return;}
8       lcntr++;
9       if (on_lpath && lcntr >= 4){on_lpath = 0;}
10    }
11    input = -input;
12  }else{
13    if (on_lpath){
14      on_lpath = !lredout(lcntr);
15      lcntr++;
16    }
17  }
18  int rem = input;
19  int res = 0;
20  while (rem > 1){
21    if (on_lpath){
22      if (!lredout(lcntr)){return;}
23      lcntr++;
24      if (on_lpath && lcntr >= 4){on_lpath = 0;}
25    }
26    rem -= 2;
27    res++;
28  }
29  if (on_lpath){
30    on_lpath = !lredout(lcntr);
31    lcntr++;
32  }
33  if (input != 2 * res + rem){
34    if (on_lpath){
35      if (!lredout(lcntr)){return;}
36      lcntr++;
37      if (on_lpath && lcntr >= 4){on_lpath = 0;}
38    }
39    abort();
40  }else{
41    if (on_lpath){
42      on_lpath = !lredout(lcntr);
43      lcntr++;
44    }
45  }
46  return res;
47 }
```

Figure A.12: Range programs generated using instrumentation with the lower bound π_τ, where $\tau = \{input \mapsto 2\}$ or the running example C_1 called `rangeProg2`

A.3.2 Detailed Analysis of the Efficiency of Ranged Program Analysis

In the following we have a more detailed look at the efficiency of ranged program analysis using two different analyses within the ranged analyses. As we employ two different analyses, we compare the wall time of the ranged program analysis separately with both basic analyses.



(a) with Se (b) with Pred

Figure A.13: Scatter plots comparing the wall time of the basic analyses and Ws-Se-Pred

In Figure A.13, we compare the execution time of Ws-Se-Pred with symbolic execution and predicate abstraction. In general, the scatter plots are more diverse, meaning that some tasks are solved faster than the basic analysis, but others are solved slower by Ws-Se-Pred. As we employ work stealing, Ws-Se-Pred can solve tasks faster than the basic analyses. For example, if predicate abstraction is rather slow for solving a task but symbolic execution finds a solution fast, Ws-Se-Pred is faster than predicate abstraction due to the fact that symbolic execution steals the second range. When looking at the runtime increase given in Figure A.14, we get a similar result as for the configurations of ranged program analysis employing two instances of the same analysis: We again see the initial overhead, that has a huge impact for tasks that are solved fast. For larger and more complex tasks, the median increase in the wall time becomes smaller. Ws-Se-Pred is for the most complex tasks taking more than 100 seconds to be solved still slightly slower than symbolic execution but faster than predicate abstraction running standalone.

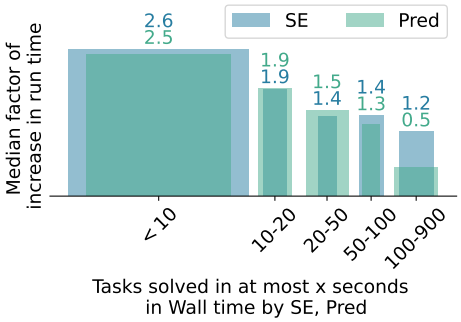
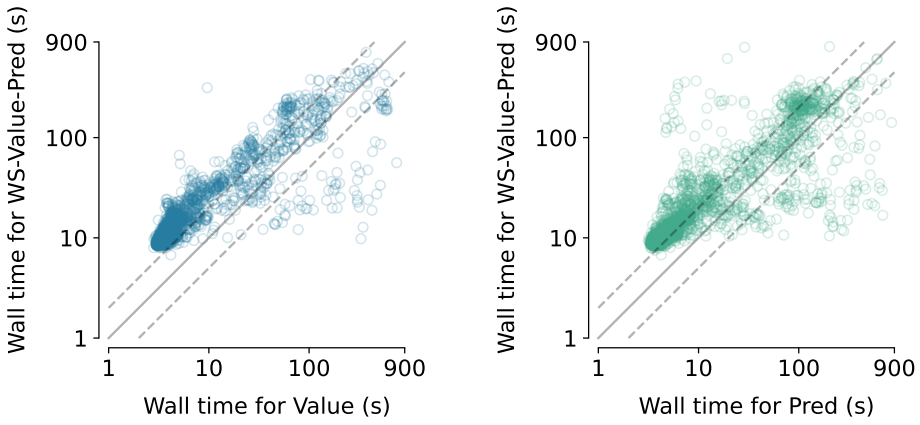


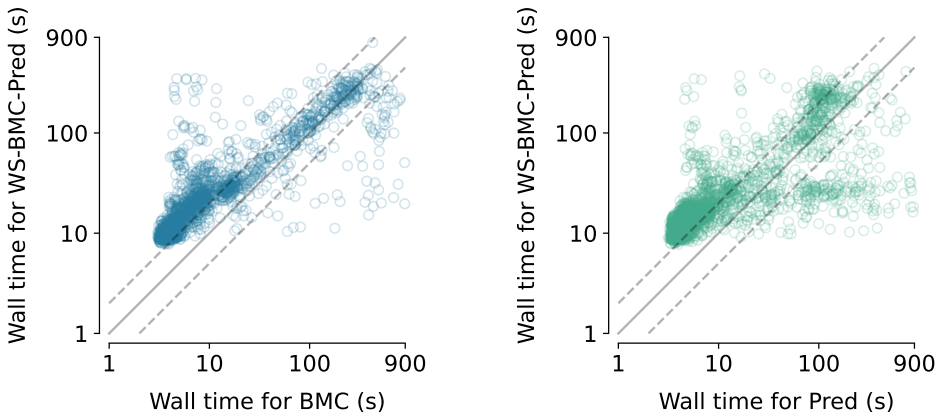
Figure A.14: Runtime increase for Ws-Se-Pred compared to the basic analyses

In Figure A.15, we compare the execution time of Ws-Value-Pred with value



(a) with Value (b) with Pred

Figure A.15: Scatter plots comparing the wall time of the basic analyses and Ws-Value-Pred

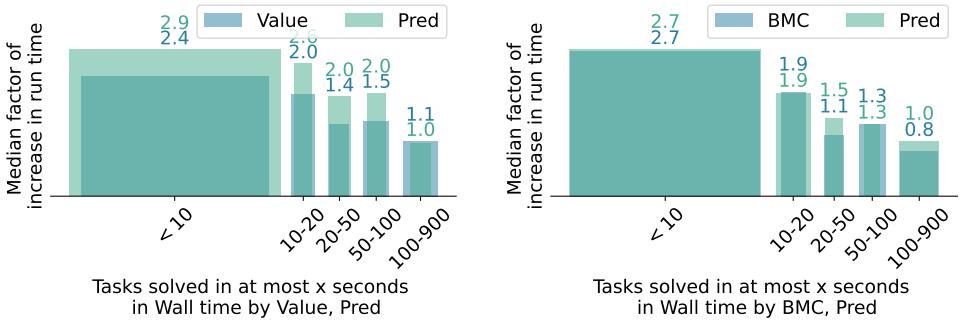


(a) with Bmc (b) with Pred

Figure A.16: Scatter plots comparing the wall time of the basic analyses and Ws-Bmc-Pred

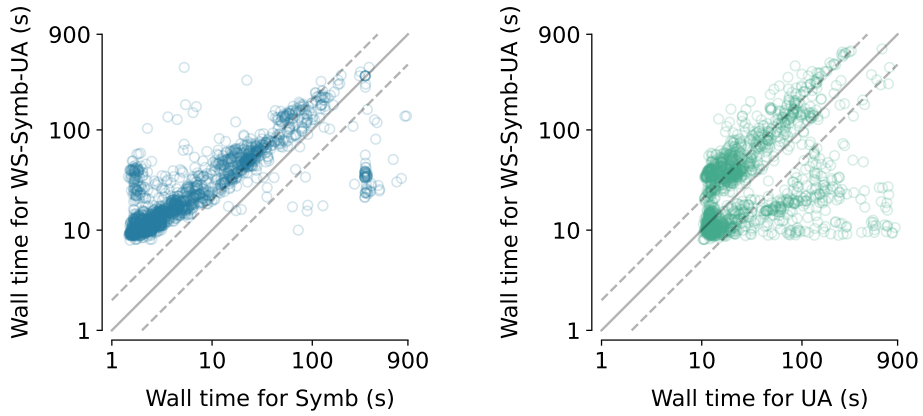
analysis and predicate abstraction and in Figure A.16 Ws-Bmc-Pred with BMC and predicate abstraction. We make the similar observations as for Ws-Se-Pred. Again, the scatter plots are more diverse, meaning that some tasks are solved faster than the basic analysis, but others are solved slower by Ws-Value-Pred or Ws-Bmc-Pred. When looking at the runtime increase given in Figure A.17, the same observations as for Ws-Se-Pred are made.

Lastly, we compare the efficiency of the two configurations of ranged program analysis using off-the-shelf tools. Now, the results look slightly different. In general, we make the same observations as before for Ws-Klee-UA compared to Klee and Ws-Symb-UA compared to Symbiotic. The main difference now is that in the UltimateAutomizer is in the experiments by far the slowest basic analysis. For example, it does not compute any result in 20 seconds or faster. Thus, a configuration of ranged program analysis employing work stealing is in many cases faster than UltimateAutomizer, but especially for small tasks slower than the other basic analysis running, as the the other



(a) for Ra-Value-Pred (b) for Ws-Bmc-Pred

Figure A.17: Runtime increase for Ws-Value-Pred and Ws-Bmc-Pred compared to the basic analyses.

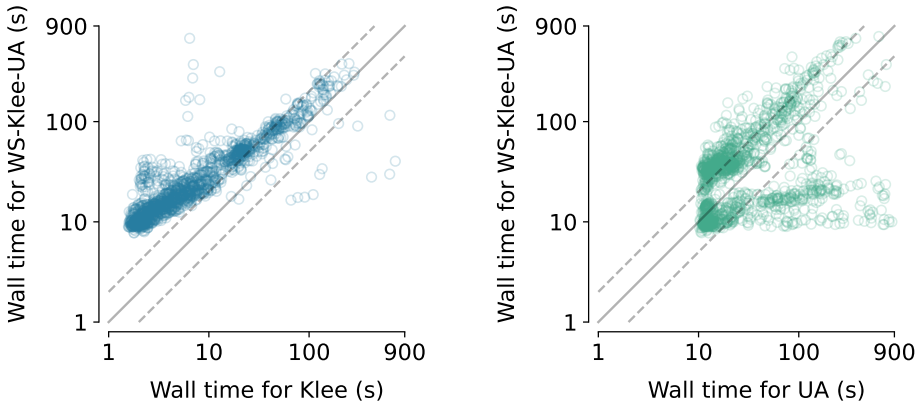


(a) with Symbiotic (b) with UltimateAutomizer

Figure A.18: Scatter plots comparing the wall time of the basic analyses and Ws-Symb-UA

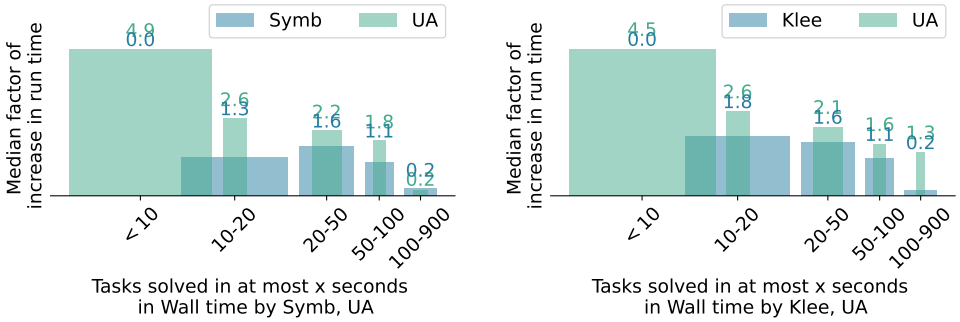
analysis solves both ranges one after another. Interestingly, for complex tasks, both Ws-Symb-UA and Ws-Klee-UA are faster than both basic analyses employed.

Finally, we compare the efficiency of instances of ranged program analysis that use work stealing with those who do not use this feature. For all five configurations we first of all observe that nearly all points are below the diagonal, meaning that the instances using work stealing are not slower than those not using work stealing. More precisely, we see for all configurations some tasks, where the instance using work stealing is faster than the other configuration. As mentioned above, UltimateAutomizer is slower than Klee and Symbiotic. Thus, Ws-Symb-UA and Ws-Klee-UA outperform Ra-Symb-UA and Ra-Klee-UA on some tasks.



(a) with Klee (b) with UltimateAutomizer

Figure A.19: Scatter plots comparing the wall time of the basic analyses and Ws-Klee-UA

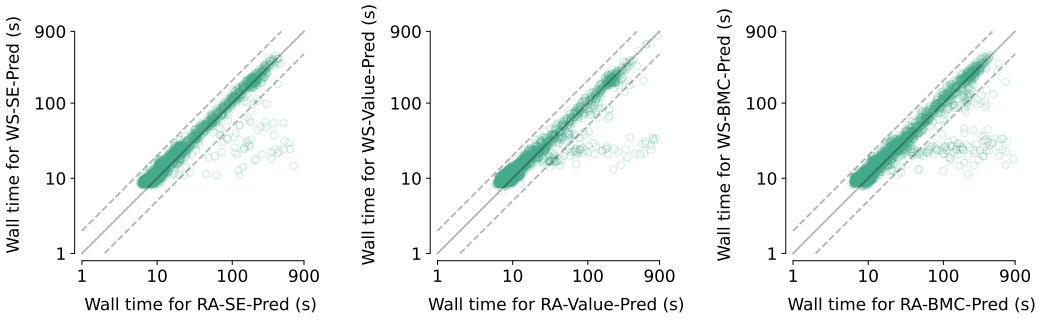


(a) for Ws-Symb-UA (b) for Ws-Klee-UA

Figure A.20: Runtime increase for Ws-Symb-UA and Ws-Klee-UA compared to the basic analyses.

Results

In summary, using ranged program analysis can in some cases increase the efficiency compared to one of the basic analyses employed, if the other one is significantly faster. For simple tasks, the overhead of ranged program analysis is observable but the effect decreases for more complex tasks. The two configurations Ws-Symb-UA and Ws-Klee-UA are for complex tasks faster than both employed basic analyses. In addition, using work stealing has no negative effect on the efficiency of ranged program analysis.



(a) Using Se and Pred (b) Using Value and Pred (c) Using Bmc and Pred

(d) Using Symbiotic and UA (e) Using Klee and UA

Figure A.21: Scatter plots comparing the wall time of instance of ranged program analysis having work stealing disabled and enabled

List of Figures

List of Tables

Index

Erklärung

Hiermit versichere ich an Eides statt, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg festgelegt sind, befolgt habe.

Ort, Datum	Unterschrift