Software and Systems Modeling
https://doi.org/10.1007/s10270-024-01155-3

SPECIAL SECTION PAPER

Exchanging information in cooperative software validation

Jan Haltermann[1] · Heike Wehrheim[1]

Received: 28 February 2023 / Revised: 20 October 2023 / Accepted: 17 January 2024
© The Author(s) 2024

Abstract

Cooperative software validation aims at having verification and/or testing tools *cooperate* on the task of correctness checking. Cooperation involves the exchange of information about currently achieved results in the form of (verification) artifacts. These artifacts are typically specialized to the type of analysis performed by the tool, e.g., bounded model checking, abstract interpretation or symbolic execution, and hence require the definition of a new artifact for every new cooperation to be built. In this article, we introduce a unified artifact (called Generalized Information Exchange Automaton, short GIA) supporting the cooperation of *over-approximating* with *under-approximating* analyses. It provides information gathered by an analysis to its partner in a cooperation, independent of the type of analysis and usage context within software validation. We provide a formal definition of this artifact in the form of an automaton together with two operators on GIAs. The first operation *reduces* a program by excluding these parts, where the information that they are already processed is encoded in the GIA. The second operation combines partial results from two GIAs into a single on. We show that computed analysis results are never lost when connecting tools via these operations. To experimentally demonstrate the feasibility, we have implemented two such cooperation: one for verification and one for testing. The obtained results show the feasibility of our novel artifact in different contexts of cooperative software validation, in particular how the new artifact is able to overcome some drawbacks of existing artifacts.

Keywords Cooperative software verification · Verification artifact · Test case generation · Component-based CEGAR

1 Introduction

Over the past years, automatic software validation (i.e., verification and testing) has become a mature field, with numerous tools providing various sorts of analyses (see, e.g., the annual competitions on software verification and testing [8, 9]). Still, the one-fits-all approach to software validation has not yet been found. All tools have their specific strengths and weaknesses, and tools efficiently solving one sort of analysis tasks might be slow at or even unable to solve other tasks.

To remedy this situation, cooperative software verification aims at having different tools cooperate on the task of software verification. This principle can not only be applied to verification but also to testing, and several different approaches combining various sorts of analyses exist today (e.g., [2, 5, 15, 16, 24, 27, 33, 34, 38, 39, 42, 48, 53–55, 75]). To achieve cooperation, tools need to *exchange information* gathered about a program during its analysis. To leverage the strengths of tools, we need to make sure that no results computed about a program are lost during this information exchange. To this end, existing cooperative approaches use various sorts of so-called *verification artifacts* [26] for information exchange, e.g., correctness witnesses [10], predicate maps [24] or violation witnesses [12]. The artifacts are, however, often specialized to the type of analysis performed, with the consequence of having to define a new form of artifact with every new cooperation.

In this work, we introduce a novel *uniform* verification artifact (called GIA) for the exchange of information, specifically focusing on the cooperation of *over- and under-approximating software analyses* (see Fig. 1), as many existing combinations successfully make use of these two types of analyses (e.g., [1, 2, 5, 20, 21, 24, 30, 33–35, 41, 42, 49,

Communicated by Holger Schlingloff and Ming Chai.

✉ Jan Haltermann
jan.haltermann@uol.de

Heike Wehrheim
heike.wehrheim@uol.de

[1] Department of Computing Science, Carl von Ossietzky Universität Oldenburg, Ammerländer Heerstraße 114-118, 26129 Oldenburg, Germany

Published online: 19 March 2024

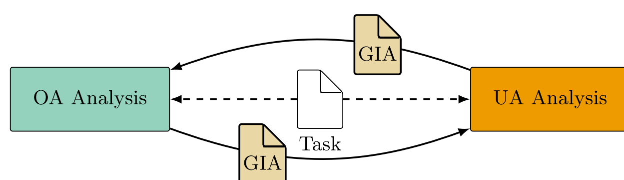


Fig. 1 Cooperation of over- and under-approximating analyses

51, 59, 70, 77, 78]). Over-approximating (OA) analyses build an over-approximation of the state space 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 program correctness.

Before defining the GIA—our new type of verification artifact—we first of all studied existing combinations of (cooperative and non-cooperative) analyses and the information they assemble and possibly exchange during an analysis. We also investigated what input formats existing tools accept. The majority of tools just take a program as input, however, there are also some tools already allowing for verification artifacts as additional inputs. With these insights at hand, we defined a new verification artifact in the form of a *generalized information exchange automaton* (GIA) which can express information generated by over- and under-approximating analyses in the context of software validation. More specifically, our artifact can encode information on (1) program paths which definitely or potentially lead to an error, i.e., (potential) counterexamples, (2) program paths which are already known to be safe, (3) program paths which are already known to be infeasible plus (4) additional constraints on program paths like state invariants. The unification of all such information in one verification artifact should in particular make the artifact *independent* of its usage, i.e., the semantics of the GIA should be the same in all usage contexts within software validation. Current artifacts, in particular the protocol automata of Beyer and Wehrheim [26], have differing meanings depending on their usage: sometimes the paths described by an automaton are the safe paths, and sometimes the paths leading to a property violation. By introducing 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 these different information exchanged in software validation while maintaining a uniform semantics.

Along with this new artifact, we also introduce two operations on it: *reducers* [18, 41] and *combiners*. A reducer allows to (syntactically) reduce a program to the part which a (prior) analysis has not yet completed (e.g., not yet proven safe). Reducers are required for cooperation of analysis tools which only take programs as inputs. A combiner allows combining computed analysis results given in two GIAs into one. We formally show that connecting tools via reducers and combiners guarantees computed analysis results to never be lost.

To demonstrate the feasibility of our approach and to show that GIAs are in fact usable in different scenarios, we have implemented two such cooperations employing GIAs as an exchange format. We have experimentally evaluated these cooperations on benchmarks of SV-COMP [9] and report on the outcomes, in particular how existing drawbacks in cooperation approaches caused by information loss can be overcome with this new artifact. Moreover, we observe that encoding information on reachable and unreachable program paths within the same artifact allows cooperative approaches to compute final results faster.

This article is an extended version of our conference paper [56], extending it with (1) a thorough discussion of related work, especially on existing artifacts and their shortcomings, (2) proofs of theorems, (3) an implementation and evaluation of an additional use case for GIAs which is cooperative test case generation and (4) a more detailed explanation of the application of GIAs in other use-cases.

2 Background

We generally aim at the validation of programs written in C. To be able to discuss and define formats for the information exchange, especially their semantics, we first provide some basic definitions on the syntax and semantics of programs, and then survey existing artifacts.

2.1 Program syntax and semantics

We represent a program as a *control-flow automaton (CFA)*. Intuitively, a CFA is a control-flow graph, where each edge is labeled with a program statement. More formally, a CFA C is a graph $C = (Loc, \ell_0, G)$ with a set of program locations Loc, the initial location $\ell_0 \in Loc$ and a transition relation $G \subseteq Loc \times Ops \times Loc$, where Ops contains all possible operations on integer variables,[1] namely assignments, conditions (for both loops and branches), function calls and return statements. We let $\mathcal{C}$ denote the set of all CFAs. Note that any program can be transferred into a CFA and any deterministic CFA into a program.

We assume the existence of two specific functions `error` and `random` which programs can call; the former can be used to represent violations of a specification (reachability of an error), the latter returns a non-deterministic value and is typically used to model inputs. We assume our programs to be deterministic except for this function `random`.

[1] We restrict the operations to integer variables for presentation only; the implementation covers C programs.

```
1 int main(){
2   int x = random();
3   if (x < 5) {
4     return 0;
5   } else {
6     x++;
7     if (x == 5) {
8       return 1;
9     } else {
10      return 2;
11    }
12  }
13 }
```

Fig. 2 An example program P for test case generation

For defining the semantics of CFAs, we let *Var* denote the set of all integer variables present in the program, *AExpr* the set of arithmetic and *BExpr* the set of Boolean expressions over the variables in *Var*. A *state* c is a mapping of the program variables to integers, i.e., $c : Var \to \mathbb{Z}$. We lift this mapping to also contain evaluations of the arithmetic and Boolean expressions, such that c maps *AExpr* to $\mathbb{Z}$ and *BExpr* to $\mathbb{B} = \{0, 1\}$. A finite *syntactic program path* is a sequence $\tau = \ell_0 \xrightarrow{g_1} \ldots \xrightarrow{g_n} \ell_n$ s.t. $(\ell_i, g_{i+1}, \ell_{i+1}) \in G$ for each transition. We extend a syntactic path to a *semantic program path* $\pi = \langle c_0, \ell_0\rangle \xrightarrow{g_1} \ldots \xrightarrow{g_n} \langle c_n, \ell_n\rangle$, by adding states to each location, where c_0 assigns the value 0 to all variables, and state changes for $\langle c_i, \ell_i\rangle \xrightarrow{g_{i+1}} \langle c_{i+1}, \ell_{i+1}\rangle$ are defined as follows: If g_{i+1} is an assignment of the form $x = a, x \in Var, a \in AExpr$, $c_{i+1} = c_i[x \mapsto c_i(a)]$, for assignments x =`random()` $c_{i+1} = c_i[x \mapsto z], z \in \mathbb{Z}$, otherwise $c_{i+1} = c_i$.

Note that we do not require that a semantic path meets all its Boolean conditions, as we want to distinguish between feasible and infeasible semantic paths: A semantic path is called *feasible*, if for each condition $g_{i+1} = b$ on the path $c_i(b) = \mathit{true}$ holds, otherwise it is called *infeasible*. We say that a path π *reaches* location $\ell \in Loc$ if $\ell = \ell_n$. If no feasible semantic path reaches a location $\ell \in Loc$, it is called *unreachable*. The set of all semantic paths (or in short, paths) of a CFA C is denoted by $\mathcal{P}(C)$.

Figure 2 contains a C-program and Fig. 3 its corresponding CFA. Let us assume our validation task on this program is *test case generation*, more specifically generating test inputs (values returned by `random`) which cover all branches of the program. A tool would then need to generate inputs leading to paths such that each node of the CFA marked in gray is reached by at least one path. A feasible path that reaches the location ℓ_3 is: $\langle\{x \mapsto 0\}, \ell_1\rangle \xrightarrow{x=\text{random();}} \langle\{x \mapsto 3\}, \ell_2\rangle \xrightarrow{x<5} \langle\{x \mapsto 3\}, \ell_3\rangle \xrightarrow{\text{return 0;}} \langle\{x \mapsto 3\}, \ell_4\rangle$. The location ℓ_7 is unreachable, as x is always greater than 5 at ℓ_6, and thus the branch cannot be covered.

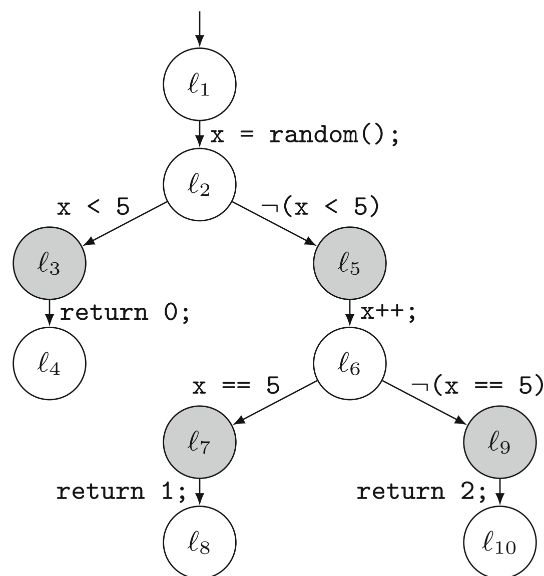


Fig. 3 CFA for program in Fig. 2, where nodes after branching points are marked gray

3 Related work

Combining different analyses is commonly applied to enha nce the performance in verification or test case generation. In verification, the goal is to check whether a program adheres to certain specifications, in test case generation tools aim at finding test cases covering a set of test goals. Following [26], these combinations of analyses can be divided into four categories: Portfolios, selection-based approaches, cooperations, and conceptual integrations. Portfolio-based approaches [7, 43, 50, 58, 61, 62, 83, 85] run multiple components sequentially or in parallel and select the first computed result, whereas selection-based approaches [6, 40, 44, 47, 68, 78, 84] select one verification component upfront based on the task. Both concepts do not foresee an information exchange between different components; hence, we do not consider them further.

In this work, we aim for finding a unified artifact that is applicable for many existing concepts of cooperative software validation combining over-approximating(OA) and under-approximating(UA) components. We present different cooperation-based approaches and approaches using a conceptual integration next. Many tools combine OA and UA tools, but there are some ideas that either combine only OA or only UA tools.

3.1 Conceptual integration

A conceptual integration is a white-box combination of multiple components, where the components exchange information not via clearly defined artifacts but rather using internal formats, method calls or accessing shared data structures.

Sequential Combinations. In sequential combinations, tools are executed in a sequential manner, where the information computed by the former tool is given to the next. Different approaches combine an OA verification tool with an UA testing approach (like dynamic symbolic execution [48] or robustness testing [63]) to guide the testing tool or analyze the non-verified program parts. FuSeBMC [3] combines different UA components for test case generation: In the first phase, a fuzzing and a bounded model checking tool are run in parallel trying to cover all test goals. Afterward, the covered goals and the inputs for covering them are given to a selective fuzzer, that uses the given information to cover the remaining test goals in the second phase.

Interleaved Combinations. Interleaved combinations [2, 4, 5, 46, 49, 51, 60, 64, 73, 76, 77, 81, 86] can be seen as an extension of the sequential combinations, where each component may be called multiple times. In Smash [49], an OA predicate analysis is combined with dynamic test generation (UA), 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, computed by the OA and UA analyses. 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 [51] (with its implementation in the tool Yogi [77]) and Dash [5] share the idea of combining predicate analysis 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 abstract reachability graph (ARG) [2].

The idea of concolic testing is to enrich a testing tool with concrete test inputs that may lead to unexplored parts of the program [28, 32, 71–74, 79, 80, 82]. The concrete inputs are computed using an over-approximating symbolic execution. Daca et al. [42] 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.

Summary. Approaches using conceptual integration 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. Several approaches use an ARG for information exchange.

3.2 Cooperative approaches

In contrast to conceptual integration, cooperative approaches use components as black boxes and information is exchanged only using clearly defined verification artifacts.

Sequential Combinations.

One of the earliest ideas for cooperative software validation in a cooperative manner is cooperative manner is conditional model checking (CMC) [15]. Therein, so-called conditional model checkers (OA tools) are executed sequentially, where each generates a predicate specifying under which condition the program adheres to the specification. The conditions are exchanged via a condition automaton. In [41], the second model checker is replaced using a testing tool, yielding a combination of an OA and an UA tool. The information from the condition automaton is transformed into a reduced program to be able to use arbitrary testing approaches. The general construction of conditional verifiers using reduction is presented in [18] and different reduction and folding strategies are proposed in [17]. In [33] and [34], an OA verification tool analyzes the program, adding conditions under which the program is safe directly into the code. These conditions are then either further analyzed, in [33] or tested dynamic symbolic by execution engine (UA) in [34].

In CoVEGI [55], a OA analysis tool is cooperating with an invariant generation tool. The invariant generation tool computes on-demand invariants for the analysis tool to enhance its performance. The invariants are encoded within correctness witnesses.

Interleaved Combinations. CoVeriTest [16, 65, 66] generalizes the idea of [42] by combining arbitrary verifiers (OA) for test case generation. 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. The condition is then used for cooperation. A similar approach only employing testing tools is presented in [7].

Counterexample-guided abstraction refinement (CEGAR) [35] is a technique for iteratively refining the abstraction. The idea is implemented in many tools [1, 20, 21, 30, 59, 70, 78], where potential counterexamples, spurious counterexamples and precision increments (mostly in form of new predicates) are exchanged between the components. A decomposed and cooperative formalization is presented in [24], where standardized formats, namely correctness and

violation witnesses, are used for exchanging the information. The concept of property-directed k-induction [52, 69] is formalized in a cooperative way in [27], where the information exchanged of generated invariants and traces is realized using no standardized format.

Summary. In cooperative software validation, components exchange information on programs partially verified (or simply explored) under certain conditions, (helpful) invariants, potential and spurious counterexamples, and newly discovered predicates. The information is encoded using the standardized formats of condition automaton, correctness, and violation witnesses.

4 Existing artifacts

As seen in the related work section, there are different artifacts that are already used either for cooperative validation, for witness validation or storage of correctness proofs [10, 12, 18, 26, 42, 55, 67]. An overview of existing verification witnesses is given in [11]. In the following, we define the artifacts (1) protocol automaton, (2) violation and (3) correctness witness, (4) condition automaton and (5) abstract reachability graph formally. We discuss their suitability for representing information exchanged between OA and UA analysis and provide concrete examples.

All of the presented formats can encode information about (non-)violation of some reachability properties, i.e., the (non-)reachability of a set $Prop \subseteq Loc$ of locations of the CFA.

4.1 Requirements

Before we discuss whether the artifacts used are suitable in the general setting, we summarize the requirements based on existing use cases for such a general format for exchanging information between OA- and UA analyses. Following existing cooperations, an 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 are either feasible and reach an error state or are infeasible (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 a context-independent semantics.

4.2 Protocol automaton

The protocol automaton, first introduced in [12] and extended in [26], in general describes a set of semantic paths. It can be used to define different existing verification artifacts in a uniform way, as the semantics of the described paths is context-dependent.

Definition 1 A *protocol automaton* $A_p = (Q, \Sigma, \delta, q_0, F)$ for a program represented as CFA $C = (Loc, \ell_0, G)$ 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 *invariants* 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* about program variables describing conditions when executing these transitions (see Def. 3 below). The connection between a semantic path π in the CFA C and paths that are described by A_p is established via matched paths. A_p *matches* a path $\pi = \langle c_0, \ell_0 \rangle \xrightarrow{g_1} \dots \xrightarrow{g_n} \langle c_n, \ell_n \rangle$ if there is a sequence $\rho = (q_0, \psi_0) \xrightarrow{(G_1, \varphi_1)} \dots \xrightarrow{(G_k, \varphi_k)} (q_k, \psi_k)$, $0 \leq k \leq n$, with $(q_{i-1}, \psi_{i-1}) \xrightarrow{(G_i, \varphi_i)} (q_i, \psi_i) \in \Sigma$, such that

1. $\forall i, 1 \leq i \leq k : g_i \in G_i$,
2. $\forall i, 1 \leq i \leq k : c_i \models \varphi_i$,
3. $\forall i, 0 \leq i \leq k : c_i \models \psi_i$.

A_p *covers* π, if A_p matches π, $k = n$ and $q_k \in F$.

In Fig. 4 to 6, three protocol automata are shown (a violation witness, a correctness witness and a condition automaton, see below), each of them covers a set of paths from the CFA of Fig. 3.

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. Thus, each tool working with protocol automata has to be aware of the type of protocol automaton given to it and its semantics. Depending on the encoded artifact, matched paths can, among others, encode paths leading to a property violation (in Fig. 4), or paths not reaching any nodes from *Prop* (in Fig. 5). Consequently, it is impossible to mark within one protocol automaton both, a path to a node from *Prop* as unreachable and state that another path reaches a different node from

Prop. Hence, (R6) and either (R1) or (R3) from Sec. 4.1 is not fulfilled.

Next, we discuss three artifacts which are specializations of protocol automata.

4.3 Violation witness

A violation witness [12] is used to encode a set of feasible semantic paths that lead to a property violation. It can be represented as protocol automaton $A_{VW} = (Q, \Sigma, \delta, q_0, F)$, where each state has only a trivial state invariant: $\forall(q, \varphi) \in Q : \varphi = true$. The assumptions in A_{VW} can contain constraints on the variable values. Semantically, paths covered by A_{VW} contain a property violation. An example for a violation witness represented as protocol automaton for the CFA from Fig. 3 with $Prop = \{\ell_4\}$ is depicted in Fig. 4. The only path covered by A_{VW} is $\pi = \langle c_0, \ell_1\rangle \xrightarrow{x\text{=random();}} \langle c_1, \ell_2\rangle \xrightarrow{x<5} \langle c_1, \ell_3\rangle \xrightarrow{\text{return 0;}} \langle c_1, \ell_4\rangle$, where $c_0 = \{x \mapsto 0\}$ and $c_1 = \{x \mapsto 1\}$. Hence, following π leads to a property violation in the example program P.

By design, the violation witness does not allow the use of state invariants. Thus, its semantics does neither allow to encode that a path does not reach a node from *Prop* (i.e., is safe) or is infeasible or some justification of this in the form of state invariants. Hence, (R3), (R4), and (R5) from Sec. 4.1 are not fulfilled.

4.4 Correctness witness

A correctness witness [10] is used to encode that a program is safe (no node from *Prop* is reachable). It can be represented as protocol automaton $A_{CW} = (Q, \Sigma, \delta, q_0, Q)$, where all states are final states and each edge is labeled with trivial assumptions: $\forall(q, (G, \psi), q') \in \delta : \psi = true$. States may contain a state invariant that justifies why the program is correct. Semantically, paths covered by A_{CW} do not contain a property violation. An example for a correctness witness represented as condition automaton for the CFA from Fig. 3 with $Prop = \{\ell_8\}$ is depicted in Fig. 5, where $*$ denotes any operation from Ops As ℓ_8 is unreachable, a correctness witness can be generated. In Fig. 5, the invariant $x > 5$ is associated with the state q_6. As the condition $c \models x > 5$ holds for all states $\langle c, \ell\rangle$ on a feasible path that is covered by A_{CW} and contains $\langle c, \ell_6\rangle$, $x > 5$ is in fact a justification for the unreachability of ℓ_8.

Correctness witnesses do not allow to specify the reachability of nodes from *Prop* nor to encode partial results. Therefore, encoding paths to nodes from *Prop* as well as marking that only certain paths of the program (and not the whole program) are safe is impossible. Hence, (R1) and (R2) from Sec. 4.1 are not fulfilled.

4.5 Condition automaton

A condition automaton [18] states which semantic paths of the program are already successfully verified and under which condition. It can be represented as a protocol automaton $A_{CA} = (Q, \Sigma, \delta, q_0, F)$, where each state has only trivial state invariants ($\forall(q, \varphi) \in Q : \varphi = true$) and accepting states cannot be left ($\forall(q_f, \cdot, q) \in \delta : q_f \in F \Rightarrow q \in F$). Semantically, paths covered by A_{CA} do not contain a property violation. In contrast to a correctness witness A_{CW}, a condition automaton can contain assumptions, allowing to specify the unreachability under that assumption. In Fig. 6, we depict a condition automaton A_{CA} for program P with $Prop = \{\ell_4\}$, where $*$ again denotes any operation from Ops. The partial result, e.g., generated by a simple reachability analysis, covers all paths containing ℓ_5 and marks them as safe (under the trivial assumption $true$). Note that A_{CA} correctly encodes the information that a part of the program satisfies the property, even though the program contains a property violation (c.f. Figure 4).

Although condition automata can mark certain regions as safe, paths (potentially) leading to a node from *Prop* cannot be encoded. In addition, condition automata do not allow adding state invariants. Hence, (R2) and (R5) from Sec. 4.1 are not fulfilled.

4.6 Abstract reachability graph

An abstract reachability graph [14] represents the abstract state space containing the analysis results computed as a graph. It is used within different tools, e.g., CPACHECKER [21]. As the ARG can be generated by any analysis, not necessarily using predicates for the abstraction, it cannot be formalized as a protocol automaton. We define an ARG $R = (N, succ, root, F, prec)$, with a set of abstract states N, a successor relation $succ \subseteq N \times G \times N$, the initial node $root \in N$, 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.

An example of an (intermediate) ARG generated by an interval analysis [37] and a location analysis [13] is depicted in Fig. 7. Each abstract state comprises a unique name, an interval for the variable x, and a location from the CFA. Frontier states are marked in gray, thus the abstract state q_6 is not fully explored. As the node q_5 is explored and has only a single successor q_6, the ARG also contains the information that ℓ_7 is unreachable, as no abstract state contains ℓ_7.

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,

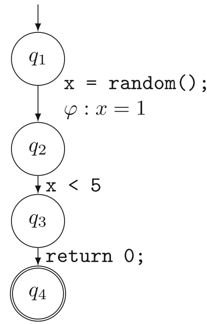


Fig. 4 Violation Witness A_{VW} with $Prop = \{\ell_4\}$ for P

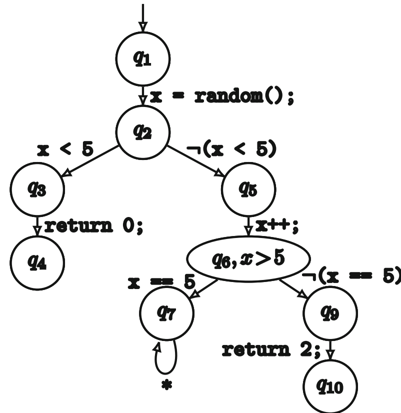


Fig. 5 Correctness Witness A_{CW} with $Prop = \{\ell_8\}$ for program P

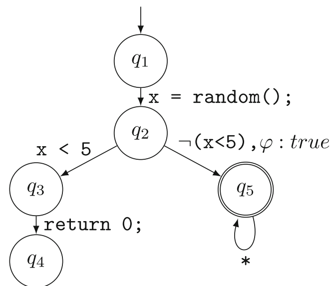


Fig. 6 Condition Automaton A_{CA} with $Prop = \{\ell_4\}$ for program P

which makes an exchange of ARGs between different analyses in a general setting impossible.

Summary. In summary, none of the existing artifacts is able to encode all desired information and is usable independent of the employed tools while maintaining one semantics. Next, we introduce a new format that overcomes these limitations.

5 Validation artifact GIA

In this work, we focus on two different validation tasks on programs, verification, and test case generation, performed by over- and under-approximating analyses. For *verification*, the goal is to show the non-reachability of certain error loca-

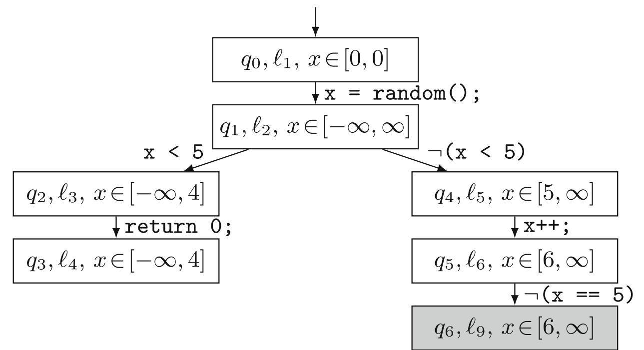


Fig. 7 ARG generated by an interval analysis with $Prop = \{\ell_8\}$ for program P. Frontier states are marked in gray

tions. To this end, we fix a *safety property* $S = (\ell, \omega)$ as a pair of location $\ell \in Loc$ and condition $\omega \in BExpr$ which has to hold at ℓ. In practice, this is encoded in the CFA using two edges $(\ell \xrightarrow{\neg\omega} \ell_e \xrightarrow{error();} \ell_{e'})$. Note that there can be multiple safety properties for a program. 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, especially `error`. To specify these paths, a sequence of return values (called *test suite*) for the calls to `random` suffices (as `random` 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 L, $L \subseteq Loc$. A target node is a node that either has a single outgoing edge labeled `error` (for verification) or is 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. In Fig. 3, the target nodes for test case generation are $L = \{\ell_3, \ell_5, \ell_7, \ell_9\}$.
Our overall objective is next to define an artifact with one semantics that is valid for most type of exchanged information. In general, UA (under-approximating) and OA (over-approximating) tools either aim at showing that target nodes are reachable (for example a call to `error` or a branch that needs to be covered) or that (a part of) the program does not reach any target node (i.e., program is safe). The overall goal is achieved when for each target node either a path reaching it is found or it is proven unreachable.

Summarizing Sec. 4.1, the information exchanged between UA and OA tools thus 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, we need the artifact 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).

So far, none of the existing artifacts discussed in Sec. 4 is able to encode all this information while maintaining one semantics for the automaton. Inspired by the idea of three-valued logics (e.g., for three-valued model checking [29]), we extend the condition automata of [18] by introducing *three* different, disjoint sets of accepting states, one for each type of exchanged information.

Definition 2 A *generalized information exchange automaton for over- and under-approximative analysis (GIA)* $A = (\mathcal{Q}, \Sigma, \delta, q_0, F_{ut}, F_{rt}, F_{cand})$ consists of

- a finite set $\mathcal{Q} \subseteq \Omega \times BExpr$ of *states* (each being a pair of a name of some set Ω and a Boolean condition) and an *initial state* $(q_0, true) \in \mathcal{Q}$,
- an *alphabet* $\Sigma \subseteq 2^G \times BExpr$,
- 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 Def. 1) 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.[2] 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. $\forall q_{cand} \in F_{cand} : (q_{cand}, *, q_{cand}) \in \delta$.

Figure 8 depicts an example of a GIA for the program of Fig. 2 with target nodes $L = \{\ell_3, \ell_5, \ell_7, \ell_9\}$, where $F_{rt} = \{q_3\}$, $F_{ut} = \{q_7\}$ and $F_{cand} = \{q_5, q_9\}$.

To fulfill the requirement (R6) from Sec. 4.1, we need to define a context independent semantics. Thus, the three sets

[2] This property is useful to have a single path π covering several nodes from F_{cand} (e.g., for branch coverage).

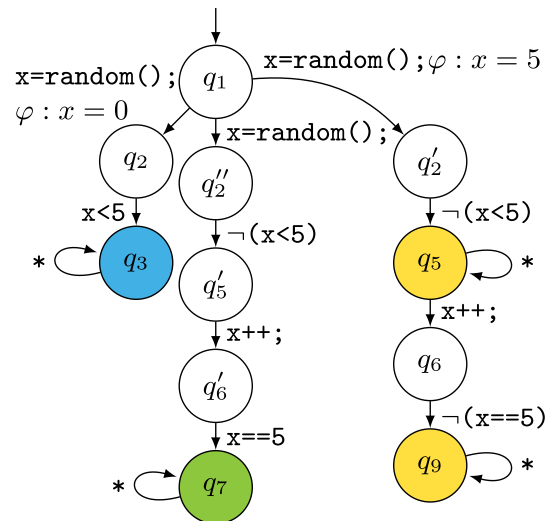


Fig. 8 A GIA generated during cooperative test case generation for the example program of Fig. 2 with states of F_{ut} marked green, of F_{rt} blue and of F_{cand} yellow. We elide state invariants (all $true$) and depict for transitions only the operation and non-true conditions

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 an automaton 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 and the program represented as CFA.

Definition 3 A GIA $A = (\mathcal{Q}, \Sigma, \delta, q_0, F_{ut}, F_{rt}, F_{cand})$ *covers a path* $\pi = \langle c_0, \ell_0 \rangle \xrightarrow{g_1} \ldots \xrightarrow{g_n} \langle c_n, \ell_n \rangle$ if there is a sequence $\rho = (q_0, \psi_0) \xrightarrow{(G_1, \varphi_1)} \ldots \xrightarrow{(G_k, \varphi_k)} (q_k, \psi_k)$, $0 \leq k \leq n$, with $(q_{i-1}, \psi_{i-1}) \xrightarrow{(G_i, \varphi_i)} (q_i, \psi_i) \in \Sigma$ (called run), such that

1. $q_k \in F_{ut} \cup F_{rt} \cup F_{cand}$,
2. $\forall i, 1 \leq i \leq k : g_i \in G_i$,
3. $\forall i, 1 \leq i \leq k : c_i \models \varphi_i$,
4. $\forall i, 0 \leq i \leq k : c_i \models \psi_i$.

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

In contrast to protocol automata, we allow that the run ρ has fewer states than the path π, as each state from $F_{ut} \cup F_{rt} \cup F_{cand}$ has a transition to itself. Depending on the parameter value for X-cover, we define three sets of paths (languages) of a GIA A: $\mathcal{P}_{ut}(A)$, $\mathcal{P}_{rt}(A)$ and $\mathcal{P}_{cand}(A)$. These three sets are then used to establish the connection between a GIA A and a CFA C: If, e.g., a path $\pi \in \mathcal{P}(C)$ reaches a target node ℓ and $\pi \in \mathcal{P}_{rt}(A)$, ℓ is denoted reachable by A. The GIA depicted in Fig. 8 thus contains the information that ℓ_3 is

reachable when the condition $x = 0$ holds, ℓ_7 is unreachable and that ℓ_5 and ℓ_9 are candidates for being reached when the condition $x = 5$ holds.

With these definitions at hand, we can formally define the *correctness* of the analysis information in a GIA. Thereby, we are able to later on reason about the correctness of combinations of tools in a cooperative setting.

Definition 4 Let A be a GIA, C a CFA and $L \subseteq Loc$ a set of target nodes. A is said to be *correct wrt. C and L* if $\mathcal{P}_{ut}(A) \subseteq \{\pi \in \mathcal{P}(C) \mid \pi$ is infeasible or π is feasible and reaches no $\ell \in L\}$ and $\mathcal{P}_{rt}(A) \subseteq \{\pi \in \mathcal{P}(C) \mid \pi$ is feasible and reaches some $\ell \in L\}$.

Correctness thus means the automaton correctly (according to the program) 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 L are encoded within the program C. The soundness is also needed to reason about the correctness of combinations of tools in a cooperative setting.

Definition 5 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 $A, A' \in \mathcal{A}, C \in \mathcal{C}$ with $\mathsf{tool}(A, C) = A'$ we have

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

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

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

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

Finally, we can define when verification or test case generation is *completed*, namely, when a correct GIA A is generated for a CFA $C = (Loc, l_0, G)$ such that for all target nodes t there exists some $\pi \in \mathcal{P}_{rt}(A) \cup \mathcal{P}_{ut}(A)$ such that π reaches t (all target nodes covered or unreachable).

6 Using GIAs in cooperative validation

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*. We next briefly summarize the existing approaches of cooperative validation presented in Sec. 3 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.

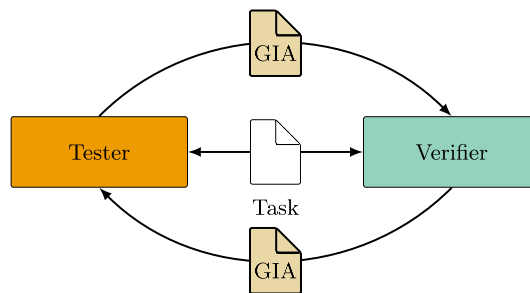


Fig. 9 Cooperative Test Case Generation using GIA as exchange formats

Cooperative Test Case Generation. 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 Fig. 9. 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. [42] 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 [16, 65, 66] and conditional testing [7]. In contrast to the approach depicted in Fig. 9, 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 within $\mathcal{P}_{rt}$. In CoVeriTest, two OA analyses are combined for test case generation, each of them reporting the candidate test cases within $\mathcal{P}_{cand}$ and explored paths in $\mathcal{P}_{ut}$. Each of them is equipped with a UA tool that validates all candidates and stores them within $\mathcal{P}_{rt}$.

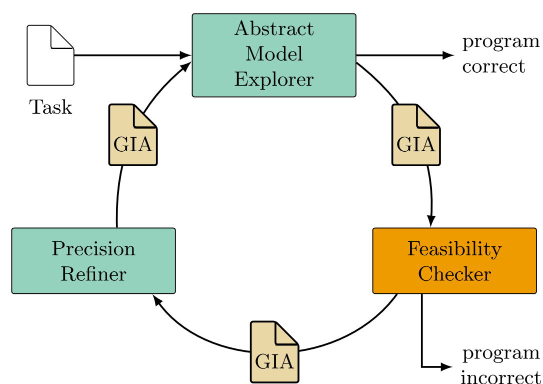


Fig. 10 Component-based CEGAR using GIA as exchange formats

Cooperative Verification Using CEGAR.

The goal of software verification is to show that none of the target nodes are reachable. CEGAR is a scheme that is commonly used in software verification. In [24], the scheme has been presented in a decomposed version applicable for cooperative verification, called CC- WIT. Therein, an abstract model explorer uses a given precision in the form of predicates to explore the state space searching for a feasible path to a target node. If no such path is detected, the program is safe. Otherwise, a potential counterexample is given to a feasibility checker checking if the counterexample is spurious. If a real counterexample is found, the verification stops with the outcome "not safe", else a precision refiner is started to refine the abstraction by generating new predicates. In [24], violation and correctness witnesses are used for exchanging the information. A unification of the information exchanged within CEGAR using GIA is depicted in Fig. 10 and called CC- GIA. The Abstract Model Explorer is an OA component building an abstraction of the state space of the program while it reports the candidates for counterexamples within $\mathcal{P}_{cand}$ and may also mark explored safe paths within $\mathcal{P}_{ut}$. The Feasibility Checker is an UA component that inspects the candidates for counterexamples and moves them to $\mathcal{P}_{rt}$ when it can show them to be real. Otherwise, the path is marked as a candidate for being infeasible by the Feasibility Checker and given to the Precision Refiner, the second OA component within CC- GIA. Its task is to find a set of predicates showing the infeasibility of the spurious path. When Precision Refiner computed the new predicates, the path is moved to $\mathcal{P}_{ut}$, where the predicates are given as state invariants for the path.

A similar combination of components appears in [5, 77] in a non-cooperative form, where precision refiner and abstract model explorer are a single component. Nevertheless, one could realize these concepts in a cooperative setting as described above.

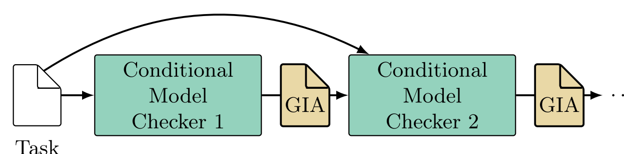


Fig. 11 Component based CEGAR using GIA as exchange formats

Cooperative Verification via Conditional Model Checking.

In contrast to the approaches discussed earlier, the concept of conditional model checking [15, 18] 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 Fig. 11. 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 [41], the second conditional verifier is replaced by a testing tool, yielding a cooperation between OA and UA tools.

Cooperation on Invariant Generation. In COVEGI [55], an OA analysis (the Main Verifier) is supported by a Helper Invariant Generator, as depicted in Fig. 12. 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, more precisely a loop invariant, 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.

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 should use a set of paths within $\mathcal{P}_{cand}$, either by using all target

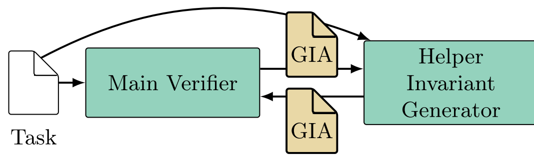


Fig. 12 Component based CEGAR using GIA as exchange formats

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.

7 GIA and off-the-shelf tools

The scenarios sketched in Sec. 6 assume that all tools potentially employed understand GIAs. This is, however (or rather, of course) not the case. To still enable cooperation of tools, in particular, while still using 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 Fig. 13 a combination of an OA tool and UA tool cooperation on the task of cooperative test case generation. In the scenario, we assume that we want to use an off-the-shelf tester that is UA and a Verifier, as depicted in Fig. 9. When the Verifier generates a GIA containing the information that certain paths of the program are unreachable, the REDUCER,[3] removes these paths from the program and generates a reduced program. This program is given to the Off-the-shelf Tester generating test cases for the reduced 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 Verifier. The resulting GIA is then given to the Verifier and the cycle starts anew.

7.1 Reducer

For the first operation on GIAs, we use the concept of *reducers* as introduced in [18, 41]. A reducer reduces a program to a certain part by removing some paths, and thereby allowing 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 them that are *rt*-covered.

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

Algorithm 1 X-REDUCER

Input: CFA $C = (Loc, \ell_0, G)$
  GIA $A = (\mathcal{Q}, \Sigma, \delta, (q_0, \psi_0), F_{ut}, F_{rt}, F_{cand})$
Output: CFA $C_r = (Loc_r, \ell_0^r, G_r)$

1: // Call existing reducer
2: (Loc_r, ℓ_0^r, G_r)
  $:= \texttt{Reducer}(C, (\mathcal{Q}, \Sigma, \delta, (q_0, \psi_0), F_X))$
3: **if** $F_{cand} \neq \emptyset$ **then**:
4:   $keep := \emptyset$
5:   **for each** $\ell = (l_i, (q_i, \psi_i)) \in Loc_r$
    s.t. $(q_i, \psi_i) \in F_{cand}$ **do**
6:     add all predecessors and successors of ℓ in Loc_r to $keep$
7:   **end for**
8:   **for each** $\ell \in Loc_r$ **do**
9:     **if** $\ell \notin keep$ **then**
10:       Remove ℓ from Loc_r;
11:       Remove all $(\ell, \cdot, \cdot), (\cdot, \cdot, \ell)$ from G_r
12:     **end if**
13:   **end for**
14: **end if**
15: **return** (Loc_r, ℓ_0^r, G_r)

Definition 6 An X-*reducer* for $X \in \{ut, rt\}$ is a mapping $red_X : \mathcal{C} \times \mathcal{A} \to \mathcal{C}$ satisfying

$$\forall C \in \mathcal{C}, A \in \mathcal{A} : P \subseteq \mathcal{P}(red_X(C, A)) \subseteq \mathcal{P}(C)$$

$$\text{where } P = \begin{cases} \mathcal{P}(C) \backslash \mathcal{P}_X(A) & \text{if } F_{cand} = \emptyset \text{ in } A \\ \mathcal{P}_{cand}(A) \backslash \mathcal{P}_X(A) & \text{otherwise.} \end{cases}$$

A reducer for $X = ut$ in the case that $F_{cand} = \emptyset$ is already existing [18]. In Alg. 1 we provide a parameterized reducer for both values of X, building on the existing one.[4] 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 3 to 13). We get the following result:

Theorem 1 *Algorithm 1 is an X-reducer according to Definition 6.*

Proof We first show that Algorithm 1 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

[4] Algorithm 1 assumes for representation purposes that the GIA does not contain state invariants. A full construction, covering this aspect is given in Appendix A.

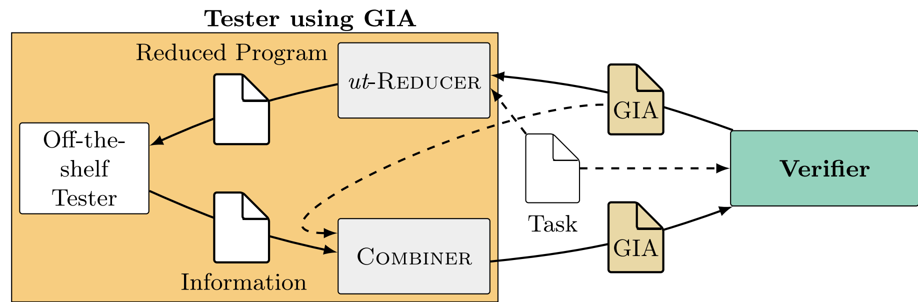


Fig. 13 Cooperative test case generation using *ut*-Reducer and Combiner

present in the original program [18]. 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 1 work correctly if $F_{cand} = \emptyset$. In case that $F_{cand} \neq \emptyset$, the reducer has to generate a program that contains at least all paths $cand$-covered by A. In lines 3-13 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 1 also works in this case concluding the proof. □

7.2 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 7 A *combiner* is a partial mapping $comb : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ which is defined on *consistent* GIAs A_1 and A_2 with $\mathcal{P}_{ut}(A_1) \cap \mathcal{P}_{rt}(A_2) = \emptyset = \mathcal{P}_{rt}(A_1) \cap \mathcal{P}_{ut}(A_2)$ such that

$$\forall A_1, A_2 \in \mathcal{A} :$$
$$\mathcal{P}_{ut}(comb(A_1, A_2)) = \mathcal{P}_{ut}(A_1) \cup \mathcal{P}_{ut}(A_2)$$
$$\wedge\ \mathcal{P}_{rt}(comb(A_1, A_2)) = \mathcal{P}_{rt}(A_1) \cup \mathcal{P}_{rt}(A_2)\ .$$

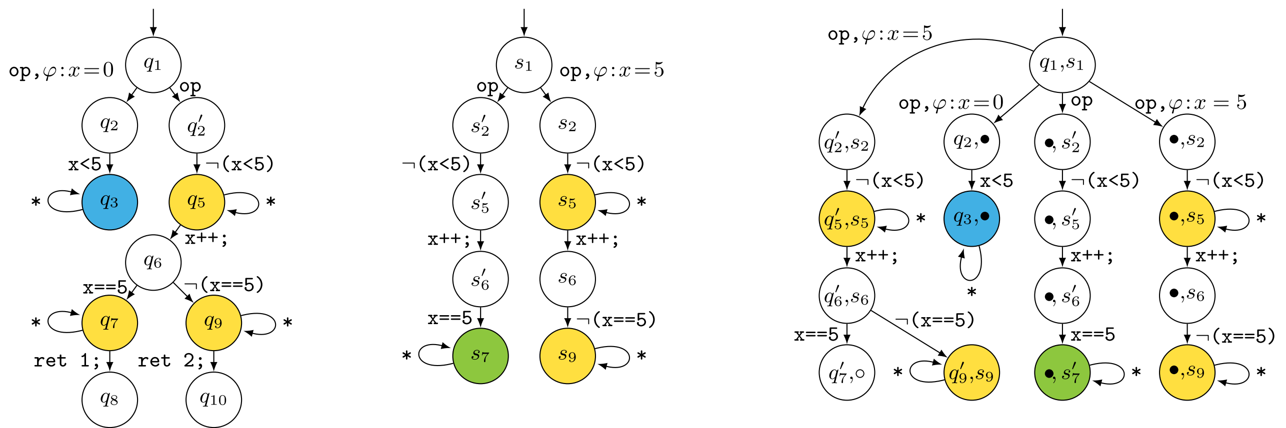
An algorithm for a combiner is given in Alg. 2, for presentation purposes assuming that each edge in δ_1, δ_2 contains only a single transition. The intuitive idea of the Combiner is to build the union of the two GIAs and consider newly computed information: For example, if there is a path π, $\pi \in \mathcal{P}_{cand}(A_1)$ and $\pi \in \mathcal{P}_{ut}(A_2)$, Combiner ensures that $\pi \in \mathcal{P}_{ut}(A_3)$ holds for the combined GIA A_3. To this end, Combiner builds the new GIA A_3 by searching for common sub-paths in the input-GIAs A_1 and A_2. A state in A_3 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

Algorithm 2 Combiner

Input: GIA $A_1 = (\mathcal{Q}_1, \Sigma, \delta_1, q_0, F^1_{ut}, F^1_{rt}, F^1_{cand})$
GIA $A_2 = (\mathcal{Q}_2, \Sigma, \delta_2, s_0, F^2_{ut}, F^2_{rt}, F^2_{cand})$
Output: GIA $A = (\mathcal{Q}, \Sigma, \delta, p_0, F_{ut}, F_{rt}, F_{cand})$
1: $\mathcal{Q} := \{((q_0, s_0), true)\}$, $p_0 := ((q_0, s_0), true)$, $\delta := \emptyset$, wl $:= \{((q_0, s_0), true)\}$
2: **while** wl $\neq \emptyset$ **do**
3: select and remove $((q_i, s_i), \psi_i)$ from wl
4: **for each** $t_1 = ((q_i, \psi_i) \xrightarrow{g_i, \varphi_i} (q_{i+1}, \psi_{i+1})) \in \delta_1$ **do**
5: **if** $\nexists ((s_i, \psi_i) \xrightarrow{g_j, \varphi_j} (s_{i+1}, \psi'_{i+1})) \in \delta_2 : g_i = g_j \vee s_i \in \{\circ, \bullet\}$ **then**
6: **if** $s_i \in \{\circ, \bullet\}$ **then** $s_{i+1} = s_i$ **else** $s_{i+1} = \circ$
7: $\mathcal{Q} := \mathcal{Q} \cup \{(q_{i+1}, s_{i+1}), \psi_{i+1}\}$,
8: $\delta := \delta \cup \{((q_i, s_i), \psi_i) \xrightarrow{g_i, \varphi_i} ((q_{i+1}, s_{i+1}), \psi_{i+1})\}$
9: **if** $q_{i+1} \notin F^1_{rt} \cup F^1_{ut}$ **then**
10: wl := wl $\cup \{((q_{i+1}, s_{i+1}), \psi_{i+1})\}$
11: **end if**
12: **else**
13: **for each** $t_2 = ((s_i, \psi_i) \xrightarrow{g_j \varphi_j} (s_{i+1}, \psi'_{i+1})) \in \delta_2 : g_i = g_j$ **do**
14: wl, $\mathcal{Q}$, δ := Merge(wl, $\mathcal{Q}$, δ, t_1, t_2)
15: **end for**
16: **end if**
17: **end for**
18: **for each** $((s_i, \psi_i) \xrightarrow{g_j, \varphi_j} (s_{i+1}, \psi_{i+1}) \in \delta_2)$ **do**
19: analogously to line 4-12
20: **end for**
21: **end while**
22: $F_{rt} = \{(q_i, s_i) \in \mathcal{Q} \mid q_i \in F^1_{rt} \vee s_i \in F^2_{rt}\}$
23: $F_{ut} = \{(q_i, s_i) \in \mathcal{Q} \mid q_i \in F^1_{ut} \vee s_i \in F^2_{ut}\}$
24: $F_{cand} = \{(q_i, s_i) \in \mathcal{Q} \mid q_i \in F^1_{cand} \cup \{\bullet\} \wedge s_i \in F^2_{cand} \cup \{\bullet\}\}$
25: **if** $F_{rt} \cap F_{ut} \neq \emptyset$ **then return** ERROR; **end if**
26: **return** $A = (\mathcal{Q}, \Sigma, \delta, p_0, F_{ut}, F_{rt}, F_{cand})$
where $\circ, \bullet$ are replacements for a state used during splitting and are not processed.

states have different state invariants. For combination, Alg. 2 applies the method `Merge`, given in Appendix B.

An application of Alg. 2 for the program from Fig. 3 is depicted in Fig. 14. We use the two GIAs A_1 (in Fig. 14a) and A_2 (in Fig. 14b) as inputs, the resulting GIA A_3 is shown in Fig. 14c, where we elided paths that are contained twice to increase readability. A_1 is generated during test case generation by an UA tool, containing the information that the target



(a) GIA A_1 after application of an UA tool

(b) GIA A_2 after application of an OA tool

(c) Combined GIA A_3 of A_1 and A_2

Fig. 14 A GIA generated during cooperative test case generation, for example, the program of Fig. 2 with states of F_{ut} marked green, of F_{rt} blue and of F_{cand} yellow. We elide state invariants (all *true*) and depict for transitions only the operation and non-true conditions. We define op:='x = random();'

node ℓ_3 is reachable when $x = 0$ holds. A_2 is produced by an OA tool, that marks the target node ℓ_7 as unreachable. As both, A_1 and A_2, contain a path to ℓ_7, but $q_7 \in F_{cand}$ in A_1 and $s_7 \in F_{ut}$ in A_2, the combiner generated a successor using '•' instead of q_7 for A_3 to maintain the information that ℓ_7 is unreachable. In contrast, the successor of the state (q_6', s_6) is $(q_7', \circ)$, as q_6 has a successor q_7 but s_6 does not. Using '○' instead of '•' ensures that $(q_7', \circ)$ is not in F_{cand} in A_3 (cf. line 24 in Alg. 2), because A_2 contains the information that this node is unreachable.

Additionally, COMBINER maintains more precise information on paths from $\mathcal{P}_{cand}$: If a path π is present in $\mathcal{P}_{cand}(A_1)$ and $\mathcal{P}_{cand}(A_2)$, once with and once without condition, the condition is also present on the path in the combined GIA. In our example, both A_1 and A_2 contain a path covering ℓ_5. A_1 has a path $(q_1, q_2', \ldots)$ with the condition true and A_2 a path $(s_1, s_2, \ldots)$ labeled with $x = 5$.

In the combined GIA A_3, the condition $x = 5$, and thus the more precise information, is maintained. The resulting GIA is not guaranteed to be minimal, meaning that it may contain some paths multiple times and contains paths that do not lead to an accepting state. For example, A_3 contains two paths both reaching ℓ_5 with the same condition.

Theorem 2 *Algorithm 2 is a combiner according to Definition 7.*

Proof Intuitively, we have to show that for the combination A of two GIAs A_1 and A_2 each path *rt*-covered by either A_1 or A_2 is also *rt*-covered by A and that the reverse holds (and analogously, that both properties hold for *ut*-covered paths). We therefore inductively construct an accepting run of A for a path π that is *rt*-covered by either A_1 or A_2 and vice versa. The full formal proof can be found in Appendix C. □

7.3 Using reducer and combiner

Finally, we can state that connecting tools via reducers and combiners do not lose any of the already computed analysis results, as depicted in Fig. 13. This property guarantees that any arbitrary combination of sound OA and UA tools achieves the same progress, as the employed tools.

Theorem 3 *Let $A \in \mathcal{A}$ 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 $A' = comb(\mathsf{tool}(red_X(A, C)), A)$ we get*

- *$\mathcal{P}_{rt}(A') = \mathcal{P}_{rt}(A) \wedge \mathcal{P}_{ut}(A') \supseteq \mathcal{P}_{ut}(A)$ if* tool *is an OA, and*
- *$\mathcal{P}_{ut}(A') = \mathcal{P}_{ut}(A) \wedge \mathcal{P}_{rt}(A') \supseteq \mathcal{P}_{rt}(A)$ 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 Theorem 2 and Definition 5. □

Let us revisit Fig. 13 used to exemplify the construction of combiner and reducer in the setting of cooperative test case generation on a more concrete level. We use an OA tool called Verifier working on GIAs as well as *ut*-REDUCER and COMBINER for an Off-the-shelf Tester, that does not understand GIAs. When started on the program from Fig. 3, *ut*-REDUCER is called with the initial, empty GIA and generates the reduced program, which is the same as the original one. Next, Off-the-shelf Tester finds a test suite covering ℓ_3 and generates the GIA A_1 depicted in Fig. 14a, that is merged with the empty GIA, not changing A_1. The original program and A_1 are given to Verifier which (1) computes that ℓ_7 is unreachable and (2) computes a path potentially leading to

ℓ_5 and ℓ_9 under the condition $x = 5$. The Verifier computes the GIA A_3, depicted in Fig. 14c, which also contains all information on A_1. As not all target nodes are covered by F_{rt} and F_{ut} in A_3, a second iteration starts: At first, the *ut*-REDUCER computes the reduced program containing only the else-branch starting in line 5. Off-the-shelf Tester confirms that ℓ_5 and ℓ_9 are reachable. This information, encoded as GIA, is finally combined with A_3. Now, all target nodes are either covered or identified as unreachable, and hence, the computation stops.

8 Implementation

To demonstrate the feasibility of GIAs as an exchange format and to show that the developed theoretical concepts work in practice, we exemplarily realized two conceptually different forms of cooperation: Cooperative test case generation as described in Sec. 6 and depicted in Fig. 13 and component-based CEGAR (C- CEGAR [24]) using only GIAs as exchange format, as explained in Sec. 6 and depicted in Fig. 10 and Fig. 13.

We implemented GIAs based on condition automata and realized our instances of cooperative test case generation and component-based CEGAR using COVERITEAM [19]. COVERITEAM is a framework that provides an easy way to build different forms of cooperative software verification. It provides a language to describe the communication between different components and their inputs and outputs. The language allows for combining different actors in sequence, in parallel, or in a cyclic manor. We integrate the GIA as an exchange format in COVERITEAM. In our setting, we use COVERITEAM for orchestration as well as for monitoring of the progress of the composition, i.e., checking whether all target nodes are already covered.

Additionally, we built modules within CPACHECKER [21] that allow processing a GIA as input as well as generating a GIA as output. Thereby, we can reuse existing analyses of CPACHECKER for the evaluation. We built *ut*-reducer and *rt*-reducer described in Alg. 3 as well as the combiner from Alg. 2 within CPACHECKER, forming a standalone-executable component, also fully integrated in COVERITEAM.

Component-based CEGAR. The original implementation of component-based CEGAR(C- CEGAR) (here called CC-WIT) contains three components, a model explorer, a feasibility checker, and a precision refiner, which are executed in a loop and exchange correctness and violation witnesses. Our re-implementation CC- GIA, also contains these three components, as depicted in Fig. 10. Note that CEGAR assumes that the feasibility checker is precise in the sense that it reports a counterexample as being spurious, only if all paths covered by the counterexample have been checked. Otherwise, the same, real counterexample may be discovered multiple times, causing an infinite refinement loop. This situation is prevented by using a feasibility checker to exhaustively check all paths in the potential counterexample.

For CC- GIA, we can use the existing realizations of model explorer, feasibility checker and precision refiner in CPACHECKER, as we updated the CPACHECKER such that it can process GIAs as input and generate them as output. As the precision refiner in C- CEGAR focuses on refining the latest infeasible counterexample generated by the model explorer, we additionally use a combiner to ensure that the precision increments computed in previous iterations are maintained. Note that exchange formats like violation and correctness witnesses can be translated into GIAs, allowing to use any off-the-shelf tool that produces these artifacts as outputs.

Cooperative test case generation. To realize cooperative test case generation using GIA as an exchange format, we follow the tooling used by Daca et al. [42]: We employ the concolic tester CREST [31] as UA tool and CPACHECKER's predicate analysis using CEGAR as OA analysis. CREST is a concolic tester, meaning that inputs are not only generated randomly or using a heuristic, but paths are encoded as formulae using symbolic inputs and solved by an SMT-solver, in this case YICES [45]. 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 analysis from CPACHECKER. The predicate analysis 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- GIA. We employed parts of TBF [22] to let CREST generate test inputs in the TESTCOMP test case format[5] 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 analysis 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 `random`. 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

[5] https://gitlab.com/sosy-lab/test-comp/test-format/blob/testcomp22/doc/Format.md

internal format. Afterward, 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.

9 Evaluation

The goal of the evaluation is twofold: First, we exemplarily show that GIAs are feasible as an exchange format and can be used in two different usage contexts, one for verification and one for test case generation. Second, we demonstrate the advantages of the clearly defined semantics of GIAs, allowing to precisely encode information for the exchange between analyses. As the goal for verification is to show the (un)-reachability of a certain error location, each program usually has a single target node. Hence, the verification task is completed if the target node is either shown to be unreachable or a concrete path leading to the target node is found. In contrast, for test case generation, most tasks contain multiple target nodes, some of them reachable and others not. Therefore, we evaluate whether the conceptual advantages of encoding information on reachable and unreachable target nodes within a single artifact can be witnessed in cooperate test case generation and lead to the computation of more precise results or a faster computation. The implementation and evaluation of this use case is an extension compared to the conference paper [56]. We therefore study the following two research questions:

RQ 1. Are GIAs feasible as exchange formats for component-based CEGAR?

RQ 2. Can (cooperative) test case generation also benefit from using GIAs for information exchange?

RQ 1 focuses on the usage of GIAs on a fine-grained scale, as CEGAR is usually employed within a single tool. In contrast, in RQ 2 we build a cooperation between two standalone, off-the-shelf tools. Note that we could also employ the component-based CEGAR using GIA in RQ 2, but as the performance of the tightly coupled version of C- CEGAR is currently better (cf. [24]), we decided to use the tightly coupled one.

9.1 RQ 1: Component-based CEGAR

The goal of verification in this setting is to either find a concrete path leading to the target node (an alarm) or to compute a proof that the target node is not reachable. To evaluate the feasibility of GIAs as an exchange format, we compare the existing implementation of C-CEGAR (CC- WIT), using violation and correctness witnesses as exchange formats between the three components, with our re-implementation (CC- GIA) which only makes use of GIAs for the information exchange. For the comparison, we are interested in the question whether the same tasks that are solved by CC- WIT can also be solved by CC- GIA and want to know if there are tasks that can only be solved using GIA as exchange format. Thus, we compare the effectiveness (number of solved tasks) of CC- WIT and CC- GIA. In addition, we want to study if there is an effect on the execution time when encoding the same information in a different format, i.e., now using GIA instead of correctness and violation witnesses. Therefore, we compare the efficiency (consumed CPU time to compute the solution) of CC- GIA and CC- WIT.

Table 1 Comparison of the existing CC- WIT with the cooperation using only GIA for information exchange (CC- GIA)

Result	CC- GIA	CC- WIT
Correct overall	2641	2819
Correct proof	2068	2100
Correct alarm	573	719
Add. solved	114	–
Incorrect	5	7

Evaluation Setup. 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. Each tool is limited to use 15 GB of memory, 4 CPU cores, and 15 min of CPU time per verification run. All experiments were executed using BENCHEXEC [25], ensuring the resource limitations. We evaluated both approaches on the SV- BENCHMARKS, the largest publicly available benchmark for C-programs, in the version used for the SV- COMP'22[6] containing in total 8 347 tasks. We used CPACHECKER in version 2.1.2, COVERITEAM in version 0.9, and BENCHEXEC in version 3.11.

Evaluation Results (Effectiveness). Table 1 contains the experimental results of CC- GIA and CC- WIT. It contains the number of overall correct answers, the correct proofs (where an approach correctly detects that no target node is reachable) and correct alarms (where a feasible path to a target node is computed). In addition, the incorrect answers are reported, as well as the number of tasks where CC- GIA computes the correct result, but CC- WIT does not (row add. solved).

For the total number of correctly solved tasks, we observe that CC- GIA can solve 94% of all tasks solved by CC- WIT. Within the 94%, the number of iterations and the computed refinements are almost always equal. The decrease originates mostly in the fact that CC- GIA is not able to compute a

[6] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/svcomp22

```
 1 int main(void) {
 2   unsigned int x = 1;
 3   unsigned int y = 0;
 4
 5   while (y < 1024) {
 6     x = 0;
 7     y++;
 8   }
 9   if (x == 0) {}
10   else
11     error();
12 }
```

Fig. 15 Program from SV- COMP, where $x = 0$ is not a valid invariant at the loop head

solution in the given time limit for 259 tasks, for which CC- WIT computes a solution within 900 s.

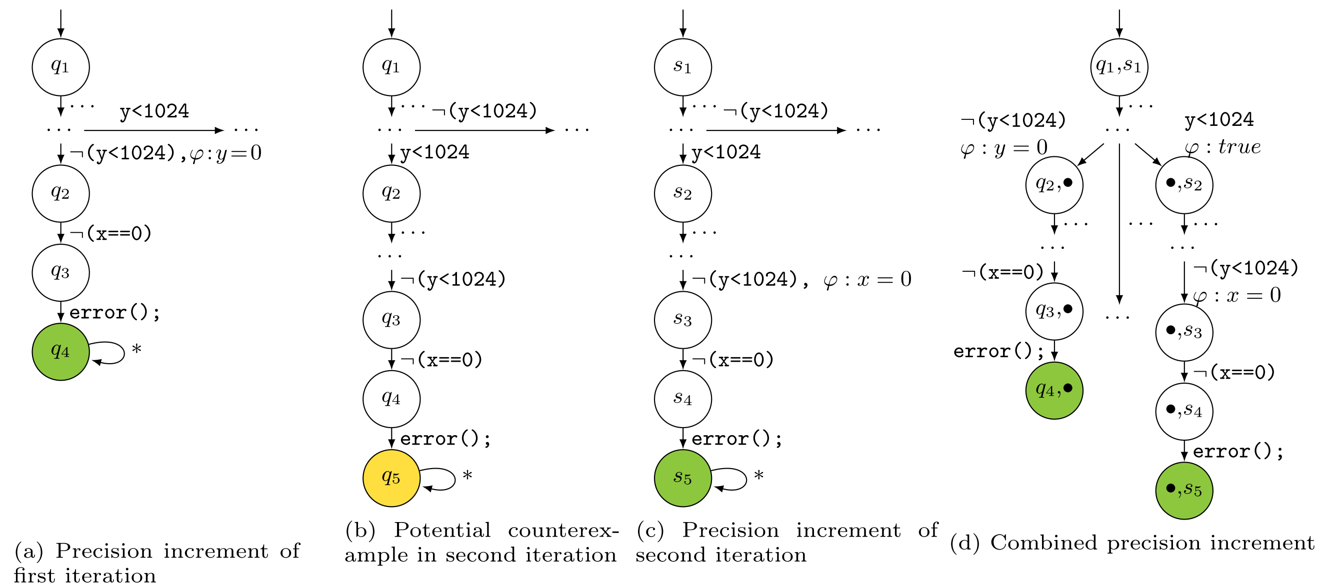
When looking at the additionally solved tasks, we can see the advantages of using GIAs: In 114 cases, CC- GIA computes the correct result, whereas CC- WIT either runs in a timeout or aborts the computation as it eventually makes no progress and gets stuck. Both situations are caused by the fact that not all information computed by the precision refiner is added in the correctness witness, a situation not happening when using GIAs. In [24], the authors argue that this situation is caused by the fact that correctness witnesses are not primarily designed for the exchange of a precision increment. The semantics of the GIA allows the precision refiner to encode the information, i.e., encode that a newly discovered predicate holds at a certain point of the infeasible counterexample path. Therefore, the refiner builds a GIA that only contains the infeasible counterexample, of which the last state is in F_{ut}, whereas the precision increment is encoded as an assumption on that path.

To exemplify the advantages, Fig. 15 presents a program taken from the SV- BENCHMARKS collection. The target node is the call to `error` in line 11. Within the first iteration of C- CEGAR, the model explorer computes a potential counterexample that does not enter the loop starting in line 5. The GIA containing the potential counterexample is given to the feasibility checker, identifying it as infeasible. Next, the precision refiner computes the interpolant $y = 0$. This formula is encoded within the GIA generated by the precision refiner (depicted in Fig. 16a) and is given to the model explorer. Now, the second iteration starts and the model explorer computes a counterexample that traverses the loop once, depicted in simplified form in Fig. 16b. Again, the feasibility checker rejects the counterexample as invalid and it is given to the precision refiner. The precision refiner now computes a new interpolant, namely $x = 0$, that is valid after the first loop iteration, but invalid before the first loop iteration. As stated in [24], the precision refiner in CC- WIT fails to encode this new predicate within the correctness witness. In contrast, using GIAd as an exchange format allows the precision refiner to build the GIA depicted in Fig. 16c, 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 the information on the interpolant computed in the first iteration, the two GIAs from Fig. 16a and from Fig. 16c are combined into the GIA depicted in Fig. 16d. 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.

Evaluation Results (Efficiency). Figure 17 compares the efficiency of CC- GIA and CC- WIT per task in a logarithmic scale. A point (x, y) contains the CPU time taken by CC- GIA (as x) and by CC- WIT (as y) for all tasks where both compute the correct solution or one run into a timeout (TO). We observe that CC- GIA needs in general more time to find a solution, as most points are below the diagonal. The increase is in the vast majority of all cases smaller than factor two (lower dashed line). The CPU time increases on average by 1.4 (standard deviation is 0.4), and the median increase is 1.3. In CC- 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- 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 increasing runtime and the number of timeouts.
The evaluation shows that GIAs are a flexible, precise, and practically suitable exchange format, applicable for C- CEGAR. In particular, we see that the drawbacks of CC- WIT, namely losing information on computed precision increments, can be overcome. As a downside, the overall efficiency slightly decreases when using GIAs, due to their size and the fact that they are non-optimized for specific applications.

9.2 RQ 2: Cooperative test case generation

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 tech-



(a) Precision increment of first iteration (b) Potential counterexample in second iteration (c) Precision increment of second iteration (d) Combined precision increment

Fig. 16 Example showing the advantages of using GIA compared to correctness witness as exchange formats

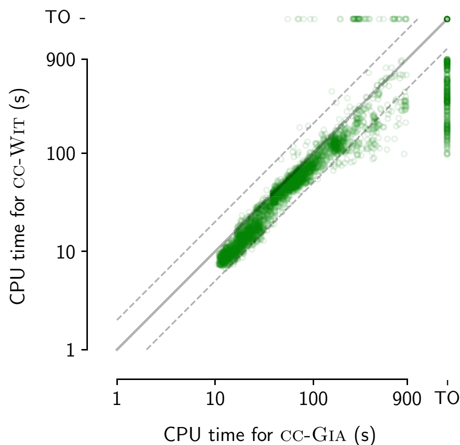


Fig. 17 Comparison of CPU time for CC- WIT and CC- GIA

nique, 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 TESTCOMP'22[7] 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 TESTCOMP (tasks `15-115.2`) and used the running example of Fig. 2 (task `a1`) and an extended version given in Appendix D in Fig. 18 (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 [23], the tool also used in the TESTCOMP. We use TESTCOV with a 30 minutes timeout, in contrast to TESTCOMP, where only a five-minute timeout is used. We used CPACHECKER in version 2.1.2, COVERITEAM in version 0.9, TESTCOV in version 3.6, and BENCHEXEC in version 3.11.

Evaluation Results. Table 2 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

[7] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/testcomp22.

Table 2 Results of test case generation for CoTest and Crest

Task	CoTest			Crest		
	#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

Note that TestCov was not able to analyze the full test suite generated by Crest in the given time limit

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 analysis 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 s.

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.

9.3 Threads to validity

There exist multiple concepts for cooperative software validation. We have implemented two of them and have experimentally shown that GIAs are suited as an exchange format that guarantees that no information is lost. For the other concepts of cooperative combinations using standardized exchange formats, namely CMC and CoVEGI, we explained how GIAs could replace the used artifacts. 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. In such cases, one can apply the concept of mapper and adapter proposed in [55] 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 verification of the correctness of a system[8] 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 as depicted in Fig. 7, 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, 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.

10 Conclusion

In this article, we have proposed general information exchange automata as an exchange format for the cooperation of over- and under-approximative analyses. It has a fixed well-defined semantics allowing its application in different scenarios. We have furthermore defined and implemented two operations on GIAs, reducing a program to the (remaining) task and combining results with previously computed information. These operations allow a re-use of off-the-shelf tools. We have formally shown that applying reducer and combiner maintains all relevant computed information. The feasibility of GIAs as exchange format has been demonstrated by applying it in an existing cooperative verification setting (C- CEGAR) and in a test case generation setting.

For future work, we plan to implement other existing forms of combinations of OA and UA off-the-shelf tools in a cooperative setting using GIAs for the information exchange, such as for conditional model checking or k-induction. GIAs are also well suited for being applied in a parallelized cooperative setting, where multiple tools work side-by-side on the same task to increase the overall performance, as the combiner of arbitrary GIAs guarantees that no information is lost.

Funding Open Access funding enabled and organized by Projekt DEAL.

Data Availability Our implementation is open-source and available as part of CPACHECKER and COVERITEAM. We archived the implementation and all experimental data for reproduction at Zenodo [57].

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

Appendix

A Full algorithm for the *X*-reducer

The full construction of X-Reducer is given in Alg. 3, extending the construction given in [18]: First, X-Reducer stops the exploration for all paths leading to a state in F_{rt}. Second, as states of a GIA consists of pairs of state name and state invariant, we need to ensure that a path is only removed, in case that its state invariant is fulfilled. This is achieved by lines 8-13, where we split the path into two sub-paths for non-trivial state invariants, one that can be taken if the state invariant is met 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. Taking this line of argument into account, we can conclude that X-Reducer is in fact a reducer, following the proof structure from [18]. Again, the resulting CFA is deterministic.

[8] according to the handbook of model checking [36]

Algorithm 3 X-Reducer (extended)

Input: CFA $C = (Loc, \ell_0, G)$ ▷ *original program*
GIA $A = (\mathcal{Q}, \Sigma, \delta, (q_0, \psi_0), F_{ut}, F_{rt}, F_{cand})$ ▷ *GIA*
$q_t \notin \mathcal{Q}$ ▷ *additional state*
Output: CFA $C_r = (Loc_r, \ell_0^r, G_r)$ ▷ *reduced program*
1: $Loc_r := \{(\ell_0, (q_0, \psi_0))\}$, $\ell_0^r := (\ell_0, (q_0, \psi_0))$, $G_r := \emptyset$
2: waitlist := Loc_r
3: **if** $(q_0, \psi_0) \in F_{cand}$ **then** US := US $\cup (q_0, \psi_0)$
4: **end if**
5: **while** waitlist $\neq \emptyset$ **do**
6: choose and remove $(\ell_i, (q_i, \psi_i))$ from waitlist
7: **for each** $g = (l_i, op, l_{i+1}) \in G$ **do**
8: **if** $(q_i, \psi_i) \in \mathcal{Q} \wedge \exists ((q_i, \psi_i), (G_i, true), (q_{i+1}, \psi_{i+1})) \in \delta$ $s.t.$ $g \in G_i$ **then**
9: **for each** $((q_i, \psi_i), (G_i, true), (q_{i+1}, \psi_{i+1})) \in \delta$ $s.t.$ $g \in G_i$ **do**
10: **if** $\psi_{i+1} \neq true$ **then**
11: **if** $(q_{i+1}, \psi_{i+1}) \notin F_X \wedge (l_{i+1}, (q_i', \psi_i)) \notin Loc$ **then**
12: waitlist := waitlist $\cup (l_{i+1}, (q_i', \psi_i))$
13: **end if**
14: **if** $(l_{i+1}, (q_t, true)) \notin Loc$ **then** waitlist := waitlist $\cup (l_{i+1}, (q_t, true))$ **end if**
15: $Loc_r := Loc_r \cup \{(l_{i+1}, (q_i', \psi_i)), (l_{i+1}, (q_{i+1}, \psi_{i+1}), (l_{i+1}, (q_t, true))\}$
16: $G_r := G_r \cup \{\big((l_i, (q_i, \psi_i)),\ op\ , (l_{i+1}, (q_i', \psi_i))\big),$ $\big(((l_{i+1}, (q_i', \psi_i)),\ \psi_{i+1}\ , (l_{i+1}, (q_{i+1}, \psi_{i+1}))\big),$ $\big(((l_{i+1}, (q_i', \psi_i)),\ \neg\psi_{i+1}\ , (l_{i+1}, (q_t, true))\big)\}$
17: **else**
18: **if** $(q_{i+1}, \psi_{i+1}) \notin F_X) \wedge (l_{i+1}, (q_{i+1}, \psi_{i+1})) \notin Loc$ **then**
19: waitlist := waitlist $\cup\{(l_{i+1}, (q_{i+1}, \psi_{i+1}))\}$
20: **end if**
21: $Loc_r := Loc_r \cup \{(l_{i+1}, (q_{i+1}, \psi_{i+1}))\}$
22: $G_r := G_r \cup \{\big(((l_i, (q_i, \psi_i)),\ op\ , ((l_{i+1}, (q_{i+1}, \psi_{i+1}))\big)\}$
23: **end if**
24: **end for**
25: **else**
26: **if** $(l_{i+1}, (q_t, true)) \notin Loc$ **then** waitlist := waitlist $\cup\{(l_{i+1}, (q_t, true))\}$ **end if**
27: $Loc_r := Loc_r \cup \{(l_{i+1}, (q_t, true))\}$
28: $G_r := G_r \cup \{\big(((l_i, (q_i, \psi_i)), op, (l_{i+1}, (q_t, true))\big)\}$
29: **end if**
30: **end for**
31: **end while**
32: **if** $F_{cand} \neq \emptyset$ **then**:
33: $toKeep := \emptyset$ ▷ *Locations on a path containing a node in F_{cand}*
34: **for each** $\ell = (l_i, (q_i, \psi_i)) \in Loc_r$ s.t. $(q_i, \psi_i) \in F_{cand}$ **do**
35: add all predecessors and successors of ℓ in Loc_r to $toKeep$
36: **end for**
37: **for each** $\ell \in Loc_r$ **do**
38: **if** $\ell \notin toKeep$ **then** Remove ℓ from Loc_r; Remove all $(\ell, \cdot, \cdot), (\cdot, \cdot, \ell)$ from G_r **end if**
39: **end for**
40: **end if**
41: **return** (Loc_r, ℓ_0^r, G_r)

B Algorithm for merge

Algorithm 4 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. We briefly summarize the five cases:

- Case line 1: State invariants and conditions of t_1 and t_2 are equal $\Rightarrow$ Path is not split.
- Case line 5: Both paths lead only to states from F_{cand}, one contains conditions, the other one does not $\Rightarrow$ Keep the non-true assumption
- Case line 9: At least one path starting in p_{i+1} eventually leads to $F_{ut}^1 \cup F_{rt}^1$ and none starting in s_{j+1} reach $F_{ut}^2 \cup F_{rt}^2$ $\Rightarrow$ Ignore s_{j+1}.
- Case line 13: At least one path starting in s_{j+1} eventually leads to $F_{ut}^2 \cup F_{rt}^2$ and none starting in p_{i+1} reach $F_{ut}^1 \cup F_{rt}^1$ $\Rightarrow$ Ignore p_{i+1}.
- Case line 17: Otherwise split the path into two paths.

C Proof of Theorem 2

Recall Theorem 2: Algorithm 2 is a combiner according to Definition 7.

Proof We assume wlog. that $\mathcal{P}_{ut}(A_1) \cup \mathcal{P}_{rt}(A_2) = \emptyset = \mathcal{P}_{rt}(A_1) \cup \mathcal{P}_{ut}(A_2)$. As Alg. 2 and Alg. 4 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 A_1, A_2 and $A = comb(A_1, A_2)$ it holds that:

$$\mathcal{P}_{ut}(A_1) \cup \mathcal{P}_{ut}(A_2) \subseteq \mathcal{P}_{ut}(A) \tag{1}$$

$$\text{and } \mathcal{P}_{ut}(A_1) \cup \mathcal{P}_{ut}(A_2) \supseteq \mathcal{P}_{ut}(A) \tag{2}$$

Let π^i denote the prefix of length i of π for a path or run. We say that a run $\rho = (q_0, \psi_0) \xrightarrow{(G_1, \varphi_1)} \ldots \xrightarrow{(G_k, \varphi_k)} (q_k, \psi_k)$, of a GIA $A = (\mathcal{Q}, \Sigma, \delta, q_0, F_{ut}, F_{rt}, F_{cand})$ *follows a path* $\pi = \langle c_0, \ell_0 \rangle \xrightarrow{g_1} \ldots \xrightarrow{g_n} \langle c_n, \ell_n \rangle$ if

1. $\forall i, 1 \leq i \leq k : g_i \in G_i$,
2. $\forall i, 1 \leq i \leq k : c_i \models \psi_i$,
3. $\forall i, 0 \leq i \leq k : c_i \models \varphi_i$.

For (1), given a path $\pi = \langle c_0, \ell_0 \rangle \xrightarrow{g_1} \ldots \xrightarrow{g_n} \langle c_n, \ell_n \rangle$ assume wlog. $\pi \in \mathcal{P}_{ut}(A_1)$. Hence, there is a run $\rho = (q_0, \psi_0) \xrightarrow{g_1, \varphi_1} \ldots \xrightarrow{g_k, \varphi_k} (q_k, \psi_k)$ of A_1. 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 $\tau = (p_0, \psi_0) \xrightarrow{g_1, \varphi_1} \ldots \xrightarrow{g_k, \varphi_k} (p_k, \psi_k)$ of A 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 $\xrightarrow{g_{i+1}, \varphi_{i+1}}$

Algorithm 4 Merge

Input: waitlist, $\mathcal{Q}$, δ
$t_1 = ((q_i, \psi_i) \xrightarrow{g_i,\varphi_i} (q_{i+1}, \psi_{i+1}))$
$t_2 = ((s_j, \psi_j) \xrightarrow{g_j,\varphi_j} (s_{j+1}, \psi_{j+1})$ ▷ *requirement:* $g_i = g_j$
Output: waitlist, $\mathcal{Q}$, δ
1: **if** $\varphi_i = \varphi_j \wedge \psi_i = \psi_j \wedge ((q_{i+1} \in F^1_{ut} \cup F^1_{rt} \cup F^1_{cand}) \Leftrightarrow (s_{i+1} \in F^2_{ut} \cup F^2_{rt} \cup F^2_{cand}))$ **then**
2: $\mathcal{Q} := \mathcal{Q} \cup \{((q_{i+1}, s_{j+1}), \psi_{i+1})\}$
3: $\delta := \delta \cup \{((q_i, s_j), \psi_i) \xrightarrow{g_i,\varphi_i} ((q_{i+1}, s_{j+1}), \psi_{i+1})\}$
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** waitlist := wailist $\cup\{((q_{i+1}, s_{j+1}), \psi_{i+1})\}$ **end if**
5: **else if** $(\psi_i = \psi_j) \wedge (reach_{cand}(q_{i+1}) \wedge reach_{cand}(s_{j+1})) \wedge (trueCond(q_i) \vee trueCond(s_j))$ **then**
6: $\mathcal{Q} := \mathcal{Q} \cup \{((q_{i+1}, s_{j+1}), \psi_{i+1})\}$
7: $\delta := \delta \cup \{((q_i, s_j), \psi_i) \xrightarrow{g_i,\varphi} ((q_{i+1}, s_{j+1}), \psi_{i+1})\}$ ▷ $\varphi \in \{\varphi_i, \varphi_j\}, \varphi \neq true$
8: **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** waitlist := wailist $\cup\{((q_{i+1}, s_{j+1}), \psi_{i+1})\}$ **end if**
9: **else if** $(\psi_i = \psi_j) \wedge (reach_{ut,rt}(q_{i+1}) \wedge reach_{cand}(s_{j+1})$ **then**
10: $\mathcal{Q} := \mathcal{Q} \cup \{((q_{i+1}, \circ), \psi_{i+1})\}$
11: $\delta := \delta \cup \{((q_i, s_j), \psi_i) \xrightarrow{g_i,\varphi_i} (q_{i+1}, \circ), \psi_{i+1})\}$ ▷ *Using* φ_i
12: **if** $q_{i+1} \notin F^1_{rt} \cup F^1_{ut}$ **then** waitlist := wailist $\cup\{((q_{i+1}, \circ), \psi_{i+1})\}$ **end if**
13: **else if** $(\psi_i = \psi_j) \wedge (reach_{cand}(q_{i+1}) \wedge reach_{ut,rt}(s_{j+1})$ **then**
14: $\mathcal{Q} := \mathcal{Q} \cup \{((\circ, s_{j+1}), \psi_{i+1})\}$
15: $\delta := \delta \cup \{((q_i, s_j), \psi_i) \xrightarrow{g_i,\varphi_j} (\circ, s_{j+1}), \psi_{i+1})\}$ ▷ *Using* φ_j
16: **if** $s_{j+1} \notin F^2_{rt} \cup F^2_{ut}$ **then** waitlist := wailist$\cup\{((\circ, s_{j+1}), \psi_{i+1})\}$ **end if**
17: **else**
18: $\mathcal{Q} := \mathcal{Q} \cup \{((q_{i+1}, \bullet), \psi_{i+1}), ((\bullet, s_{j+1}), \psi_{i+1})\}$
19: newS := newS $\cup\{((q_{i+1}, \bullet), \psi_{i+1}), ((\bullet, s_{j+1}), \psi_{i+1})\}$
20: $\delta := \delta \cup \{((q_i, s_j), \psi_i) \xrightarrow{g_i,\varphi_i} ((q_{i+1}, \bullet), \psi_{i+1}), ((q_i, s_j), \psi_j) \xrightarrow{g_j,\varphi_j} ((\bullet, s_{j+1}), \psi_{i+1})\}$
21: **end if**
22: **for each** $((q_k, s_l), \psi_m) \in$ newS **do**
23: **if** $q_k \notin F^2_{rt} \cup F^2_{ut} \wedge s_l \notin F^2_{rt} \cup F^2_{ut}$ **then** waitlist := wailist $\cup\{(q_k, s_l)\}$ **end if**
24: **end for**
25: **return** waitlist, $\mathcal{Q}$, δ

where $\circ$, $\bullet$ are placeholder that are not processed,
$trueAsmp(q_i) = true$ if all paths 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 $F_{ut} \cup F_{rt}$

(q_{i+1}, ψ_{i+1}). We distinguish, if δ_2 also contains a transition $((s_i, \psi_i) \xrightarrow{g_{i+1},\varphi'_{i+1}} (s_{i+1}, \psi'_{i+1}))$:

If there is no $((s_i, \psi_i) \xrightarrow{g_{i+1},\varphi'_{i+1}} (s_{i+1}, \psi'_{i+1})) \in \delta_2$ or $S_i \in \{\circ, \bullet\}$ then τ^i is extended by $\xrightarrow{g_{i+1}\varphi_{i+1}} ((q_{i+1}, \circ), \psi_{i+1})$ (by line 5-9 of Alg. 2) and accepts π^{i+1}.

Otherwise, if there is $((s_i, \psi_i) \xrightarrow{g_{i+1},\varphi'_{i+1}} (s_{i+1}, \psi'_{i+1})) \in \delta_2$, then τ is extended either

by $\xrightarrow{g_{i+1},\varphi_{i+1}} ((q_{i+1}, s_{i+1}), \psi_{i+1})$ in line 1-4 of Alg. 4,
by $\xrightarrow{g_{i+1},\varphi_{i+1}} ((q_{i+1}, \circ), \psi_{i+1})$ in line 9-12 of Alg. 4
or
by $\xrightarrow{g_{i+1},\varphi_{i+1}} ((q_{i+1}, \bullet), \psi_{i+1})$ and by $\xrightarrow{g_{i+1},\varphi'_{i+1}} ((\bullet, s_{i+1}), \psi'_{i+1})$ in line 18-20 of Alg. 4.

In all cases, τ^{i+1} covers π^{i+1}.

As $(q_k, \psi_k) \in F^1_{ut}$ and $((q_k, s_k), \psi_k) \in F_{ut}$, we know that $\tau \in A$ and A covers π. Thus, we can conclude that $\mathcal{P}_{ut}(A_1) \cup \mathcal{P}_{ut}(A_2) \subseteq \mathcal{P}_{ut}(A)$ holds.

For (2), given a path $\pi = \langle c_0, \ell_0 \rangle \xrightarrow{g_1} \ldots \xrightarrow{g_n} \langle c_n, \ell_n \rangle$ from $\mathcal{P}_{ut}(A)$, accepted by a run $\tau = ((q_0, s_0), \psi_0) \xrightarrow{g_1,\varphi_1} \ldots \xrightarrow{g_k,\varphi_k} ((q_k, s_k), \psi_k)$. We inductively construct a run $\rho = (p_0, \psi_0) \xrightarrow{g_1,\varphi_1} \ldots \xrightarrow{g_k,\varphi_k} (p_k, \psi_k) \in A_1$ or A_2 accepting π.

As both A_1 and A_2 may contain such a run, we start with constructing two runs $\rho_1 \in A_1$ and $\rho_2 \in A_2$ and show that at least one of them accepts π.

Induction start:
$((q_0, s_0), true) \in \tau^0$, $(q_0, true) \in \rho_1^0$ and $(s_0, true) \in \rho_2^0$. Hence, both ρ_1^0 and ρ_2^0 follow π^0.

Induction step:
Let $0 \leq i \leq k$ be arbitrary but fixed. τ has the transition $((q_i, s_i), \psi_i) \xrightarrow{g_{i+1},\varphi_{i+1}} ((q_{i+1}, s_{i+1}), \psi_{i+1})$. We now distinguish, if both runs are still under construction (*2.1*) or not (*2.2*).

Case 2.1: By induction hypothesis, we know that there are two runs $\rho_1^i \in A_1$ and $\rho_2^i \in A_2$ following π^i. The next transition of τ, $t = \xrightarrow{g_{i+1},\varphi_{i+1}} ((q_{i+1}, s_{i+1}), \psi_{i+1})$ is either added by line 5 to 8 (2.1.1) of Alg. 2, by line 14 (2.1.2) of Alg. 2 or by Alg. 4 (2.1.3).

Case 2.1.1: There is a transition $((q_i, \psi_i) \xrightarrow{g_{i+1},\varphi_{i+1}} (q_{i+1}, \psi_{i+1})) \in \delta_1$, but no transition $((s_i, \psi_i) \xrightarrow{g_{i+1},\varphi'_{i+1}} (s_{i+1}, \psi'_{i+1})) \in \delta_2$ (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 $\xrightarrow{g_i,\varphi_i} (q_{i+1}, \psi_{i+1}))$, 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 $t_1 = ((q_i, \psi_i) \xrightarrow{g_{i+1},\varphi'_{i+1}} (q_{i+1}, \psi'_{i+1})) \in \delta_1$ and $t_2 = ((s_i, \psi_i) \xrightarrow{g_{i+1},\varphi''_{i+1}} (s_{i+1}, \psi''_{i+1})) \in \delta_2$. As τ is constructed using t_1 or t_2, ρ_1^i or ρ_2^i can be extended, depending whether $\psi_{i+1} = \psi'_{i+1} \wedge \varphi_{i+1} = \varphi'_{i+1}$ (τ is in A_1) or $\psi_{i+1} = \psi''_{i+1} \wedge \varphi_{i+1} = \varphi''_{i+1}$ (τ is in A_2). Note that paths accepted by a run with true condition are also accepted with condition Thus, 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 $\xrightarrow{g_{i+1},\varphi_{i+1}}$

$((q_{i+1}, s_i), \psi_{i+1}) \in \tau$ must be added by line 5 to 9 of Alg. 2. This works analogously to Case 2.1.1 ✓

As $((q_k, s_k), \psi_k) \in F_{ut}$, $(q_k, \psi_k) \in F^1_{ut}$ for ρ_1 or $(s_k, \psi_k) \in F^2_{ut}$ for ρ_2, at least one of them accepts π, concluding the proof. □

D Additional example

```
1  int main(){
2    int y = 0;
3    int x = random();
4    if (x < 5){
5      x--;
6      if (x == 0){return 1;}
7      return 0;
8    }
9    else{
10     x++;
11     if (x == 5){return 0;}
12     else{
13       x++;
14       if (x == 9){return 3;}
15       else{return 5;}
16       return 0;
17     }
18   }
19 }
```

Fig. 18 An extended version of the example program from Fig. 2

References

1. Ádám, Z., Sallai, G., Hajdu, Á.: Gazer-theta: Llvm-based verifier portfolio with BMC/CEGAR (competition contribution). In: Groote, J.F., Larsen, K.G. (eds.) Proceedings TACAS. LNCS, vol. 12652, pp. 433–437. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_27
2. Albarghouthi, A., Gurfinkel, A., Chechik, M.: From under-approximations to over-approximations and back. In: Flanagan, C., König, B. (eds.) Proceedings of the TACAS. LNCS, vol. 7214, pp. 157–172. Springer (2012). https://doi.org/10.1007/978-3-642-28756-5_12
3. Alshmrany, K.M., Aldughaim, M., Bhayat, A., Cordeiro, L.C.: Fusebmc: an energy-efficient test generator for finding security vulnerabilities in C programs. In: Proceedings of the TAP. LNCS, vol. 12740, pp. 85–105. Springer (2021). https://doi.org/10.1007/978-3-030-79379-1_6
4. Avgerinos, T., Rebert, A., Cha, S.K., Brumley, D.: Enhancing symbolic execution with veritesting. In: Proceedings of the ICSE, pp. 1083–1094. ACM (2014). https://doi.org/10.1145/2568225.2568293
5. Beckman, N.E., Nori, A.V., Rajamani, S.K., Simmons, R.J.: Proofs from tests. In: Ryder, B.G., Zeller, A. (eds.) Proceedings of the ISSTA, pp. 3–14. ACM (2008). https://doi.org/10.1145/1390630.1390634
6. Beyer, D., Dangl, M.: Strategy selection for software verification based on Boolean features: a simple but effective approach. In: Proceedings of the ISoLA. LNCS, vol. 11245, pp. 144–159. Springer (2018). https://doi.org/10.1007/978-3-030-03421-4_11
7. Beyer, D., Lemberger, T.: Conditional testing: off-the-shelf combination of test-case generators. In: Proceedings of the ATVA. LNCS, vol. 11781, pp. 189–208. Springer (2019). https://doi.org/10.1007/978-3-030-31784-3_11
8. Beyer, D.: Advances in automatic software testing: test-comp 2022. In: Johnsen, E.B., Wimmer, M. (eds.) Proceedings of the FASE. LNCS, vol. 13241, pp. 321–335. Springer (2022). https://doi.org/10.1007/978-3-030-99429-7_18
9. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Fisman, D., Rosu, G. (eds.) Proceedings of the TACAS. LNCS, vol. 13244, pp. 375–402. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_20
10. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: exchanging verification results between verifiers. In: Zimmermann, T., Cleland-Huang, J., Su, Z. (eds.) Proceedings of the FSE, pp. 326–337. ACM (2016). https://doi.org/10.1145/2950290.2950351
11. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. **31**(4), 57:1-57:69 (2022). https://doi.org/10.1145/3477579
12. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Nitto, E.D., Harman, M., Heymans, P. (eds.) Proceedings of the ESEC/FSE, pp. 721–733. ACM (2015). https://doi.org/10.1145/2786805.2786867
13. Beyer, D., Gulwani, S., Schmidt, D.A.: Combining model checking and data-flow analysis. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 493–540. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_16
14. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. Int. J. Softw. Tools Technol. Transf. **9**(5–6), 505–525 (2007). https://doi.org/10.1007/s10009-007-0044-z
15. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: a technique to pass information between verifiers. In: Tracz, W., Robillard, M.P., Bultan, T. (eds.) Proceedings of the FSE, p. 57. ACM (2012). https://doi.org/10.1145/2393596.2393664
16. Beyer, D., Jakobs, M.: CoVeriTest: cooperative verifier-based testing. In: Hähnle, R., van der Aalst, W.M.P. (eds.) Proceedings of the FASE. LNCS, vol. 11424, pp. 389–408. Springer (2019). https://doi.org/10.1007/978-3-030-16722-6_23
17. Beyer, D., Jakobs, M.: Fred: Conditional model checking via reducers and folders. In: de Boer, F.S., Cerone, A. (eds.) Proceedings of the SEFM. LNCS, vol. 12310, pp. 113–132. Springer (2020). https://doi.org/10.1007/978-3-030-58768-0_7
18. Beyer, D., Jakobs, M., Lemberger, T., Wehrheim, H.: Reducer-based construction of conditional verifiers. In: Chaudron, M., Crnkovic, I., Chechik, M., Harman, M. (eds.) Proceedings of the ICSE, pp. 1182–1193. ACM (2018). https://doi.org/10.1145/3180155.3180259
19. Beyer, D., Kanav, S.: CoVeriTeam: on-demand composition of cooperative verification systems. In: Fisman, D., Rosu, G. (eds.)

Proceedings of the TACAS. LNCS, vol. 13243, pp. 561–579. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_31
20. Beyer, D., Kanav, S., Richter, C.: Construction of verifier combinations based on off-the-shelf verifiers. In: Johnsen, E.B., Wimmer, M. (eds.) Proceedings of the FASE. LNCS, vol. 13241, pp. 49–70. Springer (2022). https://doi.org/10.1007/978-3-030-99429-7_3
21. Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) Proceedings of the CAV. LNCS, vol. 6806, pp. 184–190. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
22. Beyer, D., Lemberger, T.: Software verification: testing vs. model checking—a comparative evaluation of the state of the art. In: Strichman, O., Tzoref-Brill, R. (eds.) Proceedings of the HVC. LNCS, vol. 10629, pp. 99–114. Springer (2017). https://doi.org/10.1007/978-3-319-70389-3_7
23. Beyer, D., Lemberger, T.: Testcov: robust test-suite execution and coverage measurement. In: Proceedings of the ASE, pp. 1074–1077. IEEE (2019). https://doi.org/10.1109/ASE.2019.00105
24. Beyer, D., Lemberger, T., Haltermann, J., Wehrheim, H.: Decomposing software verification into off-the-shelf components: an application to CEGAR. In: Proceedings of the ICSE, pp. 536–548. ACM (2022). https://doi.org/10.1145/3510003.3510064
25. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. Int. J. Softw. Tools Technol. Transf. **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
26. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: survey and unifying component framework. In: Margaria, T., Steffen, B. (eds.) Proceedings of the ISoLA. LNCS, vol. 12476, pp. 143–167. Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_8
27. Blicha, M., Hyvärinen, A.E.J., Marescotti, M., Sharygina, N.: A cooperative parallelization approach for property-directed k-induction. In: Beyer, D., Zufferey, D. (eds.) Proceedings of the VMCAI. LNCS, vol. 11990, pp. 270–292. Springer (2020). https://doi.org/10.1007/978-3-030-39322-9_13
28. Braione, P., Denaro, G., Mattavelli, A., Pezzè, M.: Combining symbolic execution and search-based testing for programs with complex heap inputs. In: Bultan, T., Sen, K. (eds.) Proceedings of the ISSTA, pp. 90–101. ACM (2017). https://doi.org/10.1145/3092703.3092715
29. Bruns, G., Godefroid, P.: Model checking partial state spaces with 3-valued temporal logics. In: Halbwachs, N., Peled, D.A. (eds.) Proceedings of the CAV. LNCS, vol. 1633, pp. 274–287. Springer (1999). https://doi.org/10.1007/3-540-48683-6_25
30. Bu, L., Xie, Z., Lyu, L., Li, Y., Guo, X., Zhao, J., Li, X.: BRICK: path enumeration based bounded reachability checking of C program (competition contribution). In: Fisman, D., Rosu, G. (eds.) Proceedings of the TACAS. LNCS, vol. 13244, pp. 408–412. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_22
31. Burnim, J., Sen, K.: Heuristics for scalable dynamic test generation. In: Proceedings of the ASE, pp. 443–446. IEEE Computer Society (2008). https://doi.org/10.1109/ASE.2008.69
32. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) Proceedings of the OSDI, pp. 209–224. USENIX Association (2008)
33. Christakis, M., Müller, P., Wüstholz, V.: Collaborative verification and testing with explicit assumptions. In: Giannakopoulou, D., Méry, D. (eds.) Proceedings of the FM. LNCS, vol. 7436, pp. 132–146. Springer (2012). https://doi.org/10.1007/978-3-642-32759-9_13
34. Christakis, M., Müller, P., Wüstholz, V.: Guiding dynamic symbolic execution toward unverified program executions. In: Dillon, L.K., Visser, W., Williams, L. (eds.) Proceedings of the ICSE, pp. 144–155. ACM (2016). https://doi.org/10.1145/2884781.2884843
35. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proceedings of the CAV, pp. 154–169. LNCS 1855, Springer (2000). https://doi.org/10.1007/10722167_15
36. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer, Berlin (2018)
37. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) Proceedings of the POPL, pp. 238–252. ACM (1977). https://doi.org/10.1145/512950.512973
38. Csallner, C., Smaragdakis, Y.: Check 'n' Crash: combining static checking and testing. In: Roman, G., Griswold, W.G., Nuseibeh, B. (eds.) Proceedings of the ICSE, pp. 422–431. ACM (2005). https://doi.org/10.1145/1062455.1062533
39. Csallner, C., Smaragdakis, Y., Xie, T.: DSD-Crasher: a hybrid analysis tool for bug finding. TOSEM **17**(2), 8:1-8:37 (2008). https://doi.org/10.1145/1348250.1348254
40. Czech, M., Hüllermeier, E., Jakobs, M., Wehrheim, H.: Predicting rankings of software verification tools. In: Proceedings of the SWAN, pp. 23–26. ACM (2017). https://doi.org/10.1145/3121257.3121262
41. Czech, M., Jakobs, M., Wehrheim, H.: Just test what you cannot verify! In: Egyed, A., Schaefer, I. (eds.) Proceedings of the FASE. LNCS, vol. 9033, pp. 100–114. Springer (2015). https://doi.org/10.1007/978-3-662-46675-9_7
42. Daca, P., Gupta, A., Henzinger, T.A.: Abstraction-driven concolic testing. In: Jobstmann, B., Leino, K.R.M. (eds.) Proceedings of the VMCAI. LNCS, vol. 9583, pp. 328–347. Springer (2016). https://doi.org/10.1007/978-3-662-49122-5_16
43. Dangl, M., Löwe, S., Wendler, P.: CPAchecker with support for recursive programs and floating-point arithmetic-(competition contribution). In: Proceedings of the TACS. LNCS, vol. 9035, pp. 423–425. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_34
44. Demyanova, Y., Pani, T., Veith, H., Zuleger, F.: Empirical software metrics for benchmarking of verification tools. In: Proceedings of the CAV. LNCS, vol. 9206, pp. 561–579. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_39
45. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) VSL. LNCS, vol. 8559, pp. 737–744. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_49
46. Gao, M., He, L., Majumdar, R., Wang, Z.: LLSPLAT: improving concolic testing by bounded model checking. In: Proceedings of the SCAM, pp. 127–136. IEEE (2016). https://doi.org/10.1109/SCAM.2016.26
47. Gargantini, A., Vavassori, P.: Using decision trees to aid algorithm selection in combinatorial interaction tests generation. In: Proceedings of the ICST, pp. 1–10. IEEE (2015). https://doi.org/10.1109/ICSTW.2015.7107442
48. Ge, X., Taneja, K., Xie, T., Tillmann, N.: DyTa: dynamic symbolic execution guided with static verification results. In: Taylor, R.N., Gall, H.C., Medvidovic, N. (eds.) Proceedings of the ICSE, pp. 992–994. ACM (2011). https://doi.org/10.1145/1985793.1985971
49. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.: Compositional may-must program analysis: unleashing the power of alternation. In: Hermenegildo, M.V., Palsberg, J. (eds.) Proceedings of the POPL, pp. 43–56. ACM (2010). https://doi.org/10.1145/1706299.1706307
50. Groce, A., Zhang, C., Eide, E., Chen, Y., Regehr, J.: Swarm testing. In: Proceedings of the ISSTA, pp. 78–88. ACM (2012). https://doi.org/10.1145/2338965.2336763
51. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: SYNERGY: a new algorithm for property checking. In: Young, M., Devanbu, P.T. (eds.) Proceedings of the FSE, pp. 117–127. ACM (2006). https://doi.org/10.1145/1181775.1181790

52. Gurfinkel, A., Ivrii, A.: K-induction without unrolling. In: Stewart, D., Weissenbacher, G. (eds.) Proceedings of the FMCAD, pp. 148–155. IEEE (2017). https://doi.org/10.23919/FMCAD.2017.8102253
53. Haltermann, J., Jakobs, M., Richter, C., Wehrheim, H.: Parallel program analysis via range splitting. In: Lambers, L., Uchitel, S. (eds.) Proceedings of the FASE. LNCS, vol. 13991, pp. 195–219. Springer (2023). https://doi.org/10.1007/978-3-031-30826-0_11
54. Haltermann, J., Jakobs, M., Richter, C., Wehrheim, H.: Ranged program analysis via instrumentation. In: Ferreira, C., Willemse, T.A.C. (eds.) Proceedings of the SEFM. LNCS, vol. 14323, pp. 145–164. Springer (2023). https://doi.org/10.1007/978-3-031-47115-5_9
55. Haltermann, J., Wehrheim, H.: CoVEGI: cooperative verification via externally generated invariants. In: Guerra, E., Stoelinga, M. (eds.) Proceedings of the FASE. LNCS, vol. 12649, pp. 108–129. Springer (2021). https://doi.org/10.1007/978-3-030-71500-7_6
56. Haltermann, J., Wehrheim, H.: Information exchange between over- and underapproximating software analyses. In: Schlingloff, B., Chai, M. (eds.) Proceedings of the SEFM. LNCS, vol. 13550, pp. 37–54. Springer (2022). https://doi.org/10.1007/978-3-031-17108-6_3
57. Haltermann, J., Wehrheim, H.: Artifact for 'information exchange between over- and underapproximating software analyses (2023). https://doi.org/10.5281/zenodo.6749669
58. Heizmann, M., Chen, Y., Dietsch, D., Greitschus, M., Hoenicke, J., Li, Y., Nutz, A., Musa, B., Schilling, C., Schindler, T., Podelski, A.: Ultimate automizer and the search for perfect interpolants-(competition contribution). In: Proceedings of the TACAS. LNCS, vol. 10806, pp. 447–451. Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_30
59. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Sharygina, N., Veith, H. (eds.) Proceedings of the CAV. LNCS, vol. 8044, pp. 36–52. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2
60. Helm, D., Kübler, F., Reif, M., Eichberg, M., Mezini, M.: Modular collaborative program analysis in OPAL. In: Proceedings of the FSE, pp. 184–196. ACM (2020). https://doi.org/10.1145/3368089.3409765
61. Holík, L., Kotoun, M., Peringer, P., Soková, V., Trtík, M., Vojnar, T.: Predator shape analysis tool suite. In: Proceedings of the HVC. LNCS, vol. 10028, pp. 202–209 (2016). https://doi.org/10.1007/978-3-319-49052-6_13
62. Holzmann, G.J., Joshi, R., Groce, A.: Swarm verification. In: Proceedings of the ASE, pp. 1–6. IEEE (2008). https://doi.org/10.1109/ASE.2008.9
63. Huster, S., Ströbele, J., Ruf, J., Kropf, T., Rosenstiel, W.: Using robustness testing to handle incomplete verification results when combining verification and testing techniques. In: Yevtushenko, N., Cavalli, A.R., Yenigün, H. (eds.) Proceedings of the ICTSS. LNCS, vol. 10533, pp. 54–70. Springer (2017). https://doi.org/10.1007/978-3-319-67549-7_4
64. Inkumsah, K., Xie, T.: Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In: Proceedings of the ASE, pp. 297–306. IEEE (2008). https://doi.org/10.1109/ASE.2008.40
65. Jakobs, M.: Coveritest with dynamic partitioning of the iteration time limit (competition contribution). In: Wehrheim, H., Cabot, J. (eds.) Proceedings of the FASE. LNCS, vol. 12076, pp. 540–544. Springer (2020). https://doi.org/10.1007/978-3-030-45234-6_30
66. Jakobs, M., Richter, C.: Coveritest with adaptive time scheduling (competition contribution). In: Guerra, E., Stoelinga, M. (eds.) Proceedings of the FASE. LNCS, vol. 12649, pp. 358–362. Springer (2021). https://doi.org/10.1007/978-3-030-71500-7_18
67. Jakobs, M., Wehrheim, H.: Compact Proof Witnesses. In: Barrett, C.W., Davies, M., Kahsai, T. (eds.) Proceedings of the NFM. LNCS, vol. 10227, pp. 389–403 (2017). https://doi.org/10.1007/978-3-319-57288-8_28
68. Jia, Y., Cohen, M.B., Harman, M., Petke, J.: Learning combinatorial interaction test generation strategies using hyperheuristic search. In: Proceedings of the ICSE, pp. 540–550. IEEE (2015). https://doi.org/10.1109/ICSE.2015.71
69. Jovanovic, D., Dutertre, B.: Property-directed k-induction. In: Piskac, R., Talupur, M. (eds.) FMCAD, pp. 85–92. IEEE (2016). https://doi.org/10.1109/FMCAD.2016.7886665
70. Kroening, D., Groce, A., Clarke, E.M.: Counterexample guided abstraction refinement via program execution. In: Davies, J., Schulte, W., Barnett, M. (eds.) Proceedings of the ICFEM. LNCS, vol. 3308, pp. 224–238. Springer (2004). https://doi.org/10.1007/978-3-540-30482-1_23
71. Liu, D., Ernst, G., Murray, T., Rubinstein, B.I.P.: LEGION: best-first concolic testing. In: Proceedings of the ASE, pp. 54–65. IEEE (2020). https://doi.org/10.1145/3324884.3416629
72. Liu, D., Ernst, G., Murray, T., Rubinstein, B.I.P.: Legion: Best-first concolic testing (competition contribution). In: Wehrheim, H., Cabot, J. (eds.) Proceedings of the TACAS. LNCS, vol. 12076, pp. 545–549. Springer (2020). https://doi.org/10.1007/978-3-030-45234-6_31
73. Majumdar, R., Sen, K.: Hybrid concolic testing. In: Proceedings of the ICSE, pp. 416–426. IEEE (2007). https://doi.org/10.1109/ICSE.2007.41
74. Marques, F., Santos, J.F., Santos, N., Adão, P.: Concolic execution for webassembly. In: Ali, K., Vitek, J. (eds.) Proceedings of the ECOOP. LIPIcs, vol. 222, pp. 11:1–11:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.ECOOP.2022.11
75. Mukherjee, R., Schrammel, P., Haller, L., Kroening, D., Melham, T.: Lifting CDCL to template-based abstract domains for program verification. In: D'Souza, D., Kumar, K.N. (eds.) Proceedings of the ATVA. LNCS, vol. 10482, pp. 307–326. Springer (2017). https://doi.org/10.1007/978-3-319-68167-2_21
76. Noller, Y., Kersten, R., Pasareanu, C.S.: Badger: complexity analysis with fuzzing and symbolic execution. In: Proceedings of the ISSTA, pp. 322–332. ACM (2018). https://doi.org/10.1145/3213846.3213868
77. Nori, A.V., Rajamani, S.K., Tetali, S., Thakur, A.V.: The YogiProject: software property checking via static analysis and testing. In: Kowalewski, S., Philippou, A. (eds.) Proceedings of the TACAS. LNCS, vol. 5505, pp. 178–181. Springer (2009). https://doi.org/10.1007/978-3-642-00768-2_17
78. Richter, C., Hüllermeier, E., Jakobs, M., Wehrheim, H.: Algorithm selection for software validation based on graph kernels. JASE **27**(1), 153–186 (2020). https://doi.org/10.1007/s10515-020-00270-x
79. Sen, K., Agha, G.: CUTE and jcute: concolic unit testing and explicit path model-checking tools. In: Ball, T., Jones, R.B. (eds.) Proceedings of the CAV. LNCS, vol. 4144, pp. 419–423. Springer (2006). https://doi.org/10.1007/11817963_38
80. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Wermelinger, M., Gall, H.C. (eds.) Proceedings of the ESES/FSE, pp. 263–272. ACM (2005). https://doi.org/10.1145/1081706.1081750
81. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: augmenting fuzzing through selective symbolic execution. In: Proceedings of the NDSS. The Internet Society (2016). https://www.ndss-symposium.org/wp-content/uploads/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf
82. Tillmann, N., de Halleux, J.: Pex-white box test generation for .net. In: Beckert, B., Hähnle, R. (eds.) Proceedings of the TAP. LNCS, vol. 4966, pp. 134–153. Springer (2008). https://doi.org/10.1007/978-3-540-79124-9_10

83. Tschannen, J., Furia, C.A., Nordio, M., Meyer, B.: Usable verification of object-oriented programs by combining static and dynamic techniques. In: Proceedings of the SEFM. LNCS, vol. 7041, pp. 382–398. Springer (2011). https://doi.org/10.1007/978-3-642-24690-6_26
84. Tulsian, V., Kanade, A., Kumar, R., Lal, A., Nori, A.V.: MUX: Algorithm selection for software model checkers. In: Proceedings of the MSR, pp. 132—141. ACM (2014). https://doi.org/10.1145/2597073.2597080
85. Yin, L., Dong, W., Liu, W., Wang, J.: Parallel refinement for multi-threaded program verification. In: Proceedings of the ICSE, pp. 643–653. IEEE (2019). https://doi.org/10.1109/ICSE.2019.00074
86. Yorsh, G., Ball, T., Sagiv, M.: Testing, abstraction, theorem proving: Better together! In: Proceedings of the ISSTA, pp. 145–156. ACM (2006). https://doi.org/10.1145/1146238.1146255

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

Jan Haltermann received the B.Sc. and M.Sc. degrees in computer science from Paderborn University, in 2019, where he started working as a Research Assistant. In 2021, he moved to the Carl von Ossietzky Universität Oldenburg, where he is currently pursuing the Ph.D. degree. His research interests include formal methods, with a focus on cooperative software verification.

Heike Wehrheim received the Diploma degree in computer science from the University of Bonn, Germany, in 1992, the Ph.D. degree in computer science from the University of Hildesheim, in 1996, and the Habilitation degree from the Carl von Ossietzky Universität Oldenburg, Germany, in 2002. From 2004 to 2021, she was first an Associate and then a Full Professor at Paderborn University, Germany. Since April 2021, she has been a Full Professor at the Carl von Ossietzky Universität Oldenburg. She has published over 100 articles in journals and conferences. Her research interests include formal methods and software analysis, in particular the verification of concurrent programs. Prof. Wehrheim is a member of the Gesellschaft für Informatik (GI) and the IFIP working group 6.1. She is on the editorial board of the journals Formal Aspects of Computing and Software Tools for Technology Transfer.