Decomposing Software Verification into Off-the-Shelf Components: An Application to CEGAR

Dirk Beyer
dirk.beyer@sosy-lab.org
LMU Munich
Munich, Germany

Jan Haltermann*
jan.haltermann@uol.de
University of Oldenburg
Oldenburg, Germany

Thomas Lemberger*
thomas.lemberger@sosy.ifi.lmu.de
LMU Munich
Munich, Germany

Heike Wehrheim
heike.wehrheim@uol.de
University of Oldenburg
Oldenburg, Germany

ABSTRACT

Techniques for software verification are typically realized as cohesive units of software with tightly coupled components. This makes it difficult to re-use components, and the potential for workload distribution is limited. Innovations in software verification might find their way into practice faster if provided in smaller, more specialized components.

In this paper, we propose to strictly decompose software verification: the verification task is split into independent subtasks, implemented by only loosely coupled components communicating via clearly defined interfaces. We apply this decomposition concept to one of the most frequently employed techniques in software verification: counterexample-guided abstraction refinement (CEGAR). CEGAR is a technique to iteratively compute an abstract model of the system. We develop a decomposition of CEGAR into independent components with clearly defined interfaces that are based on existing, standardized exchange formats. Its realization *component-based CEGAR (C-CEGAR)* concerns the three core tasks of CEGAR: abstract-model exploration, feasibility check, and precision refinement. We experimentally show that — despite the necessity of exchanging complex data via interfaces — the efficiency thereby only reduces by a small constant factor while the precision in solving verification tasks even increases. We furthermore illustrate the advantages of C-CEGAR by experimenting with different implementations of components, thereby further increasing the overall effectiveness and testing that substitution of components works well.

CCS CONCEPTS

• **Software and its engineering** → **Formal software verification**; **Abstraction, modeling and modularity**; • **Theory of computation** → **Logic and verification**.

*Both authors contributed equally to this research.

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05.
https://doi.org/10.1145/3510003.3510064

KEYWORDS

Software engineering, Software verification, Abstraction refinement, CEGAR, Decomposition, Cooperative verification

ACM Reference Format:
Dirk Beyer, Jan Haltermann, Thomas Lemberger, and Heike Wehrheim. 2022. Decomposing Software Verification into Off-the-Shelf Components: An Application to CEGAR. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510064

1 INTRODUCTION

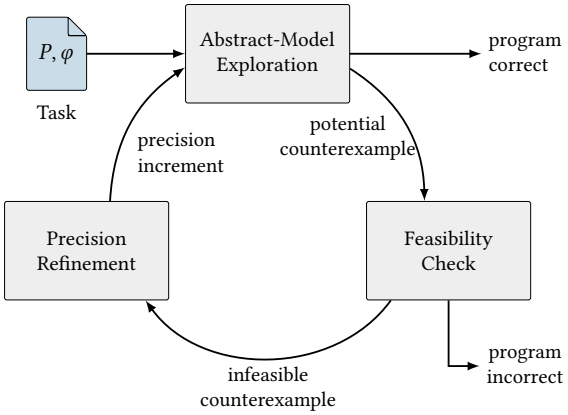


Figure 1: Workflow of classic CEGAR

Over the past decades, software verification has emerged as an area with continuous research innovations and also with increasing tool development. Competitions on software verification (SV-COMP [15], VerifyThis [67]) showcase and conserve the rapid process of tool building and application, and have also observed an interest in standardization of verification artifacts (e.g., of verification witnesses). The general task of automatic verification tools is to compute a proof or counterexample for specified requirements.

Today, the majority of existing verification tools, whether configurable or not, are strongly cohesive software units. Though software verification as a task clearly consists of individual subtasks, verifiers are typically made up of tightly coupled, stateful components

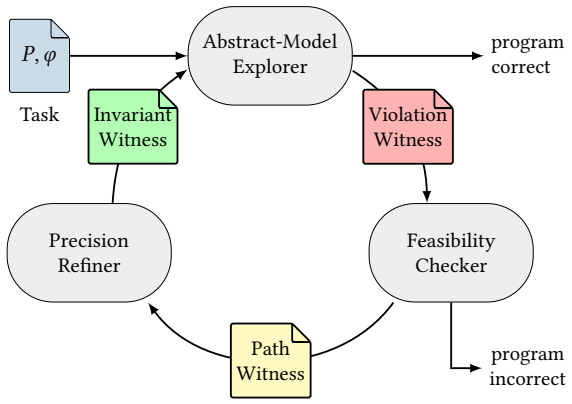


Figure 2: Workflow of component-based CEGAR

that operate on shared data structures. This architecture complicates reuse of components, impacts scalability (e.g., parallelization) and hampers exchange and integration of new components. In consequence, it often requires major implementation effort to integrate innovations in verification technology into existing tools, or is even prohibitive because the strong cohesion between existing components can not be broken easily.

To avoid this issue, we propose to employ *decomposition* concepts in the construction of verifiers. Instead of having all components integrated into a single tool, we opt for *cooperative* verification [41] where independent executable units cooperate on a verification task. Every such unit is only responsible for one well-defined subtask, and the units communicate via clearly defined interfaces.

To investigate the feasibility of such ideas, we have realized this strict decomposition into components for one of the most frequently employed techniques in software verification, *counterexample-guided abstraction refinement* (CEGAR) [53, 54]. CEGAR is a technique for automatically finding an abstract model of the software to be verified which is *as abstract as possible*, but *as precise as necessary* to successfully construct a proof of correctness or a refutation. Many tools for software verification include this CEGAR principle (e.g., [2, 4, 25, 34, 45, 47, 71, 76, 81, 83, 94, 96, 101–103]). CEGAR is also successfully employed in other areas, like probabilistic or timed-automata model checking [79, 80]. CEGAR readily lends itself to a decomposition which we realize here as *component-based CEGAR (C-CEGAR)*. Figure 1 first of all illustrates the iterative procedure of classic CEGAR: For a given level of abstraction, the exploration of an abstract model of the software (top) either proves the program correct and terminates the procedure, or finds a potential counterexample. The feasibility check (right) analyzes the counterexample. It either proves the counterexample feasible and terminates the procedure, or passes an infeasible counterexample to the next phase. The precision refinement (left) analyzes the infeasible counterexample and extracts from it a precision increment refining the abstract model which the abstract-model exploration employs in the next iteration. This cycle continues until either a correctness or a violation proof is found.

But while this general concept of CEGAR has overall proven successful (witnessed by CEGAR-based tools scoring high at SV-COMP), research into specialized techniques for the three subtasks is still ongoing. This can best be illustrated by proposals of and discussions on precision refiners [37, 38, 65, 72]. Precision refinement techniques rely on heuristics, and hence their effectiveness can only be evaluated through experiments. Due to the tight coupling of components in verifiers, new precision refiners can however neither be evaluated in isolation nor can they be integrated into existing tools without reimplementation. The past has thus unfortunately already seen multiple reimplementations of precision refiners: A vast amount of tools [4, 25, 34, 45, 47, 71, 76, 83, 96, 101] contain implementations of a refiner based on Craig interpolation and at least three tools [34, 72, 75] contain (re-)implementations of so called Newton refinement.

C-CEGAR overcomes these disadvantages of classic CEGAR by a consequent decomposition, implementing each of the three conceptual units as a stand-alone component and defining clear-cut interfaces between components. Figure 2 illustrates the workflow of C-CEGAR. For the interfaces, we employ existing standards for verification artifacts, namely violation, path, and invariant *witnesses* [20, 21], but also new formats. Witnesses are already produced by many verifiers, which allows us to partially reuse tools.

We have implemented C-CEGAR as a particular form of cooperative verification, and implemented it using the framework CoVeriTeam [33]. With this implementation at hand, we have then investigated the effects of decomposition into components on the overall effectiveness and efficiency. Our experiments show that while efficiency is slightly impacted by the necessity of data exchange via external interfaces, the overall effectiveness can even be increased. We have moreover performed the now possible independent evaluation of precision refiners, comparing Craig and Newton refinement.

Novelty. We provide the following contributions:

- We develop the concept of C-CEGAR as a composition of three independent (software) units with clearly defined interfaces based on verification witnesses and invariant maps.
- We implement C-CEGAR for C programs using the framework CoVeriTeam.
- We show the feasibility and effectiveness of C-CEGAR through a **sound** experimental evaluation on an extensive benchmark set with 8 347 verification tasks (written in C). We use 3 off-the-shelf verification tools for the three base units.
- We experimentally demonstrate that C-CEGAR makes it possible to independently evaluate components like precision refiners, which was not possible before.
- Our results are **verifiable**: all data and software are publicly available for inspection and reproduction (Sect. 6).

Significance and Potential Impact. Evaluations in the research literature and competitions show that different approaches have different strengths to solve the problem of software verification. It is therefore imperative to leverage the possibility of substituting components by alternatives, instead of (re-)implementing whole tools. Exchangeability is a key feature in component-based design and our approach can lead to components that are tuned to excel in their specific task. SAT and SMT solvers are a success story because there already is such a clearly defined interface (SAT queries, SMTlib exchange format): many applications build

on SAT and SMT solvers as components and many tools implement these component interfaces [8, 9, 11, 82].

Software verification needs to be integrated into the continuous-integration process [50], and therefore, it is important to reduce its response time. Most of the currently available software verifiers are not constructed in a way that supports massively parallel execution, but the decomposition of verification techniques into stand-alone components would enable this. With C-CEGAR, we try to improve the state of the art in this respect. We see our work as a catalyst for further research in the domain of CEGAR and as a first step towards a microservice architecture for software verification.

1.1 Related Work

There is a large body of literature on decomposition, interfaces, and cooperative verification. We restrict ourselves here to provide a few pointers to literature that directly inspired our work.

Compositionality and Decomposition. Decomposition is a central general problem-solving approach in computer science, and in particular in software engineering [63, 64]. The goal is to "divide and conquer", that is, split the problem into "easier-to-solve" sub-problems and solve them as independently as possible. Compositionality means that a system can be composed from components.

Cooperative Verification. The electronic tools integration platform (ETI) [86–88, 99] was an effort to collect and conserve tools from the formal-methods community. Wide (and public) availability of tools is the precondition to any kind of cooperation. The evidential tool bus [58, 59, 97] arose also in the formal-methods community, and tries to integrate tools that cooperate, in particular, to compose assurance claims. Conditional model checking (CMC) [26] is an approach in which several tools exchange information about the progress of the verification. CMC introduced a condition as artifact that describes which parts of the system are successfully verified so far. Conditional testing [36] applies the same idea to software testing. Sets of test goals are used as artifact to describe what has been tested so far. Conditions are also used to test what could not be verified [51, 60]. CoDiDroid [91] is a broker that delegates queries to the tools that are best suited to answer them. This way, several tools cooperate to achieve the goal.

Composition in Software Verification. There are several approaches to compose new tools from existing binary components. Reducers [31] can be composed from off-the-shelf verifiers to construct conditional verifiers and the artifact that is passed from the reducer to the verifier is a residual program. MetaVal [40] is an approach to construct a witness-based result validator from a program transformer and an off-the-shelf verifier. If the specification is large, it could be promising to decompose the specification [6].

Interfaces. Components are connected via interfaces. The interface specifies what the outside should know about the component and what types of data (or, more general, artifacts) are expected as input and output. Signatures of functions are often used in programming languages to document how a function can be used, and abstract classes (in Java: interfaces) are used to document a cohesive component or subsystem by a set of functions with their signatures which describe the service that the component or subsystem delivers. Behavioral interfaces were found to be useful for concurrent systems [61], for timed systems [62], for resources [49], for web services [16], and for program APIs and their behavior [28, 32, 77].

Verification Artifacts and Exchange Formats. Artifacts and formats that are relevant for cooperative verification were discussed recently [41]. To give some examples, artifacts for cooperative verification can be (a) programs (exchange format C: [3]), (b) specifications (exchange format: [12, 92]), (c) results (exchange format: [20, 21, 48]), and (d) conditions (exchange format: [10, 11, 26]).

Libraries and Components. Many verification approaches are based on formulas in a certain logic, and theorem provers [98] or SMT solvers [11] are used to reason about the programs or systems. SMT solvers support a standard exchange format [10], and there are even API frameworks [46, 55, 68, 84, 85] that make SMT-solvers exchangeable. There was already an idea to make reachability queries via a defined interface [18], because several verification approaches can be solved with the help of reachability queries, such as termination analysis [56, 93], test-case generation [17, 69], Impact [42, 90], and PDR [19, 43].

CEGAR. The full, concrete system implementation is often complex and abstraction can help to ignore details that are not important for proving correctness or for finding bugs. CEGAR [53, 54] is an approach that can be used to compute an abstraction of the system. There are many verification tools that use CEGAR as a component, for example, the most recent SV-COMP report mentions the following: Brick, CPA-BAM-BnB [101], CPALockator [5], CPAchecker [34], Gazer-Theta [1], JayHorn [83], PeSCo [95], UAutomizer [76], UKojak [66], UTaipan [71], and VeriAbs [2]. CEGAR is a research topic itself because of its importance [27, 37, 38, 44, 73, 90, 100].

This paper stands on the shoulders of the fine works described above: we use CEGAR, decompose it into components, use verification witnesses as interfaces, and reuse existing components for the construction of the components.

2 BACKGROUND

We start by explaining some basic notations and concepts.

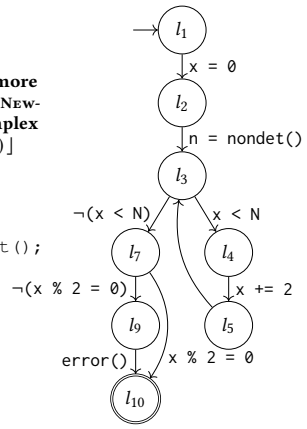
Programs. For a simplified presentation, we assume that each program contains at most one program statement on each source-code line, and that the only variable type is integer ($\mathbb{Z}$). Figures 3a and 3b show two programs. For a program P, we define the set L of all program locations (uniquely identifiable by source-code line), the program counter $pc \in L$, the set X of all program variables, the set Op of all program operations over integer variables and the program states $C = (X \to \mathbb{Z}) \cup (\{pc\} \to L)$. A program state $c \in C$ assigns a value to each program variable, and a line number to pc. A program path $c_0 \xrightarrow{op_0} \ldots \xrightarrow{op_{n-1}} c_n$ is a sequence of program states, where c_0 is an initial state with arbitrary value assignments for program variables, op_i is the program statement at program location $c_i(pc)$, and c_{i+1} is a possible successor state of c_i after executing op_i. A *control-flow automaton* (CFA) (L, ℓ_0, G) for a program P consists of the locations L, the program entry ℓ_0 and transitions $G \subseteq L \times Op \times L$, modeling the execution of a statement when moving the program counter from one location to a successor location. When control-flow branches conditionally (e.g., because of an `if-else` or `while`),

```
1  int main() {
2    unsigned int y = 1;
3    while (1) {
4      y = y + 2U * nondet();
5      if (y != 0) {}
6      else
7        error();
8    }
9  }
```

(a) Craig interpolation finds the more meaningful precision ($y \bmod 2 = 1$), Newton finds the equivalent, but more complex precision $1 \leq y + 2 * \lfloor((y * -1 + 1)/2)\rfloor$

```
1  int main(void) {
2    unsigned int x = 0;
3    unsigned short N = nondet();
4    while (x < N) {
5      x += 2;
6    }
7    if (x % 2 == 0) {}
8    else
9      error();
10 }
```

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



(c) CFA for program Fig. 3b

Figure 3: Code examples for Craig interpolation and Newton refinement

the two corresponding edges of a CFA are labeled with the condition (e.g., for the if-branch) and the negated condition (e.g., for the else-branch). Figure 3c gives the CFA of the program in Fig. 3b.

Software verification aims at analyzing the correctness of software. In this work, we suppose the verifier to check C programs for the reachability of calls to the specific error function `error`, which represents a specification violation[1]. A program P is *correct* if there is no program path that contains the statement op_i = `error()`. A state condition ϕ is a logical expression over program variables (e.g., $y = 1$), used to express state-space restrictions. A program state c fulfills a state condition ϕ when $c \models \phi$. We define the type Φ of state conditions.

Witnesses. The interfaces in our component-based CEGAR approach all come in the form of *witnesses*. An *invariant witness* describes a set of potential invariants for a program, using the formal definition of the common exchange format of correctness witness [20][2]. Intuitively, an invariant witness automaton is a CFA equipped with invariants, explaining why a property is not violated on a path or in a program. An example of an invariant witness is given in Fig. 4. More formally, the invariant witness automaton consists of a set of states Q, an initial state q_0 and a transfer relation δ. A state $q \in Q$ may summarize several concrete states of a CFA. In the example, the state q_2 represents the CFA node l_3, q_3 summarizes l_4 and l_5 and q_4 summarizes l_7, l_9 and l_{10}. In addition, states can contain invariants, e.g. state q_2 contains the invariant $x \leq 2 * (x/2)$. A transition between two states is labeled with the line number of a program location. If the location is a branch or a loop-head, the transition is in addition either

[1]We do not lose generality, as any safety property can be reduced to the call to an arbitrary function.

[2]We use the term invariant witness, as a correctness witness contains correct invariants only, in contrast to the invariant witness.

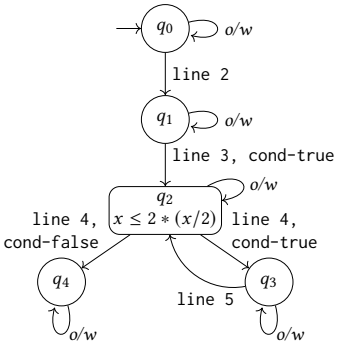


Figure 4: Invariant-witness automaton for Fig. 3b

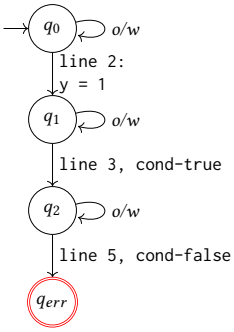


Figure 5: (Invalid) violation-witness automaton for Fig. 3a

labeled with `cond-true` or `cond-false`, indicating whether the condition is assumed to be true or false. Each state has an additional self-loop labeled o/w (*otherwise*) that can be taken if no other transition is applicable (when a state summarizes several CFA nodes). Thereby, the invariant witness covers all paths present in a CFA.

A *violation witness* in the common exchange format for witnesses [21] describes a set of program states of which at least one represents a specification violation. These program states are described by a violation witness automaton. A violation witness automaton is similarly defined to invariant witness automata, with three differences: (1) As it only represents a subset of the CFA, it may limit the CFA by not providing a o/w transition for each state, (2) it does not contain invariants and (3) its transitions may, in addition, contain state conditions, to model assumptions on the programs state. Figure 5 contains a violation witness for Fig. 3a, that describes the path of line 1 to line 7. A violation witness is called *valid*, if at least one concrete path in the program matches the described path, otherwise it is *invalid*. As the program in Fig. 3a is correct, the violation witness is invalid.

To increase the confidence in verification results, SV-COMP requires since 2017 [13] that all participating verifiers report a violation witness or correctness witness as part of each verification result.

Precision Refiners. One component of C-CEGAR for which conceptually different techniques exist is the *precision refinement*. In our evaluation, we will illustrate the advantage of C-CEGAR with the possibility of an independent evaluation of precision refiners. Here, we first of all give an example to show that different forms of precision refiners have different benefits. The programs of Figures 3a and 3b (taken from SV-Benchmarks[3]) illustrate the precisions (for a predicate domain) computed by Craig interpolation [57, 78, 89] and Newton refinement [7]. In Fig. 3a, an indefinite while-loop adds a non-deterministically computed even value to program variable y (line 4) and then asserts that $y \neq 0$ (line 5). Because y is initialized with 1 in line 2, y will always stay uneven and thus unequal to 0, even if an overflow occurs. This means that the assertion always holds. Trying to prove this, Craig interpolation (implemented in CPAchecker [34]) computes the predicate $y \bmod 2 = 1$ for the loop head in line 3, which a verifier can use to construct the right level of abstraction for proving the assertion in line 5. Newton refinement

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

(implemented in Ultimate Automizer [65]) computes a predicate with the same meaning, but it is more complex and increases verification overhead: $1 \leq y+2*\lfloor((y*-1+1)/2)\rfloor$[4] In Fig. 3b, a while-loop adds value 2 to program variable x until x is greater than or equal to non-deterministic value N. Afterwards, it asserts that $x \bmod 2 = 0$. Because x is initialized with 0 and N is of type `unsigned short`, the loop can at most add $2*65536$ to x. The type of x, `unsigned int`, is large enough to hold this value. Thus, there will be no overflow on x, and x will always be even (or 0). This means that the assertion always holds. Trying to prove this, Craig interpolation creates a large number of predicates that enumerate all possible values for x, i.e., $x = 0$, $x = 2$, $x = 4$, etc. Computing these predicates requires many precision refinements and is costly. In contrast, Newton refinement finds the more helpful predicate, $x \leq 2 * (x/2)$, which encodes $x \bmod 2 = 0$ and hints to a more suited, coarse abstraction. These small examples show that there is no single technique that is optimal for all programs. C-CEGAR is designed to open up the possibility for systematically performing exactly these kind of comparisons.

3 COMPONENT-BASED CEGAR

For a decomposition of CEGAR, we need to identify its individual *components*, and precisely define the *interfaces* between components. Figure 2 depicts the resulting workflow.

3.1 Interfaces of C-CEGAR

The components of CEGAR pass (infeasible) counterexamples and precision increments among each other. We briefly discuss the information passed:

Paths. Both potential and infeasible counterexamples are typically described by (sets of) *program paths*. Program paths are sequences of program locations and program states. An exchange format for paths should allow to describe program paths both concrete and abstract, so that multiple paths can be described and information can be restricted to the important. The exact path information that is exchanged must balance precision and abstraction: A more precise description of program paths avoids imprecision, but may become very large (think about a precise description of many loop unrollings) and lead the precision refiner to produce very specific precision increments. A more abstract description of program paths may guide a precision refiner to produce more generic precision increments (which are often better), but it may also increase imprecision and require more time to analyze. It may be beneficial to not only describe syntactic program paths, but to include information about the program state, like constraints on variable values for reaching a certain program location. This can help the feasibility checker and precision refiner to reconstruct relevant information.

Precision Increment. The precision increment produced for an infeasible counterexample helps the abstract-model exploration to not explore the same infeasible counterexample again, but to prove it infeasible. The concrete type of precision depends on the abstract model explorer, but for communicating precision increments, we propose the use of partial invariants, i.e., invariants that hold for a subset of program paths. From these partial invariants, an abstract model explorer can infer its precision. For example, predicate abstraction can split partial invariants into atoms and create a mapping from program locations to predicates. When exchanging partial invariants, a balance between weak and strong invariants must be found. In addition, most of the time, smaller invariants are easier to parse and reuse than equivalent, but more complex invariants (consider the example invariants of Fig. 3a).

[4]It can be shown that this term is equivalent to $y \bmod 2 = 1$ based on the C datatypes. A detailed reasoning is given on our supplementary webpage, https://www.sosy-lab.org/research/component-based-cegar/.

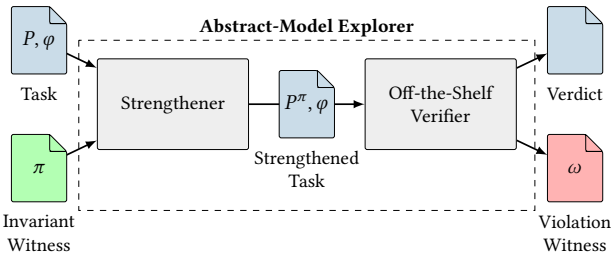


Figure 6: Construction of an abstract-model explorer from an off-the-shelf verifier

The types of these artifacts are arbitrary and information other than the proposed are possible. But to achieve the goals of C-CEGAR, common (and at the best standardized) interfaces are required. To exchange sets of program paths and precision increments, we propose to use the existing exchange formats for verification artifacts [21, 41]:

Violation Witness. We use violation witnesses for describing the (potentially infeasible) counterexample obtained from the abstract-model exploration.

Path Witness. If a violation witness is rejected by the feasibility checker, then it describes an infeasible counterexample path and no valid violation. To signify this change in the meaning of the witness, we call a rejected violation witness *path witness*. A path witness is passed from feasibility checker to precision refiner.

Invariant Witness. Precision increments are described by invariant witnesses which give (partial) invariants in a program, e.g., for invariants associated to loop heads.

With the existence of a general format for witnesses [21], we thus have tool-independent interfaces.

3.2 Components of C-CEGAR

Next, we describe the three components of C-CEGAR in more detail. The three components use the above interfaces to pass information from one to the next component. Furthermore, the components take input from and provide output to the environment.

Abstract-Model Explorer. C-CEGAR uses an *abstract-model explorer* to compute the abstraction. The abstract-model explorer takes two inputs: (1) the program P under verification and the specification φ (together called *task*), and (2) an invariant witness, and provides two outputs: (1) potentially the final verdict 'correct', and, if the verdict is not 'correct', (2) a violation witness that describes at least one potential counterexample path. The input invariant witness describes the precision increment. The contained (partial) invariants are parsed and used for improving the precision of the abstraction employed during model exploration.

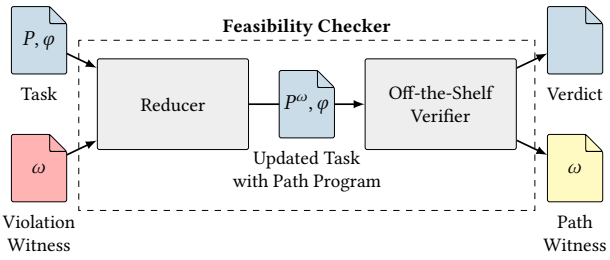


Figure 7: Construction of a feasibility checker from an off-the-shelf verifier

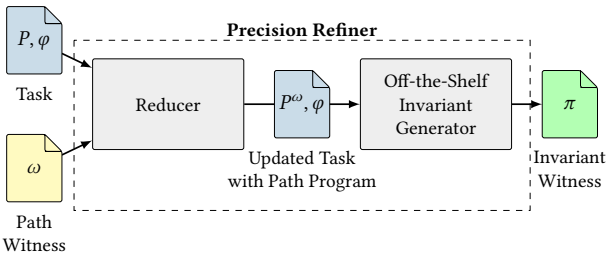


Figure 8: Construction of a precision refiner from an off-the-shelf invariant generator

Feasibility Checker. The feasibility checker is responsible for checking counterexample paths for feasibility. A feasibility checker takes two inputs: (1) the task and (2) a violation witness. It provides two outputs: (1) potentially the final verdict 'incorrect' and, if the violation witness contains no feasible counterexample path, (2) a path witness that describes no feasible and at least one of the infeasible counterexample paths contained in the input violation witness.

Precision Refiner. The task of the precision refiner is to compute new (refined) precision increments for the abstraction. A precision refiner takes two inputs: (1) the task and (2) a path witness. It provides as single output an invariant witness that describes a precision increment, computed based on the path witness.

3.3 Usage of Off-the-Shelf Components

For a realization of C-CEGAR as a cooperation of off-the-shelf components, implementations of all three components are required. A key advantage guaranteed by the usage of witnesses is the fact that such components can (partially) be *generated* with existing off-the-shelf verifiers.

Abstract-Model Explorer. Any off-the-shelf verifier can be turned into an abstract-model explorer (Fig. 6) by encoding the invariants in the invariant witness (which the verifier might not natively understand) as additional code (assertions) in the program using MetaVal [40] — we call a task enriched with such invariants *strengthened task* (Fig. 6). In addition, verifiers CPAchecker [34] and UAutomizer [75] natively support parsing invariant witnesses and using them for abstract-model exploration.

Feasibility Checker. Any existing results validator [21] for violation witnesses, of which there are plenty [14], can work as feasibility checker (Fig. 8). Furthermore, any off-the-shelf verifier can be turned into a feasibility checker by transforming the violation witness into program code [27, 40]. This so-called *path program* only encodes the program parts encoded by the witness.

Precision Refiner. Finally, we can generate precision refiners out of invariant-generation tools (like [27]). To this end, we combine the current task and a path witness into an *updated* task [31] which only contains those parts of the program which cover the infeasible counterexample paths contained in the path witness. This updated task is then passed to invariant generation (Fig. 8). If an invariant generator does not support the output format of invariant witnesses, existing techniques [74] can perform this transformation. In addition, any feasibility checker that is able to output an invariant witness can be used as precision refiner.

Overall, this shows the advantage of the decomposition of CEGAR: once such a component-based framework is available, different tools can be plugged into it, with the help of various program transformations even those that do not natively support these interfaces.

4 IMPLEMENTATION

To realize a first C-CEGAR instance, we started with a decomposition of CPAchecker's implementation of CEGAR with predicate abstraction. Afterwards, we experimented with the usage of other off-the-shelf components as Feasibility Checker and Precision Refiner.

4.1 CPAchecker's Predicate Abstraction

CPAchecker [34] is a configurable tool for software verification offering many different analysis techniques, especially providing an implementation of predicate abstraction [70] using CEGAR. It has a mature code base and has proven its ability to verify and falsify programs by winning medals in the category *Overall* in SV-COMP '22 for the fifth year in a row; among others, by applying predicate abstraction.

CPAchecker's implementation of predicate abstraction (*Pred*) is a program analysis comprising two modules, a model explorer and a combined feasibility checker and precision refiner. The analysis is based on the CPA algorithm [29] with precision adjustment [30] and adjustable-block encoding (ABE) [35]. In general, the analysis information is stored in an *abstract reachability graph* (ARG), linking the analysis information with CFA nodes. The analysis stores a set of available predicates as precision for each ARG node together with a boolean formula abstracting the current state using predicates from the precision. Initially, only the predicates `true` and `false` are available. The abstraction is computed using the strongest-postcondition semantics. If the model explorer finds an (abstract) path in the ARG to an error location, this path is analyzed for feasibility.

Within Pred, a potential counterexample path is checked for feasibility by validating the path formula which is build using the strongest-postcondition semantics. It is represented in an internal format, similar and compatible with SMT-2-LIB [9]. The obtained formula is checked for satisfiability using MathSAT5 [52]. An unsatisfiable formula indicates an infeasible path. In this case, MathSAT5's Craig interpolation is used to compute a precision increment. The newly discovered predicates are added to the precision and the ARG is recomputed. Otherwise, a violation witness is computed for the found counterexample. In addition, the precision is reported using a predicate map, which is a format containing the predicates in SMT-2-LIB.

```
1 explorer = ActorFactory.create(ProgramValidator,
2     "cpa-predicate-NoRefinement.yml");
3 checker = ActorFactory.create(ProgramValidator,
4     "cpa-validate-violation-witnesses.yml");
5 refiner = ActorFactory.create(ProgramValidator,
6     "uautomizer.yml");
```

Figure 9: Example configuration of C-CEGAR components in CoVeriTeam

4.2 Decomposing CPAchecker's Predicate Abstraction

Besides the existing CEGAR implementation, CPAchecker also provides additional helpful configurations: (1) validating potential counterexamples given as violation witnesses and (2) analyzing only the part of a program described by a violation witnesses and computing a precision increment for the infeasible part. To decompose this existing implementation, we created new configurations to ensure that only the desired functionality is executed. Thereby, we obtained three standalone and stateless components:

Abstract-Model Explorer. This configuration computes only the ARG and checks whether a counterexample path is present. A potential counterexample path is exported as violation witnesses. The initial precision is given as predicate map.

Feasibility Checker. To check a given violation witness for feasibility, we use CPAchecker's existing witness-based result validation [20, 21, 23], working with violation witnesses.

Precision Refiner. The precision refiner takes the path witness as input and uses strongest-postcondition semantics to build a path formula for each path within the witness. It then computes Craig interpolants for each path and exports the computed interpolants in a predicate map.

4.3 Implementation in CoVeriTeam

The C-CEGAR implementation using these three components, realized using CoVeriTeam [33], is called *C-Pred*. CoVeriTeam is a framework for cooperative verification, allowing for the definition of new verifiers as compositions of stateless components. It provides an easy-to-use language for describing the components' inputs and outputs as well as the communication among them. Compositions are defined in a domain-specific language and components can be exchanged easily (see Fig. 9 for an example).

4.4 Off-the-Shelf Components

Besides the thus constructed C-Pred, we used several off-the-shelf components to evaluate the benefits of C-CEGAR. We searched for tools applying techniques conceptually different to CPAchecker which can either work with the exchange formats directly or can analyze the corresponding path program. We have chosen the following two tools:

FShell-witness2test. FShell-witness2test [22] is an execution-based result validator for violation witnesses that is implemented independent of any existing verification tool. FShell-witness2test extracts test vectors encoded in the violation witness automaton, converts this test vector into a compilable test harness, compiles the test harness against the program under verification, and executes it. If a specification violation is observed during execution, the violation witness is shown to be valid through program execution. If no specification violation is observed during execution, the violation witness is rejected. Due to concrete program execution, FShell-witness2test is very precise. But FShell-witness2test can only validate violation witnesses that contain, for each non-deterministic value in the program, a state condition that encodes the corresponding concrete value. For example, FShell-witness2test would neither be able to validate nor reject Fig. 5, because this violation-witness automaton does not define a concrete value for the nondeterministic method call `nondet()` in line 4 of Fig. 3a.

Ultimate Automizer. Ultimate Automizer [75, 76] is a verification tool that uses a finite state automaton for the program and encodes property violation as final states. Accepting runs of the automaton are then analyzed for feasibility. By default, it applies a CEGAR based predicate abstraction wherein the precision increment is computed using Newton refinement. Newton refinement is conceptually different from Craig interpolation which is employed by the CPAchecker precision refiner. Whenever a path in the automaton is proven infeasible, a generalization is aimed for to reduce the accepted language of the automaton. A program is thus proven correct if the language of the automaton is empty. Ultimate Automizer offers the option to only analyze the paths of a program covered by a violation witness and to store the computed precision increment in an invariant witness. Hence, we can directly use Ultimate Automizer as both feasibility checker and precision refiner – off-the-shelf.

5 EVALUATION

Within a C-CEGAR implementation, components can be easily exchanged by others which implement the same interface via different concepts. We thus allow researchers to focus on enhancing individual components, instead of reimplementing the whole CEGAR scheme. For our evaluation, we are interested in examining the overhead associated with such a decomposition. Moreover, we want to investigate whether the component-based implementation can bring an improvement over the tightly coupled one by studying novel combinations of components.

5.1 Research Questions

We have already shown the feasibility of C-CEGAR in Sect. 4. Here, we want to study the following three research questions:

RQ 1. How large is the overhead of a component-based approach that uses off-the-shelf components?

RQ 2. What are the cost for using standardized formats in C-CEGAR?

RQ 3. Can the use of different off-the-shelf components in C-CEGAR increase the overall effectiveness to solve verification tasks?

5.2 Evaluation Setup

We run our experiments on machines with an Intel Core i5-1230 v5, 3.40 GHz (8 cores), 33 GB of memory, and Ubuntu 18.04 LTS with Linux kernel 5.4.0-96-generic. To increase the reproducibility of our results, we run our experiments with BenchExec [39]. Each verification run is limited to use 15 GB of memory, 4 CPU cores,

Table 1: Comparison of CPAchecker's predicate abstraction and the component-based version in two variations

	correct			incorrect	
	overall	proof	alarm	proof	alarm
Pred	3 769	2 556	1 213	3	9
C-Pred	3 524	2 450	1 074	0	3
C-PredWit	2 854	2 110	744	0	1

and 15 min of CPU time. The used setup is comparable to the setup used in the SV-COMP.

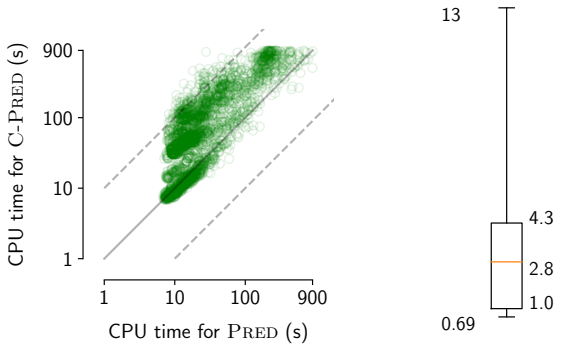
We use SV-Benchmarks, the largest available benchmark set for verification of C programs, in the version used in SV-COMP '22 [5]. We use all 8 347 verification tasks with a reachability property. A verification task can be safe (contains no violation) or unsafe (contains a violation).

We use CPAchecker version 2.1.1[6], CoVeriTeam version `c-cegar-icse2022`[7], FShell-witness2test[8] and UAutomizer[9] in its SV-COMP '22 version, UAutomizer uses a wrapper script to determine the correct configuration to use in SV-COMP. By default, this does not produce invariant witnesses if a violation witness is given. We communicated with the developers of UAutomizer and adjusted the wrapper script according to their instructions, so that UAutomizer always creates invariant witnesses. This adjusted wrapper script (and all other data and tools) is available in our supplementary artifact [24].

5.3 Evaluation Results

RQ 1 (Overhead of Component-Based Design). *Evaluation Plan:* To analyze the cost of using a component-based approach, we compare the effectiveness (RQ 1.1) and efficiency (RQ 1.2) of Pred, described in Sect. 4.1, and our component-based version C-Pred, described in Sect. 4.3. To improve comparability, we configure the model explorers of both Pred and C-Pred to start the exploration at the root of the ARG in each iteration.

RQ 1.1 (Effectiveness). Table 1 shows the experimental results of the comparison: The number of tasks solved by the component-based version C-Pred reduces from 3 769 to 3 524. There are 25 tasks that C-Pred solves even though Pred does not, but also 270 tasks that C-Pred fails to solve but Pred does (a 6.4 % decrease). For most of these tasks, the reason for failure is the decrease in efficiency: When increasing the time limit for C-Pred by the factor of twelve (to 180 min), C-Pred only fails to solve 60 tasks that Pred can solve. This is only a 1.7 % decrease compared to Pred. Reason for the remaining 60 unsolved tasks is the feasibility checker used by C-Pred. It (a) rejects more counterexamples because it is more precise than the internal check of Pred, (b) explores paths with unsupported program features that Pred does not visit, or (c) triggers SMT errors because different interpolation sequences are queried. These three issues are not related to C-CEGAR, but to the inconsistency between the internal feasibility checker of Pred and the one used by C-Pred. Except for these issues, we can conclude that C-Pred has the same expressive power and can solve the same verification tasks as the classic version Pred (when more time is given). Because the feasibility check of C-Pred is more precise than that of Pred, the number of false alarms reduces from 9 to 3.



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

(b) Distribution of run time factor for C-Pred, compared to Pred

Figure 10: Comparison of efficiency of Pred and C-Pred (across all successful verification runs)

> Decomposing an existing CEGAR implementation into components has (almost) no negative effects on the effectiveness of the approach. Moreover, the new tool can have a higher precision because better components can be used.

RQ 1.2 (Efficiency). In general, C-Pred takes more CPU time to compute the result. This effect is illustrated in the scatter-plot in Fig. 10a. The plot shows all tasks that Pred and C-Pred both solved correctly. Each point (x, y) represents a task where Pred takes x CPU-seconds and C-Pred y CPU-seconds. To visualize overlapping datapoints, each point is displayed with a transparency of 90 %. Figure 10a clearly visualizes that C-Pred has a lower efficiency compared to Pred, whereas the factor for the increased CPU time is bounded by roughly 10 (dashed line). More precisely, C-Pred uses on mean average the 3.3-fold CPU time, whereas the median increase is 2.8. Therefore, we provide a more precise insight on the time differences in Fig. 10b: In 25 % of all cases, C-Pred takes at most as much CPU time as the non-composed version (factor of 1.0). For 50 % the increase is bounded by the factor 2.8 and in 75 % of the cases, the CPU time increases by at most 4.3. The upper whisker at 13, which includes 99 % of all data, shows that there are some tasks for which C-Pred takes notably longer. Thus the median is more meaningful. To increase readability, 35 outliers, ranging from factor 13 to 31, are not shown.

We also observed that the median increase strongly correlates with the number of CEGAR iterations needed to solve a task. Figure 11 visualizes the median increase, grouped by the number of CEGAR iterations needed. Note that the i-th bars's *width* represents the number of tasks that can be solved in i iterations. When the task can be solved within a single CEGAR iteration, in the median, the CPU time does not increase (factor of 0.9). Almost 95 % of all tasks are solved within at most 5 CEGAR iterations. As the number of tasks solved with more than five iterations is smaller than 200, the

[5] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/svcomp22
[6] https://doi.org/10.5281/zenodo.5898968
[7] https://gitlab.com/sosy-lab/software/coveriteam/-/tree/c-cegar-icse2022/
[8] https://gitlab.com/sosy-lab/sv-comp/archives-2022/-/raw/main/2022/val_fshell-witness2test.zip
[9] https://doi.org/10.5281/zenodo.5898990

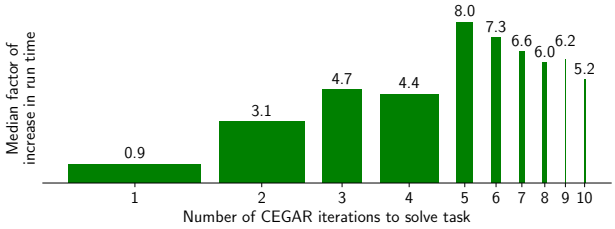
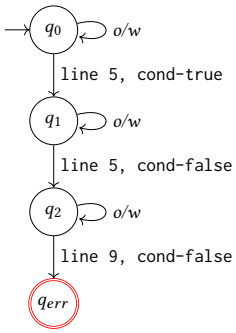


Figure 11: Median factor of run-time increase by C-Pred compared to Pred, for the first 10 numbers of CEGAR iterations. The width of the bar for i corresponds to the number of verification tasks that require exactly i CEGAR iterations

```
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 }
```

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



(b) Counterexample computed by model explorer

Figure 12: Example comparing predicate map and invariant witness as exchange formats

median may not perfectly summarize these iterations. We present the full figure and the raw data on our supplementary webpage [10]. The additional run time consumed by C-Pred stems mostly from the following facts: (1) Due to the three stateless components, less caching is possible (e.g. for incremental solver usage), (2) each component has to recompute basic information for the program, especially the CFA, which yields non-negligible I/O-overhead, and (3) redundant counterexample checks may be performed because feasibility check and precision refinement are fully decoupled.

> The efficiency of C-Pred decreases only by a constant factor (median smaller than three).

RQ 2 (Cost of Standardized Formats).

Evaluation Plan: Instead of encoding the precision increment computed by the precision using the CPAchecker internal format predicate map, we use a standardized format, namely the invariant witness. We call this configuration C-PredWit. We compare the effectiveness and efficiency of C-Pred with C-PredWit.

Table 1 also contains the experimental results of C-PredWit. This configuration can solve in total 2 854 tasks, computing 2 110 correct proofs and 744 correct alarms. Compared to C-Pred, the effectiveness reduces by 670 tasks, a decrease of around 20 %. This decrease follows mostly from the fact that the precision refiner does not add the computed precision increment to the invariant

[10] https://www.sosy-lab.org/research/component-based-cegar/

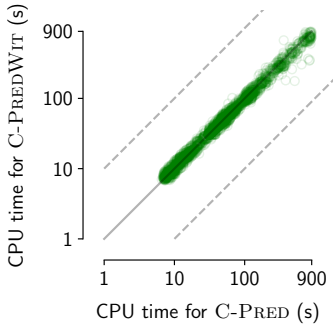


Figure 13: Comparison of run time per task of C-Pred and C-PredWit (in CPU time seconds)

witnesses. As a result, C-PredWit gets stuck in an endless loop and eventually aborts the computation.

Since invariant witnesses are not primarily designed for the exchange of a precision increment, we regularly observe this behavior. We exemplify its main reason in Fig. 12: Figure 12a contains a simplified C program from our evaluation. Before the loop body is executed for the first time, the value of x is 1 and y has the value 0. After the first loop iteration, x has the value of 0 and y's value is unequal to 0. The model explorer computes the spurious counterexample visualized in Fig. 12b. The path contains exactly one loop iteration (q_0 to q_1) and leads to the error location afterwards (q_1 to q_2 to q_{err}). A helpful precision increment which can be used to prove the counterexample to be spurious and the program to be correct contains the predicate $(y = 0 \wedge x \neq 0)$ for state q_0 and the predicate $(y \neq 0 \wedge x = 0)$ for state q_1. Although the invariant witnesses format can conceptually be used to express loop unrollings and thus can contain these two predicates, none of the precision refiners used encode these or comparable predicates in an invariant witness.

In contrast, the predicate map used to exchange information in C-Pred contains the predicates $y = 0$, $y \neq 0$, $x = 0$, and $x \neq 0$, which enable the model explorer to remove the spurious counterexample.

Next, we compare the efficiency of C-Pred and C-PredWit. Figure 13 compares the CPU time used to compute the correct solution for a task. It is visible that, except for a few outliers, both tools have the same efficiency.

> The effectiveness of C-CEGAR reduces by 20 % when using standardized formats, whereas the efficiency is not influenced.

RQ 3 (Benefit of Different Components). Finally, we analyze the advantages of the component-based design by replacing the CPAchecker components by existing off-the-shelf implementations. Here, we consider two separate questions, exchanging the feasibility checker in RQ 3.1 and the precision refiner in RQ 3.2. In the following, we are using violation and invariant witnesses as exchange formats.

RQ 3.1 (Benefit of Different Feasibility Checkers). Evaluation Plan: To analyze how different feasibility checkers influence the effectiveness and efficiency, we replace CPAchecker's witness validation with both FShell-witness2test and UAutomizer. Then, we compare the effectiveness of the three resulting C-CEGAR configurations.

Table 2: C-CEGAR using different components

RQ 3.1: C-PredWit + different feasibility checker (with precision refiner CPAchecker)

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

RQ 3.2: C-PredWit + different precision refiner (with feasibility checker CPAchecker)

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

Table 2 shows the experimental results: For each of the three configurations, it shows the overall correct results, the correct proofs, the unique proofs, the correct alarms, the unique alarms, the incorrect proofs and alarms, and the unknown results. A proof or alarm is considered unique if the corresponding configuration is the only one that achieves that result. The table shows that C-Pred with CPAchecker as feasibility checker produces the best results. Considering the unique results among these three configurations, it is visible that all three feasibility checkers allow the verification of tasks that neither of the other two configurations can solve.

> C-CEGAR allows a simple exchange of feasibility checkers. The use of conceptually different off-the-shelf checkers can increase the effectiveness of C-CEGAR.

RQ 3.2 (Benefit of Different Precision Refiners). Evaluation Plan: We replace CPAchecker's precision refiner (which uses Craig interpolation) by an existing tool, applying a conceptually different refinement strategy. Therefore, we use a configuration of Ultimate Automizer for the path described in the violation witness, computing the Newton refinement. To the best of our knowledge, Ultimate Automizer is the only formal-verification tool that is able to process violation witnesses as additional input and that also outputs invariant witnesses. Note that, in theory, any verification tool can be transformed to process violation witnesses through program transformation (as explained in Sect. 3.2). Unfortunately (based on SV-COMP '21) no other verification tool is able to produce meaningful invariant witnesses (this evaluation is available on our supplementary webpage [11]).

Our objective is to show the most important advantage of C-CEGAR, namely that using complementary techniques can lead to an increased effectiveness through uniquely solved tasks. Table 2 also contains the results for C-Pred using CPAchecker and Ultimate Automizer as precision refiner. C-PredWit with Ultimate Automizer as precision refiner is able to find 1 430 proofs (vs. 2 110) and 309 alarms (vs. 744). These numbers are lower than C-PredWit with CPAchecker as precision refiner, but this combination is still able to find 29 proofs and 1 alarm that are not found by C-PredWit with CPAchecker. This shows that different precision refiners have different strengths and weaknesses, so the easy replacement offered by C-CEGAR can be beneficial.

Taking a closer look at the two tasks given as motivating examples in Fig. 3a and Fig. 3b, we observe the following: For Fig. 3b, Ultimate Automizer is able to export a meaningful precision increment when CPAchecker in contrast starts enumerating valid assignments for x. In this case, the configuration with Ultimate Automizer as precision refiner can continue the analysis and solve tasks that cannot be solved by the other configuration. On the other hand, the precision increment computed by Ultimate Automizer often contains correct but complex predicates, for which the model explorer runs into a timeout. One example is given in Fig. 3a, where the precision increment $(1 \leq y + 2 * \lfloor((y * -1 + 1)/2)\rfloor)$ is logically equivalent to $(y \bmod 2 = 1)$, found by CPAchecker, but expressed in a more complex way.

> C-CEGAR allows a simple exchange of precision refiners. The use of conceptually different off-the-shelf refiners can increase the effectiveness of C-CEGAR.

5.4 Threats to Validity

We have conducted our evaluation using the dataset SV-Benchmarks, (https://github.com/sosy-lab/sv-benchmarks), which is the largest publicly available benchmark set for C program verification and also used by competitions. Although this dataset is widely used and accepted for benchmarking, our findings may not completely carry over to real-world C programs or other programming languages. Regarding resources, we limited the CPU limit to 15 min and memory to 15 GB. More resources will lead to improved results; but both the new approach and the baseline would benefit from more resources.

We considered only off-the-shelf tools that participated in SV-COMP, because we consider them state of the art. For verification witnesses, we used the standardized format https://github.com/sosy-lab/sv-witnesses, which is also used in SV-COMP. Using other existing tools in addition may lead to different results. To the best of our knowledge, there are no other standardized formats applicable in the C-CEGAR setting or other tools that can process the

[11] https://www.sosy-lab.org/research/component-based-cegar/

used exchange formats properly. Even if such formats would exist or other tools are applicable but do not increase the effectiveness, our findings remain valid. In addition, we cannot guarantee that decomposing other existing CEGAR schemes into components lead to the same results.

As the results from C-Pred and Pred show a high agreement in the results, we are confident that the implementation does not suffer from bugs. Anyhow, such bugs would influence the effectiveness only negatively and our findings would remain valid.

The reported data may deviate on reproduction due to different experimentation environments and measurement errors. To guarantee that our reported data has the highest precision possible, we conducted the experiments using the benchmarking framework BenchExec. To account for small, expected measurement errors, we restrict the presentation of our data to two significant digits.

6 CONCLUSION

Software verification is an *important* and *complex* problems in computer science, important because our society depends on correctly functioning software, and complex because the problem is in general undecidable. Software engineering offers the idea of decomposition [63, 64] to tackle complexity, in order to be able to focus on subproblems which are easier to solve than the overall problem.

This paper investigated the problem of decomposing the often-used CEGAR approach into components for which we can take publicly available binary components ("off-the-shelf"). This opens up many new opportunities. In particular, researchers can now focus on developing highly tuned components for each of the subproblems, and there are easy ways to parallelize software verification in order to reduce the response time. However, tool developers also have to make sure that their components deliver high-quality information to other components, at the best in a standardized format.

In future work, we will investigate the decomposition of further verification approaches as well as explore the options for parallelization. An obvious first idea is to slightly change the outer CEGAR loop in such a way that the abstract-model exploration generates a stream of counterexamples, each of which is investigated independently by feasibility checks and precision refinements running in subprocesses, which feed the precision increments on-the-fly back to the abstract-model exploration.

DECLARATIONS

Data Availability Statement. Our implementation is open source and available online as part of CoVeriTeam; minor adjustments for the C-Pred configuration were checked in to the project repository for CPAchecker. The implementation and all experimental data are archived and available at Zenodo [24].

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

REFERENCES

[1] Zs. Ádám, Gy. Sallai, and Á. Hajdu. 2021. Gazer-Theta: LLVM-Based Verifier Portfolio with BMC/CEGAR (Competition Contribution). In *Proc. TACAS (2) (LNCS 12652)*. Springer, 433–437. https://doi.org/10.1007/978-3-030-72013-1_27

[2] M. Afzal, A. Asia, A. Chauhan, B. Chimdyalwar, P. Darke, A. Datar, S. Kumar, and R. Venkatesh. 2019. VeriAbs: Verification by Abstraction and Test Generation. In *Proc. ASE*. 1138–1141. https://doi.org/10.1109/ASE.2019.00121

[3] American National Standards Institute. 1999. *ANSI/ISO/IEC 9899-1999: Programming Languages — C*. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA.

[4] Pavel Andrianov, Vadim Mutilin, and Alexey Khoroshilov. 2018. Predicate Abstraction Based Configurable Method for Data Race Detection in Linux Kernel. In *Proc. TMPA (CCIS 779)*. Springer. https://doi.org/10.1007/978-3-319-71734-0_2

[5] P. S. Andrianov. 2020. Analysis of Correct Synchronization of Operating System Components. *Program. Comput. Softw.* 46 (2020), 712–730. Issue 8. https://doi.org/10.1134/S0361768820080022

[6] S. Apel, D. Beyer, V. O. Mordan, V. S. Mutilin, and A. Stahlbauer. 2016. On-the-Fly Decomposition of Specifications in Software Model Checking. In *Proc. FSE*. ACM, 349–361. https://doi.org/10.1145/2950290.2950349

[7] T. Ball and S. K. Rajamani. 2002. *Generating abstract explanations of spurious counterexamples in C programs*. Technical Report MSR-TR-2002-09. Microsoft Research.

[8] Clark Barrett, Morgan Deters, Leonardo de Moura, Albert Oliveras, and Aaron Stump. 2013. 6 Years of SMT-COMP. *J. Autom. Reasoning* 50, 3 (2013), 243–277. https://doi.org/10.1007/s10817-012-9246-5

[9] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2015. *The SMT-LIB Standard: Version 2.5*. Technical Report. University of Iowa. Available at www.smt-lib.org.

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

[11] Clark Barrett and Cesare Tinelli. 2018. Satisfiability Modulo Theories. In *Handbook of Model Checking*. Springer, 305–343. https://doi.org/10.1007/978-3-319-10575-8_11

[12] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. 2020. ACSL: ANSI/ISO C Specification Language Version 1.15.

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

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

[15] D. Beyer. 2021. Software Verification: 10th Comparative Evaluation (SV-COMP 2021). In *Proc. TACAS (2) (LNCS 12652)*. Springer, 401–422. https://doi.org/10.1007/978-3-030-72013-1_24 preprint available.

[16] D. Beyer, A. Chakrabarti, and T. A. Henzinger. 2005. Web Service Interfaces. In *Proc. WWW*. ACM, 148–159. https://doi.org/10.1145/1060745.1060770

[17] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. 2004. Generating Tests from Counterexamples. In *Proc. ICSE*. IEEE, 326–335. https://doi.org/10.1109/ICSE.2004.1317455

[18] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. 2004. The Blast Query Language for Software Verification. In *Proc. SAS (LNCS 3148)*. Springer, 2–18. https://doi.org/10.1007/978-3-540-27864-1_2

[19] D. Beyer and M. Dangl. 2020. Software Verification with PDR: An Implementation of the State of the Art. In *Proc. TACAS (1) (LNCS 12078)*. Springer, 3–21. https://doi.org/10.1007/978-3-030-45190-5_1

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

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

[22] D. Beyer, M. Dangl, T. Lemberger, and M. Tautschnig. 2018. Tests from Witnesses: Execution-Based Validation of Verification Results. In *Proc. TAP (LNCS 10889)*. Springer, 3–23. https://doi.org/10.1007/978-3-319-92994-1_1

[23] D. Beyer and K. Friedberger. 2020. Violation Witnesses and Result Validation for Multi-Threaded Programs. In *Proc. ISoLA (1) (LNCS 12476)*. Springer, 449–470. https://doi.org/10.1007/978-3-030-61362-4_26

[24] D. Beyer, J. Haltermann, T. Lemberger, and H. Wehrheim. 2022. Reproduction Package (VM Version) for ICSE 2022 Article 'Decomposing Software Verification into Off-the-Shelf Components: An Application to CEGAR'. Zenodo. https://doi.org/10.5281/zenodo.5301636

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

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

[27] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. 2007. Path Invariants. In *Proc. PLDI*. ACM, 300–309. https://doi.org/10.1145/1250734.1250769

[28] D. Beyer, T. A. Henzinger, and V. Singh. 2007. Algorithms for Interface Synthesis. In *Proc. CAV (LNCS 4590)*. Springer, 4–19. https://doi.org/10.1007/978-3-540-73368-3_4

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

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

[31] D. Beyer, M.-C. Jakobs, T. Lemberger, and H. Wehrheim. 2018. Reducer-Based Construction of Conditional Verifiers. In *Proc. ICSE*. ACM, 1182–1193. https://doi.org/10.1145/3180155.3180259

[32] D. Beyer and S. Kanav. 2020. An Interface Theory for Program Verification. In *Proc. ISoLA (1) (LNCS 12476)*. Springer, 168–186. https://doi.org/10.1007/978-3-030-61362-4_9

[33] D. Beyer and S. Kanav. 2022. CoVeriTeam: On-Demand Composition of Cooperative Verification Systems (forthcoming). In *Proc. TACAS*. Springer.

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

[35] D. Beyer, M. E. Keremoglu, and P. Wendler. 2010. Predicate Abstraction with Adjustable-Block Encoding. In *Proc. FMCAD*. FMCAD, 189–197.

[36] D. Beyer and T. Lemberger. 2019. Conditional Testing: Off-the-Shelf Combination of Test-Case Generators. In *Proc. ATVA (LNCS 11781)*. Springer, 189–208. https://doi.org/10.1007/978-3-030-31784-3_11

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

[38] D. Beyer, S. Löwe, and P. Wendler. 2015. Sliced Path Prefixes: An Effective Method to Enable Refinement Selection. In *Proc. FORTE (LNCS 9039)*. Springer, 228–243. https://doi.org/10.1007/978-3-319-19195-9_15

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

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

[41] D. Beyer and H. Wehrheim. 2020. Verification Artifacts in Cooperative Verification: Survey and Unifying Component Framework. In *Proc. ISoLA (1) (LNCS 12476)*. Springer, 143–167. https://doi.org/10.1007/978-3-030-61362-4_8

[42] D. Beyer and P. Wendler. 2012. Algorithms for Software Model Checking: Predicate Abstraction vs. Impact. In *Proc. FMCAD*. FMCAD, 106–113.

[43] Johannes Birgmeier, Aaron R. Bradley, and Georg Weissenbacher. 2014. Counterexample to Induction-Guided Abstraction-Refinement (CTIGAR). In *Proc. CAV (LNCS 8559)*. Springer, 831–848. https://doi.org/10.1007/978-3-319-08867-9_55

[44] Ingo Brückner, Klaus Dräger, Bernd Finkbeiner, and Heike Wehrheim. 2007. Slicing Abstractions. In *Proc. FSEN (LNCS 4767)*. Springer, 17–32. https://doi.org/10.1007/978-3-540-75698-9_2

[45] Ingo Brückner, Klaus Dräger, Bernd Finkbeiner, and Heike Wehrheim. 2008. Slicing Abstractions. *Fundam. Inform.* 89, 4 (2008), 369–392.

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

[47] F. Cassez, A. M. Sloane, M. Roberts, M. Pigram, P. Suvanpong, and P. González de Aledo Marugán. 2017. Skink: Static Analysis of Programs in LLVM Intermediate Representation (Competition Contribution). In *Proc. TACAS (LNCS 10206)*. Springer, 380–384. https://doi.org/10.1007/978-3-662-54580-5_27

[48] R. Castaño, V. A. Braberman, D. Garbervetsky, and S. Uchitel. 2017. Model Checker Execution Reports. In *Proc. ASE*. IEEE, 200–205. https://doi.org/10.1109/ASE.2017.8115633

[49] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and M. Stoelinga. 2003. Resource interfaces. In *Proc. EMSOFT*. Springer. https://doi.org/10.1007/978-3-540-45212-6_9

[50] N. Chong, B. Cook, J. Eidelman, K. Kallas, K. Khazem, F. R. Monteiro, D. Schwartz-Narbonne, S. Tasiran, M. Tautschnig, and M. R. Tuttle. 2021. Code-level model checking in the software development workflow at Amazon Web Services. *Softw. Pract. Exp.* 51, 4 (2021), 772–797. https://doi.org/10.1002/spe.2949

[51] M. Christakis, P. Müller, and V. Wüstholz. 2012. Collaborative Verification and Testing with Explicit Assumptions. In *Proc. FM (LNCS 7436)*. Springer, 132–146. https://doi.org/10.1007/978-3-642-32759-9_13

[52] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. 2013. The MathSAT5 SMT Solver. In *Proc. TACAS (LNCS 7795)*. Springer, 93–107. https://doi.org/10.1007/978-3-642-36742-7_7

[53] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. 2000. Counterexample-Guided Abstraction Refinement. In *Proc. CAV (LNCS 1855)*. Springer, 154–169. https://doi.org/10.1007/10722167_15

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

[55] D. R. Cok. 2011. jSMTLIB: Tutorial, Validation, and Adapter Tools for SMT-LIBv2. In *Proc. NFM (LNCS 6617)*. Springer, 480–486. https://doi.org/10.1007/978-3-642-20398-5_36

[56] B. Cook, A. Podelski, and A. Rybalchenko. 2006. Terminator: Beyond Safety. In *Proc. CAV (LNCS 4144)*. Springer, 415–418. https://doi.org/10.1007/11817963_37

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

[58] Simon Cruanes, Grégoire Hamon, Sam Owre, and Natarajan Shankar. 2013. Tool Integration with the Evidential Tool Bus. In *Proc. VMCAI (LNCS 7737)*. Springer, 275–294. https://doi.org/10.1007/978-3-642-35873-9_18

[59] Simon Cruanes, Stijn Heymans, Ian Mason, Sam Owre, and Natarajan Shankar. 2014. The Semantics of Datalog for the Evidential Tool Bus. In *Specification, Algebra, and Software*. Springer, 256–275.

[60] M. Czech, M.-C. Jakobs, and H. Wehrheim. 2015. Just Test What You Cannot Verify!. In *Proc. FASE (LNCS 9033)*. Springer, 100–114. https://doi.org/10.1007/978-3-662-46675-9_7

[61] L. de Alfaro and T. A. Henzinger. 2001. Interface automata. In *Proc. FSE*. ACM, 109–120. https://doi.org/10.1145/503271.503226

[62] L. de Alfaro, T. A. Henzinger, and M. Stoelinga. 2002. Timed interfaces. In *Proc. EMSOFT*. Springer, 108–122. https://doi.org/10.1007/3-540-45828-x_9

[63] W. P. de Roever, H. Langmaack, and A. Pnueli (Eds.). 1998. *Compositionality: The Significant Difference, Proc. COMPOS'97*. Springer. https://doi.org/10.1007/3-540-49213-5

[64] T. DeMarco. 1979. *Structured Analysis and System Specification* (facsimile ed.). Prentice Hall. ISBN: 978-0138543808

[65] D. Dietsch, M. Heizmann, B. Musa, A. Nutz, and A. Podelski. 2017. Craig vs. Newton in software model checking. In *Proc. ESEC/FSE*. ACM, 487–497. https://doi.org/10.1145/3106237.3106307

[66] Evren Ermis, Jochen Hoenicke, and Andreas Podelski. 2012. Splitting via Interpolants. In *Proc. VMCAI (LNCS 7148)*. Springer, 186–201. https://doi.org/10.1007/978-3-642-27940-9_13

[67] Gidon Ernst, Marieke Huisman, Wojciech Mostowski, and Mattias Ulbrich. 2019. VerifyThis: Verification Competition with a Human Factor. In *Proc. TACAS (LNCS 11429)*. Springer, 176–195. https://doi.org/10.1007/978-3-030-17502-3_12

[68] M. Gario and A. Micheli. 2015. PySMT: A solver-agnostic library for fast prototyping of SMT-Based algorithms. In *Proc. SMT*.

[69] P. Godefroid and K. Sen. 2018. Combining Model Checking and Testing. In *Handbook of Model Checking*. Springer, 613–649. https://doi.org/10.1007/978-3-319-10575-8_19

[70] S. Graf and H. Saïdi. 1997. Construction of Abstract State Graphs with Pvs. In *Proc. CAV (LNCS 1254)*. Springer, 72–83. https://doi.org/10.1007/3-540-63166-6_10

[71] M. Greitschus, D. Dietsch, and A. Podelski. 2017. Loop Invariants from Counterexamples. In *Proc. SAS (LNCS 10422)*. Springer, 128–147. https://doi.org/10.1007/978-3-319-66706-5_7

[72] Á. Hajdu and Z. Micskei. 2020. Efficient Strategies for CEGAR-Based Model Checking. *J. Autom. Reasoning* 64, 6 (2020), 1051–1091. https://doi.org/10.1007/s10817-019-09535-x

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

[74] J. Haltermann and H. Wehrheim. 2021. CoVEGI: Cooperative Verification via Externally Generated Invariants. In *Proc. FASE (LNCS 12649)*. 108–129. https://doi.org/10.1007/978-3-030-71500-7_6

[75] M. Heizmann, Y.-F. Chen, D. Dietsch, M. Greitschus, J. Hoenicke, Y. Li, A. Nutz, B. Musa, C. Schilling, T. Schindler, and A. Podelski. 2018. Ultimate Automizer and the Search for Perfect Interpolants (Competition Contribution). In *Proc. TACAS (2) (LNCS 10806)*. Springer, 447–451. https://doi.org/10.1007/978-3-319-89963-3_30

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

[77] T. A. Henzinger, R. Jhala, and R. Majumdar. 2005. Permissive Interfaces. In *Proc. FSE*. ACM, 31–40. https://doi.org/10.1145/1095430.1081713

[78] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. 2004. Abstractions from proofs. In *Proc. POPL*. ACM, 232–244. https://doi.org/10.1145/964001.964021

[79] F. Herbreteau, B. Srivathsan, and I. Walukiewicz. 2013. Lazy Abstractions for Timed Automata. In *Proc. CAV (LNCS 8044)*. Springer. https://doi.org/10.1007/978-3-642-39799-8_71

[80] H. Hermanns, B. Wachter, and L. Zhang. 2008. Probabilistic CEGAR. In *Proc. CAV (LNCS 5123)*. Springer. https://doi.org/10.1007/978-3-540-70545-1_16

[81] Lukáš Holík, Martin Hruška, Ondřej Lengál, Adam Rogalewicz, Jirí Simácek, and Tomáš Vojnar. 2017. Forester: From Heap Shapes to Automata Predicates (Competition Contribution). In *Proc. TACAS (LNCS 10206)*. Springer, 365–369.

https://doi.org/10.1007/978-3-662-54580-5_24
[82] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. 2012. The International SAT Solver Competitions. *AI Magazine* 33, 1 (2012). https://doi.org/10.1609/aimag.v33i1.2395
[83] T. Kahsai, P. Rümmer, H. Sanchez, and M. Schäf. 2016. JayHorn: A Framework for Verifying Java programs. In *Proc. CAV (LNCS 9779)*. Springer, 352–358. https://doi.org/10.1007/978-3-319-41528-4_19
[84] E. G. Karpenkov, K. Friedberger, and D. Beyer. 2016. JavaSMT: A Unified Interface for SMT Solvers in Java. In *Proc. VSTTE (LNCS 9971)*. Springer, 139–148. https://doi.org/10.1007/978-3-319-48869-1_11
[85] M. Mann, A. Wilson, C. Tinelli, and C. W. Barrett. 2020. SMT-Switch: A solver-agnostic C++ API for SMT Solving. *arXiv/CoRR* 2007.01374 (2020). arXiv:2007.01374 https://arxiv.org/abs/2007.01374.
[86] Tiziana Margaria. 2005. Web services-Based tool-integration in the ETI platform. *Software and Systems Modeling* 4, 2 (2005), 141–156. https://doi.org/10.1007/s10270-004-0072-z
[87] Tiziana Margaria, Ralf Nagel, and Bernhard Steffen. 2005. jETI: A Tool for Remote Tool Integration. In *Proc. TACAS (LNCS 3440)*. Springer, 557–562. https://doi.org/10.1007/978-3-540-31980-1_38
[88] T. Margaria, R. Nagel, and B. Steffen. 2005. Remote integration and coordination of verification tools in jETI. In *Proc. ECBS*. 431–436. https://doi.org/10.1109/ECBS.2005.59
[89] K. L. McMillan. 2003. Interpolation and SAT-Based Model Checking. In *Proc. CAV (LNCS 2725)*. Springer, 1–13. https://doi.org/10.1007/978-3-540-45069-6_1
[90] K. L. McMillan. 2006. Lazy Abstraction with Interpolants. In *Proc. CAV (LNCS 4144)*. Springer, 123–136. https://doi.org/10.1007/11817963_14
[91] F. Pauck and H. Wehrheim. 2019. Together Strong: Cooperative Android App Analysis. In *Proc. ESEC/FSE*. ACM, 374–384. https://doi.org/10.1145/3338906.3338915
[92] Nir Piterman and Amir Pnueli. 2018. Temporal Logic and Fair Discrete Systems. In *Handbook of Model Checking*. Springer, 27–73. https://doi.org/10.1007/978-3-319-10575-8_2
[93] A. Podelski and A. Rybalchenko. 2005. Transition predicate abstraction and fair termination. In *Proc. POPL*. ACM, 132–144. https://doi.org/10.1145/1040305.1040317
[94] Z. Rakamarić and M. Emmi. 2014. SMACK: Decoupling Source Language Details from Verifier Implementations. In *Proc. CAV (LNCS 8559)*. Springer, 106–113. https://doi.org/10.1007/978-3-319-08867-9_7
[95] C. Richter, E. Hüllermeier, M.-C. Jakobs, and H. Wehrheim. 2020. Algorithm selection for software validation based on graph kernels. *Autom. Softw. Eng.* 27, 1 (2020), 153–186. https://doi.org/10.1007/s10515-020-00270-x
[96] C. Richter and H. Wehrheim. 2019. PeSCo: Predicting Sequential Combinations of Verifiers (Competition Contribution). In *Proc. TACAS (3) (LNCS 11429)*. Springer, 229–233. https://doi.org/10.1007/978-3-030-17502-3_19
[97] John M. Rushby. 2005. An Evidential Tool Bus. In *Proc. ICFEM (LNCS 3785)*. Springer, 36–36. https://doi.org/10.1007/11576280_3
[98] N. Shankar. 2018. Combining Model Checking and Deduction. In *Handbook of Model Checking*. Springer, 651–684. https://doi.org/10.1007/978-3-319-10575-8_20
[99] Bernhard Steffen, Tiziana Margaria, and Volker Braun. 1997. The Electronic Tool Integration Platform: Concepts and Design. *STTT* 1, 1-2 (1997), 9–30. https://doi.org/10.1007/s100090050003
[100] Cong Tian, Zhenhua Duan, and Zhao Duan. 2014. Making CEGAR More Efficient in Software Model Checking. *IEEE Trans. Softw. Eng.* 40, 12 (2014), 1206–1223. https://doi.org/10.1109/TSE.2014.2357442
[101] Anton R. Volkov and Mikhail U. Mandrykin. 2017. Predicate Abstractions Memory Modeling Method with Separation into Disjoint Regions. *Proceedings of the Institute for System Programming (ISPRAS)* 29 (2017), 203–216. Issue 4. https://doi.org/10.15514/ISPRAS-2017-29(4)-13
[102] D. Wang, C. Zhang, G. Chen, M. Gu, and J. Sun. 2016. C Code Verification based on the Extended Labeled Transition System Model. In *MoDELS 2016 (CEUR 1725)*. CEUR-WS.org, 48–55.
[103] Liangze Yin, Wei Dong, Wanwei Liu, and Ji Wang. 2018. On Scheduling Constraint Abstraction for Multi-Threaded Program Verification. *IEEE Trans. Softw. Eng.* (2018). https://doi.org/10.1109/TSE.2018.2864122