


An Interface Theory for Program Verification

Dirk Beyer  and Sudeep Kanav

LMU Munich, Germany

Abstract. Program verification is the problem, for a given program P and a specification ϕ , of constructing a proof of correctness for the statement “program P satisfies specification ϕ ” ($P \models \phi$) or a proof of violation ($P \not\models \phi$). Usually, a correctness proof is based on inductive invariants, and a violation proof on a violating program trace. Verification engineers typically expect that a verification tool exports these proof artifacts. We propose to view the task of program verification as constructing a behavioral interface (represented e.g. by an automaton). We start with the interface I_P of the program itself, which represents all traces of program executions. To prove correctness, we try to construct a more abstract interface I_C of the program (overapproximation) that satisfies the specification. This interface, if found, represents more traces than I_P that are all *correct* (satisfying the specification). Ultimately, we want a compact representation of the program behavior as a *correctness interface* I_C in terms of *inductive invariants*. We can then extract a correctness witness, in standard exchange format, out of such a correctness interface. Symmetrically, to prove violation, we try to construct a more concrete interface I_V of the program (underapproximation) that violates the specification. This interface, if found, represents fewer traces than I_P that are all *feasible* (can be executed). Ultimately, we want a compact representation of the program behavior as a *violation interface* I_V in terms of a *violating program trace*. We can then extract a violation witness, in standard exchange format, out of such a violation interface. This viewpoint exposes the duality of these two tasks — proving correctness and violation. It enables the decomposition of the verification process, and its tools, into (at least!) three components: interface synthesizers, refinement checkers, and specification checkers. We hope the reader finds this viewpoint useful, although the underlying ideas are not novel. We see it as a framework towards modular program verification.

Keywords: Program verification, Interface theory, Cooperative verification, Software verification, Verification interface, Verification witness, Conditional model checking, Tool combination, Modular verification

1 Introduction

Software verification solves the problem of finding out, for a given program P and a behavioral specification ϕ , whether the program fulfills the specification, writ-

Funded in part by Deutsche Forschungsgemeinschaft (DFG) – [378803395](#) (ConVeY).

ten $P \models \phi$, or not, written $P \not\models \phi$. The problem is in general undecidable [26, 48], but we can create verification tools that solve some practical instances of the problem with reasonable performance. The society and industry depends on correctly working software. As often with difficult problems, there are many different heuristics that lead to different verification tools with different strengths [7, 15, 37]. Software verification is applied more and more to industry-scale software [5, 24, 29, 39].

Our motivation is to decompose the problem of software verification in such a way that parts of the problem can be given to different verification tools, which can be specialized to solve their part of the problem. Tools for software verification usually work on an internal representation of the program, which is an overapproximation (to prove correctness), or an underapproximation (to prove violation), or neither of the two (intermediate result). We call these internal representations *verification interfaces*, and we would like to make them explicit and ideally export them to the user, such that the verification problem can be composed into sub-problems that can be solved by different tools.

In theory, the answer to the verification problem is TRUE or FALSE, and early tools only reported those answers. It became clear quickly that in practice, the value lays not in the short answer, but in the explanation—a verification witness—that describes the answer TRUE or FALSE in more detail. Thus, model checkers started exporting counterexamples when the answer was FALSE [28]. It took another 20 years to make counterexamples exchangeable using a standard XML format for violation witnesses [11]. The format was quickly adopted by many publicly available tools for software verification¹ and got extended to correctness witnesses later [10]. Exporting witnesses for decisions computed by algorithms seems to be standard also in other areas [42, 50].

Contributions. As a first step towards the decomposition of verification tools, we define interfaces, state the interface theorems (known from refinement calculus [44] and interface automata [3]) to enable modular verification, discuss the various proof flows, including the connection to verification witnesses, and discuss a few approaches as we see them through the lens of interfaces.

Related Work. The insights in this paper stem from our work on capturing the essence of the program-verification process in verification witnesses [10, 11], which is a large project that started seven years ago [21]. The basic idea is to summarize, materialize, and conserve the information that the verification system uses internally for the proof of correctness or violation.

The foundational ideas that we use in this paper are well-known, such as seeing the correctness proofs as a modular two-step approach that consists of (i) capturing the semantics and deduct specification satisfaction (e.g., using a correctness logic [34] or an incorrectness logic [45]) and (ii) base the proof on refinements [44].

The inspiration to call the objects of interest *interface* comes from the interface theories for concurrent systems [3], for timed systems [4], for resources [25], for web services [8], and for program APIs [17, 33].

¹ For C programs: <https://gitlab.com/sosy-lab/sv-comp/archives-2020/-/tree/svcomp20/2020>

2 Verification via Interfaces

For simplicity, we restrict our consideration to specifications of safety properties, and to programs that contain only variables of type integer and no function calls. The theory can be extended naturally.

2.1 Verification Interfaces

A program P is usually represented as a control-flow graph (CFA) [1, 40] or control-flow automaton [15, 16]. A control-flow automaton $P = (L, l_0, G)$ consists of a set L of program locations, an initial program location l_0 , and a set $G \subseteq L \times Ops \times L$ of control-flow edges, which transfer from one program location to another on a program operation from Ops . The program operations operate on a set of program variables X . For defining interfaces, we use protocol automata from the literature on verification witnesses [11, 20], in order to emphasize the similarity of verification interfaces with verification witnesses.

A *verification interface* $(Q, \Sigma, \delta, q_{init}, F)$ for a program P is a nondeterministic finite automaton and its components are defined as follows (the set Φ contains all predicates of a given theory over the set X of variables of P):

1. The set $Q \subseteq \Gamma \times \Phi$ is a finite set of control states, where each control state $(\gamma, \varphi) \in Q$ has a name γ from a set Γ of names, which can be used to uniquely identify a control state q within Q , and an invariant $\varphi \in \Phi$, which is a predicate over program variables that evaluates to *true* whenever a program path reaches a program location that is matched by this control state.²
2. The set $\Sigma \subseteq 2^G \times \Phi$ is the alphabet, in which each symbol $\sigma \in \Sigma$ is a pair (S, ψ) that comprises a finite set $S \subseteq G$ of CFA edges and a state condition $\psi \in \Phi$.
3. The set $\delta \subseteq Q \times \Sigma \times Q$ contains the transitions between control states, where each transition is a triple (q, σ, q') with a source state $q \in Q$, a target state $q' \in Q$, and a guard $\sigma = (S, \psi) \in \Sigma$ comprising a *source-code guard* S (syntax), which restricts a transition to the specific set $S \subseteq G$ of CFA edges, and a *state-space guard* $\psi \in \Phi$ (semantics), which restricts the state space to be considered by an analysis that consumes the protocol automaton. We also write $q \xrightarrow{\sigma} q'$ for $(q, \sigma, q') \in \delta$.
4. The control state $q_{init} \in Q$ is the initial control state of the automaton.
5. The subset $F \subseteq Q$ contains the accepting control states.

For a given interface $(Q, \Sigma, \delta, q_{init}, F)$, a sequence $\langle q_0, \dots \rangle$ of states from Q is called *path* if it starts in the initial state, i.e., $q_0 = q_{init}$, and there exists a transition between successive control states, i.e., $q_i \rightarrow q_{i+1}$ for all $i \in [0, n-1]$. A *test vector* [9] specifies the values for input variables of a program. A path p is called *P-feasible*, if a test vector exists³ for which p can be executed in P ,

² For example, an invariant that is matched for a loop-head location is called *loop invariant* of the program.

³ Note that a test vector can have length zero if no input values are necessary to execute a path.

```

1 x = nondet();
2 if (x < -10)
3   exit(1);
4
5 if (x < 0)
6   x = -x;
7
8 if (x >= 0)
9   return x;
10 else
11   error();
12

```

Listing 1: Correct program

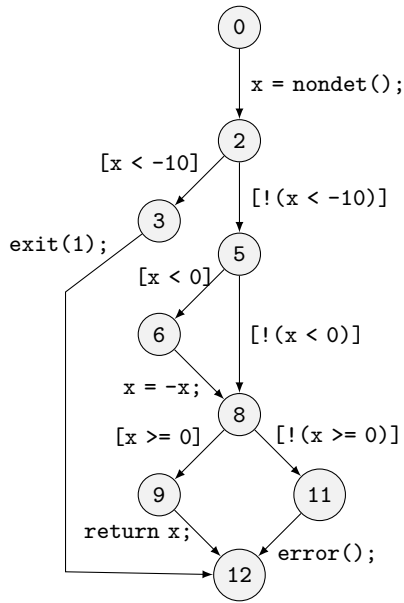


Fig. 1: Program interface for Listing 1

```

1 x = nondet();
2 if (x < -10)
3   exit(1);
4
5 if (x > 0)
6   x = -x;
7
8 if (x >= 0)
9   return x;
10 else
11   error();
12

```

Listing 2: Violating program

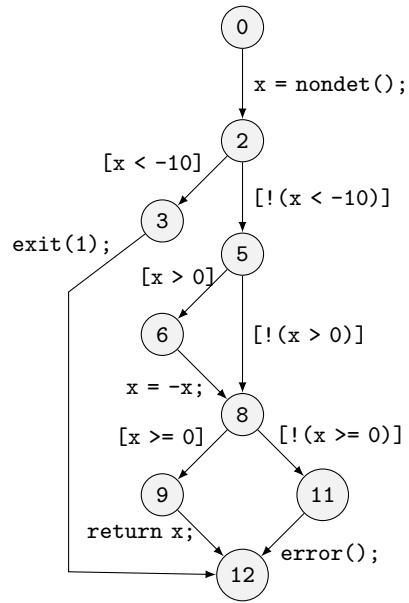


Fig. 2: Program interface Listing 2

otherwise the path is called *P-infeasible*. The *semantics* $L(I)$ of a verification interface I is defined as the set of all paths of I .

Refinement. Given two verification interfaces I_1 and I_2 , we say that I_1 *refines* I_2 , written $I_1 \preceq I_2$, if $L(I_1) \subseteq L(I_2)$.

Program Interface. If our goal is to reason about interfaces, we need to be able to represent the control-flow automaton of a program also as an interface. For a given program P , the corresponding program interface $I_P = (Q_P, \Sigma_{P \rightarrow P}, (l_0, true), Q_P)$ consists of the following components:

1. The set $Q_P = L \times \{true\}$ of control states represents the program locations, where the set L models the program-counter values, and the invariant is $true$ for all program locations.
2. The set $\Sigma_P = G_P \times \{true\}$ of alphabet symbols represents the program operations, where the set $G_P = \{\{g\} \mid g \in G\}$ models the program operations when control flows from one location to the next, and the guard is $true$ for all operations. Each transition is labeled with exactly one control-flow edge (therefore the singleton construction above).
3. The set $\rightarrow_P \subseteq Q_P \times \Sigma_P \times Q_P$ of transitions represents the control-flow edges of the program.
4. The initial control state $(l_0, true) \in Q_P$ consists of the program-entry location and the invariant $true$.
5. The set of final control states is the set Q_P of all control states, which models that the program executions can potentially end at any given time (e.g., by termination from the operating system).

The set $L(I_P)$ of paths in I_P contains (by definition) exactly the paths of P , in other words, each program execution corresponds to a P -feasible path of the verification interface I_P .

Example 1. We consider as example a program that is supposed to compute the absolute value of an integer number, if the value is not smaller than -10 . The program first reads an integer value into variable x , and exits if the value is smaller than -10 . Then, if the value is smaller than zero, the value is inverted. If the operation was successful, the new value is returned, otherwise an error is signaled. Listings 1 and 2 show two C programs, one correct and one with a typo as bug: in line 5, the programmer mistyped the less-than as a larger-than. Figures 1 and 2 show the program interfaces for the two C programs from Listings 1 and 2. We use a compact notation for a transition label (S, ψ) , where we omit the set braces for the set S of CFA edges, if S is a singleton, we omit the source and target control states and only print the operation, and we omit the state-space guard if it is $true$. The background color of a control state indicates membership in the set F : gray for final (accepting) and light-red for non-final (non-accepting) control states.

Specification Interface. Specifications are typically given as LTL formulas [46] or as monitor automata [15, 47]. Since we focus on safety specifications, we use monitor automata. In order to use a uniform formalism, we use interfaces here also. A specification interface $I_\phi = (Q_\phi, \Sigma_\phi, \rightarrow_\phi, q_{init}, F_\phi)$ consists of the following components:

1. The set $Q_\phi \subseteq \Gamma \times \{true\}$ of control states (all state invariants are $true$).
2. The set $\Sigma_\phi = 2^G \times \{true\}$ of labels that match control-flow edges, where each label has a set of control-flow edges for the matching, and the guard is $true$ for all transitions.
3. The set $\rightarrow_S \subseteq Q_\phi \times \Sigma_\phi \times Q_\phi$ of transitions represents the state changes according to the monitored control-flow edges of the program.

4. The initial control state is $q_{init} \in Q_\phi$, with the invariant *true*, i.e., $q_{init} = (\cdot, true)$.
5. The set $F_\phi \subseteq Q_\phi$ of final control states are those control states in which the interface accepts the path, that is, the represented specification is satisfied.

Correctness and Violation. Given a verification interface I and a specification ϕ , the verification interface is *correct*, written as $I \models \phi$, if $L(I) \subseteq L(I_\phi)$, or, using the notion of refinement of verification interfaces, $I \preceq I_\phi$, otherwise the verification interface is *violating*.

Verification Problem. Given a program P and a specification ϕ , *verification* is the problem of finding either a correctness proof for $P \models \phi$ or a violation proof for $P \not\models \phi$.

Since we know that the program interface I_P is path-equivalent to the program P , and that the specification interface I_ϕ represents a monitor automaton for the specification ϕ , we can restate the verification problem in terms of verification interfaces:

*Given a program P and a specification ϕ , verification is the problem of finding either a correctness proof for $I_P \preceq I_\phi$ or a violation proof for $I_P \not\preceq I_\phi$.*⁴

Traditionally, the verification problem is solved in one monolithic procedure, or in an alternating sequence of attempts to prove $P \models \phi$ or $P \not\models \phi$. Our goal is to decompose the proof-finding process into smaller parts.

Figure 3 illustrates the space of verification interfaces. Each node represents an interface and each dotted line represents that the lower interface refines the upper interface. On the very top, we have the interface I_\top , which accepts all paths, and $I \preceq I_\top$ holds for all interfaces I . On the very bottom, we have the interface I_\perp , which accepts no paths, and $I_\perp \preceq I$ holds for all interfaces I . These two parts of the picture are not interesting and we will not revisit them.

The program interface I_P is the center of the interface space, and the verification problem is to answer the question whether it belongs to the area of *correctness interfaces* (marked by $\models \phi$, light blue) or to the area of the *violation interfaces* (marked by $\not\models \phi$, red).

The specification interface I_ϕ is the top-most element in the refinement hierarchy inside the area of correctness interfaces, that is, I_ϕ is the most abstract correctness interface. If $I_P \preceq I_\phi$ holds, then there exists a refinement path through the area of correctness interfaces from the program interface to the specification interface. This is well-known from refinement calculus [44] and is applied for proving correctness. There is a symmetry for proving violation, which was not yet emphasized in the literature:

The test-vector interface I_T contains one feasible violating path and is the bottom-most element in the refinement hierarchy inside the area of violation

⁴ There are various ways for reasoning in order to obtain a proof, for example, strongest post-conditions [34] are traditionally used for correctness proofs and incorrectness logic [45] was recently proposed for violation proofs.

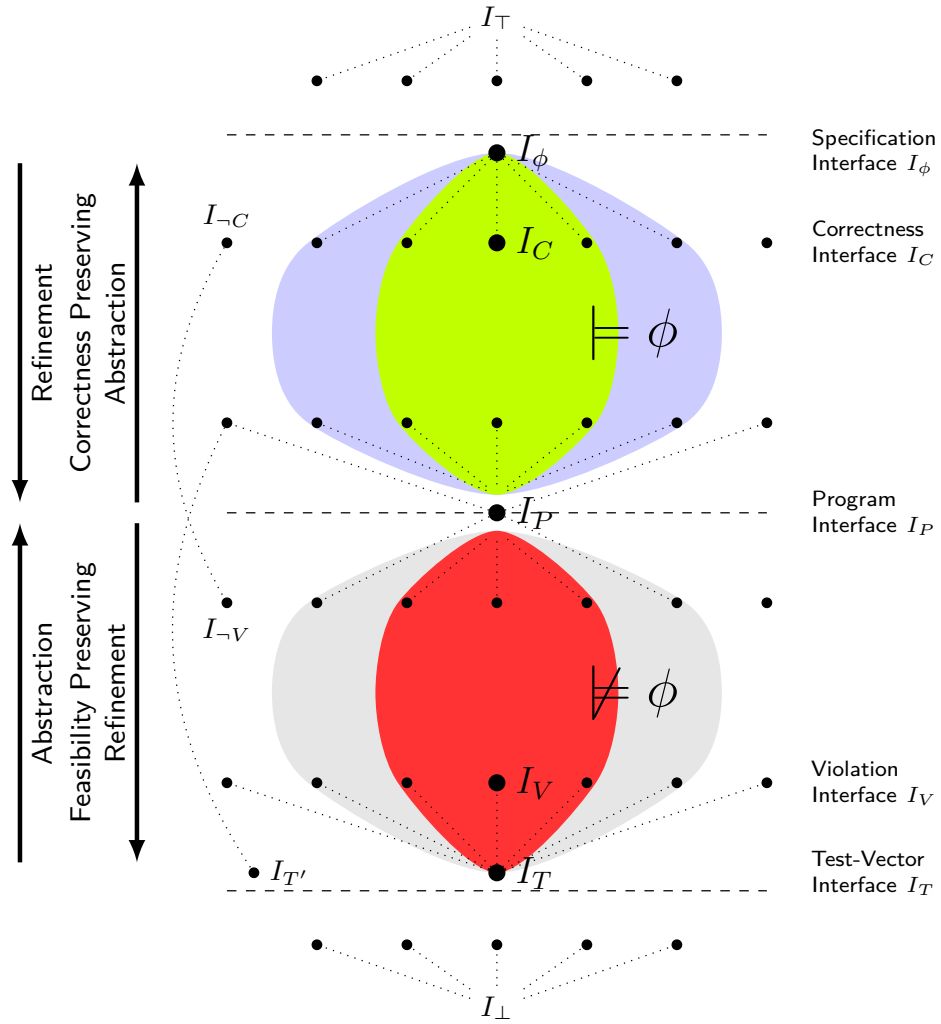


Fig. 3: Space of verification interfaces

interfaces, that is, I_T is the most concrete violation interface.⁵ If $I_T \preceq I_P$ holds, then there exists a refinement path through the area of violation interfaces from the test-vector interface to the program interface.

⁵ There might be several violating test-vectors for different bugs (as there might be different specifications for the overall correctness of the program), but let us assume for simplicity that there is only one violating test vector.

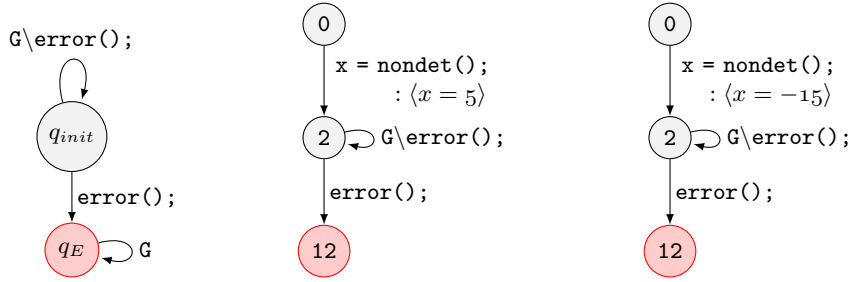


Fig. 4: Specification interface

Fig. 5: Feasible interface (test vector for Listing 2)

Fig. 6: Infeasible interface (test vector for Listing 2)

Example 2. Figure 4 shows an example specification interface (I_ϕ in Fig. 3) for representing a safety specification. The specification interface starts from an initial state q_{init} and transitions to the non-final (non-accepting, violating) control state q_E when it encounters a call to function `error`. A program is *correct* if the non-accepting state is never reached during any execution, otherwise it is said to *violate* the specification.

Figure 5 shows an example interface (I_T in Fig. 3) representing a test vector for our violating example program in Listing 2. Here, the test vector assumes that variable x was assigned the value 5 (expressed by the state-space guard after the colon) by the call to function `nondet`. Note that the label of a transition is a pair (S, ψ) and here we have S is the set $\{(0, x = \text{nondet}()); 2\}$ and ψ is the predicate $x = 5$. Then, the automaton either keeps on looping in control state 2, or transitions to the non-accepting (violating) control state 12 on a call to function `error`.

Figure 6 shows an example test-vector interface ($I_{T'}$ in Fig. 3) that is infeasible for our violating example program in Listing 2. Here, the test vector assumes that variable x was assigned the value -15 by the call to function `nondet`. Then, the automaton either keeps on looping in the control state 2, or transitions to the non-accepting (violating) control state 12 on a call to function `error`. This interface is infeasible because our program would exit (line 3 of Listing 2) if x was assigned -15.

2.2 Modular Verification using Interfaces

As illustrated in Fig. 3, there are intermediate correctness interfaces between the program and the specification, and there are intermediate violation interfaces between the program interface and the test-vector interface.

Theorem 1 (Refinement Preserves Correctness). *Given a program P , a specification ϕ , and an interface I_C , if $I_C \models \phi$ and $I_P \preceq I_C$, then $P \models \phi$.*

According to Theorem 1 [44], we can now use an intermediate correctness interface to construct a correctness proof via the interface: Given a program P , a specifi-

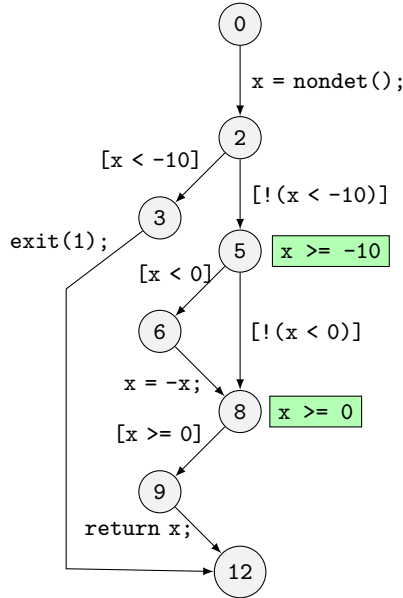


Fig. 7: Correctness interface for correct program (Listing 1)

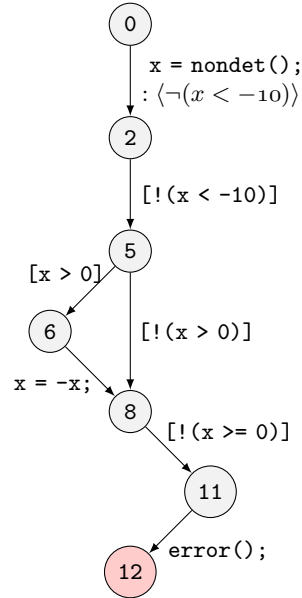


Fig. 8: Violation interface for violating program (Listing 2)

cation ϕ , and an interface I_C , to prove $P \models \phi$ it is sufficient to prove (i) $I_C \models \phi$ and (ii) $I_P \preceq I_C$. An intermediate correctness interface I_C is also drawn in Fig. 3.

The requirement for constructing correctness interfaces is to represent (a) only correct program paths (satisfying the specification) and (b) try to enlarge the set of paths until a compact form is reached. The quality of a correctness interface I_1 is often felt better than the quality of I_2 , if $I_2 \preceq I_1$, or $L(I_2) \subseteq L(I_1)$. Requirement (a) can be proven with a Hoare logic [34].

To construct an induction proof, we would like to add another requirement: (c) all the invariants in the control states of the correctness interfaces are *inductive*. Therefore, Fig. 3 has two marked areas between the program and the specification interface: The large (light-blue) area represents all correctness interfaces, the smaller (green) area represents all correctness interfaces whose invariants are *inductive*. We use the notion of *inductive invariants* as used in the literature [30].

Example 3. Figure 7 shows an example correctness interface I_C for the program in Listing 1. The green rectangles at control states show the state invariants. The paths leading to the violating program location (i.e., taking the violating transition) in the program interface of Fig. 1 are not contained in the correctness interface because they are infeasible.

To emphasize the symmetry between correctness and violation proofs, we write the below text using a wording as close as possible to the above.

Theorem 2 (Abstraction Preserves Violation). *Given a program P , a specification ϕ , and an interface I_V , if $I_V \not\models \phi$ and $I_V \preceq I_P$, then $P \not\models \phi$.*

According to [Theorem 2](#), we can now use an intermediate violation interface to construct a violation proof via the interface: Given a program P , a specification ϕ , and an interface I_V , to prove $P \not\models \phi$ it is sufficient to prove (i) $I_V \not\models \phi$ and (ii) $I_V \preceq I_P$. An intermediate violation interface I_V is also drawn in [Fig. 3](#).

The requirement for constructing violation proofs is to represent (a) only feasible program paths (being executable) and (b) try to reduce the set of paths until only one is left. The quality of a violation interface I_1 is often felt better than the quality of I_2 , if $I_1 \preceq I_2$, or $L(I_1) \subseteq L(I_2)$. Requirement (a) can be proven with an incorrectness logic [45].

To construct a counterexample proof, we would like to add another requirement: (c) *all* the feasible paths of the violation interfaces are *violating*. Therefore, [Fig. 3](#) has two marked areas between the program and the test-vector interface: The large (light gray) area represents all feasible interfaces, the smaller (red) area represents all violation interfaces that contain *only violating* paths.

Example 4. [Figure 8](#) shows an example violation interface I_V for the program in [Listing 2](#). This interface only shows the paths leading to the non-accepting (violating) control state (i.e., taking the violating transition) in [Fig. 2](#).

Theorem 3 (Substitutivity of Interfaces). *Given two verification interfaces I_1 and I_2 with $I_1 \preceq I_2$ and a specification ϕ , if $I_2 \models \phi$, then $I_1 \models \phi$ (and if $I_1 \not\models \phi$, then $I_2 \not\models \phi$).*

Using [Theorem 3](#), we can use the concept of step-wise refinement in proofs of correctness [44] and in proofs of violation [11]. [Theorem 3](#) lets us *substitute* one interface by another one while preserving the (dis-) satisfaction of the specification.

2.3 Proof Flows using Interfaces and Witnesses

[Figure 9](#) illustrates the possible ways to construct proofs. In the interface domain on the left, the figure shows the program interface I_P , a correctness interface I_C , and a violation interface I_V . In the domain of the software engineer, we have the specification ϕ , the program P , the test vector T , and two verification witnesses W_C and W_V . The correctness witness W_C [10] is a representation of the verification results if the verification tool constructed a correctness proof; the violation witness W_V [11] is a representation of the verification results if the verification tool constructed a violation proof.

Proving Correctness. To prove the correctness $P \models \phi$ for a given program P and a specification ϕ , we can use interfaces in the following way: First we embed the program P into the interface domain by constructing I_P . This is simply done by applying the definition. The creative part of the proof construction is to come up with the correctness interface I_C that contains invariants that are inductive. So the actual proof consists of three steps: (a) construct I_C , (b) show $I_P \preceq I_C$,

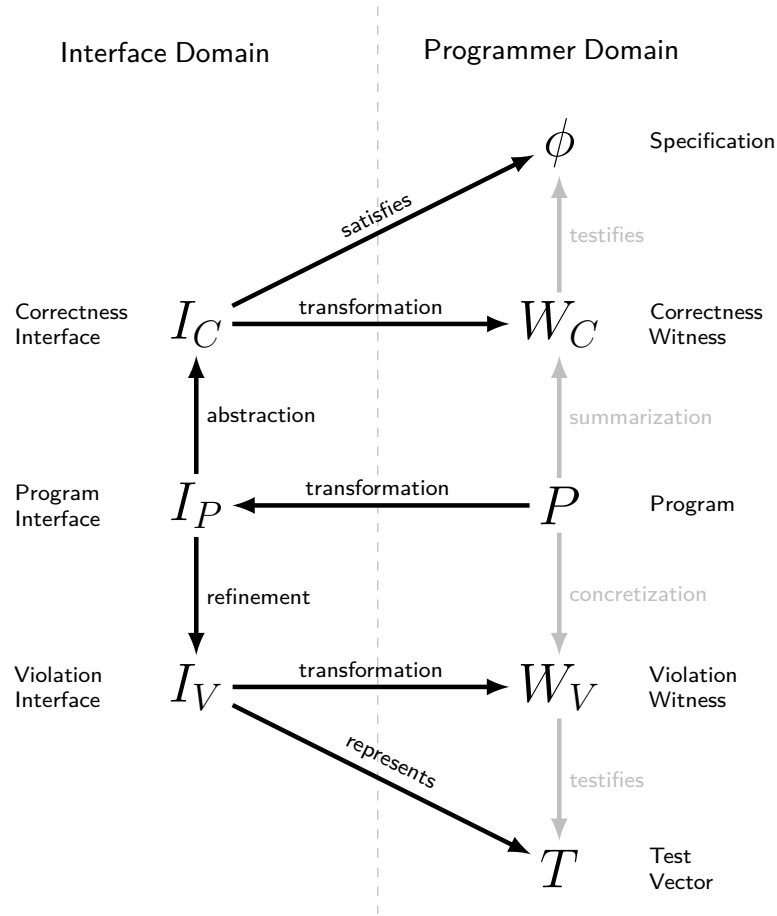


Fig. 9: Proof flows using the interface domain

and (c) show $I_C \models \phi$. At the end, we can extract a correctness witness W_C in an exchange format to share with tools and users.

A correctness witness overapproximates the correctness interface that it is extracted from. The intention of a correctness witness is to represent useful information to help reconstructing a correctness proof [10], but it might be overapproximating too much, that is, having invariants that are not inductive, or even weaker than the specification. In other words, a correctness witness might describe a set of paths that includes also violating paths, while a correctness interface is guaranteed to represent only correct (and inductive) paths.

Proving Violation. To prove the violation $P \not\models \phi$ for a given program P and a specification ϕ , we can use interfaces in the following way: First we embed the program P into the interface domain by constructing I_P . Again, this is simply done by applying the definition. The creative part of the proof construction is

to come up with the violation interface I_V that describes paths that all violate the specification. So the actual proof consists of three steps: (a) construct I_V , (b) show $I_V \preceq I_P$, and (c) show $I_V \not\models \phi$. At the end, we can extract a violation witness W_V in an exchange format to share with tools and users.

A violation witness overapproximates the violation interface that it is extracted from. The intention of a violation witness is to represent useful information to help reconstructing a violation proof [11], but it might be overapproximating too much, that is, including paths that are not violating, or not even feasible. In other words, a violation witness might describe a set of paths that includes also correct paths, while a violation interface is guaranteed to represent only feasible (and violating) paths.

3 Decomposing Verification and Cooperative Verification

The original goal of our work is to find ways to decompose verification tasks in such a way that several tools, written by different development teams, cooperate to solve the verification task. In fact, the proof flows that were explained in the previous section are actually used in practice, but their three steps are usually hidden under the hood of the verification engine, and the flow is mostly implemented in a monolithic way.

Our proposal is to make the interfaces eminent, and to explicitly separate the steps of the overall proof. From this it follows that the steps need not necessarily be taken care of by the same verifier. The idea is to decompose the overall verification process into parts that can be performed by specific tools, optimized for their part of the proof. Verification interfaces are a great tool to make program verification compositional, involving different tools that solve the problem together in a cooperative manner [20]. Thus, we need three kinds of tools:

- Interface synthesizers, to construct an interface
- Refinement checkers, to check $I_1 \preceq I_2$
- Specification checkers, to check $I \models \phi$

In the following, we put new and existing approaches to verification into the perspective of interfaces, by motivating their existence (for new or recent ones) and by trying to explain the internal working of some existing approaches.

3.1 Decomposed Approaches

Learning and Approximate Methods. Classically, we need approaches to construct interfaces that are valid, that is, interfaces with inductive invariants for correctness proofs and interfaces that are feasible and validating for violation proofs. But given existing checkers as explained above, we can use approximate methods to construct interfaces that are not guaranteed to be helpful for the proof construction. Since the interfaces can be checked, it is easy to refute them or prove that they are indeed useful. Also, such interfaces might

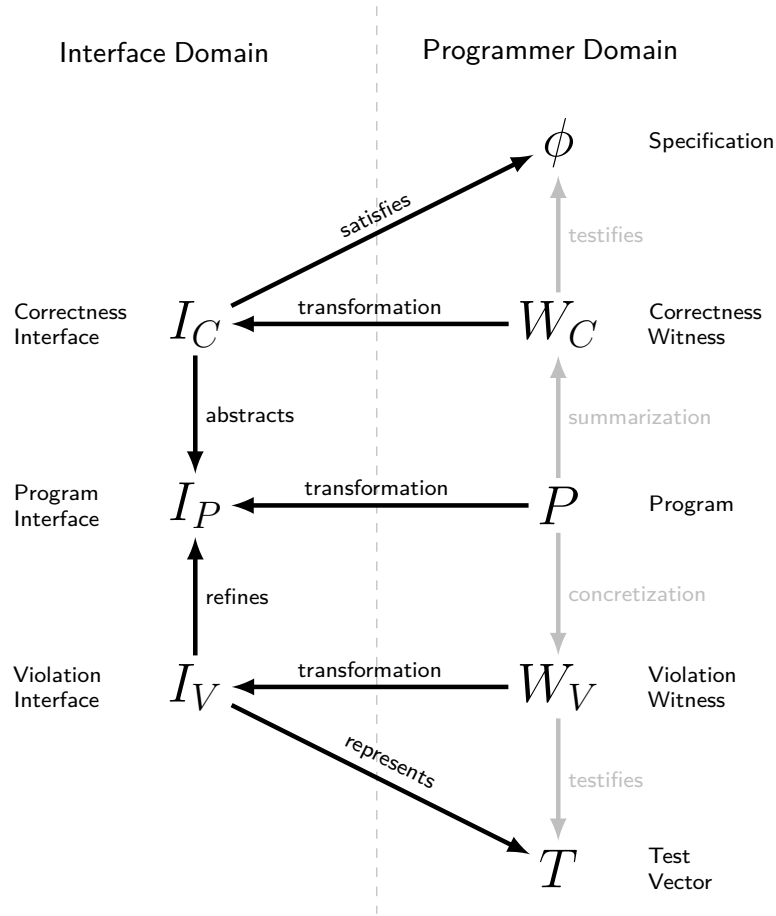


Fig. 10: Validation flows using the interface domain

be helpful to be further refined or abstracted to become more useful for the proof process. Furthermore, it might be interesting to come up with violation interfaces via learning-based testing [43].

Refiners. Besides the above-mentioned checkers, we can imagine tools that take an interface I_1 as input and refine (e.g., reduce) it in order to construct a new interface I_2 such that $I_2 \preceq I_1$. This idea is already used in the context of conditional model checking [18] (Reducers).

Abstracters. For the other direction, we can imagine tools that take an interface I_1 as input and abstract (e.g., extend, slice) it in order to construct a new interface I_2 such that $I_1 \preceq I_2$. This is an old but effective idea and used in program slicing [49].

Interactive Verification. The process of interactively constructing a proof in software verification using tools like Dafny [41], KeY [2], and Why3 [31] can be

seen through the interface lens as follows: The human defines the correctness interface, usually by injecting the invariants in the program source code using annotations, and the verifier checks the refinement and specification satisfaction.

Witness-Based Results Validation. A validator for verification results takes the correctness witness W_C and transforms it to the internal interface representation I_C , that is, the validator does not need to come up with I_C (and the contained invariants) but applies only a (syntactic) transformation. [Figure 10](#) tries to illustrate this flow. Then, the validator tries to prove $I_P \preceq I_C$ and $I_C \models \phi$. Symmetrically, for validating a violation result, the validator takes the violation witness W_V and transforms it to the internal interface representation I_V , which ideally describes an error path that it can easily replay and check for feasibility and violation, i.e., $I_V \preceq I_P$ and $I_V \not\models \phi$. Regarding multi-threaded programs, there is support for verification witnesses and their validation already [\[14\]](#).

k -Induction. There are verification approaches that consist of two engines, (a) an invariant-generator and (b) an inductiveness checker [\[12, 13, 38\]](#). The former constructs the most essential parts of the correctness interface I_C (the invariants, done in parallel in an isolated separate process), while the latter performs the checks $I_P \preceq I_C$ and $I_C \models \phi$, with ever increasing values for length k of the inductive-step.

3.2 Integrated Approaches

CEGAR — Explained using Interfaces. Counterexample-guided abstraction refinement (CEGAR) [\[27\]](#) is an approach that uses the following steps in a loop until a proof of either correctness or violation is constructed:

1. construct an abstract model I_a using a given precision
2. check $I_a \models \phi$; if it holds, terminate with answer (TRUE, W_C) (the interface I_a corresponds to an interface I_C in [Fig. 3](#), the correctness witness W_C in [Fig. 9](#) is an abstraction of I_C)
3. extract counterexample interface I_b from I_a (interface I_a corresponds to interface I_{-C} in [Fig. 3](#))
4. check $I_b \not\models \phi$; if it holds, terminate with answer (FALSE, W_V) (the interface I_b corresponds to an interface I_V in [Fig. 3](#), the violation witness W_V in [Fig. 9](#) is an abstraction of I_V)
5. extract new facts to refine the precision (derived from the infeasibility of I_b) and continue with step (1); (the interface I_b corresponds to an interface I_{-V} in [Fig. 3](#))

[Theorems 1](#) and [2](#) explain the correctness of CEGAR-based software model checking: The interfaces I_C and I_V can be used to prove the correctness and violation, respectively, using an internal specification checker and feasibility checker. Note that the feasibility checker in CEGAR is given by the above-described refinement checker (all refinements of the program interface I_P are

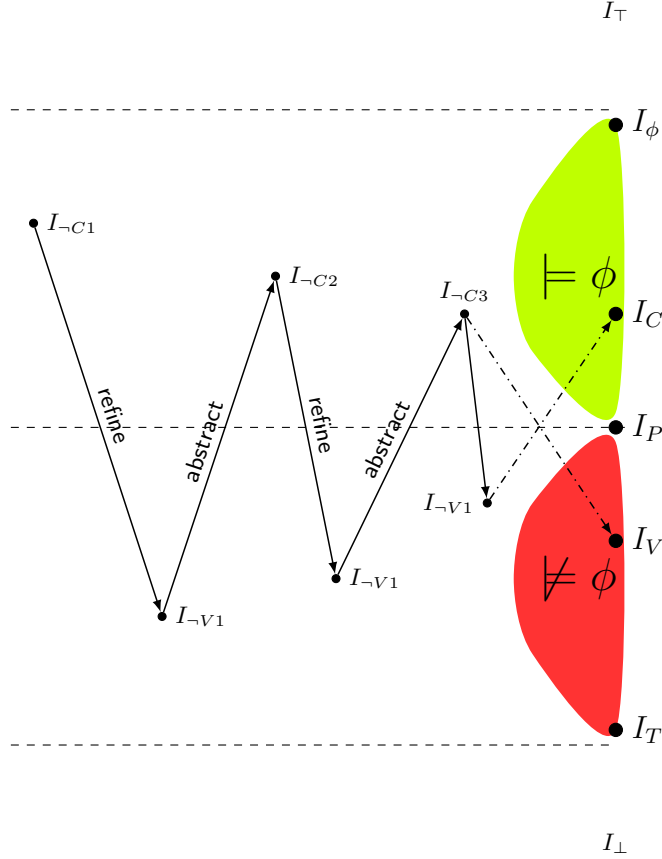


Fig. 11: Explaining CEGAR using interfaces

feasible, see Fig. 3). Figure 11 illustrates the alternation of the CEGAR loop between trying to prove correctness and trying to prove violation.

The resulting correctness interface I_C (in case of outcome `TRUE`) contains predicates describing inductive invariants (overapproximation of I_P), and the resulting violation interface I_V (in case of outcome `FALSE`) contains (at least one) feasible and violating path (underapproximation of I_P).

Test Generation. Theorem 2 explains the process of symbolic-execution-based test generation (as done, e.g., by KLEE [23]): The approaches leverage concretization mechanisms to construct a refined interface (constraints describing error paths, underapproximation) and the process must ensure feasibility, until a violating interface is found.

Explicit-State Model Checking. In some approaches to verification, the complete state space is exhaustively enumerated and checked [6, 19, 32, 36]. When proving correctness of a program, those approaches operate on the same level of

abstraction as the program itself, there is neither over- nor under-approximation. Thus, the most compact correctness interface used by such a verifier is the program interface I_P — these approaches cannot benefit from abstraction. However, when proving violation of a program, once an error path is encountered, the verifier can terminate the exploration and the partially explored state space can be seen as violation interface (which represents only a subset of all paths). Similar observations hold for SMT-based bounded model checking [22].

4 Conclusion

Software verification is a grand challenge of computer science [35]. Many powerful tools and approaches have been developed for program verification. Different approaches come with different strengths, and in order to join forces, we need to investigate ways to combine approaches. We are looking into possibilities to decompose a verification problem into smaller sub-problems in such a way that we can assign them to different tools (cooperative verification [20]). To achieve this, we extended the schema for proving correctness from refinement calculus by a symmetric schema for proving violation of program specifications. We hope that our interface-based viewpoint stimulates discussion on how we can achieve more modularity and decomposition in software verification. As future work, we plan to integrate compositional proofs into CoVeriTeam⁶ — a tool to compose verification actors.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (1986)
2. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The key tool. *Software and System Modeling* **4**(1), 32–54 (2005). <https://doi.org/10.1007/s10270-004-0058-x>
3. de Alfaro, L., Henzinger, T.A.: Interface automata. In: Proc. FSE. pp. 109–120. ACM (2001). <https://doi.org/10.1145/503271.503226>
4. de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Timed interfaces. In: Proc. EMSOFT, pp. 108–122. LNCS 2491, Springer (2002). https://doi.org/10.1007/3-540-45828-x_9
5. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with SLAM. *Commun. ACM* **54**(7), 68–76 (2011). <https://doi.org/10.1145/1965724.1965743>
6. Baranová, Z., Barnat, J., Kejstová, K., Kučera, T., Lauko, H., Mrázek, J., Ročkai, P., Štill, V.: Model checking of C and C++ with DIVINE 4. In: Proc. ATVA. pp. 201–207. LNCS 10482, Springer (2017). https://doi.org/10.1007/978-3-319-68167-2_14
7. Beckert, B., Hähnle, R.: Reasoning and verification: State of the art and current trends. *IEEE Intelligent Systems* **29**(1), 20–29 (2014). <https://doi.org/10.1109/MIS.2014.3>
8. Beyer, D., Chakrabarti, A., Henzinger, T.A.: Web service interfaces. In: Proc. WWW. pp. 148–159. ACM (2005). <https://doi.org/10.1145/1060745.1060770>

⁶ <https://gitlab.com/sosy-lab/software/coveriteam/>

9. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: Proc. ICSE. pp. 326–335. IEEE (2004). <https://doi.org/10.1109/ICSE.2004.1317455>
10. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE. pp. 326–337. ACM (2016). <https://doi.org/10.1145/2950290.2950351>
11. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE. pp. 721–733. ACM (2015). <https://doi.org/10.1145/2786805.2786867>
12. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Proc. CAV. pp. 622–640. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_42
13. Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. *J. Autom. Reasoning* **60**(3), 299–335 (2018). <https://doi.org/10.1007/s10817-017-9432-6>
14. Beyer, D., Friedberger, K.: Violation witnesses and result validation for multi-threaded programs. In: Proc. ISO LA. LNCS , Springer (2020)
15. Beyer, D., Gulwani, S., Schmidt, D.: Combining model checking and data-flow analysis. In: Handbook of Model Checking, pp. 493–540. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_16
16. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker BLAST. *Int. J. Softw. Tools Technol. Transfer* **9**(5-6), 505–525 (2007). <https://doi.org/10.1007/s10009-007-0044-z>
17. Beyer, D., Henzinger, T.A., Singh, V.: Algorithms for interface synthesis. In: Proc. CAV. pp. 4–19. LNCS 4590, Springer (2007). https://doi.org/10.1007/978-3-540-73368-3_4
18. Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Reducer-based construction of conditional verifiers. In: Proc. ICSE. pp. 1182–1193. ACM (2018). <https://doi.org/10.1145/3180155.3180259>
19. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Proc. FASE. pp. 146–162. LNCS 7793, Springer (2013). https://doi.org/10.1007/978-3-642-37057-1_11
20. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. In: Proc. ISO LA. LNCS , Springer (2020)
21. Beyer, D., Wendler, P.: Reuse of verification results: Conditional model checking, precision reuse, and verification witnesses. In: Proc. SPIN. pp. 1–17. LNCS 7976, Springer (2013). https://doi.org/10.1007/978-3-642-39176-7_1
22. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. TACAS. pp. 193–207. LNCS 1579, Springer (1999). https://doi.org/10.1007/3-540-49059-0_14
23. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. OSDI. pp. 209–224. USENIX Association (2008)
24. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O’Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: Proc. NFM. pp. 3–11. LNCS 9058, Springer (2015). https://doi.org/10.1007/978-3-319-17524-9_1
25. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Resource interfaces. In: Proc. EMSOFT. LNCS 2855, Springer (2003). https://doi.org/10.1007/978-3-540-45212-6_9

26. Church, A.: A note on the Entscheidungsproblem. *Journal of Symbolic Logic* **1**(1), 40–41 (1936). <https://doi.org/10.2307/2269326>
27. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* **50**(5), 752–794 (2003). <https://doi.org/10.1145/876638.876643>
28. Clarke, E.M., Grumberg, O., McMillan, K.L., Zhao, X.: Efficient generation of counterexamples and witnesses in symbolic model checking. In: *Proc. DAC*. pp. 427–432. ACM (1995). <https://doi.org/10.1145/217474.217565>
29. Cook, B.: Formal reasoning about the security of Amazon web services. In: *Proc. CAV (2)*. pp. 38–47. LNCS 10981, Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_3
30. Cousot, P.: On fixpoint/iteration/variant induction principles for proving total correctness of programs with denotational semantics. In: *Proc. LOPSTR 2019*. pp. 3–18. LNCS 12042, Springer (2020). https://doi.org/10.1007/978-3-030-45260-5_1
31. Filliâtre, J.C., Paskevich, A.: Why3: Where programs meet provers. In: *Programming Languages and Systems*. pp. 125–128. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_8
32. Havelund, K., Pressburger, T.: Model checking Java programs using Java PATHFINDER. *Int. J. Softw. Tools Technol. Transfer* **2**(4), 366–381 (2000)
33. Henzinger, T.A., Jhala, R., Majumdar, R.: Permissive interfaces. In: *Proc. FSE*. pp. 31–40. ACM (2005). <https://doi.org/10.1145/1095430.1081713>
34. Hoare, C.A.R.: An axiomatic basis for computer programming. *Commun. ACM* **12**(10), 576–580 (1969). <https://doi.org/10.1145/363235.363259>
35. Hoare, C.A.R.: The verifying compiler: A grand challenge for computing research. *J. ACM* **50**(1), 63–69 (2003)
36. Holzmann, G.J.: The SPIN model checker. *IEEE Trans. Softw. Eng.* **23**(5), 279–295 (1997)
37. Jhala, R., Majumdar, R.: Software model checking. *ACM Computing Surveys* **41**(4) (2009). <https://doi.org/10.1145/1592434.1592438>
38. Kahsai, T., Tinelli, C.: PKIND: A parallel k-induction based model checker. In: *Proc. Int. Workshop on Parallel and Distributed Methods in Verification*. pp. 55–62. EPTCS 72 (2011). <https://doi.org/10.4204/EPTCS.72.6>
39. Khoroshilov, A.V., Mutilin, V.S., Petrenko, A.K., Zakharov, V.: Establishing Linux driver verification process. In: *Proc. Ershov Memorial Conference*. pp. 165–176. LNCS 5947, Springer (2009). https://doi.org/10.1007/978-3-642-11486-1_14
40. Kildall, G.A.: A unified approach to global program optimization. In: *Proc. POPL*. pp. 194–206. ACM (1973). <https://doi.org/10.1145/512927.512945>
41. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: *Proc. LPAR*. pp. 348–370. LNCS 6355, Springer (2010). https://doi.org/10.1007/978-3-642-17511-4_20
42. McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. *Computer Science Review* **5**(2), 119–161 (2011). <https://doi.org/10.1016/j.cosrev.2010.09.009>
43. Meinke, K.: Learning-based testing: Recent progress and future prospects. In: *Machine Learning for Dynamic Software Analysis: Potentials and Limits*. pp. 53–73. LNCS 11026, Springer (2018). https://doi.org/10.1007/978-3-319-96562-8_2
44. Morris, J.M.: A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comput. Program.* **9**(3), 287–306 (1987). [https://doi.org/10.1016/0167-6423\(87\)90011-6](https://doi.org/10.1016/0167-6423(87)90011-6)
45. O’Hearn, P.W.: Incorrectness logic. *Proc. ACM Program. Lang.* **4**(POPL) (2020). <https://doi.org/10.1145/3371078>

46. Piterman, N., Pnueli, A.: Temporal logic and fair discrete systems. In: Handbook of Model Checking, pp. 27–73. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_2
47. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. **3**(1), 30–50 (2000). <https://doi.org/10.1145/353323.353382>
48. Turing, A.: On computable numbers, with an application to the Entscheidungsproblem. In: Proc. LMS. vol. s2-42, pp. 230–265. London Mathematical Society (1937). <https://doi.org/10.1112/plms/s2-42.1.230>
49. Weiser, M.: Program slicing. IEEE Trans. Softw. Eng. **10**(4), 352–357 (1984). <https://doi.org/10.1109/tse.1984.5010248>
50. Wetzler, N., Heule, M.J.H., Jr., W.A.H.: DRAT-TRIM: Efficient checking and trimming using expressive clausal proofs. In: Proc. SAT. pp. 422–429. LNCS 8561, Springer (2014). https://doi.org/10.1007/978-3-319-09284-3_31