International Journal on Software Tools for Technology Transfer (2020) 22:135–161
https://doi.org/10.1007/s10009-019-00547-x

SPIN 2018

IC3 software model checking

Tim Lange[1] · Martin R. Neuhäußer[2] · Thomas Noll[1] · Joost-Pieter Katoen[1]

Published online: 5 November 2019

Abstract
In recent years, the inductive, incremental verification algorithm IC3 had a major impact on hardware model checking. Also for software model checking, a number of adaptations of Boolean IC3 and combinations with CEGAR and ART-based techniques have been developed. However, most of them exploit the peculiarities of software programs, such as the explicit representation of control flow, only to a limited extent. In this paper, we present an approach that supports this explicit representation in the form of control-flow automata, and integrates it with symbolic reasoning about the data state space of the program. By maintaining reachability information specifically for each control location, we arrive at a "two-dimensional" extension of IC3, which provides a true lifting from hardware to software model checking. Moreover, we address the problem of generalization in this setting, an essential feature to ensure the scalability of IC3. We introduce several improvements that range from efficient caching of generalizations over variable reductions to syntax-oriented generalization by means of weakest preconditions. Using a prototypical implementation, we evaluate our approach on a number of case studies, including a significant subset of the SV-COMP 2018 benchmarks, and compare the outcomes with results obtained from other IC3 software model checkers.

Keywords Program verification · Safety properties · Software model checking · IC3

1 Introduction

IC3 [10] is one of the most prominent state-of-the-art model-checking algorithms designed for bit-level verification of hardware systems represented as finite-state transition systems. The IC3 algorithm attempts to prove the correctness of an invariant property by iteratively deriving overapproximations of the reachable state space until a fixpoint is reached. Its impressive impact on the model-checking community and its superior performance are based on two important aspects: In contrast to most other (bounded) model-checking algorithms, IC3 does not rely on unrolling the transition relation, but instead generates clauses that are inductive relative to stepwise reachability information. In addition, IC3 applies aggressive abstraction to the explored state space by so-called generalization.

There have been several attempts to lift Boolean IC3 to the domain of software model checking. As this setting usually induces infinite-state systems, more advanced symbolic reasoning techniques are required. The most prominent one is satisfiability modulo theories (SMT). Here, sets of states are symbolically specified by first-order formulae over constraints from the respective theory, and SMT-solving techniques are employed to rule out spurious counterexamples.

One of the first attempts to integrate SMT into IC3 has been presented in [14]. In addition to the generalization of SAT to SMT solving, it exploits the partitioning of the program's state space as induced by its control-flow graph. This is achieved by unwinding the latter into an abstract reachability tree (ART) in which each node is associated with a control location and a formula, resulting in an "explicit-symbolic" approach named Tree-IC3. Candidate counterexamples are handled by computing under-approximations of pre-images. The advantages in comparison with Boolean IC3 are twofold. First, Tree-IC3 eliminates the possible redundancy of subformulae that can be present at frames where the corresponding

✉ Joost-Pieter Katoen
katoen@cs.rwth-aachen.de

Tim Lange
tim.lange@cs.rwth-aachen.de

Martin R. Neuhäußer
martin.neuhaeusser@siemens.com

Thomas Noll
noll@cs.rwth-aachen.de

1 RWTH Aachen University, Aachen, Germany

2 Siemens AG, Munich, Germany

location cannot be reached. Second, due to the disjunctive partitioning of the control state space, the solver is exposed to simpler and smaller formulae.

On the downside, the key idea underlying IC3, relative inductiveness, cannot be directly applied in this setting due to the partitioned representation that leads to a path-wise unwinding of the transition system. In earlier work [32], we have therefore developed another adaptation of IC3 to software model checking, called IC3CFA. Here, the control flow of a program is explicitly represented as a control-flow automaton (CFA), while its data space is handled symbolically to support inductive reasoning. Thus, our computation model is similar to the one underlying the abstract interpretation framework [18], which also combines the explicit handling of control flow with abstract representation of data. By adding the control locations as another dimension to the reachability analysis of IC3, this hybrid structure provides a true lifting of IC3 from hardware to software model checking. While IC3CFA was originally developed for reasoning about programmable logic controller code, it can successfully be applied to the verification of C programs, too.

As the author of the original IC3 algorithm [10] states, “one of the key components of IC3 is inductive generalization” [25]. Even though IC3 is sound and complete without this feature, by explicitly enumerating models in a finite state space, it largely depends on its ability to abstract from specific states in order to scale to huge or even infinite state spaces. While literal elimination on the Boolean skeleton can be applied to IC3 in the first-order setting [14], the situation in IC3CFA is more involved as one has to deal with both the underlying SMT theory and the explicit representation of control flow. Based on the results described in [33], we present some small and elegant modifications of the generalization procedure of IC3CFA that significantly improve its overall performance. Our techniques are generic in that they are independent of the underlying theory, and some of them are even applicable to IC3 in general as they support existing generalization methods, such as unsatisfiable cores [10], obligation ordering [23], and interpolation [8].

In summary, this paper gives a uniform representation of IC3 software model checking that combines the advantage of explicitly handling the control flow of a program, employing a corresponding automata model, with relative inductive reasoning over a symbolic representation of its data space. We start by introducing some general concepts in Sect. 2. Next, Sect. 3 presents the ideas underlying the IC3 approach. Sect. 4 describes the extension of the original IC3 algorithm by control-flow automata. Section 5 introduces our concepts for generalization. Results of the experimental evaluation based on the implementation described in Sect. 6 are given in Sect. 7. Finally, Sect. 8 discusses related work, and Sect. 9 concludes the paper with a summary and a description of future work.

2 Preliminaries

In order to enable the application of our verification framework to different concrete settings, we parametrize it by various syntactic categories. This gives us the freedom to dynamically configure it with first-order theories on demand. On the lowest level, we have the set of (program) variables, X. From these, expressions in $E(X)$ can be built by applying (arithmetic) operations to variables and constants. Finally, (first-order) formulae in $\mathcal{F}(X)$ are constructed by applying predicates (such as relational operations) to expressions and combining the resulting terms using Boolean operations.

To model our input program, we use the standard guarded command language (GCL) [19,20]:

Definition 1 *(Guarded command language)* The syntax of the guarded command language (GCL) is given by

$$\mathsf{cmd} ::= x := e \,|\, \mathsf{assume}\ b \mid \mathsf{cmd}_1;\ \mathsf{cmd}_2 \mid \mathsf{cmd}_1 \,\Box\, \mathsf{cmd}_2,$$

where $x \in X$ is a variable, $e \in E(X)$ is an expression, and $b \in \mathcal{F}(X)$ is a formula (Boolean expression). The set of all GCL commands is denoted by GCL.

Thus, commands can be of one of the four following types. An assignment $x := e$ changes the value of x to the value of expression e. The assume statement $\mathsf{assume}\ b$ behaves like a skip operation if b evaluates to *true* but blocks execution in case the guard evaluates to *false*. To compose a program from several commands, the sequencing construct $\mathsf{cmd}_1;\ \mathsf{cmd}_2$ can be used. In order to support branching, GCL features the choice command $\mathsf{cmd}_1 \,\Box\, \mathsf{cmd}_2$ that takes two arbitrary commands and non-deterministically executes one of them. Deterministic branching (as in standard if-then-else constructs) can be implemented by combining choice and assume commands, guarding both branches by disjoint Boolean expressions. Note that this simple languages does not feature advanced control-flow features such as while loops. They will be introduced later in the form of control-flow automata (cf. Def. 4).

A formal interpretation of GCL commands is given by the following operational semantics.

Definition 2 *(GCL semantics [20])* A (program) state σ is a function that maps every variable to some value in the underlying domain. The set of all states is denoted by Σ. The execution relation, $\rightarrow \subseteq (GCL \times \Sigma) \times \Sigma$, is represented by triples of the form $\langle \mathsf{cmd}, \sigma \rangle \rightarrow \sigma'$ which are derived using the following operational rules. Here, $\sigma(e)$ denotes the value of expression e in state σ, and $\sigma[x \mapsto \sigma(e)]$ updates σ by setting x to this value.

$$\text{(assign)}\frac{}{\langle x := e, \sigma\rangle \rightarrow \sigma[x \mapsto \sigma(e)]}$$

$$\text{(assume)}\frac{\sigma(b) = true}{\langle \mathsf{assume}\ b, \sigma\rangle \rightarrow \sigma}$$

$$\text{(seq)}\frac{\langle \mathsf{cmd}_1, \sigma\rangle \rightarrow \sigma' \langle \mathsf{cmd}_2, \sigma'\rangle \rightarrow \sigma''}{\langle \mathsf{cmd}_1;\ \mathsf{cmd}_2, \sigma\rangle \rightarrow \sigma''}$$

$$\text{(choice1)}\frac{\langle \mathsf{cmd}_1, \sigma\rangle \rightarrow \sigma'}{\langle \mathsf{cmd}_1 \,\square\, \mathsf{cmd}_2, \sigma\rangle \rightarrow \sigma'}$$

$$\text{(choice2)}\frac{\langle \mathsf{cmd}_2, \sigma\rangle \rightarrow \sigma'}{\langle \mathsf{cmd}_1 \,\square\, \mathsf{cmd}_2, \sigma\rangle \rightarrow \sigma'}$$

Given this operational semantics, we can investigate the effect of GCL commands not only on single states, but on sets characterized by arbitrary FO formulae [20]. As we will see later, our verification framework requires to do this in a backward fashion. We therefore employ the concept of weakest preconditions to capture all states that are mapped by a given command cmd to a state that satisfies a given postcondition φ.

Note that GCL does not feature loops, implying that the execution of a program necessarily terminates in a state or gets stuck in an invalid assume command. However, due to the presence of the choice command, we have to deal with nondeterminism. Here, it is important to observe that the postcondition φ will later be used to represent "bad" states, that is, the violation of a (safety) condition. We are therefore interested in every initial state which ensures the postcondition under *some* possible execution. This is accomplished by the following variant of the weakest precondition that guarantees this existential property, and that moreover ignores executions that get stuck, which is consistent with the usage of the assume command as a "filter" operation on states.

To formalize this notion, we use the notation $\sigma \models \varphi$ to indicate that a state σ satisfies a formula φ, and write $\models \varphi$ (often omitting the $\models$ symbol) if this applies to every σ.

Definition 3 *(Weakest existential precondition)* Given $\mathsf{cmd} \in GCL$ and $\varphi \in \mathcal{F}(X)$, the weakest existential precondition (WEP) of φ with respect to cmd, denoted by $wep(\mathsf{cmd}, \varphi)$, is given by

$$wep(\mathsf{cmd}, \varphi) = \{\sigma \in \Sigma \mid \exists \sigma' \in \Sigma.\langle \mathsf{cmd}, \sigma\rangle \rightarrow \sigma' \wedge \sigma' \models \varphi\}.$$

Note that this notion is related to the concept of weakest *liberal* preconditions (wlp), as introduced in [20], in the following way:

$$wep(\mathsf{cmd}, \varphi) \equiv \neg wlp(\mathsf{cmd}, \neg\varphi).$$

Using induction on the structure of commands, it is straightforward to show that syntactic representations of WEPs can be obtained by simple syntactic transformations as follows.

Lemma 1 **(WEP transformers)** *The WEP of a formula $\varphi \in \mathcal{F}(X)$ can be represented by:*

$$\begin{aligned} wep(x := e, \varphi) &= \varphi[x \mapsto e] \\ wep(\mathsf{assume}\ b, \varphi) &= \varphi \wedge b \\ wep(\mathsf{cmd}_1;\ \mathsf{cmd}_2, \varphi) &= wep(\mathsf{cmd}_1, wep(\mathsf{cmd}_2, \varphi)) \\ wep(\mathsf{cmd}_1 \,\square\, \mathsf{cmd}_2, \varphi) &= wep(\mathsf{cmd}_1, \varphi) \vee wep(\mathsf{cmd}_2, \varphi) \end{aligned}$$

where $\varphi[x \mapsto e]$ denotes the FO-formula obtained by replacing all free occurrences of x in φ by e.

While predicate transformers offer a simple and elegant way to map postconditions to corresponding preconditions, the size of the resulting predicate can become exponential in the size of the command. As the size of a query can influence the runtime of the SMT solver, the actual verification time may increase substantially. However, this phenomenon was not observed in the experimental evaluation (see Sect. 7).

To represent full control flow, e.g., loops, we leverage the expressiveness by wrapping our GCL with a so-called control-flow automaton [17]. It differs from the common definition of control-flow graphs [36] by associating program instructions with edges rather than nodes, which is more appropriate in our setting.

Definition 4 *(Control-flow automaton)* A control-flow automaton (CFA) is a tuple $\mathcal{A} = (L, G, \ell_0, \ell_E)$ consisting of finite sets of locations L and edges $G \subseteq L \times GCL \times L$ labeled with GCL commands, an initial location $\ell_0 \in L$, and an error location $\ell_E \in L$ which does not have any G-successor.

Moreover, by $out : L \rightarrow 2^G$ and $succ/pred : L \rightarrow 2^L$ we, respectively, denote the outgoing edges and successor/predecessor locations of a location $\ell \in L$, that is,

$$\begin{aligned} out(\ell) &= \{g \in G \mid \exists \mathsf{cmd} \in GCL, \ell' \in L.g = (\ell, \mathsf{cmd}, \ell')\} \\ succ(\ell) &= \{\ell' \in L \mid \exists \mathsf{cmd} \in GCL.(\ell, \mathsf{cmd}, \ell') \in G\} \\ pred(\ell) &= \{\ell' \in L \mid \exists \mathsf{cmd} \in GCL.(\ell', \mathsf{cmd}, \ell) \in G\}. \end{aligned}$$

By introducing control-flow automata, the guarded command language and first-order formulae, we have established three different layers of separation of concerns to abstract from different aspects of the concrete input program. On the top-most level, the CFA represents the control-flow structure of the input program by showing the program paths that can be chosen during execution. The guarded commands that label the CFA's edges form the central tier and distinguish whether taking an edge changes a variable assignment or conditionally manipulates the control flow based on the current variable assignment, namely by means of assume statements. This way, the guarded command language connects the top-level CFA with the low-level formula that gives the semantics of the expressions used in the guarded commands.

This gives us the freedom to dynamically configure our verification framework with first-order theories on demand. For example, for programs with integers only, we use bit-vector theory for bit-precise analysis or linear integer arithmetic if this level of precision is not necessary.

3 Preliminaries on IC3

This section introduces the main definitions and concepts of the class of IC3 algorithms [10,12] that will be lifted in Sect. 4 to operate on control-flow automata. As a common semantic basis, these algorithms use (symbolic representations of) transition systems, which are structures of the form $\mathcal{M} = (X, I, T)$. Here, $I \in \mathcal{F}(X)$ specifies the initial states and $T \in \mathcal{F}(X \cup X')$ the transition relation where $X' = \{x' \mid x \in X\}$ represents the updated values of variables after taking a transition. (For example, $x' = x + 1$ describes an increment operation on variable $x \in X$.) Moreover, for $e \in E(X)$ and $\varphi \in \mathcal{F}(X)$, e' and φ' denote the result of substituting each free variable by its primed version.

A *cube c* over X is defined as a conjunction of *literals*, each literal being a variable or its negation in the propositional case and a theory atom or its negation in the quantifier-free first-order case. The negation of a cube, i.e., a disjunction of literals, is called a *clause*, and usually denoted by d. A *frame F* is defined as a conjunction of clauses. For simplicity, we will often refer to cubes and frames as sets of implicitly conjoined literals and clauses, respectively.

Definition 5 *(Inductive formula)* Given a transition system $\mathcal{M} = (X, I, T)$, a formula $\varphi \in \mathcal{F}(X)$ is called inductive in $\mathcal{M}$ if $I \Rightarrow \varphi$ and $\varphi \wedge T \Rightarrow \varphi'$.

Any formula that is inductive in $\mathcal{M}$ subsumes $\mathcal{M}$'s reachable states. Moreover, proving that φ is an invariant in $\mathcal{M}$ (denoted $\mathcal{M} \models \varphi$) is equivalent to showing that its negation (i.e., the set of bad states) is unreachable from any of $\mathcal{M}$'s initial states.

Corollary 1 *Every inductive formula is an invariant.*

The reverse, however, does not hold in general: Most invariants are not inductive. This is crucial for IC3-like algorithms which aim at proving that a formula φ is an invariant in $\mathcal{M}$ by constructing a sequence of strengthenings of φ that converges to an inductive formula or fails when encountering a counterexample.

Definition 6 (*Frame sequence* [10]) Let $\mathcal{M}$ be a transition system, $\varphi \in \mathcal{F}(X)$ a formula. A sequence of formulae $F_0, F_1, \ldots, F_k \in \mathcal{F}(X)$ is a frame sequence for φ if

$$I \Rightarrow F_0, \tag{1}$$

$$\forall i < k. \qquad F_i \Rightarrow F_{i+1}, \tag{2}$$

$$\forall i \le k. \qquad F_i \Rightarrow \varphi, \text{ and} \tag{3}$$

$$\forall i < k. \qquad F_i \wedge T \Rightarrow F'_{i+1}. \tag{4}$$

IC3 terminates successfully once it has constructed a frame sequence $F_0, F_1, \ldots, F_k$ such that $F_{k-1} \equiv F_k$. For finite-state systems, this is guaranteed to eventually happen due to the monotonicity property as expressed by Eq. (2). In that case, IC3 has proven that φ is an invariant in $\mathcal{M}$ by constructing the inductive strengthening F_k of φ:

Corollary 2 (Soundness of IC3) *Let $\mathcal{M}$ be a transition system, $\varphi \in \mathcal{F}(X)$ a formula, and $F_0, \ldots, F_{k-1}, F_k$ a frame sequence for φ where $F_{k-1} \equiv F_k$. Then φ is an invariant in $\mathcal{M}$.*

Proof By Eq. (1), $I \Rightarrow F_0$ and by Eq. (2), $F_i \Rightarrow F_{i+1}$ for all $i < k$; hence, $I \Rightarrow F_k$. By Eq. (4) we have that $F_{k-1} \wedge T \Rightarrow F'_k$. From our assumption that $F_{k-1} \equiv F_k$, we conclude that $F_k \wedge T \Rightarrow F'_k$. Thus F_k is an inductive formula, i.e., it subsumes all reachable states in $\mathcal{M}$. Finally, $F_k \Rightarrow \varphi$ holds by Eq. (3); hence F_k is a strengthening of φ. Therefore φ is an invariant. □

Definition 7 *(Relative inductivity)* Let $\mathcal{M}$ be a transition system and $\varphi, \psi \in \mathcal{F}(X)$ be formulae. Then φ is inductive relative to ψ if $I \Rightarrow \varphi$ and $\varphi \wedge \psi \wedge T \rightarrow \varphi'$ holds.

4 IC3CFA

This section presents the search phase of our *IC3CFA* algorithm for IC3-style software model checking. It is applied to a CFA without unrolling the transition relation. We first introduce the main idea underlying IC3CFA and later give a pseudocode sketch with annotated pre- and postconditions. We will present the algorithm in a way that is similar to [10] and show that the proof of correctness is mostly analogous. Afterward we will review the relative inductivity query [10]:

$$F_{i-1} \wedge \varphi \wedge T \Rightarrow \varphi' \tag{5}$$

$$I \Rightarrow \varphi \tag{6}$$

and present a relaxation of it.

4.1 Basic concepts

In order to apply IC3 to a CFA, let us reconsider IC3-SMT [14]. While not the best performing algorithm, its conceptual simplicity supports formal analysis, and its correctness directly follows from the correctness of IC3. Given the state space of IC3-SMT, extracting the control-flow variable *pc* into a CFA with $n + 1$ locations $L = \{\ell_0, \ldots, \ell_n\}$ yields a partitioning of $n+1$ disjoint state spaces S_ℓ that are induced by the program variables in X, where each S_ℓ is implicitly conditioned by $pc = \ell$. The program counter is initially ℓ_0 and is incremented after each (non-branching) instruction.

In order to formalize this, we define *data regions* and *regions* along the lines of [26].

Definition 8 *(Data region)* A data region represented by a quantifier-free FO-formula φ over X consists of all variable assignments σ satisfying φ, i.e., $\{\sigma \in \Sigma \mid \sigma \models \varphi\}$.

Following Def. 8, we can now reason about sets of states that are defined only by constraints on the values of program variables. In order to also incorporate the program location, we define *regions*, in [26] also referred to as *atomic regions*.

Definition 9 *(Region)* A region $r = (\ell, \varphi)$ consists of a location $\ell \in L$ and a data region φ, and is represented by $\{\psi \mid \psi \equiv (pc = \ell \wedge \varphi)\}$. Analogously, $\neg r$ corresponds to $\{\psi \mid \psi \equiv \neg(pc = \ell \wedge \varphi)\}$.

Assume an IC3-SMT transition $pc = \ell \wedge \psi \wedge pc' = \ell'$ with some predicate transformer $\psi(X, X')$ transferring variable assignment σ to successor assignment σ'. This transformer will be the label of the edge from location ℓ to ℓ', i.e., $(\ell, \psi, \ell') \in G$[1] in CFA $\mathcal{A} = (L, G, \ell_0, \ell_E)$. For convenience, we define local transition formulae that map to the corresponding transition formula whenever such an edge exists.

Definition 10 *(Local transition formula)* The *transition formula* between ℓ and ℓ' is defined by:

$$T_{\ell \to \ell'} = \begin{cases} \psi & \text{, if } (\ell, \psi, \ell') \in G \\ \mathit{false} & \text{, otherwise.} \end{cases} \tag{7}$$

We define the global transition formula as follows.

Definition 11 *(Global transition formula)* The global transition relation T of CFA $\mathcal{A}$ is defined as:

$$T = \bigvee_{(\ell,\psi,\ell') \in G} \psi. \tag{8}$$

Example 1 Consider the example C program from Fig. 1. A corresponding CFA $\mathcal{A}$ is depicted in Fig. 2. It induces local transition formulae such as:

$$T_{\ell_3 \to \ell_4} = (y' = b),$$
$$T_{\ell_4 \to \ell_5} = (a < 0), \textit{ and}$$
$$T_{\ell_3 \to \ell_5} = \mathit{false}.$$

[1] Note that, while CFA edges are defined over $L \times GCL \times L$, either the edge label cmd or its corresponding formula representation $\psi(X, X')$ will be used where appropriate. For analogy to IC3-SMT, $\psi(X, X')$ is more appropriate, whereas for WEP computations we use cmd.

```
#include "assert.h"

int main() {
    signed char a = __VERIFIER_nondet_char();
    signed char b = __VERIFIER_nondet_char();
    signed char y, t;
    y = b;
    if (a < (signed char)0) {
        a = -a; }
    if (b < (signed char)0) {
        b = -b; }
    while (b != (signed char)0) {
        t = b;
        b = a % b;
        a = t; }
    if (y > (signed char)0) {
        assert(y >= a); }
    return a; }
```

Fig. 1 Running example C program

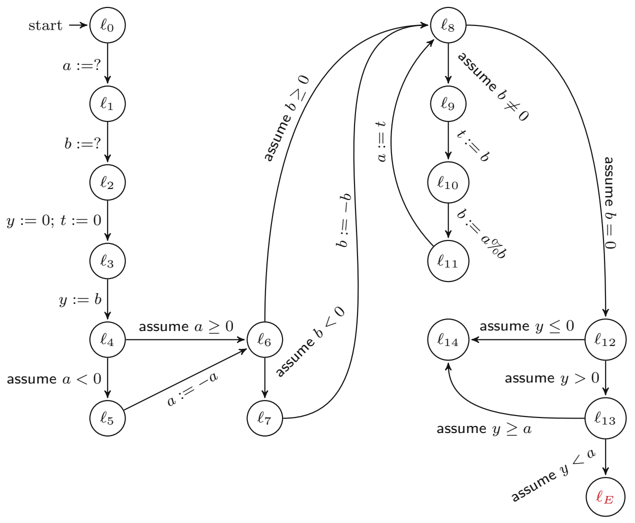


Fig. 2 CFA of our sample C program

The explicit handling of control flow enables us to verify safety properties by simply checking whether the dedicated error location ℓ_E as introduced in Def. 4 is reachable. More formally, such properties can simply be represented by the formula $P = (pc \neq \ell_E)$, as in [14].

In the remainder of this paper, we assume that an uninitialized variable has an arbitrary value from its domain. The initial state representation is thus $I = (pc = \ell_0)$. Just like for the error location, the extraction of control flow yields the symbolic representation $\bigwedge \emptyset = \mathit{true}$ for the initial location ℓ_0 of CFA $\mathcal{A}$ and *false* for all locations $\ell \in L \setminus \{\ell_0\}$.

Based on the partitioning of the state space by locations, we split the frame sequence $F_0, \ldots, F_k$ into a disjunctive set of frame sequences

$$\{F_{(0,\ell_0)}, \ldots, F_{(k,\ell_0)}; \ldots; F_{(0,\ell_n)}, \ldots, F_{(k,\ell_n)}\},$$

one for each location $\ell_i \in L$. Thus, entry $F_{(i,\ell)}$ characterizes the states in the variable's state space that are reachable in location ℓ in at most i steps. Note that we follow IC3 by not creating frames for the error location ℓ_E, just like IC3 does not include $\neg P$-states in any F_i.

As a consequence of *location-local* frames, we have to adapt the IC3 invariants (1)-(4) slightly: Given a CFA $\mathcal{A}$, for all $\ell \in L \setminus \{\ell_E\}$ it holds:

$$true \Rightarrow F_{(0,\ell_0)}, \qquad (9)$$
$$F_{(i,\ell)} \Rightarrow F_{(i+1,\ell)}, \qquad \forall i < k \qquad (10)$$
$$F_{(i,\ell)} \Rightarrow true, \qquad \forall i \le k \qquad (11)$$
$$F_{(i,\ell)} \wedge T_{\ell\to\ell'} \Rightarrow F'_{(i+1,\ell')}, \qquad \forall i < k. \qquad (12)$$

As we can see, the invariants (9) and (11) become fairly simple, due to the explicit handling of control-flow locations. Our algorithm will initialize $F_{(0,\ell_0)}$ to *true* and all other $F_{(0,\ell)}$ to *false* to cover (9). Similarly, we initialize every frame $F_{(i,\ell)}$ with $i \ge 1$ by *true*.

Lemma 2 *If, for some $0 \le i < k$ and all $\ell \in L \setminus \{\ell_E\}$, $F_{(i,\ell)} = F_{(i+1,\ell)}$, then $\bigwedge_{\ell \in L\setminus\{\ell_E\}} F_{(i,\ell)}$ is an inductive strengthening.*

Proof We reuse the corresponding result for IC3 given in Cor. 2 by showing the equivalence of the termination conditions for IC3 and IC3CFA. Assume that for some i in IC3, $F_i = F_{i+1}$. Let S be the set of all possible states of the program. We partition S into disjoint sets S_ℓ such that $s \in S_\ell$ iff $(pc = ` \wedge s \in S)$.

We start by considering the termination criterion of IC3: $F_i = F_{i+1}$. In order to prove the equivalence, we also need to take into account the invariant $F_i \wedge T \Rightarrow F'_{i+1}$, since both conditions are required for inductivity. Next, we partition the state space into disjoint state, each indicating a different pc value. The partitioning of the state space obviously implies that frames which specify different pc values have to be split, too. By definition, a location can only have a fixed set of successors, such that we do not have to consider $F_{(i+1,\ell')}$ for all $\ell' \in L$, but only for the subset $\{F_{(i+1,\ell')} \mid \ell' \in succ(\ell)\}$ of successors of the current location ℓ. In the last step, we can remove the successor condition, since it is ensured by invariant (12). Thus we arrive at the following sequence of equivalences:

$$\exists F_i, F_{i+1} \subseteq S.\ F_i = F_{i+1}$$
$$\overset{(4)}{\Longleftrightarrow} \exists F_i, F_{i+1} \subseteq S.\ F_i = F_{i+1}$$
$$\wedge\ (\forall s \in F_i.\ succ(s) \in F_{i+1})$$
$$\Longleftrightarrow \forall \ell \in L \setminus \{\ell_E\}\ \exists F_{(i,\ell)}, F_{(i+1,\ell)} \subseteq S_\ell.$$
$$F_{(i,\ell)} = F_{(i+1,\ell)}$$
$$\wedge \left(\forall f \in F_{(i,\ell)}.\ succ(f) \in \bigcup_{\ell' \in succ(\ell)} F_{(i+1,\ell')} \right)$$
$$\overset{(12)}{\Longleftrightarrow} \forall \ell \in L \setminus \{\ell_E\}\ \exists F_{(i,\ell)}, F_{(i+1,\ell)} \subseteq S_\ell.F_{(i,\ell)}$$
$$= F_{(i+1,\ell)}.$$
□

The next step is to define a notion of *inductivity relative* to some set of states. Just like for general inductivity, the extraction of control flow allows us to exploit the control structure of the program in order to consider only small parts of arbitrarily large and complex programs for a single step of the transition relation. We refer to inductivity relative to some state set along the edge $T_{\ell_1\to\ell_2}$ as *edge-relative inductivity*.

Definition 12 *(Edge-relative inductivity for formulae)* Given a CFA $\mathcal{A}$ and locations $\ell, \ell' \in L$, a formula φ is edge-relative inductive to formula ρ if:

$$\rho \wedge \varphi \wedge T_{\ell\to\ell'} \implies \varphi' \qquad (13)$$

We can omit the implication $I \Rightarrow \varphi$ (cf. Def. 7) here, since the implication is trivially valid in IC3CFA for all locations ℓ except for ℓ_0, for which the algorithm terminates with a counterexample and inductivity is not checked.

Note that edge-relative inductivity does also hold for any ρ if $(\ell, \varphi, \ell') \notin G$. Following [14], we will slightly modify relative inductivity by tailoring it to the setting of IC3: inductivity of a negated formula $\neg\varphi$ relative to a non-negated formula ρ. In analogy, we will reason about inductivity of a negated region r_2 relative to a non-negated region r_1.

Lemma 3 **(Relative inductive regions)** *Edge-relative inductivity of $\neg r_2 = \neg(\ell_2, \varphi_2)$ to $r_1 = (\ell_1, \varphi_1)$ is equivalent to:*

$$\varphi_1 \wedge T_{\ell_1\to\ell_2} \Rightarrow \neg\varphi'_2,\ \textit{if}\ \ell_2 \ne \ell_1 \qquad (14)$$
$$\varphi_1 \wedge \neg\varphi_2 \wedge T_{\ell_1\to\ell_2} \Rightarrow \neg\varphi'_2,\ \textit{if}\ \ell_2 = \ell_1. \qquad (15)$$

Proof Consider the extended transition formula

$$\hat{T}_{\ell_1\to\ell_2} := \left(pc = \ell_1 \wedge T_{\ell_1\to\ell_2} \wedge pc' = \ell_2\right)$$

and replace all occurrences of $T_{\ell_1\to\ell_2}$ in Lem. 3 with $\hat{T}_{\ell_1\to\ell_2}$. This is a valid substitution, since it preserves validity of the implications in Lem. 3 and only adds additional, explicit assignments to pc and pc'. For regions $r_1 = (\ell_1, \varphi_1)$ and $r_2 = (\ell_2, \varphi_2)$, let

$$\psi_1 \equiv (pc = \ell_1 \wedge \varphi_1) \text{ and } \neg\psi_2 \equiv \neg(pc = \ell_2 \wedge \varphi_2)$$

be the corresponding (negated) formulae. Definition 12 yields:

$$(pc = \ell_1 \wedge \varphi_1) \wedge \neg(pc = \ell_2 \wedge \varphi_2) \wedge \hat{T}_{\ell_1\to\ell_2}$$
$$\Rightarrow \neg(pc' = \ell_2 \wedge \varphi'_2)$$
$$\equiv (pc = \ell_1 \wedge \varphi_1) \wedge (pc \ne \ell_2 \vee \neg\varphi_2) \wedge \hat{T}_{\ell_1\to\ell_2}$$
$$\Rightarrow (pc' \ne \ell_2 \vee \neg\varphi'_2).$$

Since $pc = \ell_1$ and $pc' = \ell_2$, for $\ell_1 \neq \ell_2$ this is equisatisfiable to:

$$(true \wedge \varphi_1) \wedge (true \vee \neg\varphi_2) \wedge \hat{T}_{\ell_1 \to \ell_2} \Rightarrow (false \vee \neg\varphi_2')$$
$$\equiv \varphi_1 \wedge \hat{T}_{\ell_1 \to \ell_2} \Rightarrow \neg\varphi_2'.$$

Otherwise, we obtain:

$$(true \wedge \varphi_1) \wedge (false \vee \neg\varphi_2) \wedge \hat{T}_{\ell_1 \to \ell_2} \Rightarrow (false \vee \neg\varphi_2')$$
$$\equiv \varphi_1 \wedge \neg\varphi_2 \wedge \hat{T}_{\ell_1 \to \ell_2} \Rightarrow \neg\varphi_2'. \qquad \square$$

Applying Lem. 3 to edge-relative inductivity for frames yields the following notion:

Definition 13 *(Edge-relative inductivity for frames)* For CFA $\mathcal{A}$ and locations $\ell, \ell' \in L$, a clause d is inductive relative to frame $F_{(i,\ell)}$ if:

$$F_{(i,\ell)} \wedge T_{\ell \to \ell'} \Rightarrow d' \quad \text{, if } \ell' \neq \ell \qquad (16)$$
$$F_{(i,\ell)} \wedge d \wedge T_{\ell \to \ell'} \Rightarrow d' \quad \text{, if } \ell' = \ell. \qquad (17)$$

Example 2 Reconsider the CFA $\mathcal{A}$ from Fig. 2. Consider the cube $c = (y \geq a)$ and frame $F_{(5,\ell_{12})} = \neg(y > 0)$. The clause $\neg c$ is inductive relative to $F_{(5,\ell_{12})}$ along $e = (\ell_{12}, \texttt{assume } y > 0, \ell_{13}) \in G$ iff

$$unsat\,(\neg(y > 0) \wedge (y > 0) \wedge (y \geq a))?$$

which is indeed unsatisfiable. Note that neither y nor a appear primed in c', since e does not assign these variables.

The next important point are predecessors. They are essential in IC3 since in order to determine whether a location in a frame F_i is reachable in at most i steps, a backward search is performed. More exactly, if clause $\neg c$ at level i is not inductive relative to F_{i-1}, IC3 needs a predecessor s to recursively check inductivity of s relative to F_{i-2}. Finding such predecessors is more involved in our setting as the IC3 approach to use a model of the SAT-solver is not applicable here, since there may be *infinitely many* predecessor states. In contrast to [9,14,28,29], we use weakest preconditions to compute all predecessors.

4.2 The IC3CFA algorithm

This section details the search phase of IC3CFA, presents the IC3CFA algorithm and states its correctness. Generalization is deferred to Sect. 5. We follow the presentation of [10] and highlight our modifications to IC3 if they are more than just notational changes. We use pre-/postconditions like [10] to construct the correctness proof more easily.

We begin with the top-level function PROVE as shown in Alg. 1, which includes the top-level loop over iterations

Algorithm 1 Outer loop [32]

```
Require: $true$
Ensure: return $false$ iff $\ell_E$ is reachable
  function PROVE($\mathcal{A}$)
  if $\ell_0 = \ell_E$ or $((\ell_0, \psi, \ell_E) \in G$ and $sat(\psi)$?) then
      return $false$
  initialize frames
  for $k = 1$ to ... do
      if not STRENGTHEN($k$) then
          return $false$
      PROPAGATECLAUSES($k$)
      if $\exists i < k. \forall \ell \in L. clauses(F_{(i,\ell)}) = clauses(F_{(i+1,\ell)})$ then
          return true
```

$1, \ldots, k$. It is akin to the original [10]. Apart from the modified termination criterion (see Lem. 2), we only have to adapt the initial checks for zero- and one-step counterexamples that cannot be detected in the main loop of IC3. Zero-step counterexamples appear when the initial location ℓ_0 is also the error location ℓ_E. For one-step counterexamples, i.e., $I \wedge T \wedge \neg\varphi'$ in [10], we have to detect whether (a) there exists a transition from ℓ_0 to ℓ_E and (b) whether this transition is actually feasible. Since the initial region as well as the error region is unconstrained in the data region, we just have to check whether there exists any assignment that satisfies the transition formula ψ. If both checks fail, we initialize frames at level $i = 0$ and $i = 1$: $F_{(0,\ell)}$ is set to *false* for all $\ell \in L$ except for $F_{(0,\ell_0)}$ which we set to *true*. This corresponds to $F_0 = I$. All frames $F_{(1,\ell)}$ are set to *true* for all $\ell \in L$, which conforms to $F_1 = \varphi$. In the remainder we iterate over the "frontier" k, starting with $k = 1$. After entering this loop, we try to strengthen the frames at level k by identifying the reason why induction fails. If strengthening is not possible, we have found a counterexample and return *false*. A successful strengthening means that no counterexample to induction (CTI), i.e., no state in $F_{(i,\ell)}$ with a successor in ℓ_E, is reachable any more and we can propagate clauses. Before advancing to the next iteration $k+1$ we check whether we have constructed a sufficient strengthening (according to Lem. 2). If this is the case, we return *true*; otherwise we increment k.

Whether a strengthening for some k exists is determined by the function STRENGTHEN presented in Alg. 2. As in IC3 [10], the function PROVE iterates as long as there exist CTI states. As we use WEP to extract an exact pre-image of $\neg\varphi$, we only have to consider a single data region per location, such that this loop degrades to iterating as long as there exists a location ℓ for which a state in $F_{(k,\ell)}$ can take a transition to the error location ℓ_E. If such ℓ with $e = (\ell, \texttt{cmd}, \ell_E) \in G$ exists, we compute the CTI data region φ as $wep\,(\texttt{cmd}, true)$ and call BACKWARDBLOCK with k and the region (ℓ, φ). If this returns *false*, it means that the backward search hit the initial location ℓ_0, i.e., we found a feasible counterexample path, such that we update *false* upward the call stack to

Algorithm 2 Strengthening [32]

```
Require: (a) $k \geq 1$
Require: (b) $\forall 0 \leq i < k, \ell \in L, F_{(i,\ell)} \Rightarrow F_{(i+1,\ell)}$
Require: (c) $\forall 0 \leq i < k, \ell, \ell' \in L$, s.t. $(\ell, \psi, \ell') \in G, F_{(i,\ell)} \wedge T_{\ell \to \ell'} \Rightarrow F'_{(i+1,\ell')}$
Ensure: $\forall 0 \leq i < k, \ell \in L, F_{(i,\ell)} \Rightarrow F_{(i+1,\ell)}$
Ensure: if ret. value then $\forall 0 \leq i < k, \ell, \ell' \in L$, s.t.$(\ell, \psi, \ell') \in G$, $F_{(i,\ell)} \wedge T_{\ell \to \ell'} \Rightarrow F'_{(i+1,\ell')}$
Ensure: if ¬ret. value, there exists a counterexample path
  function STRENGTHEN($k$)
  while $\exists \ell$, s.t. $sat\left(F_{(k,\ell)} \wedge T_{\ell \to \ell_E}\right)$? do
    @assert (b),(c)
    $\varphi$ := predecessor data region
    if not BACKWARDBLOCK($k$, $(\ell, \varphi)$) then
      return false
    @assert $\varphi \not\models F_{(k,\ell)}$
  return true
```

Algorithm 3 Inner loop [32]

```
Require: (b),(c)
Require: $sat\left(F_{(\hat{\imath},\hat{\ell}')} \wedge \hat{\varphi} \wedge T_{\hat{\ell}' \to \ell_E}\right)$?
Ensure: if ret. value, then $\neg\hat{\varphi}$ is inductive relative to $F_{(\hat{\imath}-1,\ell)}$, $\forall \ell$, $(\ell, t, \hat{\ell}') \in G$
Ensure: if ret. value, then (b),(c)
Ensure: if ¬ret. value, there exists a feasible path $\ell_0 \rightsquigarrow \hat{\ell}'$
  function BACKWARDBLOCK($\hat{\imath}$, $(\hat{\ell}', \hat{\varphi})$)
  $Q.add(\hat{\imath}, \hat{\ell}', \hat{\varphi})$
  while $|Q| > 0$ do
    @assert $\forall (i, \ell', \varphi) \in Q.\ 0 \leq i \leq k$
    @assert $\forall (i, \ell', \varphi) \in Q.\ \exists$ path $(\ell', \varphi) \rightsquigarrow (\ell_E, true)$
    $(i, \ell', \varphi) = Q.pop$
    if $i = 0$ then
      return false
    else
      @assert $(\ell', \neg\varphi)$ is ind. rel. to $F_{(j,\ell)}$,
                                    $\forall 0 \leq j < i, \ell \in L \setminus \{\ell_E\}$
      for each $\ell$, s.t. $(\ell, \varphi, \ell') \in G$ do
        if $\ell = \ell'$ and $sat\left(F_{(i-1,\ell)} \wedge \neg\varphi \wedge T_{\ell \to \ell'} \wedge \varphi'\right)$?
            OR $\ell \neq \ell'$ and $sat\left(F_{(i-1,\ell)} \wedge T_{\ell \to \ell'} \wedge \varphi'\right)$? then
          generate predecessor $\psi$ of $\varphi$
          @assert $\forall (i, \ell', \varphi) \in Q, \psi \neq \varphi$
          add $(i-1, \ell, \psi)$ and $(i, \ell', \varphi)$ to $Q$
        else
          GENERALIZECLAUSE($\varphi$)
          block $\varphi$ in frames $F_{(j,\ell')}$ for all $0 \leq j \leq i$
    return true
```

PROVE. On the other hand, if BACKWARDBLOCK succeeds to block the path up to φ, we can continue to the next iteration of the loop and see whether there exists another CTI region to be considered. If the loop terminates and there are no more CTI regions available, we have constructed a strengthening for level k and can return to PROVE. Note that in the search for CTI regions, according to Def. 11 we do not have to specifically handle locations ℓ which do not have a transition to ℓ_E, since for those locations $T_{\ell \to \ell_E} = false$ by definition, such that $F_{(k,\ell)} \wedge T_{\ell \to \ell_E}$ is not satisfiable.

Calling BACKWARDBLOCK (see Alg. 3) first pushes the CTI's level $\hat{\imath}$, location $\hat{\ell}$ and data region $\hat{\varphi}$ into the obligation queue Q. As in [21], the remainder of BACKWARDBLOCK consists of a loop that terminates if the obligation queue is empty. Inside the loop, we take the first proof obligation out of Q. If this obligation has level 0, we return *false* without further investigation since the obligation *must* intersect the initial states. If the obligation has level $i \neq 0$, then we have to check inductivity, like IC3 [10] and PDR [21]. Since we split F_{i-1} into $\{F_{(i-1,\ell_1)}, \ldots, F_{(i-1,\ell_n)}\}$ we check inductivity of φ relative to $F_{(i-1,\ell)}$ for every incoming edge $\left(\ell, \varphi, \ell'\right) \in G$. As the in-degree of CFAs for programs is usually small, the additional loop does not add much overhead, given an efficient data structure for storing predecessor locations. The three cases inside the inner loop in Alg. 3 are inspired by Lem. 3: If the pre-location and postlocation are identical and φ is not inductive relative to $F_{(i-1,\ell)}$, we determine ψ, the predecessor of φ, using WEP, add the new proof obligation $(i-1, \ell, \psi)$ to the obligation queue Q and put the current proof obligation back into Q. If $\ell \neq \ell'$, we do the same thing but consider the simpler relative inductivity that resembles reachability of φ from $F_{(i-1,\ell)}$. If both conditions fail, i.e., if φ is inductive relative to $F_{(i-1,\ell)}$, we block φ in all $F_{(j,\ell')}$ for $0 \leq j \leq i$.

Example 3 Let us illustrate how IC3CFA proves a property, using the running example of Fig. 1. As there are neither 0-step nor 1-step counterexamples, we proceed to the initialization of frames: $F_{(0,\ell_0)} = true$ and $F_{(0,\ell)} = false, \forall \ell \in L_P \setminus \{\ell_0\}$, where $L_P = L \setminus \{\ell_E\}$. Furthermore, we set $F_{(1,\ell)} = true, \forall \ell \in L_P$.

Afterward, we invoke the strengthening procedure (see Alg. 2) with k=1. Since $\ell_{13} \to \ell_E$ and the query $sat\left(F_{(1,\ell_{13})} \wedge T_{\ell_{13} \to \ell_E}\right)$?, i.e., $sat\,(true \wedge y \geq a)$?, yields *true*, we obtain the CTI data region $\varphi = (y \geq a)$ using the WEP. We enter BACKWARDBLOCK with the parameters $(1, (\ell_{13}, y \geq a))$ and add the corresponding proof obligation $(1, (\ell_{13}, y \geq a))$ to the queue Q. Since $i \neq 0$, we check for each predecessor location, in our case just ℓ_{12}, whether φ is inductive relative to $F_{(0,\ell_{12})}$ along edge e=$(\ell_{12}$, assume $y > 0, \ell_{13})$, i.e., whether $sat\left(F_{(0,\ell_{12})} \wedge T_{\ell_{12} \to \ell_{13}} \wedge y \geq a\right)$? holds or in our case $sat\,(false \wedge y > 0 \wedge y \geq a)$?, which is obviously false. We therefore block $\varphi = (y \geq a)$ in $F_{(1,\ell_{13})}$, i.e., $F_{(1,\ell_{13})} \leftarrow \neg(y \geq a)$.

As Q is empty now, we return to STRENGTHEN and find no remaining CTI region since for location ℓ_{13}, the query $sat\left(F_{(1,\ell_{13})} \wedge T_{\ell_{13} \to \ell_E}\right)$? yields *false* and no other predecessor location to ℓ_E exists. We have thus found a strengthening for level $k = 1$ and can check whether for some $0 \leq i \leq k$ and all $\ell \in L_P$, the clauses of $F_{(i,\ell)}$ and $F_{(i+1,\ell)}$ are identical. In our case, $i = 0$ is the only applicable level to check and the equivalence check already fails for ℓ_1, since $F_{(0,\ell_2)} = false \neq true = F_{(1,\ell_2)}$.

We therefore proceed to the next iteration $k = 2$. Here we find again the CTI data region $\varphi = (y \geq a)$ and enter BACKWARDBLOCK with $2, (\ell_{13}, y \geq a)$, which is not inductive relative to $F_{(1,\ell_{12})}$ along e. We thus determine the predecessor data region $\psi = (y \geq a) \wedge (y > 0)$ via WEP, add the new obligation $o_2 = (1, (\ell_{12}, \psi))$ to the queue and put the original obligation $o_1 = (2, (\ell_{13}, \varphi))$ back into the queue. In the next iteration of the while loop we pop o_2, since it has the lowest index. However, o_2 is not inductive relative to $F_{(0,\ell_8)}$ along $e_2 = (\ell_8, \texttt{assume } b = 0, \ell_{12})$ because $F_{(0,\ell_8)}$ is *false*. We therefore block ψ in $F_{(1,\ell_{12})}$ and continue with o_1, which has become inductive by blocking ψ in $F_{(1,\ell_{12})}$ and can therefore be blocked in $F_{(2,\ell_{13})}$, such that no CTI exists any more. In the termination check, we now have two indices to test: For $i = 0$ it fails due to $F_{(0,\ell_2)} = \mathit{false} \neq \mathit{true} = F_{(0,\ell_2)}$, while for $i = 1$ it fails because of $F_{(1,\ell_{12})} = \neg\psi \neq \mathit{true} = F_{(2,\ell_{12})}$. We omit further iterations here.

Our implementation of the IC3CFA algorithm is able to prove the correctness of the C program in Fig. 1 (without any minimizations and optimizations on the C code) after 101 iterations, which takes a total verification time of 177.8 seconds, of which 151.6 seconds are spent on 15602 solver calls.

Theorem 1 *[32] Function PROVE of Alg. 1 returns true on termination iff there exists an inductive strengthening F for property P such that $F \wedge P$ is inductive.*

4.3 Possible improvements

We consider three improvements:

1. Reuse existing obligations from the obligation queue;
2. Skip obligations after recursive descent in the obligation queue;
3. Construct counterexample paths.

Reusing obligations. Using this approach we statically determine all CTI regions upfront rather than recomputing the same region in every iteration. This can be pursued a bit further: In practice, the WEP computation leads to recomputing a set of obligations in every iteration. Assume a CFA $\mathcal{A}$ and iteration k where a path of length k exists in $\mathcal{A}$, but $\mathcal{A}$ has no counterexample path of length at most k. Then the set of all proof obligations created in the k^{th} iteration is the disjoint union of the set of newly explored obligations of the k^{th} iteration

$$\{(\ell, \varphi) \mid (0, \ell, \varphi) \in Q \text{ at some point in iteration } k\}$$

and the set of obligations created in the $k - 1^{st}$ iteration

$$\{(\ell, \varphi) \mid (i, \ell, \varphi) \in Q \text{ in iteration } k - 1\}.$$

This can be seen as follows: The search phase of IC3CFA explores all maximal, reachable path fragments of length up to k that reach a CTI state. In other words, it implicitly creates a tree of regions that are backward reachable from CTI regions and implicitly encodes the nodes of this tree as proof obligations. After proceeding to iteration $k+1$, IC3CFA may be able to explore direct predecessors of all those regions that are leaves in the implicit backward reachability tree. However, so as to explore these states, IC3CFA traverses the whole backward reachability tree of iteration k.

This suggests to reuse obligations of iteration k in iteration $k + 1$, just like Tree-IC3 continues to unroll the ART rather than deleting and re-unrolling it after every iteration [34]. To do so we only have to modify the internal behavior of the obligation queue Q: Rather than taking obligation $(i, (\ell, c))$ out of the queue, we keep $(i, (\ell, c))$ in Q and just mask the obligation such that it will not be reconsidered. In case we attempt to put a previously popped obligation $(i, (\ell, c))$ back into the queue, i.e., when IC3CFA detects a non-inductive region and needs to create a predecessor obligation, we remove the masking of $(i, (\ell, c))$ in Q. If Q only contains masked obligations, we consider Q empty. After checking inductivity on the frame sequences, we can then recover the obligations computed during this iteration by removing all markings from obligations in Q. But in order to reuse these obligations in the next iteration, we first have to modify them: Due to the backward search approach of IC3 that starts at the iteration count k, new obligations in $k + 1$ are created one level after the ones created in iteration k. Therefore we increment the level entry i of every obligation $(i, (\ell, c))$ in Q. The resulting obligation queue allows us to start our backward search at those regions that we were not able to explore any further due to the iteration's bounded search length k.

Skipping obligations. Another aspect related to the computation of exact pre-images using WEP arises when a region has been shown to be inductive relative to all predecessor locations' frames. If a proof obligation $(i, (\ell, c))$ succeeds, the cube c is blocked in $F_{(i,\ell)}$ and two situations are possible: Either there exists another obligation $(i, (\ell', c'))$ at level i or $(i, (\ell, c))$ was the last obligation at level i and the next obligation appears at level $i + 1$. In the first case, ℓ' is a predecessor of some $\bar{\ell}$ that appears in a proof obligation at level $i + 1$ and we have to check whether $(i, (\ell', c'))$ is inductive relative to their respective predecessor frames. In the latter case, we return to obligation $(i + 1, (\bar{\ell}, \bar{c}))$ after exploring all paths leading up to $\bar{\ell}$ of length up to i. While for standard IC3 we might encounter new predecessor states, the use of WEP guarantees that no predecessor region can reach the region $(\bar{\ell}, \bar{c})$. This implies that $(\bar{\ell}, \bar{c})$ is inductive and the proof obligation $(i + 1, (\bar{\ell}, \bar{c}))$ succeeds. As a result, we can immediately block each obligation that has been considered before and which has been put back into the obligation queue Q.

Counterexample generation. Concrete counterexamples are of large value for practical applications. For the representation of such counterexamples two common approaches exist: The first is an initial variable assignment on which the program can be simulated to provide a violating run. Depending on the structure of the program, this approach can be sufficient and offers the advantage that simulation is very efficient and allows the user to see the violating run directly in the source code. For IC3CFA such an initial assignment can be efficiently extracted: If we encounter a violation in BACKWARDBLOCK, we have popped an obligation $(0, (\ell, c))$ from Q. Furthermore, by construction, $\ell = \ell_0$ and c represents the data region that has successors leading to the error location ℓ_E. So in order to extract a violating initial variable assignment, we just take one state from the violating set of states represented by c, i.e., we query the SMT solver with $sat\,(c)$?, which returns SAT by construction, and extract the model which represents one possible variable assignment of the data region c. However, in the presence of nondeterminism it can be difficult to reconstruct a violating run from the initial state.

The second, more involved approach to counterexample generation is to construct a full counterexample trace directly in the model checker. In order to achieve this with IC3CFA, or with the algorithms in [10,21], we have to store the dependencies between obligations in Q, e.g., parent pointers that indicate based on which other obligation an obligation has been created. Such relations between obligations allow us to trace paths through the implicit backward reachability tree that is given by the proof obligations in Q. Tracing the dependencies of obligations allows us to extract a set of obligations $(0, (\ell_0, c_0)), \dots, (k, (\ell_k, c_k))$. After ordering the obligations based on their level i in ascending order, the list of obligations implies a full path $\pi := \ell_0, \ell_1, \dots, \ell_k, \ell_E$ through the CFA $\mathcal{A}$ with annotated data regions $c_0, \dots, c_k$ except for ℓ_E. This approach compensates for many of the shortcomings of simulating initial variable assignments by not relying on a simulation and being able to handle nondeterministic assignments through the data regions learnt during exploration. But without additional overhead, the counterexample trace can only be mapped back to the input CFA. As typically the input program is reduced prior to the verification, a counterexample on the CFA of the reduced program generally cannot be mapped back to the original program.

5 Generalization

In the previous section, we introduced the basic search phase of the IC3CFA algorithm in analogy to the IC3 algorithm [10] and its PDR variant [21] (cf. Sect. 6.3 for more details on the latter). However, in our presentation of IC3CFA in Alg. 1 to 3 we did omit a crucial part of the IC3 algorithm: generalization. This means that by weakening of first-order logic formulae, we compute overapproximations of pre-images of symbolic states, which allows us to deal with software modeled as a (fully) symbolic transition system. As stated before, IC3 is sound and complete without generalization, but is hardly scalable. As such, generalization is not strictly necessary but the success of IC3 is mainly due to its ability to efficiently prune the state space. For IC3CFA, however, we need to answer three main questions:

1. how can we generalize a cube of some first-order theory, rather than pure propositional logic;
2. how do we handle generalization in the presence of multiple incoming edges and multiple predecessor frames; and
3. what are the effects of weakest predecessor computation on the generalization?

We will consider these issues in the following subsections.

5.1 Generalization of cubes

To generalize a first-order cube c, we use theory-unaware syntactic weakening operations on formulae and employ SMT solvers. We first abstract all theory aspects from cube c, resulting in its Boolean skeleton. Based on this skeleton, we execute standard IC3 generalization. This results in dropping predicate symbols from c. This lifting of the syntactic generalization of IC3 offers a simple way to generalize first-order cubes in IC3CFA.

5.2 Generalization on multiple edges

In order to generalize a cube c at location ℓ' and index i, we consider a generalization of c that is inductive relative to $F_{(i-1,\ell')}$ along all incoming edges $T_{\ell \to \ell'}$ for $\ell \in pred(\ell')$. To represent this n-dimensional inductivity for $n = |pred(\ell')|$, we construct an SMT query of the form

$$\left(\left(F_{(i-1,\ell_1)} \wedge T_{\ell_1 \to \ell'}\right) \vee \dots \vee \left(F_{(i-1,\ell_n)} \wedge T_{\ell_n \to \ell'}\right)\right) \wedge c'. \tag{18}$$

Inspired by (14), this query allows us to check inductivity of c relative to all predecessor frames along all incoming edges and thus to determine a valid generalization of c for ℓ'.

It is easy to see that cube c satisfies (18) iff c satisfies (13) for all $T_{\ell_j \to \ell'}$, $1 \le j \le n$. However, (18) combines the inductivity of all incoming edges of ℓ' into one monolithic query. This contradicts IC3's principle of avoiding monolithic approaches in favor of modularity. We therefore strive to find generalizations over multiple incoming edges in an *incremental* fashion. To do so, we first consider isolated generalizations of each incoming edge of ℓ' and combine these into a valid generalization for c at ℓ'.

For location ℓ' with $pred(\ell') = \{\ell_1, \ldots, \ell_n\}$, the set of incoming edges into ℓ' is $in(\ell') = \{e_j \in G \mid \exists 1 \leq j \leq n, \mathsf{cmd}_j \in GCL.e_j = (\ell_j, \mathsf{cmd}_j, \ell')\}$. For index i and cube c, we determine a generalization g_j of c that is inductive relative to $F_{(i-1,\ell_j)}$ for each incoming edge e_j, which yields a generalization for location ℓ' as follows.

Corollary 3 *If g_j is a generalization for edge $e_j \in in(\ell')$, for each $1 \leq j \leq n$, then $g = \bigcup_{j=1}^{n} g_j$ is a valid generalization for c at ℓ'.*

Proof A generalization for c along all incoming edges can only drop literal $lit \in c$ if it can be dropped along each edge e_j. Therefore the union of literals of all edge-local generalizations is a valid generalization for location ℓ'. □

Example 4 Consider our example program from Fig. 1 with CFA $\mathcal{A}$ in Fig. 2. Let $F_{(2,\ell_4)} = \neg(y > 0 \wedge b = 0)$ and $F_{(2,\ell_5)} = \neg true$. The proof obligation $(3, (\ell_6, y \geq a \wedge y > 0 \wedge b = 0 \wedge b \geq 0))$ is inductive relative to $F_{(2,\ell_4)}$ along $e_4 = (\ell_4, \mathsf{assume}\ a \geq 0, \ell_6)$, as well as to $F_{(2,\ell_5)}$ along $e_5 = (\ell_5, a := -a, \ell_6)$. We thus generalize the cube with respect to both e_4 and e_5. Generalizing $c = y \geq a \wedge y > 0 \wedge b = 0 \wedge b \geq 0$ along e_4 results in $g_4 = y > 0 \wedge b = 0$. For edge e_5, we obtain $g_5 = true$. Thus $g = g_4 \cup g_5 = (y > 0 \wedge b = 0)$ is blocked in $F_{(3,\ell_6)}$.

The standard IC3 generalization with the lifting to CFAs enables significant improvements (cf. Sect. 4.2): We are able to solve the example after 56 iterations in 9.3 seconds, of which 8.6 seconds are spent for 14,583 SMT calls.

We now improve on considering generalizations of incoming edges separately. Consider an edge-local generalization $g_h = c \setminus \delta$ with δ containing all literals that g_h dropped from c. Since $g = \bigcup_{j=1}^{n} g_j$, $g_h \subseteq g$ and hence for $g = c \setminus \Delta$ it follows that $\Delta \subseteq \delta$. This allows us to execute the full syntactic generalization only on the first incoming edge and then check whether g_h is inductive along the other edges. If it is not inductive on some other edge e_m, we backtrack based on δ. Note that it does not suffice to iteratively shift literals back from δ into g_m (which is initially identical to g_h) and to check inductivity on g_m since this may introduce many unnecessary literals. A more formal perception can be given in terms of the induced subclause lattice [12]. When dropping literals from clause $d = \neg c$, we traverse this lattice downward toward the empty clause $\perp$. If $d_j = \neg g_j$ is not inductive along e_m, a non-inductive clause in the lattice is found. From [10,12] and the restriction given in Cor. 3, there must exist a clause d_m that is a valid generalization along e_m with $d_j \subset d_m$, i.e., d_m is reachable from d_j by traversing the subclause lattice upward. However, there are exponentially many subclauses d_m for which $d_j \subset d_m$, precisely 2^n with $n = |c \setminus g_j|$, of which not many are inductive. It is thus very likely that we end up with a subclause that is not minimal. In order to obtain a minimal subclause, we could traverse the subclause lattice back down from d_m, similar to the procedure given in [12], but we might also just give up minimality and content ourselves with d_m. A third approach would be to refrain from traversing the subclause lattice upward in the first place and rather restart generalization of c but only check the literals of δ for dropping. In practice, the question which of these three methods performs best heavily depends on the input model, and the answer varies from case to case. For the remainder we will choose the third approach for its simplicity.

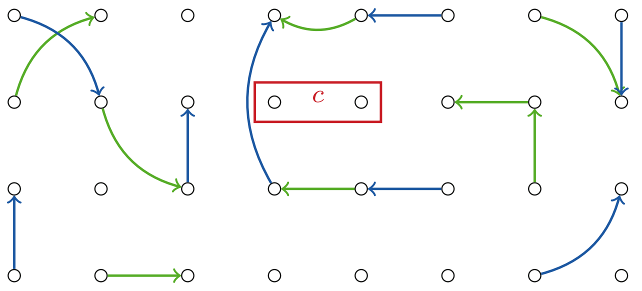


Fig. 3 Example transition system (e_1 green, e_2 blue)

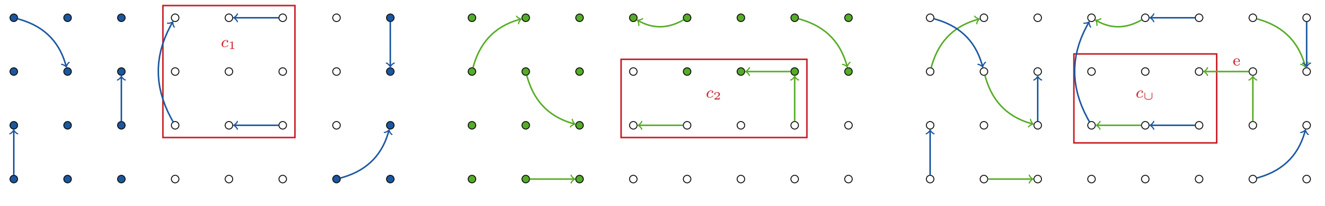
Example 5 We now illustrate why the stronger inductivity query of Lem. 3 cannot be applied in the presence of generalization. Let $(i, (\ell, c))$ be an obligation consisting of cube c and location ℓ at index i, with ℓ having incoming edge e_1 and self-loop e_2.

To simplify things we sketch a simplified state space for unprimed state variables without extracting control flow like in IC3-SMT [14]. The transition system is depicted in Fig. 3. For simplicity, arrows depict a transition between an unprimed state and a primed state. Blue arrows represent transitions according to edge e_1 and green arrows stand for the self-loop edge e_2. Red boxes represent the states symbolically encoded by the annotated cube. As proven in [32], we can use the original inductivity query

$$F_{(i,\ell)} \wedge \neg c \wedge T_e \Rightarrow \neg c' \tag{19}$$

for IC3CFA without generalization.

However, the presence of $\neg c$ in the premise excludes all states in c from considering their successors. We now isolate edge e_1 and e_2 as depicted in Fig. 4a, b. We construct generalizations $g_1 = \neg c_1$ and $g_2 = \neg c_2$ that are inductive relative to the corresponding frames, where a state, depicted by a dot, is in $F_{(i-1,\ell_{pre})}$ iff the dot is filled. Generalizations g_1 and g_2 overapproximate c and g_i is inductive relative to the predecessor frame along edge e_i. To block a safe superset of states of c, we block the states that are not reachable via both edges, i.e., $g = g_1 \cup g_2$. However as depicted in Fig. 4c, we have not considered reachability of g_2 via transitions that



(a) Transitions and frames for e_1 **(b)** Transitions and frames for e_2 **(c)** Result of merging generalizations

Fig. 4 Example of different edges in generalization

originate from states in g_2, in Fig. 4a depicted by arrows inside of g_2. By taking the intersection of the respective state sets, the result cuts transitions that were not considered in the inductivity query of g_2, due to the use of $\neg c$ in the premise of (19). This leads to a violation of the inductivity query for g.

Theorem 2 *Generalization does not preserve* (19)*:*

$$\left(F_1 \wedge T_1 \Rightarrow \neg g_1'\right) \wedge \left(F_2 \wedge T_2 \wedge \neg g_2 \Rightarrow \neg g_2'\right) \;\not\Longrightarrow\; \left(\left(F_1 \wedge T_1\right) \vee \left(F_2 \wedge T_2 \wedge \neg\left(g_1 \wedge g_2\right)\right) \Rightarrow \neg\left(g_1' \wedge g_2'\right)\right).$$

Proof Assume

a) $F_1 \wedge T_1 \wedge g_1'$ unsatisfiable, and
b) $F_2 \wedge T_2 \wedge \neg g_2 \wedge g_2'$ unsatisfiable.

Then the conclusion is invalid: We derive that

$$\begin{aligned}
&\left(F_1 \wedge T_1 \wedge g_1' \wedge g_2'\right) \\
&\quad \vee \left(F_2 \wedge T_2 \wedge \neg\left(g_1 \wedge g_2\right) \wedge g_1' \wedge g_2'\right) \\
&\overset{a)}{\Longleftrightarrow} F_2 \wedge T_2 \wedge \neg\left(g_1 \wedge g_2\right) \wedge g_1' \wedge g_2' \\
&\Longleftrightarrow F_2 \wedge T_2 \wedge \left(\neg g_1 \vee \neg g_2\right) \wedge g_1' \wedge g_2' \\
&\Longleftrightarrow \left(F_2 \wedge T_2 \wedge \neg g_1 \wedge g_1' \wedge g_2'\right) \\
&\quad \vee \left(F_2 \wedge T_2 \wedge \neg g_2 \wedge g_1' \wedge g_2'\right) \\
&\overset{b)}{\Longleftrightarrow} F_2 \wedge T_2 \wedge \neg g_1 \wedge g_1' \wedge g_2'.
\end{aligned}$$

This formula is satisfiable. □

To fix this deficiency we strengthen the inductivity query by weakening the premise and removing $\neg c$ from (19).

Theorem 3 *Generalization preserves* (16)*:*

$$\begin{aligned}
&\left(F_1 \wedge T_1 \Rightarrow \neg g_1'\right) \wedge \left(F_2 \wedge T_2 \Rightarrow \neg g_2'\right) \\
&\quad \Longrightarrow \left(\left(F_1 \wedge T_1\right) \vee \left(F_2 \wedge T_2\right) \Rightarrow \neg\left(g_1' \wedge g_2'\right)\right)
\end{aligned}$$

Proof We derive:

$$\begin{aligned}
&\left(\left(F_1 \wedge T_1\right) \vee \left(F_2 \wedge T_2\right)\right) \wedge g_1' \wedge g_2' \\
&\quad \Longleftrightarrow \exists s . s \models \left(\left(\left(F_1 \wedge T_1\right) \vee \left(F_2 \wedge T_2\right)\right) \wedge g_1' \wedge g_2'\right) \text{ satisf.} \\
&\quad \Longleftrightarrow \exists s . s \models \left(F_1 \wedge T_1 \wedge g_1' \wedge g_2'\right) \vee \left(F_2 \wedge T_2 \wedge g_1' \wedge g_2'\right) \\
&\quad \Longrightarrow \exists s . s \models \left(F_1 \wedge T_1 \wedge g_1'\right) \vee \left(F_2 \wedge T_2 \wedge g_2'\right).
\end{aligned}$$

This contradicts validity of $(F_1 \wedge T_1 \Rightarrow \neg g_1')$ and $(F_2 \wedge T_2 \Rightarrow \neg g_2')$. □

A simple improvement to generalization in order to save calls to the SMT solver is to statically check for duplicate literals in the formula, especially those in the cube c and in the frame. If the frame contains a clause $\neg lit$ which also appears as $\neg lit \in c$, then the satisfiability of the formula does not change when we remove $\neg lit$ from c. Therefore we can statically drop literals. We call these *don't care literals*, and refer to statically removing them as *don't care generalization*.

5.3 Generalization with weakest preconditions

Weakest existential preconditions (WEP) allow to efficiently compute the exact set of predecessor states for a cube that is not inductive relative to its predecessor frame along some edge. This section describes the implications of WEP to generalization [37]. We assume all edges to contain straight-line code, i.e., no GCL choice command □. Choice commands are removed as follows:

Definition 14 The function *split* translates GCL command cmd into a set of choice-free GCL commands as follows:

$$\mathit{split}(\mathsf{cmd}) = \begin{cases} \mathit{split}(\mathsf{cmd}_1) \cup \mathit{split}(\mathsf{cmd}_2) & \text{if } \mathsf{cmd} = \mathsf{cmd}_1 \,\square\, \mathsf{cmd}_2 \\ \{c_1;\ c_2 \mid i \in \{1,2\}, \\ \qquad c_i \in \mathit{split}(\mathsf{cmd}_i)\} & \text{if } \mathsf{cmd} = \mathsf{cmd}_1;\ \mathsf{cmd}_2 \\ \{\mathsf{cmd}\} & \text{otherwise.} \end{cases}$$

The function *split* is applied to transform edge $e = (\ell, \mathsf{cmd}, \ell') \in G$ into a set G' of choice-free edges $e' = (\ell, \mathsf{cmd}', \ell') \in G'$. Furthermore, we restrict all Boolean guards b of assume b commands to not contain disjunctions. These restrictions establish that all CTIs are cubes and that the WEP of a cube is a cube.

Using WEP for inductivity queries. The query

$$unsat\left(F_{(i-1,\ell)} \wedge T_{\ell\to\ell'} \wedge c'\right)?$$

asks whether some state characterized by $F_{(i-1,\ell)}$ can reach a state in c via $T_{\ell\to\ell'}$. A semantically equivalent formulation is whether there exists some state in the pre-image of c under $T_{\ell\to\ell'}$ *and* in $F_{(i-1,\ell)}$. Since the first part corresponds to the WEP, this corresponds to the query

$$unsat\left(F_{(i-1,\ell)} \wedge wep(T_{\ell\to\ell'}, c)\right)?$$

This observation motivates a WEP-based alternative to relative inductivity.

Definition 15 (WEP-relative inductivity) Let $i \geq 1$ and CFA $\mathcal{A} = (L, G, \ell_0, \ell_E)$ with edge $e = (\ell, \mathsf{cmd}, \ell') \in G$. For frame $F = F_{(i-1,\ell)}$ and a cube c, let:

$$relInd(F, e, c) \iff unsat\left(F \wedge T_{\ell\to\ell'} \wedge c'\right)?$$
$$relIndAlt(F, e, c) \iff unsat\left(F \wedge wep(\mathsf{cmd}, c)\right)?$$

The following result shows that relInd and relIndAlt are equivalent.

Theorem 4 *Let $i \geq 1$ and (L, G, ℓ_0, ℓ_E) be a CFA with edge $e = (\ell, \mathsf{cmd}, \ell') \in G$. Furthermore, let c be a cube and $F_{(i-1,\ell)}$ be a frame. It holds that:*

$$relIndAlt(F_{(i-1,\ell)}, e, c) \iff relInd(F_{(i-1,\ell)}, e, c).$$

Proof

$$\begin{aligned} &relIndAlt(F_{(i-1,\ell)}, e, c)\\ &\iff unsat(F_{(i-1,\ell)} \wedge wep(\mathsf{cmd}, c))\\ &\iff unsat(F_{(i-1,\ell)} \wedge T_{\ell\to\ell'} \wedge c') && \text{(Def. 3)}\\ &\iff relInd(F_{(i-1,\ell)}, e, c). \end{aligned}$$
□

While both equivalent formulations seem more or less identical, their execution time in the SMT solver may vary heavily. First, *relInd* refers to primed and unprimed variables, while *relIndAlt* only relies on unprimed variables. The SMT query of *relInd* may, in worst case, contain twice as many variables. The number of variables is certainly not a precise metric for the complexity of a formula and the time to solve it, but they are correlated [7,11,30]. In addition, *relIndAlt* can be beneficial for SMT solvers with caching, since for edges with different GCL commands $\mathsf{cmd} \neq \mathsf{cmd}'$ such that $wep(\mathsf{cmd}, c) = wep(\mathsf{cmd}', c)$, the queries *relInd* are different, but the ones for *relIndAlt* are identical and can thus benefit from caching. On the other hand, for *relIndAlt* we have to compute the WEP for every inductivity query, whereas for *relInd* this is only needed if c is not inductive relative to $F_{(i-1,\ell)}$. A detailed empirical evaluation of the effects of *relIndAlt* is provided in Sect. 7.

Assumed literals. The assume command in GCL is a liberal version of the assert command.[2] While an assert command should never evaluate to false in a correct program, assume can evaluate to *false* without any side-effect apart from terminating control flow. The assume command together with the choice command □ allows for modeling deterministic branching. We can thus consider assume statements as guards, i.e., for edge $e = (\ell, \mathsf{assume}\ b, \ell') \in G$, *every* state in ℓ that terminates in ℓ' *must* satisfy the guard b and every $\neg b$-state in ℓ has no successor along e. For literal *lit* of cube c that is inductive relative to $F_{(i-1,\ell)}$ along e, $c \setminus \{lit\}$ is inductive relative to $F_{(i-1,\ell)}$ along e. We can statically check whether such literals are contained in c in order to use it as generalization.

To formalize this generalization based on literals contained in assume, we first show that WEP is distributive over conjunction of cubes.

Lemma 4 *Let* cmd *be a choice-free GCL command. Given two cubes c_1, c_2, it holds that*

$$wep(\mathsf{cmd}, c_1 \wedge c_2) \iff wep(\mathsf{cmd}, c_1) \wedge wep(\mathsf{cmd}, c_2).$$

Proof By structural induction over GCL command cmd without choice. □

For choice-free GCL command cmd, all literals *lit* that are assumed in cmd can be removed from cube c without any SMT query.

Theorem 5 *Let g be the generalization of cube c at index $i \in \mathbb{N}$ and location $\ell' \in L$ for edge $e = (\ell, \mathsf{cmd}, \ell') \in G$. For all literals $lit \in g$, $\mathsf{cmd}' \in split(\mathsf{cmd})$ and "virtual" edges $e' = (\ell, \mathsf{cmd}', \ell')$ the following holds:*

$$\begin{aligned} &wep(\mathsf{cmd}', true) \subseteq wep(\mathsf{cmd}', lit)\\ &\quad\implies relInd(F_{(i-1,\ell)}, e', c \setminus \{lit\}). \end{aligned}$$

Proof Let $\widehat{c} = c \setminus \{lit\}$, and assume $wep(\mathsf{cmd}', lit) \subseteq wep(\mathsf{cmd}', true)$.

$$\begin{aligned} &relInd(F_{(i-1,\ell)}, e', c)\\ &\iff relInd(F_{(i-1,\ell)}, e', \widehat{c} \wedge lit)\\ &\iff relIndAlt(F_{(i-1,\ell)}, e', \widehat{c} \wedge lit) && \text{(Thm. 4)}\\ &\iff unsat(F_{(i-1,\ell)} \wedge wep(\mathsf{cmd}', \widehat{c} \wedge lit)) && \text{(Def. 15)} \end{aligned}$$

[2] Though being part of our implementation and GCL [19,20], we omitted the assert command since, in a preprocessing phase, we transform all edges with assert commands into one edge that assumes the assert to be valid and one that assumes it to be false, where the former one maintains the original source and target location and the latter leads to the error location.

$$\begin{aligned}
&\iff \mathit{unsat}(F_{(i-1,\ell)} \wedge \mathit{wep}(\mathsf{cmd}', \widehat{c}) \wedge \mathit{wep}(\mathsf{cmd}', \mathit{lit})) && \text{(Lem. 4)}\\
&\implies \mathit{unsat}(F_{(i-1,\ell)} \wedge \mathit{wep}(\mathsf{cmd}', \widehat{c}) \wedge \mathit{wep}(\mathsf{cmd}', \mathit{true})) && \text{(Ass.)}\\
&\iff \mathit{unsat}(F_{(i-1,\ell)} \wedge \mathit{wep}(\mathsf{cmd}', \widehat{c} \wedge \mathit{true})) && \text{(Lem. 4)}\\
&\implies \mathit{unsat}(F_{(i-1,\ell)} \wedge \mathit{wep}(\mathsf{cmd}', \widehat{c}))\\
&\iff \mathit{relIndAlt}(F_{(i-1,\ell)}, e', \widehat{c}) && \text{(Def. 15)}\\
&\iff \mathit{relInd}(F_{(i-1,\ell)}, e', \widehat{c}) && \text{(Thm. 4)}
\end{aligned}$$

where $\mathit{gen}(F_{(i-1,\ell)}, e', c)$ denotes the set of valid generalizations of cube c relative to frame $F_{(i-1,\ell)}$ and edge e'.

Since edge e has been split into a set of "virtual" edges, thanks to this theorem we can treat them as if they were regular edges, to obtain a generalization for e. □

Example 6 We would like to check whether cube $c = (x > y \wedge y > 0)$ along edge $e = (\ell, \mathsf{assume}\ y > 0, \ell')$ is relative to $F_{(i,\ell)}$. The above theorem allows us to generalize c to $g = (x > y)$ without any SMT calls.

Dropping a literal based on the GCL assumptions is a very simple, yet efficient enhancement. Checking whether an edge contains assume statements is linear in the size of the GCL command.

Static inductivity checks and generalization using WEP. WEP can also be used to statically detect that a cube c *must* be inductive. As we will see, the static check is incomplete: if it fails, we cannot conclude that c is not inductive. The static check is favorable as it is computationally cheap and can save expensive SMT calls. In addition, it yields generalizations of good quality. The key idea is as follows.

Given that c is inductive relative to $F_{(i-1,\ell)}$ along $e = (\ell, \mathsf{cmd}, \ell')$, the formula $F_{(i-1,\ell)} \wedge \mathit{wep}(\mathsf{cmd}, c)$ is unsatisfiable by Thm. 4. In other words, no state in $\mathit{wep}(\mathsf{cmd}, c)$ is contained in $F_{(i-1,\ell)}$. Due to the exclusion of disjunctions in Boolean guards and the restriction to choice-free GCL commands, $\mathit{wep}(\mathsf{cmd}, c)$ will yield a cube c_{pre} and all states $s \in c_{pre}$ are blocked in $F_{(i-1,\ell)}$. Possibly all $s \in c_{pre}$ will be blocked by a single cube $c_F \in F_{(i-1,\ell)}$ with $c_F \subseteq c_{pre}$, i.e., c_F blocks a superset of the states in c_{pre}. We formalize this static check as follows:

Theorem 6 *Let $i \in \mathbb{N}$, (L, G, ℓ_0, ℓ_E) be a CFA with edge $e = (\ell, \mathsf{cmd}, \ell') \in G$. Consider cube c at location ℓ' and index i with corresponding frame $F_{(i-1,\ell)}$, then*

$$\exists c_F.\ \big(\neg c_F \in F_{(i-1,\ell)} \wedge c_F \subseteq \mathit{wep}(\mathsf{cmd}, c)\big) \implies \mathit{relInd}(F_{(i-1,\ell)}, e, c).$$

Proof

$$\begin{aligned}
&\exists c_F.\ \neg c_F \in F_{(i-1,\ell)} \wedge c_F \subseteq \mathit{wep}(\mathsf{cmd}, c)\\
&\implies \exists c_F.\ \mathit{unsat}(F_{(i-1,\ell)} \wedge c_F) \wedge c_F \subseteq \mathit{wep}(\mathsf{cmd}, c)\\
&\implies \mathit{unsat}(F_{(i-1,\ell)} \wedge \mathit{wep}(\mathsf{cmd}, c))\\
&\iff \mathit{relIndAlt}(F_{(i-1,\ell)}, e, c) && \text{(Def. 15)}\\
&\iff \mathit{relInd}(F_{(i-1,\ell)}, e, c). && \text{(Thm. 4)}
\end{aligned}$$

□

Thus, if c_F appears negated in $F_{(i-1,\ell)}$ and only has literals in $\mathit{wep}(\mathsf{cmd}, c)$, then c is inductive relative to $F_{(i-1,\ell)}$ along edge e. If this check fails, we cannot deduce that c is not inductive relative $F_{(i-1,\ell)}$ along edge e, since c may be blocked in $F_{(i-1,\ell)}$, by a set of cubes each of which blocks a certain subset of the states in c. As IC3CFA frames are typically smaller than those for IC3, the cost of statically checking whether such c exists in $F_{(i-1,\ell)}$ is negligible.

Example 7 Reconsider the CFA $\mathcal{A}$ from Fig. 2 and consider $(3, (\ell_6, c))$ with cube $c = (y \geq a \wedge y > 0 \wedge b = 0 \wedge b \geq 0)$ along $e_4 = (\ell_4, \mathsf{assume}\ a \geq 0, \ell_6)$. The WEP of c is $\mathit{wep}(\mathsf{assume}\ a \geq 0, c) = c \wedge a \geq 0$. Since $F_{(2,\ell_4)} = \neg(y \geq 0 \wedge b = 0)$, the clause $\neg c_F = \neg(y \geq 0 \wedge b = 0)$ is contained in $F_{(2,\ell_4)}$ and $c_F \subseteq \mathit{wep}(\mathsf{assume}\ a \geq 0, c)$, such that c is inductive relative to $F_{(2,\ell_4)}$ along e_4. Stated differently, $\mathit{wep}(\mathsf{assume}\ a \geq 0, c)$ contains all states that can possibly go to c via e_4. These are states where y is at least 0 *and* b equals 0. But from previously learnt information we conclude that reachable states in $F_{(2,\ell_4)}$ satisfy $y < 0$ or $b \neq 0$ and thus conflict all states in $\mathit{wep}(\mathsf{assume}\ a \geq 0, c)$.

If $\neg c_F \in F_{(i-1,\ell)}$, then also every cube c_{post} with $c_F \subseteq \mathit{wep}(\mathsf{cmd}, c_{post})$ is inductive relative to $F_{(i-1,\ell)}$. Thus, cube c_g with $c_F = \mathit{wep}(\mathsf{cmd}, c_g)$ is the best generalization in that case. From a semantic point of view, c_g is the strongest postcondition of cmd w.r.t. c_F. However, the syntactic inclusion $c_g \subseteq c$ does not hold in general and thus c_g is not a valid generalization. We therefore search for a slightly different function that inverts the wep function and preserves the syntactical subset relation.

In order to define function wep^{-1}, we exploit that wep distributes over conjunctions (cf. Lem. 4). After partitioning the cube c into its literals $c = \{\mathit{lit}_1, \ldots, \mathit{lit}_n\}$, the wep of literal lit_i is determined individually. Assumed literals have some side effects and are directly treated.

Lemma 5 *For choice-free GCL command* cmd *and cube c:*

$$\mathit{wep}(\mathsf{cmd}, c) = \bigwedge_{\mathit{lit} \in c} \mathit{wep}(\mathsf{cmd}^{\mathsf{assume}}, \mathit{lit}) \wedge \mathit{wep}(\mathsf{cmd}, \mathit{true})$$

where $\mathsf{cmd}^{\mathsf{assume}}$ *is the command* cmd *with all* assume *statements removed.*

Proof Structural induction over cmd. □

This decomposition allows us to store a mapping $m : \mathit{Literal} \dashrightarrow \mathit{Literal}$ from $\varphi = \mathit{wep}(\mathsf{cmd}^{\mathsf{assume}}, \mathit{lit}_i)$ to lit_i of the assume-free GCL command.

Corollary 4 *If there exists $c_F \in F_{(i-1,\ell)}$ such that $c_F \subseteq wep(\mathsf{cmd}, c)$, then*

$$c_F \subseteq \bigwedge_{lit \in c} wep(\mathsf{cmd}^{\mathsf{assume}}, lit) \wedge wep(\mathsf{cmd}, true).$$

Using Cor. 4, we apply the mapping m to the set of literals $\{\varphi_j \mid \varphi_j \in (c_F \backslash wep(\mathsf{cmd}, true))\}$, i.e.,

$$m(\varphi_j) = lit_j \wedge \varphi_j \in c_F \implies lit_j \in c_m.$$

Definition 16 Given cubes $c_1 = \{lit_1, \ldots, lit_n\}$ and $c_2 \subseteq c_1$, choice-free GCL command cmd and partial mapping $m : Literal \dashrightarrow Literal$ with

$$m(\varphi_i) = lit_i \iff wep(\mathsf{cmd}^{\mathsf{assume}}, lit_i) = \varphi_i$$

we define $wep_{c_1}^{-1} : GCL \times Cube \to Cube$ by

$$wep_{c_1}^{-1}(\mathsf{cmd}, c_2) \\ = \bigwedge\{lit_j \mid m(\varphi_j) = lit_j \wedge \varphi_j \in c_2 \cap wep(\mathsf{cmd}^{\mathsf{assume}}, c_1)\}.$$

Example 8 Reconsider Ex. 7. After identifying $c_F = (y \geq 0 \wedge b = 0) \in F_{(2,\ell_4)}$, we can deduce the generalization of c along edge $e_4 = (\ell_4, \mathsf{assume}\ a \geq 0, \ell_6)$ from c_F. In this case, the mapping g is very simple: Since the GCL command of e_4 does not contain any assignments, it is the identity of the literals of c. Thus we first remove the literals in $wep(\mathsf{assume}\ a \geq 0, true) = (a \geq 0)$ from c_F, which yields $c_F \backslash wep(\mathsf{assume}\ a \geq 0, true) = \{y \geq 0, b = 0\}$. Given that $g = id$, we obtain the result

$$wep_{c_1}^{-1}(\mathsf{cmd}, c_2) = \bigwedge\{y \geq 0, b = 0\},$$

which is identical to the result that we would have obtained using IC3 generalization, but without the use of a solver.

Note that we define individual functions $wep_{c_1}^{-1}$ based on cube c_1 that is used to construct the mapping m.

Lemma 6 *For choice-free GCL command* cmd *and cubes $c_1 \subseteq c$, $c_2 \subseteq c$, it holds:*

$$wep_c^{-1}(\mathsf{cmd}, c_1 \wedge c_2) \\ \iff wep_c^{-1}(\mathsf{cmd}, c_1) \wedge wep_c^{-1}(\mathsf{cmd}, c_2)$$

and

$$c_1 \subseteq c_2 \implies wep_c^{-1}(\mathsf{cmd}, c_1) \subseteq wep_c^{-1}(\mathsf{cmd}, c_2).$$

Proof Both claims immediately follow from Def. 16:

$$\begin{aligned} & wep_c^{-1}(\mathsf{cmd}, c_1 \wedge c_2) \\ & = \bigwedge\{lit_j \mid m(\varphi_j) = lit_j \wedge \\ & \qquad \varphi_j \in (c_1 \wedge c_2) \cap wep(\mathsf{cmd}^{\mathsf{assume}}, c)\} \\ & \iff \bigwedge\{lit_j \mid m(\varphi_j) = lit_j \wedge \\ & \qquad \varphi_j \in c_1 \cap wep(\mathsf{cmd}^{\mathsf{assume}}, c)\} \wedge \\ & \quad \bigwedge\{lit_j \mid m(\varphi_j) = lit_j \wedge \\ & \qquad \varphi_j \in c_2 \cap wep(\mathsf{cmd}^{\mathsf{assume}}, c)\} \\ & \iff wep_c^{-1}(\mathsf{cmd}, c_1) \wedge wep_c^{-1}(\mathsf{cmd}, c_2) \end{aligned}$$

and

$$\begin{aligned} & c_1 \subseteq c_2 \\ & \implies \bigwedge\{lit_j \mid m(\varphi_j) = lit_j \wedge \\ & \qquad \varphi_j \in c_1 \cap wep(\mathsf{cmd}^{\mathsf{assume}}, c)\} \subseteq \\ & \quad \bigwedge\{lit_j \mid m(\varphi_j) = lit_j \wedge \\ & \qquad \varphi_j \in c_2 \cap wep(\mathsf{cmd}^{\mathsf{assume}}, c)\} \\ & \implies wep_c^{-1}(\mathsf{cmd}, c_1) \subseteq wep_c^{-1}(\mathsf{cmd}, c_2) \end{aligned}$$

□

Lemma 7 *Given GCL command* cmd *and cubes c and $\bar{c} = wep(\mathsf{cmd}, c)$, then*

1. *wep_c^{-1} is a left inverse of wep: $wep_c^{-1}(\mathsf{cmd}, \bar{c}) = c$,*
2. *$wep(\mathsf{cmd}, wep_c^{-1}(\mathsf{cmd}, \bar{c})) = \bar{c}$.*

Proof 1.

$$\begin{aligned} & wep_c^{-1}(\mathsf{cmd}, wep(\mathsf{cmd}, c)) \\ & = wep_c^{-1}(\mathsf{cmd}, \bigwedge_{lit \in c} wep(\mathsf{cmd}^{\mathsf{assume}}, lit) \\ & \quad \wedge wep(\mathsf{cmd}, true)) && \text{(Lem. 5)} \\ & = \bigwedge_{lit \in c} wep_c^{-1}(\mathsf{cmd}, wep(\mathsf{cmd}^{\mathsf{assume}}, lit)) \\ & \quad \wedge wep_c^{-1}(\mathsf{cmd}, wep(\mathsf{cmd}, true)) && \text{(Lem. 6)} \\ & = \bigwedge_{lit \in c} lit \wedge \bigwedge \emptyset && \text{(Def. 16)} \\ & = c. \end{aligned}$$

2.

$$\begin{aligned} & wep(\mathsf{cmd}, wep_c^{-1}(\mathsf{cmd}, \bar{c})) \\ & = wep(\mathsf{cmd}, \bigwedge\{m(\varphi) \mid \varphi \in \bar{c} \cap wep(\mathsf{cmd}^{\mathsf{assume}}, c)\}) \\ & && \text{(Def. 16)} \\ & = wep(\mathsf{cmd}, \bigwedge\{m(\varphi) \mid \varphi \in wep(\mathsf{cmd}^{\mathsf{assume}}, c)\}) \\ & = wep(\mathsf{cmd}, c) \\ & = \bar{c}. \end{aligned}$$

□

In the last step of the proof we use that $\bar{c}$ equals $wep(\mathsf{cmd}, c)$. Together with Lem. 5 this means that $\bar{c} \supseteq$

$wep(\mathsf{cmd}^{\mathsf{assume}}, c)$ and therefore $\bar{c} \cap wep(\mathsf{cmd}^{\mathsf{assume}}, c)$ equals $wep(\mathsf{cmd}^{\mathsf{assume}}, c)$. As a consequence, each literal φ of the $wep(\mathsf{cmd}^{\mathsf{assume}}, c)$ is mapped back to the original literal $lit \in c$, such that the conjunction over the set of literals is exactly c.

Using these results it follows that if clause $\neg c_F \in F_{(i-1,\ell)}$, and cube c_F is a subset of $wep(\mathsf{cmd}, c)$, then applying wep_c^{-1} to c_F yields a subset of the original cube c.

Lemma 8 *Let cube c be inductive relative to $F_{(i-1,\ell)}$ along $e = (\ell, \mathsf{cmd}, \ell')$. Then*

$$\exists c_F.\, \neg c_F \in F_{(i-1,\ell)} \wedge c_F \subseteq wep(\mathsf{cmd}, c)$$
$$\implies wep_c^{-1}(\mathsf{cmd}, c_F) \subseteq c.$$

Proof Follows from Lem. 6 and 7. □

We now show that if c is inductive relative to $F_{(i-1,\ell)}$, then the same holds for $wep_c^{-1}(\mathsf{cmd}, c_F)$.

Lemma 9 *Let cube c be inductive relative to $F_{(i-1,\ell)}$ along $e = (\ell, \mathsf{cmd}, \ell')$. Then:*

$$\exists c_F.\, \neg c_F \in F_{(i-1,\ell)} \wedge c_F \subseteq wep(\mathsf{cmd}, c)$$
$$\implies relInd(F_{(i-1,\ell)}, e, wep_c^{-1}(\mathsf{cmd}, c_F)).$$

Proof

$$\begin{aligned}
&\neg relInd(F_{(i-1,\ell)}, e, wep_c^{-1}(\mathsf{cmd}, c_F))\\
&\implies \neg relIndAlt(F_{(i-1,\ell)}, e, wep_c^{-1}(\mathsf{cmd}, c_F))\\
&\iff \neg unsat(F_{(i-1,\ell)} \wedge wep_c^{-1}(\mathsf{cmd}, c_F))\\
&\iff sat(F_{(i-1,\ell)} \wedge wep_c^{-1}(\mathsf{cmd}, c_F))\\
&\iff sat(F_{(i-1,\ell)} \wedge c_F)\\
&\implies \neg\left(\exists c_F \in F_{(i-1,\ell)}\right)\\
&\implies \neg\left(\exists c_F \in F_{(i-1,\ell)}.\, c_F \subseteq wep(\mathsf{cmd}, c)\right).
\end{aligned}$$
□

As a next step, we prove that applying wep_c^{-1} to c_F yields a valid generalization of c if c is inductive relative to $F_{(i-1,\ell)}$ and for some $\neg c_F$ in $F_{(i-1,\ell)}$, c_F is a subset of $wep(\mathsf{cmd}, c)$.

Theorem 7 *Let cube c be inductive relative to $F_{(i-1,\ell)}$ along $e = (\ell, \mathsf{cmd}, \ell')$. Then:*

$$\exists c_F.\, \neg c_F \in F_{(i-1,\ell)} \wedge c_F \subseteq wep(\mathsf{cmd}, c)$$
$$\implies wep^{-1}(\mathsf{cmd}, c_F) \in gen(F_{(i-1,\ell)}, e, c).$$

Proof Directly from Lem. 8 and 9. □

This result allows to enhance the search for predecessor cubes (see Thm. 6) by considering wep_c^{-1} of c_F in order to obtain a generalization for cube c. This does not involve any SMT solver queries.

5.4 Ordering obligations

Let us reconsider the *split* operation from Def. 14. Let edge $e = (\ell, \mathsf{cmd}, \ell')$ where cmd contains a choice operator. The function *split* returns a set of GCL commands that can be considered as parallel edges in the CFA. Whenever we encounter such an edge e and cube c is not inductive relative to $F_{(i-1,\ell)}$, we obtain a set of commands $\{\mathsf{cmd}_1, \ldots, \mathsf{cmd}_n\}$ to each of which *wep* is applied, resulting in a set of predecessor cubes $\{c_1, \ldots, c_n\}$ of c all being one-step predecessors of states in c. We have to create proof obligations for each of these cubes. This raises the question whether the order in which obligations for the same index are handled matters and if so, which order is optimal or at least best overall.

Since the created obligations only differ in the cube they contain, we have to find some order based on a metric for the "quality" of the cube. Like in standard IC3, we aim to find generalizations that block large regions of states, i.e., those that contain the fewest literals. For that reason we prefer cubes with few literals, since these cubes more likely produce small generalizations. To that end, we change the priority queue of obligations to order cubes based on ascending index and for obligations with the same index sort them in ascending order of their cube's size. The insight that employing this order performs well has also been confirmed in [23]. In [23], proof obligations are also created for cubes that may be helpful to be proven inductive in order to block larger regions of the state space. Since multiple of these so-called *may*-obligations can be created for the same index, the problem of ordering obligations also arose.

5.5 Caching generalizations

As seen in this section, IC3CFA makes heavy use of the WEP for predecessor computation, which enables multiple optimizations to the generalization procedure of IC3CFA. Apart from exploiting the presence of exact predecessors as seen above, we will now focus on a different aspect: When looking at the process of how IC3CFA searches the state space for counterexamples and how it creates stepwise reachability information in the frame sequence, we can see that due to the deterministic search on the CFA, as well as the precise predecessor computation, we determine the same cubes over and over again, but at different indices, leading to repeated generalizations of the same cube. This suggests to cache generalizations. This is accomplished by means of so-called *generalization contexts*, which store for a given cube and edge the context of the generalization.

Definition 17 *(Generalization context)* A generalization context GC_i at index i contains quadruples (c, e, F, g) where e is an edge, F is a frame and $g \subseteq c$ generalizes cube c.

A generalization context stores the exact setting in which a cube c was generalized to cube g. Generalizations at index j are available in each generalization context GC_i with $i \geq j$. As a first step, this cached information is used if we generalize cube c at index i and $(c, e, F, g) \in GC_i$ relative to $F = F_{(i-1,\ell)}$ and edge e to the generalized cube g. This however will not produce many cache hits, since frames evolve and change over time. As frames only grow, the set of states represented by frame F decreases. Therefore, if no state in F has a successor state in cube c (and g), then no frame $F' \supseteq F$ has a successor state in g.

Theorem 8 *Let edge $e = (\ell, \mathsf{cmd}, \ell')$ and c be a cube to be generalized at index i and ℓ' relative to $F_{(i-1,\ell)}$ and e. Then:*

$$\begin{aligned}&(c, e, F, g) \in GC_i \wedge F \subseteq F_{(i-1,\ell)}\\ &\quad\implies g \in gen(F_{(i-1,\ell)}, e, c).\end{aligned}$$

where again $gen(F_{(i-1,\ell)}, e, c)$ denotes the set of valid generalizations of cube c relative to frame $F_{(i-1,\ell)}$ and edge e.

Proof

$$\begin{aligned}&(c, e, F, g) \in GC_i \wedge F \subseteq F_{(i-1,\ell)}\\ &\quad\implies \exists\, j \leq i.\ g \in gen(F_{(j-1,\ell)}, e, c)\\ &\qquad \wedge F = F_{(j-1,\ell)} \wedge F_{(j-1,\ell)} \subseteq F_{(i-1,\ell)}\\ &\quad\implies relInd(F_{(j-1,\ell)}, e, g) \wedge F_{(j-1,\ell)} \subseteq F_{(i-1,\ell)}\\ &\quad\implies unsat(F_{(j-1,\ell)} \wedge T_e \wedge g') \wedge F_{(j-1,\ell)} \subseteq F_{(i-1,\ell)}\\ &\quad\implies unsat(F_{(i-1,\ell)} \wedge T_e \wedge g')\\ &\quad\implies relInd(F_{(i-1,\ell)}, e, g)\\ &\quad\implies g \in gen(F_{(i-1,\ell)}, e, c).\end{aligned}$$ □

This result enables us to use entry (c, e, F, g) of the generalization context GC_i if c is to be generalized along edge e relative to a stronger frame $F_{(i-1,\ell)} \supseteq F$. While this improves the number of cache hits dramatically, we should not immediately block g as the generalization of c. Even though g is a valid generalization, as $g \subseteq c$ and is relative inductive to $F_{(i-1,\ell)}$, it may not be optimal and in practice it rarely is. Due to the stronger frame $F_{(i-1,\ell)} \supseteq F$, there are now many more states blocked in $F_{(i-1,\ell)}$ than there were in F, such that larger cubes $\bar{g} \subseteq g$ are now one-step unreachable from $F_{(i-1,\ell)}$. We call the cube g to which Thm. 8 applies an *upper bound* of the final generalization of c.

Example 9 Assume that the generalization from Ex. 8 has happened and has been cached. Furthermore, assume $F_{(2,\ell_4)}$ has grown from $F_{(2,\ell_4)} = \neg(y \geq 0 \wedge b = 0)$ to $F_{(2,\ell_4)} = \neg(y \geq 0 \wedge b = 0) \wedge \neg(a\%b = 0 \wedge b \neq 0)$ (with the new lemma stemming from one iteration of the loop). Our aim is to generalize $c = (y \geq a \wedge y > 0 \wedge b = 0 \wedge b \geq 0)$ along $e_4 = (\ell_4, \mathsf{assume}\ a \geq 0, \ell_6)$ relative to the new $F_{(2,\ell_4)}$. Then we can automatically deduce that we only need to check whether we can drop the literals $(y \geq 0)$ or $(b = 0)$. This saves two SMT queries. However, we can also see that it is in fact advisable to check the remaining literals, since we can drop the literal $(b = 0)$, too, after blocking $(a\%b = 0 \wedge b \neq 0)$.

We can improve the *upper bound* for the generalization of cube c even further. So far, we have only used entries $(c, e, F, g) \in GC_i$ for identical matches of cube c. We can however also use $(\hat{c}, e, F, g) \in GC_i$ for cubes $g \subseteq c$ and some arbitrary cube $\hat{c}$ and frames $F \subseteq F_{(i-1,\ell)}$ to increase the level of information reuse even further.

Corollary 5 *Let edge $e = (\ell, \mathsf{cmd}, \ell')$ and let c be a cube to be generalized at index i and ℓ' relative to $F_{(i-1,\ell)}$ and e. Then*

$$\begin{aligned}&\big((\hat{c}, e, F, g) \in GC_i \wedge F \subseteq F_{(i-1,\ell)} \wedge g \subseteq c\big)\\ &\quad\implies g \in gen(F_{(i-1,\ell)}, e, c).\end{aligned}$$

Example 10 Consider the generalization of cube $c = (y \geq a \wedge y > 0 \wedge -b = 0 \wedge b < 0)$ that originates from the backward trace $\ell_8 \rightarrow \ell_7 \rightarrow \ell_6$ along edge $e_4 = (\ell_4, \mathsf{assume}\ a \geq 0, \ell_6)$. Let $F_{(2,\ell_4)} = \neg(y \geq 0 \wedge b = 0) \wedge \neg(a\%b = 0 \wedge b \neq 0)$ and $(\hat{c}, e, F, g) \in GC_i$ with

$$\begin{aligned}\hat{c} &= (y \geq a \wedge y > 0 \wedge b = 0 \wedge b \geq 0),\\ e &= (\ell_4, \mathsf{assume}\ a \geq 0, \ell_6),\\ F &= \neg(y \geq 0 \wedge b = 0),\ \text{and}\\ g &= (y \geq 0 \wedge b = 0).\end{aligned}$$

Since $F \subseteq F_{(2,\ell_4)}$ and $g \subseteq c$, the generalization g provides an upper bound on the literals for the generalization of c.

In addition to *upper bounds* on the literals for the generalization of cube c, the generalization context GC_i can be used to derive a *lower bound* on the literals. Intuitively, these are the literals of which we know that they definitely cannot be dropped from c. The intuition is that if we encounter a generalization context (c, e, F, g) with $F_{(i-1,\ell)} \subseteq F$, then we assume that each $\widehat{g} \subseteq g$ was not a valid generalization relative to F. Since $F_{(i-1,\ell)} \subseteq F$, such $\widehat{g}$ is not a valid generalization relative to $F_{(i-1,\ell)}$ either, and thus no literal in g can be dropped from c.

Theorem 9 *Let edge $e = (\ell, \mathsf{cmd}, \ell')$ and let c be a cube to be generalized at index i and ℓ' relative to $F_{(i-1,\ell)}$ and e. Then:*

$$\begin{aligned}&\big((c, e, F, g) \in GC_i \wedge F_{(i-1,\ell)} \subseteq F \wedge \widehat{g} \subset g\big)\\ &\quad\implies \widehat{g} \notin gen(F_{(i-1,\ell)}, e, c)\\ &\quad\implies g \subseteq g_{fin} \subseteq c,\ \forall g_{fin} \in gen(F_{(i-1,\ell)}, e, c).\end{aligned}$$

Proof

$$\begin{aligned}
&(c, e, F, g) \in GC_i \wedge F_{(i-1,\ell)} \subseteq F \wedge \widehat{g} \subset g \\
&\quad \Longrightarrow \exists j \le i.\, g \in gen(F_{(j-1,\ell)}, e, c) \wedge F = F_{(j-1,\ell)} \\
&\quad \Longrightarrow \exists j \le i.\, \widehat{g} \notin gen(F_{(j-1,\ell)}, e, c) \wedge F = F_{(j-1,\ell)} \\
&\quad \Longrightarrow \neg relInd(F_{(j-1,\ell)}, e, \widehat{g}) \\
&\quad \Longrightarrow \neg relInd(F_{(i-1,\ell)}, e, \widehat{g}) \qquad (F' \subseteq F) \\
&\quad \Longrightarrow \widehat{g} \notin gen(F_{(i-1,\ell)}, e, c).
\end{aligned}$$
□

Example 11 Let $c = (y \ge a \wedge y > 0 \wedge b = 0 \wedge b \ge 0)$ to be generalized relative to $F_{(2,\ell_4)} = \neg(y \ge 0 \wedge b = 0)$ along $e_4 = (\ell_4, \text{assume } a \ge 0, \ell_6)$. Assume that $(c, e, F, g) \in GC_i$ with

$$\begin{aligned}
e &= (\ell_4, \text{assume } a \ge 0, \ell_6), \\
F &= \neg(y \ge 0 \wedge b = 0) \wedge \neg(a\%b = 0 \wedge b \ne 0), \text{ and} \\
g &= (y \ge 0).
\end{aligned}$$

By Thm. 9, $(y \ge 0)$ *must* be part of the final generalization of c. If additionally some entry in GC_i gives an upper bound like $(y \ge 0 \wedge b = 0)$, we effectively only have to check a single literal for dropping, reducing the number of SMT checks in this case by 75%.

Combining the caching of generalization contexts and the presented methods to derive *lower bounds* and *upper bounds* on the literals to be generalized can save a significant amount of effort in generalization. This is due to the fact that the set of literals that needs to be checked is decreased.

6 Tool implementation

After introducing several optimizations for IC3 that adapt it toward software model checking, this section gives an account of the verifier's architecture.

6.1 Architecture

IC3CFA and all optimizations described in Sect. 5 have been implemented on top of an existing proprietary framework for software verification, which is not publicly available. Figure 5 depicts the verifier's core architecture: We assume that the input programs are already augmented with the specification, i.e., assumptions and assertions. Several input languages are supported, including a subset of C, a proprietary intermediate language, and a textual representation of the verifier's own intermediate verification language (IVL). The latter is used as a unifying representation of the input program. The C language fragment features complex data structures such as (nested) records and arrays as well as pointer arithmetic. It excludes recursive procedures and procedure pointers (as procedures are handled by inlining of calls) and dynamic memory allocation (as it is generally not used in our application setting).

First, several model minimization techniques are applied to reduce the size and complexity of the input program. Subsequently, the resulting IVL program is mapped to a CFA representation. Finally, one can select one of several memory models that express the semantics of IVL language constructs as guarded command language expressions and predicate logic formulae. Finally, large-block encoding [4] is applied to further reduce the number of locations in the CFA before it is fed to the verification algorithms. Besides IC3CFA, the verifier includes engines for bounded model checking (BMC), counterexample-guided abstraction refinement (CEGAR) and TreeIC3 [14]. In case a property violation is found by an engine, the counterexample is given in a common representation that allows us to reconstruct diagnostic feedback for the user.

All engines rely on an SMT solver interface which allows us to direct decision problems to different solvers (currently Z3 [35], CVC4 [3], and MathSAT5 [16]) and also to make use of solver-specific features such as, e.g., Craig interpolation, UNSAT-core extraction, and optimized AllSAT computations to improve generalization [8,10].

6.2 Preprocessing

As depicted in Fig. 5, the input program is pre-processed before it is given to the verification engine. Data flow analyses that are applied on the level of the intermediate verification language include points-to, initialized variables, needed variables, and reaching definitions analysis [36]. The analysis results drive program transformations such as forward propagation [31], constant folding, and program slicing. In particular, the latter removes all parts of the program whose execution does not have any impact on the properties to be verified. From our practical experience, the resulting reduction of the size of the verification problem leads to noteworthy performance improvements in the overall verification process.

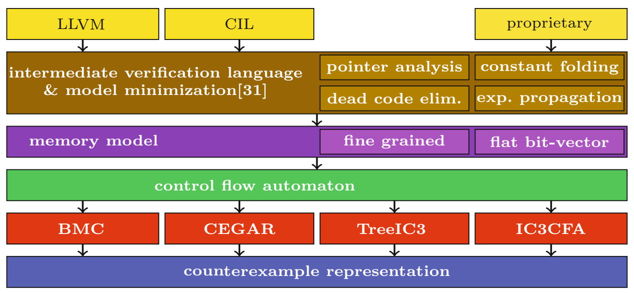


Fig. 5 Tool architecture

Once the input program has been simplified, we construct its CFA representation where large-block encoding (LBE) [4] is employed to reduce the number of program locations even further by applying the following transformation rules:

1. The *error sink* rule removes all outgoing edges of the error location ℓ_E, which is justified by the fact that we only consider safety properties. It needs to be applied only once.
2. The *sequence* rule merges consecutive edges into a single edge. More formally, if $e_0 = (\ell, \mathsf{cmd}, \ell')$ and $e_i = (\ell', \mathsf{cmd}_i, \ell_i)$ for some $i > 0$, LBE introduces new edges $e_i' = (\ell, \mathsf{cmd};\ \mathsf{cmd}_i, \ell_i)$ and removes e_0 as well as all e_i from the CFA.
3. The *choice* rule merges parallel edges. Any two edges of the form $e_1 = (\ell, \mathsf{cmd}_1, \ell')$ and $e_2 = (\ell, \mathsf{cmd}_2, \ell')$ are replaced by a single edge $e = (\ell, \mathsf{cmd}_1 \,\square\, \mathsf{cmd}_2, \ell')$, effectively moving the nondeterministic choice from CFA- to GCL-level.

In principle, rules 2 and 3 could be applied until a fixed point is reached. However, as also noted in [5], there is a practical trade-off between the number of program locations that are kept and the complexity of the GCL statements that determine the transition semantics. Therefore we use a heuristic that depends on a complexity measure of the CFA's edge labels to determine when to stop the LBE reduction phase.

6.3 Implementation details

Cubes vs. clauses and delta-encoding. The property-directed reachability (PDR) algorithm from [21] improves upon the data structures proposed in the original paper on IC3 [10]:

– In [10], each frame in the sequence $F_1, \ldots, F_n$ is represented in conjunctive normal form, i.e., as a set of clauses where each clause symbolically encodes admissible states. Conversely, each frame in a sequence $R_1, \ldots, R_n$ of PDR is in disjunctive normal form where each cube encodes a set of blocked states. The two are dual, i.e., $F_i \equiv \neg R_i$ and taken individually, the particular choice for representing a frame sequence does not alter the IC3/PDR algorithm.
– Whereas [10] explicitly stores the set of clauses that constitute a frame, PDR uses delta-encoding. More formally, let $F_1, \ldots, F_n$ be a frame sequence with $F_i \Rightarrow F_{i+1}$ for $1 \leq i < n$. The latter implication is also syntactic [10, Sect. 4], i.e., it holds that $clauses(F_{i+1}) \subseteq clauses(F_i)$. The same holds for PDR where $\neg R_i \Rightarrow \neg R_{i+1}$. However, the fact that $cubes(R_i) \subseteq cubes(R_{i+1})$ allows us to use a more efficient (delta) encoding: Instead of storing all cubes explicitly, any frame R_{i+1} is uniquely determined by recording the cubes in $R_{i+1} \setminus R_i$. Hence, each frame in PDR only stores those cubes that have not been blocked in a preceding frame.

Both IC3 and PDR only need to store a single frame sequence, whereas IC3CFA stores one such sequence for each control location of the program. Hence, we heavily rely on delta-encoding to efficiently represent frame sequences. Moreover, to reduce the risk of polluting frames by including redundant, i.e., semantically equivalent but syntactically different cubes, we use theory-aware rewrite rules that simplify and partly normalize cubes before they are included in a frame.
Caching. For caching SMT queries, our implementation uses least-recently used (LRU) caches in almost all situations.
Counterexamples. A practical aspect not considered in [10] and [21] is the extraction of counterexamples. To do so, we implement the method for counterexample generation as described in Sect. 4.3.

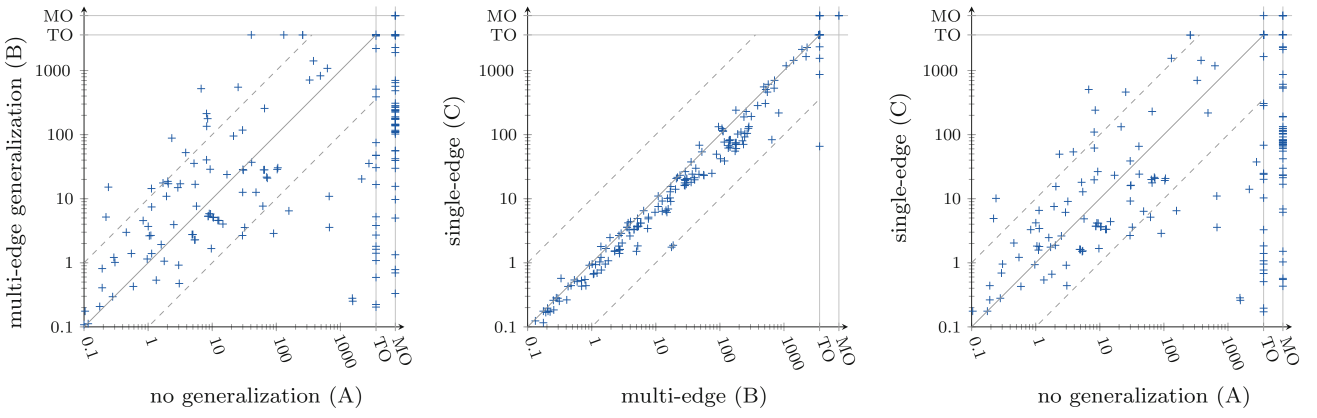
7 Evaluation

In what follows, we quantify the effect of the optimizations that have been introduced so far and investigate how they relate to each other. The section concludes with a short comparison of our verifier's performance with other state-of-the-art software model checkers that use inductive incremental verification schemes.

7.1 Setup

All benchmarks are executed on a host system with Intel® Xeon E5-2670 CPUs at 2.3 GHz, without hyper-threading, and with 64 GB of main memory. The verifier uses Z3 version 4.8.3 (as of commit `#ccf6ca3`) for all queries except for Craig interpolation where MathSAT5, version 5.5.2, is used. Each verification run is restricted to a single core and a maximum memory usage of 3000 MB. The time limit was set to one hour. All measurements were conducted using the BenchExec tool [6] which was configured to enforce the above constraints.

A total of 406 C files were chosen from different sources. These include all benchmarks from [14]. In addition, we selected several categories from the annual SV-COMP competition [39], in particular from the ReachSafety category. More precisely, the subcategories for bit-vectors, control flow, floats, and loops are included in the results that we report. Other subcategories such as the one on product lines have been omitted, as the respective C files employ language constructs that our frontend does not support, such as recursive procedures or dynamic memory allocation.



(a) multi-edge generalization **(b)** single- vs. multi-edge generalization **(c)** single-edge generalization

Fig. 6 Comparing no, single-edge, and multiple-edge generalization

7.2 Configurations

Table 1 summarizes the different configurations of the verifier that are used in the subsequent discussion. The columns are labeled with the verifier's command line switches that can be used to activate the corresponding technique.

7.3 Discussion of results

Effect of generalization. In order to compare the performance impact of several generalization strategies, we start by comparing the verification times of IC3CFA without generalization with those obtained by generalizing along all incoming edges (cf. Sect. 5.2). The scatter plot in Fig. 6a depicts the results and clearly shows the benefit of generalization. In particular, there is a large number of verification tasks that can only be solved using generalization.

However, generalization does not reduce the verification time for those instances that can be solved without generalization. This comes as no surprise, as generalization is computationally expensive; in particular, it induces additional invocations of the underlying decision procedure. If a proof succeeds without generalization, its incurred overhead does not pay off.

Single- vs. multiple-edge generalization. Corollary 3 improves generalization by replacing the rather complex inductivity queries for multi-edge generalization with several simpler queries that check inductivity along single edges. The scatter plot in Fig. 6b supports the intuition that breaking down larger queries improves performance. In particular, one can accumulate those literals that cannot be dropped while processing edges. This reduces the literals that remain candidates for generalization and thereby the number of calls to the decision procedure. As a result, single-edge generalization succeeds on all verification instances that multi-edge generalization is able to decide and on another six instances where multi-edge generalization fails.

Figure 6c depicts the effect of single-edge generalization compared to an approach that does not generalize at all.

Cube ordering within proof obligations. As mentioned in Sect. 5.4, DNF-splitting of WEP formulae gives rise to multiple cubes that need to be discharged within one proof obligation. To evaluate the effect of a particular order in which those cubes are processed, we order them based on their cardinality and compare an increasing, a decreasing, and a random order (in Table 1 denoted by $\sqsubseteq$, $\sqsupseteq$, and ?, respectively). The scatter plots in Fig. 7 confirm the results from [23] that processing cubes by increasing cardinality yields the overall best result.

Obligation reuse. Another optimization is obligation reuse, i.e., remembering obligations from previous iterations and reusing them to proceed the backward search where the previous iteration was stopped due to the depth bound k. The scatter plot in Fig. 8a clearly shows that verification times generally improve. Moreover, the positive impact of obligation reuse grows for larger verification times. This can be explained as follows: For harder tasks, the search depth k and the amount of obligations that are recomputed without obligation reuse increase. Hence, for those instances, obligation reuse allows us to save large numbers of computationally expensive calls to the underlying decision procedure. Accordingly, the scatter plot in Fig. 8b shows significant savings in SMT solving times, especially for long-running instances.

Our preliminary conclusion is that both cube sorting in increasing order and obligation reuse generally improve performance. Moreover, generalization enables significantly more verification tasks to be solved, where the single-edge approach turns out to be superior to multi-edge generalization. We therefore use configuration F, which employs all

Table 1 Benchmark configurations

	Generalization mode `-ic3-generalize`	Cube order in obligations `-ic3-obligation-cube-order`	Obligation reuse `-ic3-obligation-reuse`	WEP-based ind. queries `-ic3-wp-for-inductivity`	Pre-cubes `-ic3-generalization-from-predecessor-frame`	Generalization caching `-ic3-generalization-caching`	Lower bounds `-ic3-generalization-caching-lower-bounds`	Interpolation `-interpolate`	Semantic subsumption `-ic3-block-semantic`
A	None	$\sqsubseteq$							
B	Multi	$\sqsubseteq$							
C	Single	$\sqsubseteq$							
D	Single	$\sqsupseteq$							
E	Single	?							
F	Single	$\sqsubseteq$	✓						
G	Single	$\sqsubseteq$	✓	✓					
H	Single	$\sqsubseteq$	✓		✓				
I	Single	$\sqsubseteq$	✓			✓			
J	Single	$\sqsubseteq$	✓			✓	✓		
K	Single	$\sqsubseteq$	✓					✓	
L	Single	$\sqsubseteq$	✓	✓	✓	✓	✓		
M	Single	$\sqsubseteq$	✓	✓	✓	✓	✓		✓

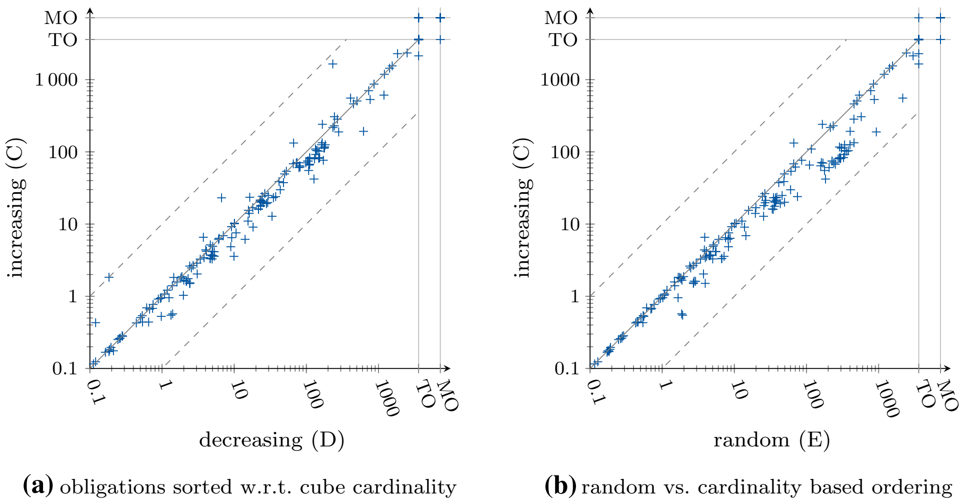


Fig. 7 Order in which cubes are discharged from proof obligations

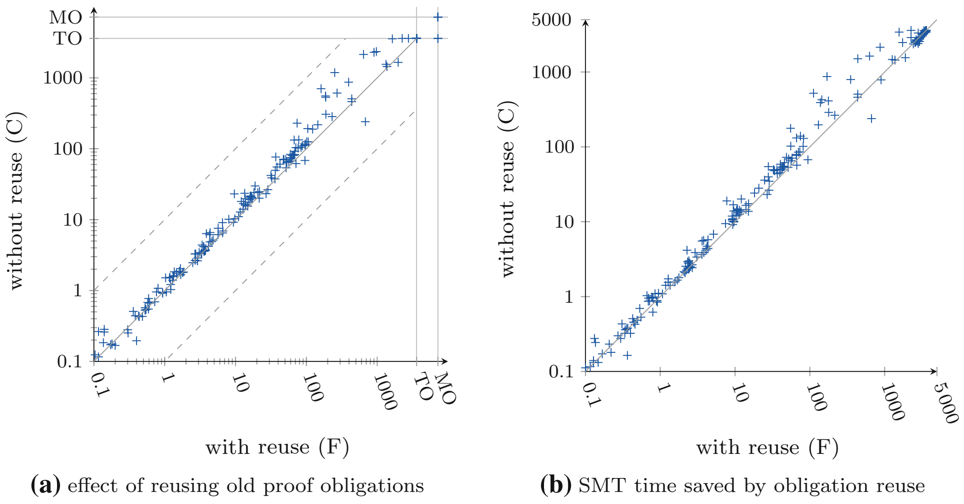


Fig. 8 Effect of reusing old proof obligations

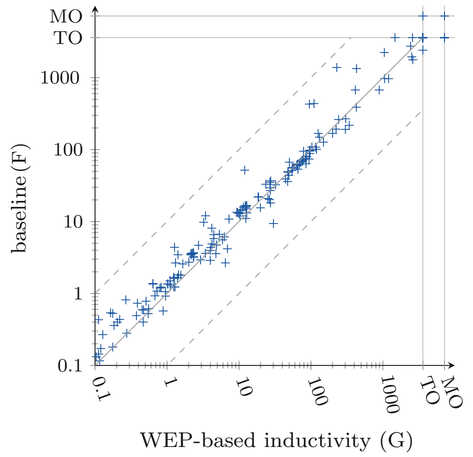
these features (cf. Table 1), as the baseline for all subsequent comparisons with other configurations.

WEP-based inductivity. Figure 9a depicts the evaluation of the WEP-based inductivity query as shown in Thm. 4. As we can see, the results are not equally good for all inputs. In fact, Fig. 9a shows that WEP-based inductivity is favorable for small and medium hard inputs with verification times up to around 100 seconds, where the results start to become indefinite with many instances performing slightly worse with WEP-based inductivity, but also some instances performing much better and one instance being almost one order of magnitude faster with WEP-based inductivity. In addition we can see from Fig. 9a that WEP-based inductivity is able to solve one instance shortly before the timeout that occurred without WEP-based inductivity. Due to the large performance gains for many instances and the marginal performance deterioration for only a few, we consider WEP-based inductivity in total as a helpful technique.

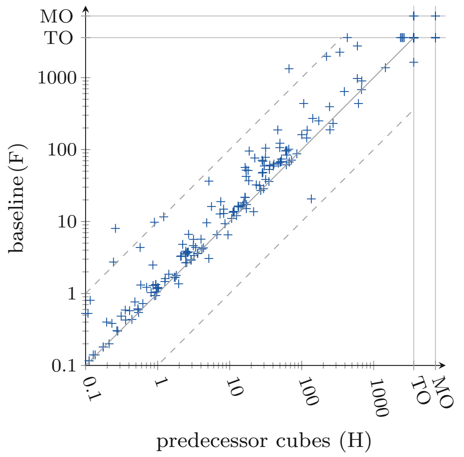
Predecessor cubes. Next, we evaluate the isolated effect of predecessor cubes as presented in Thm. 7, by comparing the *Baseline* configuration with one that is identical except that the static predecessor cube generalization is enabled. The resulting scatter plot is depicted in Fig. 9b and nicely illustrates how much performance can be gained by static optimizations. Except for one noticeable outlier, almost all other instances can be verified faster with two instances even more than one order of magnitude faster with predecessor than *Baseline*. Furthermore, we can see that using predecessor cubes, IC3CFA is able to solve four instances that run into a timeout in *Baseline*.

Generalization context. As presented in Sect. 5, the generalization context (Def. 17) can be used in several ways. We start with the simple version as shown in Thm. 8 where we just check whether a subset of the current frame has been stored in the context and use the result as upper bound on the literals of the new generalization. As we can see from the corresponding scatter plot in Fig. 10a, the effect is negligible for easy instances and the curve gets a slight bump for harder benchmarks but converges to the diagonal near the timeout. This effect is caused by the overhead introduced by manag-

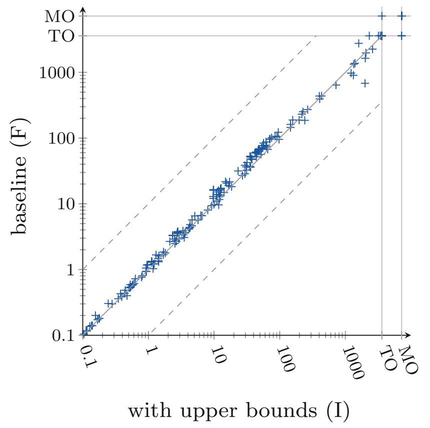
Fig. 9 Effects of WEP-based inductivity queries and predecessor cubes



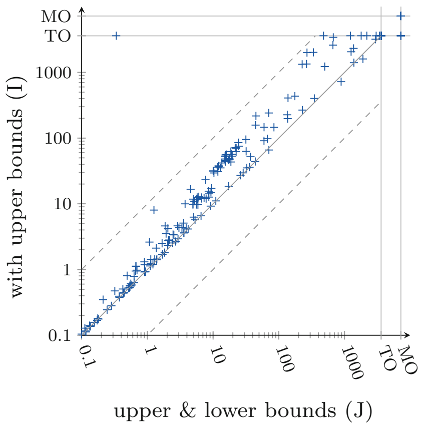
(a) WEP for inductivity queries



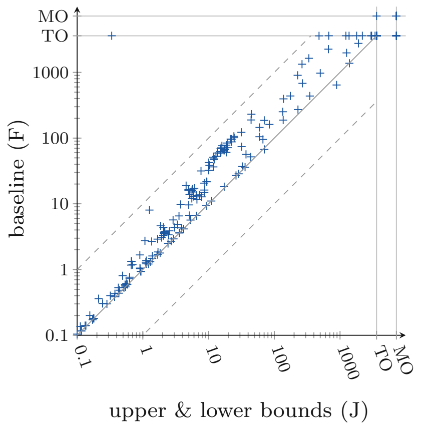
(b) predecessor cubes for generalization



(a) generalization context (upper bounds)



(b) isolated



(c) upper & lower bounds

Fig. 10 Generalization contexts for caching upper and lower bounds

ing and filling the cache and searching all the cache entries. For small/easy instances, the overhead of filling and managing the cache compensates the benefits given by the few cache hits. The larger/harder the instances get, the more cache hits occur and outweigh the costs of managing the cache. However, when the cache gets full and especially when the generalization contexts contain very large frames, the subset search becomes more and more costly until at some point it again outweighs the positive effects.

Figure 10b depicts the scatter plot comparing configurations I and J (see Table 1), which allows us to evaluate the isolated effect of lower bounds obtained from the generalization context as they are presented in Thm. 9. We can see that the configuration with lower bounds is better in every single instance, which is due to the fact that lower bounds use the same caches as upper bounds, such that the overhead of managing the cache incurs only once and therefore each cache hit improves the performance. In addition, there are some instances that can be solved with lower bounds but that would have otherwise run into a timeout.

To set the isolated results of lower and upper bounds into context, Fig. 10c compares the baseline with a configuration with both upper and lower bounds enabled. Here, we can see that the marginal performance gain of upper bounds, which is mainly due to the overhead of managing the cache, is heavily boosted by adding lower bound extraction, such that all instances benefit from generalization caching with many instances almost reaching one order of magnitude improvement and additionally many new instances can be solved.

Negative findings. Using Craig interpolation for generalization turns out to have a negative effect on verification times, as depicted in Fig. 11a. We ascribe this to the computational cost associated with finding interpolants. In particular, most instances use a bit-vector encoding of program variables where Craig interpolation is known to be difficult [2].

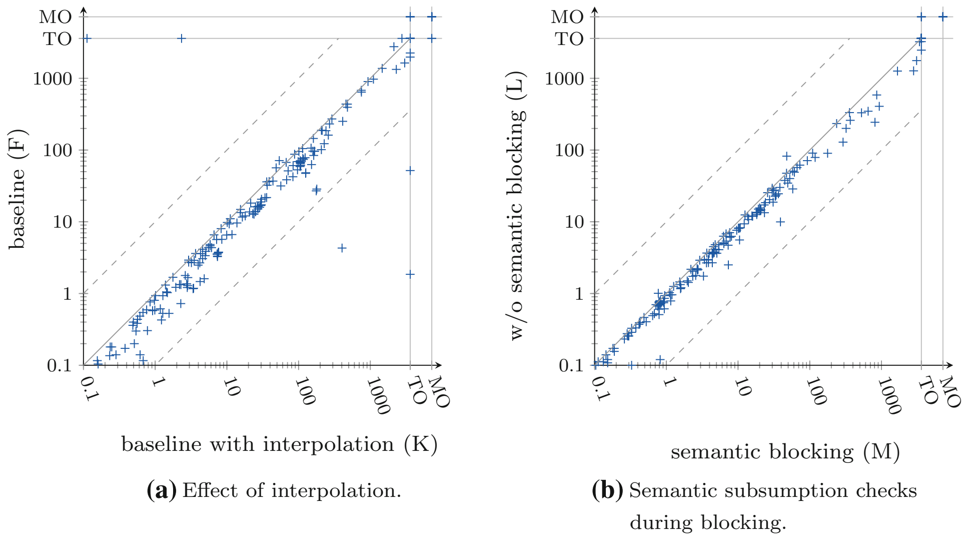


Fig. 11 Negative findings regarding Craig interpolation and semantic subsumption checks

(a) Effect of interpolation. **(b)** Semantic subsumption checks during blocking.

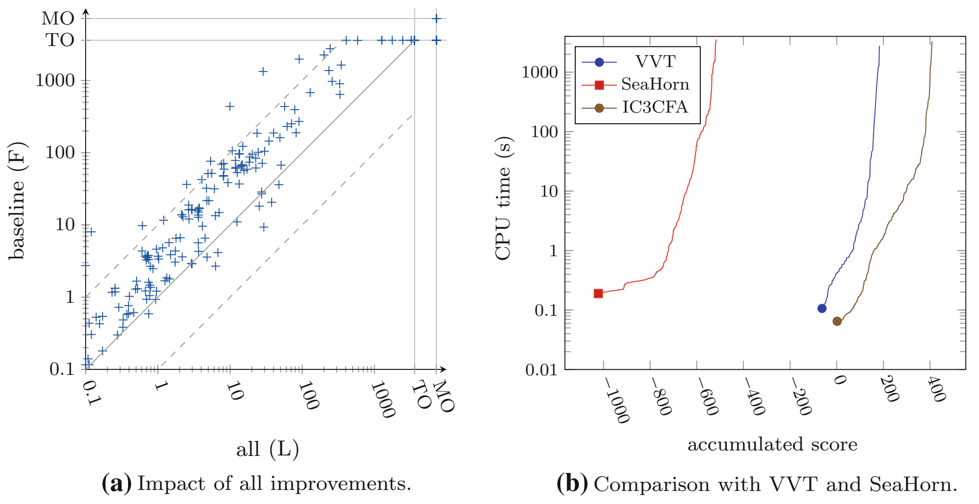


Fig. 12 Overall evaluation

(a) Impact of all improvements. **(b)** Comparison with VVT and SeaHorn.

Moreover, we evaluate a configuration which introduces additional semantic subsumption checks during the blocking phase. However, their additional overhead does not pay off either, as depicted in Fig. 11b.

Combining the optimizations. The scatter plot depicted in Fig. 12a compares a configuration that makes use of all beneficial optimizations (that is, only omits interpolation and semantic subsumption, i.e., configuration L in Table 1) against our baseline (F). As we can see, verification times improve for almost all instances. In particular, many more instances can be solved and some speedups are by orders of magnitude. This configuration is therefore chosen as the baseline for the following comparison with other tools.

7.4 Comparison with other software model checkers

In the following, we compare the performance of the verifier (in configuration L) with other state-of-the-art tools that implement IC3 algorithms for software model checking. The only available tools are the Vienna Verification Tool (VVT, [22]), which implements the CTIGAR approach, and SeaHorn [24] with Spacer/Z3PDR. We compare against the version and configuration that participated in SV-COMP 2016.[3] All tools are executed on the same machine and with the same time and memory limits as before.

To graphically illustrate the comparison between the different tools, we use a score-based quantile plot. The score is given according to the rules of SV-COMP [39], which are widely accepted. It is determined as follows: A correct result reporting a violation of the property (*correct FALSE*) gives a score of one point, while a correct result reporting no property violation (*correct TRUE*) scores two points. This is supposed to balance the hardness of the verification task, as finding an error is generally considered easier than proving the absence of errors, which requires completeness. If, how-

[3] Neither SeaHorn nor VVT took part in the subsequent competitions SV-COMP'17 or SV-COMP'18.

ever, the results given by the verification tool are incorrect, an *incorrect FALSE* result gives a penalty of 16 points, while for an *incorrect TRUE* result 32 points are deducted. Again, an incomplete analysis that finds a spurious error will just cause some additional work of the developer to find out that it was a *false alarm*, where a missed bug due to an unsound analysis will give a dangerous confidence that everything is safe and thus has to be punished harder.

The score-based quantile plot is computed as follows: The graph itself depicts only correct results, which are ordered based on their runtime and for each of these results, the score is added up. In other words, each point (x, y) on the graph means that all verification runs up to y seconds achieve a total score of x points. To incorporate the penalties for incorrect results, it does not matter how fast they were at producing the incorrect result, such that we can ignore the time and just take the sum of all penalties collected. This summed penalties determine the value by which the graph is shifted along the x-axis.[4]

As we can see from Fig. 12b, the curve of SeaHorn is less steep than the one of VVT; hence SeaHorn solves more instances in a shorter amount of time. Moreover, its graph is wider, which indicates that SeaHorn is able to solve more instances than VVT. But we can also see that SeaHorn's graph is shifted much more to the left: SeaHorn collected 1024 penalty points for 48 incorrect "false" and eight incorrect "true" results. As SeaHorn makes heavy use of LLVM's compiler optimizations to simplify the input problem, we suspect that too optimistic program transformations on the LLVM level lead to this high number of incorrect results; at least, similar effects have been discussed in [1]. In contrast, VVT only gives two incorrect *TRUE* and no incorrect *FALSE*, resulting in 64 penalty points. Finally, we observe that IC3CFA does not produce any incorrect results, hence starts at a score of zero and achieves the highest score of all, ending at 410 points for 161 correctly proven and 88 correctly disproven instances.

8 Related work

In the previous sections we illustrated the process of lifting the IC3 algorithm to software model checking on the example of IC3-SMT and Tree-IC3 [14]. To complete the presentation of our IC3CFA algorithm, we will have a brief look at a selection of other algorithms for IC3-style software model checking.

We start with the IC3+IA algorithm of [15], which builds upon the work of [14]. The main motivation of IC3+IA is to integrate IC3 with (predicate) abstraction, in this case *Implicit Abstraction* (IA) as presented in [38]. In the interaction between IC3 and IA, IC3 operates purely on the Boolean level of the abstracted state space and discovers inductive clauses over the abstraction predicates. Like IC3CFA, IC3+IA is able to handle a variety of background theories because it does not rely on ad hoc extensions such as quantifier elimination or theory-aware generalization procedures. But in contrast to IC3CFA, the IC3+IA algorithm does not explicitly represent the transition relation but keeps it symbolic. In addition, the use of IA allows IC3+IA to abandon the explicit computation of the abstract system. The algorithm proceeds as follows: Given some set of predicates $\mathbb{P}$, we consider the Boolean state space that is spun by $\mathbb{P}$ and apply standard IC3 on it, with the exception that the transition relation is given in terms of IA in order to avoid quantifier elimination. To check inductivity of a $\mathbb{P}$-cube in this state space relative to a $\mathbb{P}$-frame, [15] introduces a modified check that includes the lifting to IA. Due to the use of predicate abstraction, a counterexample found in the search phase of IC3 may be spurious, i.e., it may not be a counterexample on the concrete state space. This situation is common to predicate abstraction and is resolved by simulating the given abstract counterexample path on the concrete system. If the counterexample also exists in the concrete system, IC3+IA detected a real counterexample. If, however, the counterexample does not exist in the concrete system, the set of predicates must be refined, just like in other predicate abstraction settings. Based on this new set of predicates, IC3 is executed again. A distinct feature of IC3+IA is that it only adds new predicates to $\mathbb{P}$, such that $\mathbb{P}$ monotonically increases, which allows IC3+IA to keep all learnt clauses and rather than restarting IC3, it can just be continued. In [15] the authors also evaluate a combination of Tree-IC3 and IA, but while being more efficient than Tree-IC3 with interpolation (the best configuration from [14]) it is outperformed by IC3+IA.

Counterexample to induction-guided abstraction refinement (CTIGAR) [8] combines abstraction and IC3. It also maintains a set of predicates $\mathbb{P}$ that abstract the concrete state space. But where IC3+IA only refines $\mathbb{P}$ whenever a spurious counterexample trace has been found in the abstract state space, CTIGAR triggers refinement of $\mathbb{P}$ over single-step queries in two situations: A so-called *lifting failure* occurs whenever an abstract cube c is not relatively inductive, i.e., whenever one of $F_{i-1} \wedge \neg c \wedge T \implies \neg c'$ or $I \implies \neg c$ fails. When this happens, IC3 extracts the state s from the solver model, as well as an assignment z to the primary inputs, and tries to lift the full assignment s to a partial assignment. The corresponding query $s \wedge z \wedge T \wedge \neg c'$ [13] is satisfied by default in IC3, however, for the abstracted version $\widehat{s}$ of s,

[4] Note that since SV-COMP 2017, the rules of the competition require witnesses for both verification results to gain the mentioned score points, which was implemented to prevent guessing the result. However, since none of the tools that we compare with is able to give such witnesses, we omit them and assign the score just based on the output of the tool.

it may fail, due to states in $\widehat{s}$ that have $\neg\widehat{c}$-successor states. The second situation in which predicate refinement must take place in CTIGAR is when the abstract cube $\widehat{c}$ is not inductive relative to some frame F_i, i.e., the abstract consecution fails, but the corresponding concrete consecution succeeds. Such a situation is called *consecution failure* and is triggered when the concrete cube c does not have F_i-predecessors, but by abstracting it to $\widehat{c}$, the abstraction includes a successor to some F_i-state. Like other predicate abstraction algorithms, CTIGAR uses interpolants for refinement if they are available in the respective theory. However, when a *consecution* or *lifting failure* occurs, CTIGAR may not eagerly refine whenever an abstraction failure occurs, but may choose a lazy approach and only refine later. In fact, the experimental evaluation in [8] revealed that in practice a lazy refinement approach is the best performing in terms of solved benchmarks, as well as in cumulative time (except for one configuration which solves six instances less).

There also exist other liftings of IC3 toward software model checking that are theory agnostic. A prominent example is *generalized property-directed reachability* (GPDR) [27]. Its main contribution is the lifting of IC3/PDR to non-linear fixed points that become important when considering procedure calls. In [27], GPDR is used to analyze times push-down systems that are modeled in linear real arithmetic.

A second approach tailored to a specific domain is [40], which applies IC3/PDR only to quantifier-free bit-vectors. As such, the algorithm models program behavior bit-precisely and all resulting formulae can be bit-blasted. Accordingly, [40] applies the standard Boolean IC3 algorithm.

9 Conclusion

In this paper we have presented *IC3CFA*, an incremental, inductive verification algorithm that lifts the ideas of IC3 from hardware to software model checking. In resemblance to other adaptations of IC3 to software verification, it supports the handling of infinite-state systems by generalizing SAT to SMT solving. Its distinguishing feature, however, is the explicit consideration of a program's control flow by maintaining location-specific frames for storing approximate reachability information. In comparison with more implicit approaches such as Tree-IC3 [14], which is based on unrolling the abstract reachability tree, this enables a stronger form of relative inductiveness.

Further performance enhancements were achieved by lifting the generalization procedure from IC3 to IC3CFA, which is employed to ensure the scalability of IC3 by overapproximating pre-images in counterexample analysis. Starting from a basic approach to generalization by dropping literals from cubes, we introduced a variety of techniques to improve the procedure, some of which are also applicable to other IC3-style verification algorithms. As we have shown, our techniques yield speedups of up to two orders of magnitude and enable many more benchmarks to be solved.

In future work, we plan to improve the handling of functions, which are currently processed by call inlining and cannot be recursive. Here, an interprocedural extension would be preferable with regard to both the expressiveness of the programming language and the efficiency of the verification approach. Our approach will be to embed the intraprocedural IC3CFA algorithm into an interprocedural context by interpreting function calls as single transitions in the global state space, using "summary" information for the respective function.

References

1. Albarghouthi, A., Gurfinkel, A., Chechik, M.: From under-approximations to over-approximations and back. In: TACAS, *LNCS*, vol. 7214, pp. 157–172. Springer (2012)
2. Backeman, P., Rümmer, P., Zeljić, A.: Bit-vector interpolation and quantifier elimination by lazy reduction. In: FMCAD18 (2018)
3. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV, LNCS, vol. 6806, pp. 171–177. Springer (2011)
4. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: FMCAD, pp. 25–32. IEEE (2009)
5. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: FMCAD, pp. 189–197. IEEE (2010)
6. Beyer, D., Löwe, S., Wendler, P.: Benchmarking and resource measurement. In: SPIN, LNCS, vol. 9232, pp. 160–178. Springer (2015)
7. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
8. Birgmeier, J., Bradley, A.R., Weissenbacher, G.: Counterexample to induction-guided abstraction-refinement (CTIGAR). In: CAV, *LNCS*, vol. 8559, pp. 831–848. Springer (2014)
9. Bjørner, N., Janota, M.: Playing with quantified satisfaction. In: LPAR (short papers), EPiC Series in Computing, vol. 35, pp. 15–27. EasyChair (2015)
10. Bradley, A.R.: SAT-based model checking without unrolling. In: VMCAI, LNCS, vol. 6538, pp. 70–87. Springer (2011)
11. Bradley, A.R., Manna, Z.: The Calculus of Computation—Decision Procedures with Applications to Verification. Springer, Berlin (2007)
12. Bradley, A.R., Manna, Z.: Checking safety by inductive generalization of counterexamples to induction. In: FMCAD, pp. 173–180. IEEE (2007)
13. Chockler, H., Ivrii, A., Matsliah, A., Moran, S., Nevo, Z.: Incremental formal verification of hardware. In: FMCAD, pp. 135–143. FMCAD Inc. (2011)
14. Cimatti, A., Griggio, A.: Software model checking via IC3. In: CAV, LNCS, vol. 7358, pp. 277–293. Springer (2012)
15. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: IC3 modulo theories via implicit predicate abstraction. In: TACAS, LNCS, vol. 8413, pp. 46–61. Springer (2014)
16. Cimatti, A., Griggio, A., Schaafsma, B., Sebastiani, R.: The MathSAT5 SMT Solver. In: TACAS, LNCS, vol. 7795. Springer (2013)

17. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (2001)
18. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252. ACM (1977)
19. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975)
20. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Upper Saddle River (1976)
21. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: FMCAD, pp. 125–134. FMCAD Inc. (2011)
22. Günther, H., Laarman, A., Weissenbacher, G.: Vienna verification tool: IC3 for parallel software (competition contribution). In: TACAS, LNCS, vol. 9636, pp. 954–957. Springer (2016)
23. Gurfinkel, A., Ivrii, A.: Pushing to the top. In: FMCAD, pp. 65–72. IEEE (2015)
24. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: CAV (1), LNCS, vol. 9206, pp. 343–361. Springer (2015)
25. Hassan, Z., Bradley, A.R., Somenzi, F.: Better generalization in IC3. In: FMCAD, pp. 157–164. IEEE (2013)
26. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL, pp. 58–70. ACM (2002)
27. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: SAT, LNCS, vol. 7317, pp. 157–171. Springer (2012)
28. Komuravelli, A., Bjørner, N., Gurfinkel, A., McMillan, K.L.: Compositional verification of procedural programs using horn clauses over integers and arrays. In: FMCAD, pp. 89–96. IEEE (2015)
29. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. In: CAV, LNCS, vol. 8559, pp. 17–34. Springer (2014)
30. Kroening, D., Strichman, O.: Decision procedures–an algorithmic point of view: Texts in theoretical computer science: An EATCS Series. Springer, Berlin (2008)
31. Lange, T., Neuhäußer, M.R., Noll, T.: Speeding up the safety verification of programmable logic controller code. In: HVC, LNCS, vol. 8244, pp. 44–60. Springer (2013)
32. Lange, T., Neuhäußer, M.R., Noll, T.: IC3 software model checking on control flow automata. In: FMCAD, pp. 97–104. IEEE (2015)
33. Lange, T., Prinz, F., Neuhäußer, M.R., Noll, T., Katoen, J.P.: Improving generalization in software IC3. In: SPIN, LNCS, vol. 10869, pp. 85–102. Springer (2018)
34. Mertens, T.: Efficient reuse of learnt information for control-flow oriented IC3 algorithms. Master thesis, RWTH Aachen University (2016)
35. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS, LNCS, vol. 4963, pp. 337–340. Springer (2008)
36. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Berlin (1999)
37. Prinz, F.: Generalisation methods for control-flow oriented IC3 algorithms. Master thesis, RWTH Aachen University (2016)
38. Tonetta, S.: Abstract model checking without computing the abstraction. In: FM, LNCS, vol. 5850, pp. 89–105. Springer (2009)
39. Vojnar, T., Beyer, D.: Competition on software verification (SV-COMP). https://sv-comp.sosy-lab.org/ (2019)
40. Welp, T., Kuehlmann, A.: QF BV model checking with property directed reachability. In: DATE, pp. 791–796. EDA Consortium (2013)