Improving Generalization in Software IC3

Tim Lange[1], Frederick Prinz[1], Martin R. Neuhäußer[2], Thomas Noll[1], and Joost-Pieter Katoen[1]

[1] RWTH Aachen University, Germany
{tim.lange, noll, katoen}@cs.rwth-aachen.de
[2] Siemens AG, Germany martin.neuhaeusser@siemens.com

Abstract. Generalization is a key feature to support state-space abstraction in IC3-based algorithms for software model checking, such as Tree-IC3 or IC3CFA. This paper introduces several improvements that range from efficient caching of generalizations over variable reductions to syntax-oriented generalization. Our techniques are generic in that they are independent of the underlying theory, and some of them are even applicable to IC3 in general. Their evaluation on multiple benchmarks, including a significant subset of the SV-COMP 2017 benchmarks, yields promising results.

1 Introduction

IC3 [4] 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 tries 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 base 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, so-called generalization.

An adaptation of IC3 to software model checking, called IC3CFA, was presented in [15]. In IC3CFA, the control-flow of a program is represented as control-flow automaton (CFA) while the data space is handled symbolically. This explicit structure can further guide the reachability analysis of IC3 when dealing with software. While IC3CFA was originally developed in the context of Programmable Logic Controller code verification, it is successfully applied to the verification of C programs, too, as shown in Sec. 4.

As the author of the original IC3 algorithm [4] states, "one of the key components of IC3 is inductive generalization" [13]. Even though IC3 is sound and complete without generalization, 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 state spaces. While the original procedure of literal elimination on the Boolean skeleton can be applied to IC3 in the first-order setting [6], generalization in IC3CFA still offers a large potential for improvements.

Our contributions are five small and elegant modifications of the generalization procedure of IC3CFA that improve the overall performance of IC3CFA by up to two orders of magnitude for certain benchmarks and allow us to verify ten instances more than using IC3-style generalization, with a verification time that is in total 10% lower. In addition, our improvements are general in the sense that the presented algorithms are independent of the underlying SMT theory and support existing generalization methods, such as unsatisfiable cores [4], obligation ordering [12] and interpolation [3].

Another approach to software model checking with IC3 is TreeIC3 [6], which iteratively derives a tree-like structure given in terms of an abstract reachability tree (ART) and computes abstractions of candidate counterexamples. However, generalization is not discussed in [6]. [18] presents an algorithm to apply IC3 to software model checking using the theory of quantifier-free bitvector formulas, thereby lifting standard IC3 from one-dimensional to multi-dimensional Boolean variables. Due to this restriction to bit vectors, the algorithm is highly theory-aware and cannot be applied to other theories such as unbounded integers. It utilizes the standard generalization of IC3. [14] also addresses IC3-style software model checking, but focuses on pushdown automata and general Horn clauses, as well as linear real arithmetic. [3] utilizes the ideas of IC3 to enhance a predicate abstraction/refinement-based analysis of infinite-state systems by using counterexamples to induction to guide the refinement process of the predicates. In [7], the authors use the Tree-IC3 algorithm [6] and combine it with implicit predicate abstraction. By abstracting the exact transition relation, they are able to verify a software system in terms of a purely abstract state space by executing a Boolean IC3 algorithm, using only the basic generalization of IC3.

In the remainder we provide basic concepts for IC3CFA generalization in Sec. 2, present our contributions to the generalization procedure in Sec. 3, and show their evaluation in Sec. 4. We conclude with a short summary and outlook in Sec. 5. Proofs are provided in the appendix.

2 Preliminaries

To abstract from different input languages and to maintain a small set of commands, we use a guarded command language (GCL) for loop-free code.

Definition 1 (GCL commands). *The syntax of the Guarded Command Language (GCL) is defined as*

$$\mathcal{C} ::= \texttt{Assume}\ b \mid x := a \mid \mathcal{C}_1 ; \mathcal{C}_2 \mid \mathcal{C}_1 \Box \mathcal{C}_2,$$

where x is a variable and

$$\begin{aligned} a &::= z \ \mid\ x \ \mid\ a_1 \diamond a_2 \\ b &::= true \ \mid\ false \ \mid\ a_1 \circ a_2 \ \mid\ b_1 \wedge b_2 \end{aligned}$$

with $z \in \mathbb{Z}$, $\diamond \in \{+, -, *, /, \%\}$, and $\circ \in \{=, \neq, \leq, \geq, <, >\}$.

For our modifications to the generalization algorithm, we disallow disjunction and negation in Boolean expressions of `Assume` commands. We can compensate this restriction by nondeterministic branching using the $\Box$ operator and pushing negations into relational operators, which are closed under negation. While this may seem overly restrictive, it is necessary to enable some of our improvements, in particular those presented in Sec. 3.1. GCL only supports loop-free code. To manipulate control flow we use a control-flow automaton [15]:

Definition 2 (Control-flow automaton). *A control-flow automaton (CFA) is a tuple $A = (L, G, \ell_0, \ell_E)$ of locations $L = \{\ell_0, \ldots, \ell_n\}$, edges $G \subseteq L \times GCL \times L$, initial location $\ell_0 \in L$ and error location $\ell_E \in L$.*

Furthermore we use the GCL command $\mathcal{C}$ of edge $e = (l, \mathcal{C}, l') \in G$ to define the quantifier-free first-order term T_e of e, i.e. the term representing the semantics of $\mathcal{C}$, transforming unprimed current states φ to primed successor states φ' by replacing every variable x in φ by x'. Given a subset V of variables in CFA edges, a *cube* $c \in \mathit{Cube}$ over V 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*. A *frame* $F \in \mathit{Frame}$ is defined as a conjunction of clauses. For simplicity, we will refer to cubes and frames as sets of implicitly conjoined literals and clauses, respectively. To facilitate this reasoning, we will distinguish between two different, converse perspectives: Given two cubes c_1 and c_2, then $c_1 \subseteq c_2$ holds if the set of literals of c_1 is a subset of the literals of c_2. However, if we think of a cube as a symbolic representation of states, then c_1 represents more states than c_2. To take this converse perspective into account, we introduce the partial order $\sqsubseteq$:

Definition 3. *Given two cubes c_1 and c_2, let $c_2 \sqsubseteq c_1$ iff $c_1 \subseteq c_2$.*

Similar to the definition of relative inductivity [4], [15] defines relative inductivity for single edges of a CFA as follows:

Definition 4 (Edge-relative inductivity [15]). *Assuming frame $F_{(i-1,\ell)}$ and cube c at location ℓ', we define inductivity of c edge-relative to $F_{(i-1,\ell)}$ along edge $e = (\ell, \mathcal{C}, \ell')$ by*

$$F_{(i-1,\ell)} \wedge T_e \Rightarrow \neg c' \qquad , \text{ if } \ell \neq \ell' \tag{1}$$

$$F_{(i-1,\ell)} \wedge \neg c \wedge T_e \Rightarrow \neg c' \qquad , \text{ if } \ell = \ell'. \tag{2}$$

We define the predicate $\mathit{relInd}_e(F_{(i-1,\ell)}, c)$ to hold iff c is inductive edge-relative to $F_{(i-1,\ell)}$ along e.

While this holds for IC3CFA without generalization, (2) cannot be applied in the presence of generalization any more.

Lemma 1. *Given two distinct edges $e_1 = (\ell_1, \mathcal{C}_1, \ell')$ $e_2 = (\ell_2, \mathcal{C}_2, \ell')$ with $\ell_1 \neq \ell_2$ and two frames $F_1 = F_{(i-1,\ell_1)}$ and $F_2 = F_{(i-1,\ell_2)}$, Generalization does not preserve* (2)*:*

$$\begin{aligned} & (F_1 \wedge T_{e_1} \Rightarrow \neg g_1') \wedge (F_2 \wedge \neg g_2 \wedge T_{e_2} \Rightarrow \neg g_2') \\ \not\Rightarrow \quad & ((F_1 \wedge T_{e_1}) \vee (F_2 \wedge \neg (g_1 \wedge g_2) \wedge T_{e_2})) \Rightarrow \neg (g_1' \wedge g_2') \end{aligned}$$

As a result of Lemma 1, we will only use (1) for all inductivity queries in the remainder. Like [4], we check validity of (1) by checking whether

$$F \wedge T_{\ell_1 \rightarrow \ell_2} \wedge c'$$

is unsatisfiable.

Analogously to a Boolean generalization of IC3 [4], we can define a generalization that preserves edge-relative inductivity.

Definition 5 (Edge-relative inductive generalization). *We call a function $gen : Frame \times Cube \times GCL \rightarrow Cube$ an edge-relative inductive generalization for edge e if the following holds for every cube c and every frame F:*

1. *$gen(F, c, e) \subseteq c$*
2. *$relInd_e(F, c) \Rightarrow relInd_e(F, gen(F, c, e))$.*

IC3

Let $S = (X, I, T)$ be a transition system over a finite set X of Boolean variables, and $I(X)$ and $T(X, X')$ two propositional formulas describing the initial condition over variables in X, and the transition relation over X and next-state primed successors X', respectively. Given a propositional formula $P(X)$, we want to verify that P is an S-invariant, i.e. that every state in S that is reachable from a state in I satisfies P. Sometimes also an inverted formulation is used like in [9] where $\neg P$ states are *bad states* and we want to show that no bad state is reachable from any of the initial states. The main idea of the IC3 algorithm [4] is to show that P is *inductive*, i.e. that $I \Rightarrow P$ and $P \wedge T \Rightarrow P'$ hold, which entails that P is an S-invariant. However, the reverse implication does generally not apply. Therefore the goal of IC3 is to produce a so-called *inductive strengthening* F of P, s.t. $F \wedge P$ is inductive. In contrast to its predecessor FSIS [5], IC3 constructs F *incrementally.* This incremental construction is based on a dynamic sequence of *frames* $F_0, \ldots, F_k$ for which

$$I \Rightarrow F_0 \tag{3}$$

$$F_i \Rightarrow F_{i+1} \qquad \text{, for } 0 \leq i < k \tag{4}$$

$$F_i \Rightarrow P \qquad \text{, for } 0 \leq i \leq k \tag{5}$$

$$F_i \wedge T \Rightarrow F_{i+1}' \qquad \text{, for } 0 \leq i < k \tag{6}$$

has to hold in order to produce an inductive invariant. Note that k is a dynamic bound that will be extended by IC3 on demand. Due to the usage of a satisfiability (SAT) solver, [4] uses the implication $F \Rightarrow F'$ to compare frames. This semantic check covers the syntactic check whether $F \sqsubseteq F'$. The algorithm starts with initial checks for 0- and 1-step reachable states in $\neg P$ and afterwards initializes the first frame F_0 to I. The rest of the algorithm can be divided into an inner and an outer loop, sometimes referred to as *blocking* and *propagation* phases.

The outer loop iterates over the maximal frame index k, looking for states in F_k that can reach $\neg P$, so-called *counterexamples to induction* (CTI). If such a CTI exists, it is analyzed in the inner loop, the blocking phase. If no such CTI exists, IC3 tries to propagate clauses learned in frame F_i forward to F_{i+1}, for $0 \leq i < k$. In the end it checks for termination, which is given if $F_i = F_{i+1}$ for some $0 \leq i < k$.

The blocking phase decides whether a CTI is reachable from I or not. For this purpose, it maintains a set of pairs of frame index and symbolic states in form of a cube, called *proof obligations*. From this set it picks a pair (i, c) with smallest frame index i. For (i, c), IC3 checks whether $\neg c$ is inductive relative to F_{i-1}. If it is, we can block c in frames F_j for $0 \leq j \leq i{+}1$. But rather than just adding $\neg c$, IC3 tries to obtain a clause $cl \subseteq \neg c$ excluding more states. This *generalization* of $\neg c$ is added to the frames. If $\neg c$ is not inductive relative to F_{i-1}, this means that there exists an F_{i-1} predecessor p reaching c. IC3 therefore adds $(i{-}1, p)$ as proof obligation. The blocking phase terminates at obligation $(0, p)$, in which case there exists a counterexample path, or when for every proof obligation (i, c) it holds that $i > k$, i.e. every predecessor of the CTI is relative inductive.

IC3CFA [15] adapts IC3 to software model checking, based on the idea to exploit the structure of the CFA of a program to guide the search of IC3. For this purpose, [15] proposes an algorithm similar to the original IC3: It starts searching for CTIs by computing states in every predecessor location of the error location ℓ_E (representing all *bad states*) using *weakest preconditions*. Those states at their respective index and location are stored as proof obligations. As long as the obligation queue is non-empty, the algorithm picks an obligation with minimal index and checks whether the obligation is inductive edge-relative to the location-specific predecessor frame on every incoming edge. If the inductivity check succeeds, the cube is generalized and blocked by adding it to the frame matrix at the respective index and location. If the check does not succeed, a predecessor is computed and added to the obligation queue. If at some point the initial location ℓ_0 is reached, a counterexample can be reported. If, on the other hand, the obligation queue becomes empty, the strengthening of frames up to bound k is complete and termination is checked. The termination condition verifies that for some i, at every location the frame with index i is equal to the frame with index $i{+}1$. If this condition is not met, the algorithm proceeds by increasing its bound to $k{+}1$. [15] shows that the original IC3CFA algorithm, without any generalization, already outperforms some other IC3 algorithms for software, such as TreeIC3 and the trivial lifting to IC3SMT [6].

3 Generalization

This section presents five improvements to the IC3CFA algorithm that pave the way towards a powerful, efficient generalization. Since IC3CFA as published in [15] does not include generalization, we must first take a look at how a generalization for a location can be computed while using the positive effects of only considering individual edges. In Lemma 1 and Def. 5, we have shown how inductivity and generalization work on a single CFA edge. Given some ℓ' with predecessor locations $\ell_1, \ldots, \ell_n$, we can compute edge-relative generalizations $g_1, \ldots, g_n$ and merge them to a generalization to be blocked at ℓ' by constructing the cube g that contains all literals appearing in each g_i $(1 \leq i \leq n)$. Using this merge to lift the syntactic generalization based on dropping literals will be referred to as *IC3-style generalization* in the remainder.

As mentioned before, IC3 uses the model generated by the SAT solver as a set of predecessor states of a non-inductive state set for further backward search. If such a cube c, or more precisely its negation, is shown to be inductive relative to a frame F, it is generalized to a subcube g that is still inductive relative to F. While this method is complete for Boolean IC3, its application to IC3 for software, such as TreeIC3 [6] or IC3CFA [15], has two main consequences that will be driving forces behind the contributions of this section: (1) there may exist infinitely many predecessor states, so model enumeration is not complete, and (2) the generalization should ideally be theory-unaware to support its use with different backend theories. While this may seem counter-intuitive, since theory-aware implementations may be more performant, the independence from any theory allows a modular, flexible approach to backend theories that enables us to solve many more problem domains. For example, a generalization that uses Linear Real Arithmetic (LRA) is only able to solve a limited number of problems, whereas we can switch from LRA to Floating Point (FP) depending on whether we need bit-precise arithmetic, including overflows, or not. Note that in contrast to IC3, IC3CFA does not use the global transition relation but only single transitions between control-flow locations.

3.1 Predecessor Computation

A solution to (1) is presented in [6, 15]: The computation of the weakest precondition (WP) yields an exact pre-image and thus provides a suitable way to extract predecessor states. While various definitions of WPs can be found in the literature [10, 16], the most commonly used is the *demonic* version [8]. For non-deterministic choices this variant only covers states that reach a state in the postcondition under *all* possible executions. However, in safety verification we are not only interested in those states that *must* lead to a failure, but also in states that *may* lead to a failure under certain conditions, which corresponds to the *angelic* interpretation of weakest preconditions. In contrast to [6], [15] splits the semantics of an edge into GCL commands and SMT expressions. By applying standard GCL predicate transformer semantics [8] and rewriting the SMT expressions, our angelic WP computation does *not introduce quantifiers* and thus there is no need to eliminate those. Based on the four types of GCL commands (Def. 1), the angelic WP is defined according to the following rules:

$$\begin{aligned}
wp(\texttt{Assume } b, \varphi) &= \varphi \wedge b \\
wp(x := a, \varphi) &= \varphi[x \mapsto a] \\
wp(\mathcal{C}_1; \mathcal{C}_2, \varphi) &= wp(\mathcal{C}_1, wp(\mathcal{C}_2, \varphi)) \\
wp(\mathcal{C}_1 \square \mathcal{C}_2, \varphi) &= wp(\mathcal{C}_1, \varphi) \vee wp(\mathcal{C}_2, \varphi).
\end{aligned}$$

where $\varphi[x \mapsto a]$ equals φ with all free occurrences of x replaced by a. When applying this WP to a state set represented symbolically, it yields a first-order term whose structure is highly dependent on the structure of the transition. However, this ambiguity exacerbates the structured processing of proof obligations and generalization. In our experiments we found that a restricted representation like cubes, which are used in [4], offers two-fold advantages: The simpler data structures improve the performance of our implementation while at the same time the generalization is able to drop more literals. To employ the cube structure of [4], we can translate the results of WP computations into disjunctive normal form (DNF), resulting in a number of cubes that can potentially lead into the target states. While this translation has exponential worst-case runtime, our experiments have revealed an overall beneficial effect.

After closer inspection of the decomposition of the WP into DNF, we found that we can organize the conversion to DNF as a translation on the GCL structure. We call this operation *split*, as it splits a single choice command into a set of parallel commands that can be represented as parallel edges in the CFA.

Definition 6 (Split). *The function spl* : $GCL \to 2^{GCL}$ *is given by:*

$$spl(\mathcal{C}) = \begin{cases} spl(\mathcal{C}_1) \cup spl(\mathcal{C}_2) & \text{if } \mathcal{C} = (\mathcal{C}_1 \square \mathcal{C}_2) \\ \{c_1; c_2 \mid \forall i \in \{1,2\}.\ c_i \in spl(\mathcal{C}_i)\} & \text{if } \mathcal{C} = (\mathcal{C}_1; \mathcal{C}_2) \\ \{\mathcal{C}\} & \text{otherwise.} \end{cases}$$

Applying the function *spl* to the commands along edges in G yields a refined set of control-flow edges

$$G' = \{(\ell, \mathcal{C}', \ell') \mid (\ell, \mathcal{C}, \ell') \in G, \mathcal{C}' \in spl(\mathcal{C})\}.$$

Intuitively, an edge $e = (\ell, \mathcal{C}, \ell') \in G$ labeled with $\mathcal{C}$ including a choice, is split into multiple edges between the same locations. Each new edge contains no choice command any more, such that it models a deterministic, sequential behaviour. This in turn means that the result of a WP computation of a cube with respect to a split edge is now guaranteed to be a cube again.

Theorem 1. *For any GCL command $\mathcal{C}$ and cube c:*

$$dnf(wp(\mathcal{C}, c)) = \bigvee \{wp(\mathcal{C}', c) \mid \mathcal{C}' \in spl(\mathcal{C})\}$$

where $dnf(\varphi)$ converts the given quantifier-free first-order formula φ into DNF.

Using the *spl* function, we can split the WP transformer into a number of transformers that directly yield cubes, rather than arbitrarily structured formulas. The disjunction of those partial WPs is equal to the DNF of the original WP, as shown in Theorem 1. While this additional split operation is not beneficial on its own, it enables a subsequent optimization that has the potential to statically derive a generalization of good quality without any SMT calls.

3.2 Predecessor Cubes

This approach computes the generalization of a given cube c based on a syntactic check of the predecessor frame. If the predecessor frame $F_{(i-1,\ell)}$ contains a clause $\neg\bar{c}$ which blocks at least $wp(\mathcal{C}, c)$, i.e. $\bar{c} \subseteq wp(\mathcal{C}, c)$ with respect to edge $e = (\ell, \mathcal{C}, \ell') \in G$, then the cube c is inductive edge-relative to $F_{(i-1,\ell)}$ w.r.t. e. This static test can be applied to every inductivity check of IC3CFA, not only to those used in generalization.

Lemma 2. *Let (L, G, ℓ_0, ℓ_E) be a CFA with edge $e = (\ell, \mathcal{C}, \ell') \in G$, and cube c to be blocked at $\ell' \in L$ and index i. For frame $F_{(i-1,\ell)}$:*

$$\begin{aligned}&\big(\exists \bar{c} \in Cube.\ \big(\neg\bar{c} \in F_{(i-1,\ell)}\big) \wedge \big(\bar{c} \subseteq wp(\mathcal{C}, c)\big)\big)\\ \implies\ & relInd_e(F_{(i-1,\ell)}, c).\end{aligned}$$

Given such a predecessor cube $\bar{c}$, we can also derive the generalization $g \subseteq c$ of the original cube c.

Lemma 3. *Let $\mathcal{C}$ be a choice-free GCL command. Given two cubes c_1, c_2, it holds that*

$$wp(\mathcal{C}, c_1 \wedge c_2) \iff wp(\mathcal{C}, c_1) \wedge wp(\mathcal{C}, c_2).$$

Given that wp distributes over conjunction for choice-free GCL commands, we can decompose cube c into its literals $l_1, \ldots, l_n$ and construct the wp for each $l_i, i \in \{1, \ldots, n\}$ individually. The result of $wp(\mathcal{C}, c)$, a conjunction of n WPs of the literals l_i, is $w = w_1 \wedge \cdots \wedge w_n$. If we now encounter a predecessor cube $\bar{c} \subseteq w$, we map each $w_i \in \bar{c}$ back to its original literal $l_i \in c$ and obtain a new cube $g = \{l_i \mid w_i \in \bar{c}, w_i = wp(\mathcal{C}, l_i)\}$. Since our approach obviously satisfies monotonicity for choice-free GCL commands that don't contain disjunctions according to Def. 1, it holds that

$$\bar{c} \subseteq w \implies g \subseteq c.$$

Furthermore, since $\neg wp(\mathcal{C}, l_i) \in F_{(i-1,\ell)}$ for each $l_i \in g$, g is inductive relative to $F_{(i-1,\ell)}$ and thus g is a valid generalization for c.

Note that semantically g is a strongest postcondition of $\bar{c}$, but differs in the syntactic structure.

Theorem 2. *Let (L, G, ℓ_0, ℓ_E) be a CFA with edge $e = (\ell, \mathcal{C}, \ell') \in G$, and cube c be blocked at $\ell' \in L$ and index i. For frame $F_{(i-1,\ell)}$ and cube g:*

$$\begin{aligned} & \big(\forall \mathcal{C}' \in spl(\mathcal{C}).\neg wp(\mathcal{C}', g) \in F_{(i\text{-}1,\ell)} \wedge wp(\mathcal{C}', g) \subseteq wp(\mathcal{C}', c)\big) \\ \implies \; & gen(F_{(i,\ell')}, c, e) = g. \end{aligned}$$

We execute the static check for predecessor cubes, as given in Lemma 2, in the generalization of IC3CFA whenever deriving a generalization from the methods proposed in Sec. 3.4 & 3.6 fails. If we find a predecessor cube, we can immediately use that to construct a generalization and skip the generalization phase entirely.

3.3 WP Inductivity

Like the original IC3 algorithm, IC3CFA makes heavy use of the underlying solver, such that small optimizations in its usage can have significant effect on the overall performance. As we compute exact pre-images by taking weakest preconditions in each step anyway, we may replace the transition part of the relative inductivity check (cf. implication (1) in Def. 4) by the simpler test whether the frame and the WP share common states. Since the resulting formula only reasons about unprimed variables, we reduce the number of variables in the SMT query by half in best case, and also decrease the size of the formula in general. However, since the inductivity check happens before the WP construction, we never know whether we can actually reuse the constructed WP afterwards. To avoid unnecessary overhead we introduce an additional step: Given a failed inductivitiy query, IC3CFA constructs the WP and decomposes it into a set of cubes. However, not all cubes from the set of WP cubes may be of interest. In fact, the likelihood that some of the cubes are already excluded by the respective frame is very high. Here we can use the modified inductivity check to filter those cubes that are actually causing the failed inductivity. An experimental evaluation is given in Sec. 4.

Definition 7 (Alternative relative inductivity). *Let (L, G, ℓ_0, ℓ_E) be a CFA with edge $e = (\ell, \mathcal{C}, \ell') \in G$ and $i \in \mathbb{N}, i \geq 1$. Given the frame $F_{(i-1,\ell)}$ and a cube c, we define the predicate $relIndAlt_e(F_{(i-1,\ell)}, c)$ by*

$$relIndAlt_e(F_{(i-1,\ell)}, c) \Leftrightarrow unsat(F_{(i-1,\ell)} \wedge wp(\mathcal{C}, c)).$$

Cube c is inductive relative to frame $F_{(i-1,\ell)}$ and edge e iff $relIndAlt_e(F_{(i-1,\ell)}, c)$ holds. In the following we will also refer to $relIndAlt$ as $relInd$.

The correctness of alternative relative inductivity is given by:

Lemma 4. *Let (L, G, ℓ_0, ℓ_E) be a CFA with edge $e = (\ell, \mathcal{C}, \ell') \in G$ and $i \in \mathbb{N}, i \geq 1$. For every frame $F_{(i-1,\ell)}$ and cube c:*

$$relIndAlt_e(F_{(i-1,\ell)}, c) \quad \Leftrightarrow \quad relInd_e(F_{(i-1,\ell)}, c).$$

3.4 Caching of Generalization Context

For IC3 software model checking algorithms that determine predecessor cubes using weakest precondition cubes and thus generalization often reappear, but so far they are always recomputed. This approach of recomputing generalizations again and again is obviously not very efficient. We therefore aim to store the information that is obtained during a generalization of a cube in order to reuse this information in subsequent generalizations. To do so, not only do we need to store the generalization g, we also have to store all other aspects of the used inductivity query: the cube c that the generalization was derived from as well as the frame F and the edge$e$ relative to which it was determined. We call this set of information the *context* of the generalization or simply *generalization context.*

Definition 8 (Generalization context). *The generalization context GC_i of a CFA (L, G, ℓ_0, ℓ_E) at index i is a set of quadruples*

$$GC_i \subseteq \mathit{Cube} \times G \times \mathit{Frame} \times \mathit{Cube}$$

where

$$\begin{aligned} &(c, e, F, g) \in GC_i \text{ with } e = (\ell, \mathcal{C}, \ell') \\ \implies &\big(\exists j \leq i.\ \mathit{gen}(F_{(j,\ell')}, c, e) = g \text{ and } F = F_{(j-1,\ell)}\big)\,. \end{aligned}$$

Following Def. 8, each generalization g of a cube c at index j and edge e relative to frame F is stored as one generalization context (c, e, F, g) available in all sets GC_i $(i \geq j)$. In our implementation we use a least-recently-used cache for this purpose such that generalizations that are older and less likely to reappear are replace.

3.5 Upper Bounds from Generalization Context

Given a cube c to be generalized along edge $e \in G$, our aim is to derive a generalization based on previous generalizations of c along e (if any) relative to frame F. Due to the monotonically growing behaviour of frames in IC3, a generalization of a fixed cube c along fixed edge e with only a shrinked frame $F \sqsubseteq F'$ will yield a result that contains *at most* the literals of the previous attempt. In other words, by excluding states from a frame, the set of states unreachable from that frame can only grow but never shrink. Therefore the old generalization gives an *upper bound* on the literals of the new generalization.

Theorem 3. *Let (L, G, ℓ_0, ℓ_E) be a CFA with edge $e = (\ell, \mathcal{C}, \ell') \in G$ and c a cube to be generalized at index $i \geq 1$ and location ℓ'. Given the frame $F_{(i-1,\ell)}$, it holds that*

$$\big((c, e, F, g) \in GC_i \wedge F_{(i-1,\ell)} \sqsubseteq F\big) \implies g = \mathit{gen}(F_{(i,\ell')}, c, e).$$

So far, we might reuse a generalization g based on the generalization context (c, e, F, g) if we encounter the exact same cube c along e again, but relative to a shrinked frame $F' \sqsubseteq F$. We extend the use of generalization contexts to cover those cases when we encounter a similar cube $\widehat{c}$. More precisely, $\widehat{c}$ has to be a superset of the previous generalization g with respect to the literals, i.e. $g \subseteq \widehat{c}$. If this is the case and the frame condition $F_{(i-1,\ell)} \sqsubseteq F$ also holds, then we also get g as generalization of $\widehat{c}$. As a result, we avoid even more computations. Corollary 1 shows the effect of the improved upper bounds.

Corollary 1. *Let (L, G, ℓ_0, ℓ_E) be a CFA with edge $e = (\ell, \mathcal{C}, \ell') \in G$, and $\widehat{c}$ a cube to be generalized at index i and location ℓ'. Given the frame $F_{(i-1,\ell)}$, it holds that*

$$\begin{aligned} &((c, e, F, g) \in GC_i) \wedge \left(F_{(i-1,\ell)} \sqsubseteq F\right) \wedge (g \subseteq \widehat{c}) \\ \implies\ &gen(F_{(i,\ell')}, \widehat{c}, e) = g. \end{aligned}$$

While this caching of generalizations may seem counter-intuitive for SAT-based IC3, it drastically improves the performance of certain types of IC3 for software verification: Due to the pseudo-random choice of predecessor cubes based on the SAT model, lazily resetting the SAT solver and re-generalizing cubes prevents IC3 from investigating bad paths too deeply [11]. However, if we swap the model-based predecessor extraction for WP-based predecessors, the computed cubes will be deterministic in every iteration, making resets and re-generalizations less useful while caching becomes more effective.

3.6 Lower Bounds from Generalization Context

Generalization contexts are used to derive a generalization based on previous computations. More precisely, if there exists a matching generalization context gc, then it will yield all literals that will *at most* be contained in the new generalization, i.e. an upper bound on the literals of the generalization. In this section we use generalization contexts to determine all literals that will *at least* be contained in the generalization, i.e. lower bounds on the literals.

Let (c, e, F, g) be a generalization context. For this previous generalization, c has been generalized to the cube $g \subseteq c$, thus it follows that all proper subsets $\widehat{g} \subset g$ are not inductive relative to F. If we later encounter c with a larger frame F', i.e. $F \sqsubseteq F'$, every cube larger than g will not be inductive relative to F'. However, there might be a state f that is part of F', but not part of F, i.e. $f \models F' \wedge \neg F$, such that f enables a transition to a g-state. Thus, the resulting generalization $\bar{g}$ will be somewhere between g and c, i.e. $g \subseteq \bar{g} \subseteq c$, making g a lower bound[3] of the literals of the new generalization. Theorem 4 shows the correctness of necessary literals based on the generalization context.

[3] Note that g is a lower bound but not necessarily the greatest one, as states in $F' \wedge \neg F$ may have a transition into g.

Theorem 4. *Let (L, G, ℓ_0, ℓ_E) be a CFA with $e = (\ell, \mathcal{C}, \ell') \in G$, and c a cube to be generalized at ℓ'. It holds for $i \geq 1$:*

$$\begin{aligned} & ((c, e, F, g) \in GC_i) \wedge \left(F \sqsubseteq F_{(i-1,\ell)}\right) \wedge (\widehat{g} \subset g) \\ \Longrightarrow\ & gen(F_{(i,\ell')}, c, e) \neq \widehat{g} \\ \Longrightarrow\ & g \subseteq gen(F_{(i,\ell')}, c, e) \subseteq c. \end{aligned}$$

Using the generalization context we have shown that we can store old generalization results and their corresponding context *to give both an upper and a lower bound for the new generalization.* This means that for a cube c to be generalized, cube g with $g \subseteq c$ as upper bound and cube g' with $g' \subseteq g \subseteq c$ as lower bound, we effectively only have to check whether we can drop the literals of $g \backslash g'$ from c.

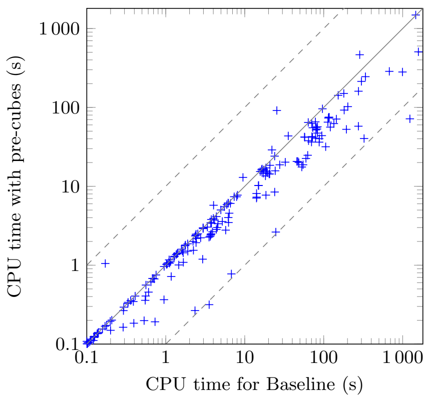
4 Evaluation

We implemented our optimizations to IC3CFA on top of an existing proprietary model-checker with bit-precise analysis. The input C file is parsed by the CIL Parser [17] and translated into an intermediate language. We apply static minimizations, construct the CFA, apply Large-Block Encoding [2] and execute IC3CFA. All results are obtained using Z3 4.6.1.

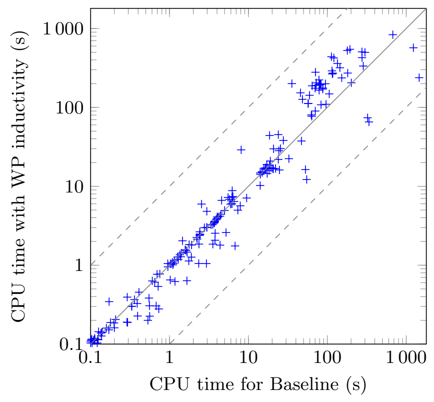
To evaluate the performance of our proposed improvements, we use a benchmark set consisting of 99 benchmarks from [6] and 254 SV-COMP 2017 benchmarks [1] from the *ReachSafety* category with the subcategories: *ReachSafety-BitVectors*, *ReachSafety-ControlFlow* and *ReachSafety-Loops*. Note that in some subsets of these categories, all files contain constructs, e.g. function pointers, that our parser and bit-precise memory model do not support. We therefore excluded the subsets *ntdrivers* and *ssh* and evaluated our contributions on the remaining 353 instances. To enable reviewers to reproduce our results, we make a Linux binary available at `http://www-i2.informatik.rwth-aachen.de/mctools/vplc/spin18`. All our results are obtained on an Intel Xeon CPU E5-2670 v3 @ 2.30GHz with a timeout of 1800s and a memory limit of 3GB, using one core per instance, executed in Benchexec 1.16.

We evaluate the improvements of each optimization using scatter plots with logarithmic axes (see Fig. 1), where every mark below/right of the diagonal indicates that the given optimization pays off, with points on the dashed line indicating a variation of one order of magnitude. To evaluate the isolated effect of each modification, we ran a *Baseline* configuration with every optimization disabled and separate configurations *Pre-Cubes*[4] and *WP Inducitivity* where we enable just the specific modification. Due to the close relation between our proposed methods for using generalization contexts as upper and lower bounds, we evaluate these in a stepwise fashion: *Baseline* disables lower and upper bounds

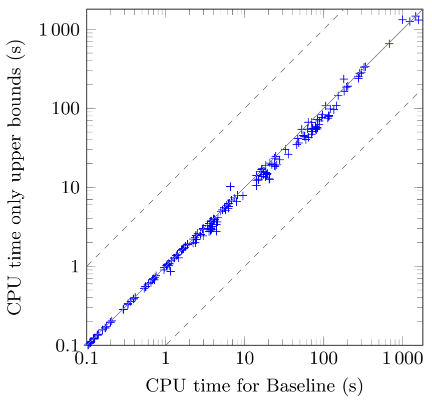
[4] Note that, as mentioned in Sec. 3.1, we apply *split* only in combination with the search for predecessor cubes and thus also evaluate their effect together.



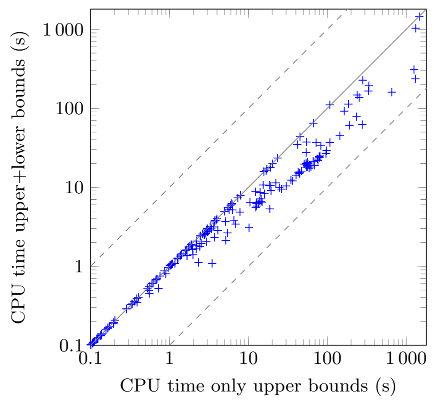
(a) Effect of Pre-Cubes



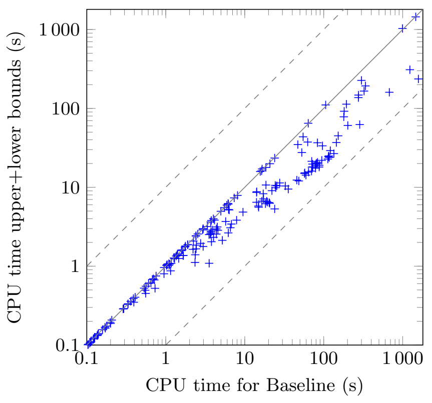
(b) Effect of WP Inductivity



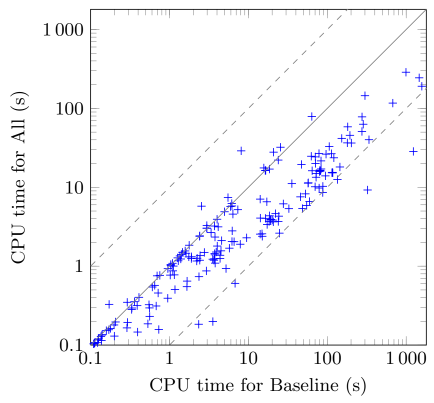
(c) Effect of Upper Bounds



(d) Effect of Lower Bounds



(e) Effect of Generalization Caching



(f) Effect of all Optimizations

Fig. 1. Isolated effect of individual optimizations and total effect of all optimizations

Configuration	# Solved	Time(s)
Baseline	234	12340
Pre-Cubes	238	9992
WP Inductivity	236	11925
Upper Bounds	234	11571
Upper+Lower Bounds	242	11913
All	**244**	11160
No Generalization	161	6075

Fig. 2. Summary of all configurations

from generalization contexts; *Upper Bounds* enables upper bounds, while lower bounds are disabled; *All Bounds* enables upper and lower bounds, thus evaluating the isolated effect of lower bounds when compared to *Upper Bounds* and evaluating the overall effect of generalization contexts when compared to *Baseline*. To further compare the overall effect of all contributions the *All* configuration has all proposed optimizations enabled. To evaluate the performance of our new IC3CFA implementation with generalization and the presented modifications against the version presented in [15], we add the *No Generalization* configuration.

Considering the configuration *Pre-Cubes*, the scatter plot (Fig. 1a) shows a strong positive impact of enabling the extraction of a generalization based on subcubes in the predecessor frame. While a very small number of instances exhibit negative performance with enabled *Pre-Cubes*, multiple benchmarks are found near the dashed line, indicating a verification time that is one order of magnitude faster with *Pre-Cubes* enabled, with the most notable case resulting in a verification time of 1200 seconds without and 72 seconds with *Pre-Cubes*. We conclude that the proposed method is highly favorable and improves the verification time of IC3CFA dramatically. Figure 2 shows that IC3CFA with *Pre-Cubes* is able to solve four more instances than without.

The proposed WP inductivity (Fig. 1b) yields a more ambivalent result. We achieve a speedup for almost all small instances with runtime up to 10 seconds. For instances between 10 and 100 seconds runtime, we see an indeterminate situation with results equally spread to both sides. For larger/harder benchmarks, enabling *WP-Inductivity* clearly yields a worse runtime than *Baseline*. Interestingly, Fig. 2 reveals that *WP-Inductivity* can solve two more instances than *Baseline*.

As mentioned, our proposed caching of generalization contexts enables multiple optimizations, allowing us to identify literals that can be dropped, as well as literals that cannot be dropped. Due to the strong connection, we evaluate the performance in a three-fold way:

We start with the effect of upper bounds (Fig. 1c), i.e. we store generalization contexts, but only use them to identify literals that can be removed, and compare the results to *Baseline*. Except for three outliers, *Upper Bounds* shows a small improvement in performance for all benchmarks. As shown in Fig. 2, the number of solved instances is identical to *Baseline* and the total verification time improves slightly. We conclude that the overhead of managing and searching the cache entries almost outweighs the saved solver calls if we use generalization contexts for upper bounds only.

Next, we evaluate the effect of lower bounds (Fig. 1d) by comparing the results of the *Upper Bounds* configuration with one where upper and lower bounds are activated, i.e. we use cached generalization contexts to identify which literals can be dropped and which cannot. In contrast to upper bounds, *also* enabling lower bounds enables a massive performance improvement with all instances being solved faster than without lower bounds.

Finally, we evaluate the performance of generalization caching in total against *Baseline* (Fig. 1e), revealing the total effect of upper and lower bounds. We can see that in total, no instance is being solved slower with generalization caching than without and some instances can be solved almost one order of magnitude faster with generalization caching than without. In particular the few instances that perform worse with just upper bounds are being compensated by the massive improvement that lower bounds yield.

To evaluate the overall effect of all our contributions, we compare a configuration with all proposed improvements activated (*All*) against *Baseline*. Fig. 1f shows the large potential that our small improvements to the generalization procedure of IC3CFA yield. The few instances with negligible deterioration are all simpler instances that are executed in less than 100 seconds. On the other hand, the vast majority of instances benefit heavily from our improved generalization with multiple instances performing about one order of magnitude better. For the most noticeable instance, the runtime of IC3CFA is reduced from 1200 seconds to only 28 seconds, a gain of almost two orders of magnitude. Enabling all optimizations, we are able to solve 10 more instances out of the set of all 353 benchmark instances.

5 Conclusion

In this paper we presented a number of very simple and easy to implement improvements of the generalization procedure in CFA-based IC3CFA software model checking. As evaluated in Sec. 4, these straight-forward improvements offer performance improvements of up to two orders of magnitude and enable 10 more benchmarks to be solved. In addition, we are able to solve 35% more benchmarks than the original IC3CFA algorithm presented in [15]. According to our experiments, the amendments of predecessor cubes and generalization, as well as generalization contexts turned out to be the most beneficial. Predecessor cubes that extract a generalization based on an occurrence of a sub-cube in the previous frame are tailored towards software IC3 with weakest preconditions and can also be applied to other IC3 software model checkers that use weakest preconditions, like TreeIC3. Our small and elegant amendment of generalization contexts is simple in the sense that it caches generalizations and the context in which they appeared to give upper as well as lower bounds for the current generalization and therefore drastically reduce the number of literals that are tested for dropping. It is also general in the sense that it can be applied to any IC3 algorithm as it operates on the elementary data structures *cube* and *frame*, which are identical in IC3CFA thanks to our DNF-based decomposition, and IC3 algorithms for word- and bit-level verification [3, 4, 7, 9, 12] that operate on cubes and clauses.

References

1. Competition on software verification (SV-COMP). `https://sv-comp.sosy-lab.org/2017/`, accessed: 2017-01-23
2. 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)
3. Birgmeier, J., Bradley, A.R., Weissenbacher, G.: Counterexample to induction-guided abstraction-refinement (CTIGAR). In: CAV. Lecture Notes in Computer Science, vol. 8559, pp. 831–848. Springer (2014)
4. Bradley, A.R.: SAT-based model checking without unrolling. In: VMCAI. Lecture Notes in Computer Science, vol. 6538, pp. 70–87. Springer (2011)
5. Bradley, A.R., Manna, Z.: Checking safety by inductive generalization of counterexamples to induction. In: FMCAD. pp. 173–180. IEEE Computer Society (2007)
6. Cimatti, A., Griggio, A.: Software model checking via IC3. In: CAV. Lecture Notes in Computer Science, vol. 7358, pp. 277–293. Springer (2012)
7. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Infinite-state invariant checking with IC3 and predicate abstraction. Formal Methods in System Design 49(3), 190–218 (2016)
8. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)
9. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: FMCAD. pp. 125–134. FMCAD Inc. (2011)
10. Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: generating compact verification conditions. In: POPL. pp. 193–205. ACM (2001)
11. Griggio, A., Roveri, M.: Comparing different variants of the IC3 algorithm for hardware model checking. IEEE Trans. on CAD of Integrated Circuits and Systems 35(6), 1026–1039 (2016)
12. Gurfinkel, A., Ivrii, A.: Pushing to the top. In: FMCAD. pp. 65–72. IEEE (2015)
13. Hassan, Z., Bradley, A.R., Somenzi, F.: Better generalization in IC3. In: FMCAD. pp. 157–164. IEEE (2013)
14. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: SAT. Lecture Notes in Computer Science, vol. 7317, pp. 157–171. Springer (2012)
15. Lange, T., Neuhäußer, M.R., Noll, T.: IC3 software model checking on control flow automata. In: FMCAD. pp. 97–104. IEEE (2015)
16. Leino, K.R.M.: Efficient weakest preconditions. Inf. Process. Lett. 93(6), 281–288 (2005)
17. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: intermediate language and tools for analysis and transformation of C programs. In: CC. Lecture Notes in Computer Science, vol. 2304, pp. 213–228. Springer (2002)
18. Welp, T., Kuehlmann, A.: QF BV model checking with property directed reachability. In: DATE. pp. 791–796. EDA Consortium San Jose, CA, USA / ACM DL (2013)

A Appendix: Proofs

Proof (Lemma 1). Given the premises

$$\begin{aligned} &a)\ F_1 \wedge T_1 \wedge g_1' \text{ unsatisfiable} \\ &b)\ F_2 \wedge \neg g_2 \wedge T_2 \wedge g_2' \text{ unsatisfiable} \end{aligned}$$

the conclusion must be invalid, i.e. there must be a satisfying assignment to

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

Given the premises a) and b) we cannot preclude that this formula has no satisfying assignment. For example we don't know anything about the relationship between F_2 and $\neg g_1$, so they may share common states. One of these common states may also have a T_2-successor that is in $g_1' \wedge g_2'$ and so the formula *may* be satisfied. This means that the conclusion of Lemma 1 is not valid and thus generalization does not preserve inductivity as given in Def. 4.

To prove the correctness of Th. 1, we use the following auxiliary lemma:

Lemma 5.
For GCL command $\mathcal{C}$ without $\square$ and cube c: $dnf(wp(\mathcal{C}, c)) = wp(\mathcal{C}, c)$, where $dnf(\varphi)$ converts the quantifier-free first-order formula φ into DNF.

Proof (Lemma 5). Note that the GCL command $\mathcal{C}$ contains no choice. We prove the lemma by structural induction over $\mathcal{C}$ without the choice case.

– $\mathcal{C} = (\texttt{Assume}\ b)$:

$$\begin{aligned} & dnf(wp(\texttt{Assume}\ b, c)) \\ =\ & dnf(c \wedge b) \\ =\ & c \wedge b \\ =\ & wp(\texttt{Assume}\ b, c) \end{aligned}$$

– $\mathcal{C} = (x := a)$:

$$\begin{aligned} & dnf(wp(x := a, c)) \\ =\ & dnf(c[x \mapsto a]) \\ =\ & c[x \mapsto a] \\ =\ & wp(x := a, c) \end{aligned}$$

– $\mathcal{C} = (\mathcal{C}_1; \mathcal{C}_2)$:

$$\begin{aligned}
& \mathit{dnf}(\mathit{wp}(\mathcal{C}_1;\mathcal{C}_2, c)) \\
= \;& \mathit{dnf}(\mathit{wp}(\mathcal{C}_1, \mathit{wp}(\mathcal{C}_2, c))) \\
= \;& \mathit{wp}(\mathcal{C}_1, \mathit{wp}(\mathcal{C}_2, c)) && \text{(by hypothesis)} \\
= \;& \mathit{wp}(\mathcal{C}_1;\mathcal{C}_2, c)
\end{aligned}$$

Proof (Theorem 1).

$$\begin{aligned}
& \mathit{dnf}(\mathit{wp}(\mathcal{C}, c)) \\
= \;& \mathit{dnf}(\bigvee\{\mathit{wp}(\mathcal{C}', c) \mid \mathcal{C}' \in \mathit{split}(\mathcal{C})\}) \\
= \;& \bigvee\{\mathit{dnf}(\mathit{wp}(\mathcal{C}', c)) \mid \mathcal{C}' \in \mathit{split}(\mathcal{C})\} \\
= \;& \bigvee\{\mathit{wp}(\mathcal{C}', c) \mid \mathcal{C}' \in \mathit{split}(\mathcal{C})\} && \text{(Lemma 5)}
\end{aligned}$$

Proof (Lemma 2).

$$\begin{aligned}
& \exists \bar{c} \in \mathit{Cube}.\ \neg\bar{c} \in F_{(i-1,\ell)} \wedge \bar{c} \subseteq \mathit{wp}(\mathcal{C}, c) \\
\Rightarrow \;& \exists \bar{c} \in \mathit{Cube}.\ \mathit{unsat}(F_{(i-1,\ell)} \wedge \bar{c}) \wedge \bar{c} \subseteq \mathit{wp}(\mathcal{C}, c) \\
\Rightarrow \;& \mathit{unsat}(F_{(i-1,\ell)} \wedge \mathit{wp}(\mathcal{C}, c)) \\
\Rightarrow \;& \mathit{relIndAlt}_e(F_{(i-1,\ell)}, c) && \text{(Def. 7)} \\
\Leftrightarrow \;& \mathit{relInd}_e(F_{(i-1,\ell)}, c) && \text{(Lemma 4)}
\end{aligned}$$

Proof (Lemma 3). We prove Lemma 3 by structural induction over GCL command $\mathcal{C}$ without choice.

– $\mathcal{C} = \texttt{Assume}\ b$:

$$\begin{aligned}
& \mathit{wp}(\texttt{Assume}\ b, c_1 \wedge c_2) \\
= \;& (c_1 \wedge c_2) \wedge b \\
= \;& (c_1 \wedge b) \wedge (c_2 \wedge b) \\
= \;& \mathit{wp}(\texttt{Assume}\ b, c_1) \wedge \mathit{wp}(\texttt{Assume}\ b, c_2)
\end{aligned}$$

– $\mathcal{C} = x := a$:

$$\begin{aligned}
& \mathit{wp}(x := a, c_1 \wedge c_2) \\
= \;& (c_1 \wedge c_2)[x \mapsto a] \\
= \;& c_1[x \mapsto a] \wedge c_2[x \mapsto a] \\
= \;& \mathit{wp}(x := a, c_1) \wedge \mathit{wp}(x := a, c_2)
\end{aligned}$$

– $\mathcal{C} = \mathcal{C}_1; \mathcal{C}_2$:

$$\begin{aligned}
& \mathit{wp}(\mathcal{C}_1;\mathcal{C}_2, c_1 \wedge c_2) \\
= \;& \mathit{wp}(\mathcal{C}_1, \mathit{wp}(\mathcal{C}_2, c_1 \wedge c_2)) \\
= \;& \mathit{wp}(\mathcal{C}_1, \mathit{wp}(\mathcal{C}_2, c_1) \wedge \mathit{wp}(\mathcal{C}_2, c_2)) && \text{(by hypothesis)} \\
= \;& \mathit{wp}(\mathcal{C}_1, \mathit{wp}(\mathcal{C}_2, c_1)) \wedge \mathit{wp}(\mathcal{C}_1, \mathit{wp}(\mathcal{C}_2, c_2)) && \text{(by hypothesis)} \\
= \;& \mathit{wp}(\mathcal{C}_1;\mathcal{C}_2, c_1) \wedge \mathit{wp}(\mathcal{C}_1;\mathcal{C}_2, c_2)
\end{aligned}$$

Proof (Theorem 2). Let $\widehat{c}$ be a cube and $\mathcal{C}$ be a GCL command without a choice

$$\begin{aligned}
& \neg wp(\mathcal{C}, \widehat{c}) \in F_{(i-1,\ell)} \wedge wp(\mathcal{C}, \widehat{c}) \subseteq wp(\mathcal{C}, c) & \\
\Rightarrow\ & relInd_e(F_{(i-1,\ell)}, \widehat{c}) \wedge wp(\mathcal{C}, \widehat{c}) \subseteq wp(\mathcal{C}, c) & \text{(Lemma 2)} \\
\Rightarrow\ & relInd_e(F_{(i-1,\ell)}, \widehat{c}) \wedge \widehat{c} \subseteq c & \text{(Monotonicity)} \\
\Rightarrow\ & gen(F_{(i,\ell')}, c, e) = \widehat{c} & \text{(Def. 5)}
\end{aligned}$$

Proof (Lemma 4).

$$\begin{aligned}
& relIndAlt_e(F_{(i-1,\ell)}, c) & \\
\Leftrightarrow\ & unsat(F_{(i-1,\ell)} \wedge wp(\mathcal{C}, c)) & \text{(Def. 7)} \\
\Leftrightarrow\ & unsat(F_{(i-1,\ell)} \wedge T_e \wedge c') & \text{(WP)} \\
\Leftrightarrow\ & relInd_e(F_{(i-1,\ell)}, c) & \text{(Def. 4)}
\end{aligned}$$

Proof (Theorem 3).

$$\begin{aligned}
& (c, e, F, g) \in GC_i \wedge F_{(i-1,\ell)} \sqsubseteq F & \\
\Rightarrow\ & \exists j \leq i.\ gen(F_{(j,\ell')}, c, e) = g & \\
& \wedge F = F_{(j-1,\ell)} \wedge F_{(i-1,\ell)} \sqsubseteq F & \text{(Def. 8)} \\
\Leftrightarrow\ & relInd_e(F_{(j-1,\ell)}, g) \wedge F = F_{(j-1,\ell)} \wedge F_{(i-1,\ell)} \sqsubseteq F & \text{(Def. 5)} \\
\Rightarrow\ & relInd_e(F_{(i-1,\ell)}, g) & \\
\Rightarrow\ & gen(F_{(i,\ell')}, c, e) = g & \text{(Def. 5)}
\end{aligned}$$

Proof (Corollary 1).

$$\begin{aligned}
& (c, e, F, g) \in GC_i \wedge F_{(i-1,\ell)} \sqsubseteq F \wedge g \subseteq \widehat{c} & \\
\Rightarrow\ & gen(F_{(i,\ell')}, c, e) = g \wedge g \subseteq \widehat{c} & \text{(Th. 3)} \\
\Rightarrow\ & gen(F_{(i,\ell')}, \widehat{c}, e) = g & \text{(Def. 5 (1))}
\end{aligned}$$

Proof (Theorem 4).

$$\begin{aligned}
& (c, e, F, g) \in GC_i \wedge F_{(i-1,\ell)} \sqsubseteq F \wedge \widehat{g} \subset g & \\
\Rightarrow\ & \exists j \leq i.\ gen(F_{(j,\ell')}, c, e) = g & \\
& \wedge F = F_{(j-1,\ell)} \wedge F_{(i-1,\ell)} \sqsubseteq F \wedge \widehat{g} \subset g & \text{(Def. 8)} \\
\Rightarrow\ & \exists j \leq i.\ gen(F_{(j,\ell')}, c, e) \neq \widehat{g} & \\
& \wedge F = F_{(j-1,\ell)}) \wedge F_{(i-1,\ell)} \sqsubseteq F & \\
\Leftrightarrow\ & \neg relInd_e(F_{(j-1,\ell)}, \widehat{g}) & \text{(Def. 5)} \\
\Rightarrow\ & \neg relInd_e(F_{(i-1,\ell)}, \widehat{g}) & \\
\Rightarrow\ & gen(F_{(i,\ell')}, c, e) \neq \widehat{g} & \text{(Def. 5)}
\end{aligned}$$