ICP and IC3 with Stronger Generalization

Felix Winterer¹, Tobias Seufert¹, Karsten Scheibler², Tino Teige², Christoph Scholl¹, Bernd Becker¹
¹ University of Freiburg, Freiburg, Germany
² BTC Embedded Systems AG, Oldenburg, Germany
{winteref, seufert, scholl, becker}@informatik.uni-freiburg.de
{karsten.scheibler, tino.teige}@btc-es.de

Abstract

Most recently, IC3 was integrated into the SMT solver iSAT3. Thus, iSAT3+IC3 introduces the first IC3 variant based on interval abstraction and Interval Constraint Propagation (ICP). As strong generalization is one of the key aspects for the IC3 algorithm to be successful, we integrate two additional generalization schemes from literature into iSAT3+IC3: Inductive Generalization and Counterexamples To Generalization (CTG). Furthermore, we evaluate the benefits and the drawbacks of different variants of these methods in the context of interval abstraction and ICP.

1 Introduction

Without doubt, IC3 [5, 16] is currently the most efficient engine for checking safety properties in sequential Hardware Verification. During the last years IC3 has been lifted to various domains [12, 25, 27, 4] and is also successfully applied to Software Verification with powerful underlying SAT Modulo Theories (SMT) solvers like Mathsat, Z3, or SMTInterpol [13, 15, 11] to just name a few. Most recently we presented the first incarnation of IC3 based on interval abstraction and Interval Constraint Propagation (ICP) [1]. We called our approach iSAT3+IC3 as IC3 was integrated into the SMT solver iSAT3 [33], which bases on interval arithmetic reasoning. Additionally, iSAT3+IC3 provides native support for floating point arithmetic making it a good fit for software verification [2]. Besides its core SMT solver component, iSAT3 provides a portfolio of model checkers using Bounded Model Checking (BMC) [3], Craig Interpolation [29] and k-Induction [18]. Besides CBMC [14, 37], it is part of the BTC EmbeddedPlatform® (EP) and is successfully used for Dead Code Detection [31] in an industrial setting.

iSAT3 can be seen as the first incarnation of Abstract Conflict-Driven Clause Learning (ACDCL [7]) based on interval abstraction as the iSAT algorithm [20] was proposed several years before [7]. Supporting a theory in iSAT3 is only a matter of ICP-contractors for its operations.

In [36] we gave an outline of the features of iSAT3+IC3 with contributions on bit-level IC3 as well as new methods which specifically apply to interval arithmetic reasoning and ICP. Especially when it comes to the generalization of learned clauses – which represent the overapproximations of reachable states computed and strengthened by IC3 – iSAT3+IC3 extends known bit-level approaches and introduces a new theory-aware technique of relaxing interval bounds, so called Bound Generalization. However, there are well-tried techniques which naturally integrate with bound generalization and allow for even stronger generalization, that have not been implemented in iSAT3+IC3 so far [36]. Here, we implement Inductive Generalization [5, 16] as well as Down() [6] and its extension to Counterexamples to Generalization (CTG) [23] in the context of ICP and also general transition relations (which do not necessarily behave like functions). We show that we can further improve the powerful iSAT3+IC3 implementation with stronger generalization and give a thorough analysis of different variants of Inductive Generalization and CTG in the context of dead code detection. Further we analyze their interplay with ICP and bound generalization.

Related Work. Inductive Generalization as well as Counterexamples To Generalization are popular techniques which are used by a wide range of hardware verification tools implementing IC3 [22, 8, 9]. Furthermore, in [21] a survey is presented for different IC3 variants on bit-level, also including some configurations of inductive generalization, Down() and CTG.

These techniques have also successfully been lifted to the SMT domain and are used in software verification approaches like [4]. However, to the best of our knowledge, there is no implementation yet which is based on interval abstraction and ICP with native support for floating point arithmetic. In most approaches (e.g. [12, 25, 27, 4, 26]) floating point arithmetic is usually approximated over the reals which is not suitable for directly detecting dead code in floating-point programs¹.

Furthermore, it still remains unclear, how these techniques for stronger generalization apply to interval abstraction and especially the iSAT algorithm.

¹For example, when comparing two large numbers like 10²⁰ and 10²⁰ + 1 in an if condition, they would be considered as equal under 64 bit floating-point arithmetic (with round-to-nearest). In contrast, using real-valued arithmetic, they are not equal – leading to spuriously detected dead code [36].
Structure of the paper. In Section 2 we provide some preliminaries including information about the SMT solver iSAT3 and its already existing IC3 extension. The methods newly integrated into iSAT3+IC3 for stronger generalization are presented in Section 3 and experimentally evaluated in Section 4. Finally, the paper is concluded in Section 5.

2 Preliminaries

As this paper is an extension of [36], it uses the same notations and bases on the same preliminaries as well as its contributions. To make this paper stand on its own, we will revisit most of the aspects here.

2.1 SAT and Notations

The Boolean satisfiability problem (SAT) is the problem of deciding whether a Boolean formula \( F \) is satisfiable or not. The Boolean formula \( F \) is satisfied iff there exists an assignment for its Boolean variables such that \( F \) evaluates to true. State-of-the-art solvers of the SAT problem build on Conflict-Driven Clause Learning (CDCL) [40] and require the formula to be in conjunctive normal form (CNF). A Boolean formula in CNF is a conjunction of clauses. Clauses are disjunctions of literals, literals represent a Boolean variable or its negation. Any Boolean formula may be transformed to CNF by applying the Tseitin-transformation [42].

In this paper we use upper case letters to denote formulas. For literals, we use lower case letters – except \( i, j, k, m \) and \( n \) which we use for indices and \( x \) which we use for non-Boolean variables. We also use lower case letters for sets of Boolean variables, e.g. \( F \). Furthermore, we denote clauses by a tilde-decorated lower case letter, e.g. \( \bar{c} \). Similarly, we denote a cube (which is a conjunction of literals) by \( \bar{c} \). A negated clause is a cube and vice versa. Hence, for simplification, we consider a negated clause \( \bar{c} \) as a cube containing the negated literals of \( \bar{c} \) and the other way around. Additionally, when writing \( F(\bar{c}) \) we indicate that the formula \( F \) depends on the Boolean variables from the set \( \bar{c} \). If the literals of clause \( \bar{c} \) and cube \( \bar{c} \) belong to the Boolean variables in \( \bar{q} \), we write \( \bar{c}(\bar{q}) \) and \( \bar{c}(\bar{q}) \).

2.2 IC3

We consider the verification of safety properties. Thus, we have to prove that a certain safety property holds on all possible execution paths. This can be reduced to a simple reachability problem. We ask the question: if the system starts in a safe or good state, is it possible to reach an unsafe or bad state in a finite number of transition steps? IC3 [5] is a way to check whether a system can reach a bad state. To achieve this, we encode the states with a set of Boolean variables (denoted by \( \bar{s} \)). Furthermore, we identify the initial states as well as the transition relation with the predicates \( I(\bar{s}) \) and \( T(\bar{s}, \bar{s}+) \) while we represent the set of good safe states with the property \( P(\bar{s}) \). For brevity, we neglect that the transition relation usually considers variables representing inputs as well. Thus, \( P \) is violated if a bad unsafe state is reached.

To conclude that the bad states are generally unreachable resp. that \( P \) is proven, we need to find a formula \( F \) which is a safe inductive invariant, i.e. \( I \implies F, F \implies P, \) and \( F(\bar{s}_0) \land T(\bar{s}_0, \bar{s}_1) \land \neg F(\bar{s}_1) \) is unsatisfiable.

In order to obtain such an \( F \), IC3 [5] builds a sequence of frame formulas \( F_i \) with \( F_0(\bar{s}) = I(\bar{s}) \) and \( F_i(\bar{s}) \implies F_{i+1}(\bar{s}) \). Each \( F_i \) is an overapproximation of the good states reachable in up to \( i \) transition steps. While other SAT-based model checking approaches, like Bounded Model Checking (BMC) [3] or its unbounded extensions (k-Induction [18] and Craig Interpolation [29]), unroll the transition relation, IC3 considers only one transition relation at a time. IC3 explicitly enumerates predecessor states of the unsafe states in a depth-first search manner, and incrementally strengths the \( F_i \). Thus, compared to BMC, k-induction, or Craig Interpolation, IC3 requires much more but less complex solver calls.

Before IC3 starts, the formulas \( I(\bar{s}_0) \land \neg P(\bar{s}_0) \) and \( I(\bar{s}_0) \land T(\bar{s}_0, \bar{s}_1) \land \neg P(\bar{s}_1) \) have to be solved to ensure that \( P \) is not violated in up to one transition step. We present the pseudo-code formulation of IC3 from [36], see also [5].

Procedure MAIN():

1. \( F_0(\bar{s}) := I(\bar{s}), F_1(\bar{s}) := P(\bar{s}), \ i := 1 \)
2. Solve \( F_i(\bar{s}_0) \land T(\bar{s}_0, \bar{s}_1) \land \neg P(\bar{s}_1) \)
   (a) If satisfiable: extract \( \bar{c}(\bar{s}_0) \), DFS(\( \bar{c}, i - 1 \))
   (b) If unsatisfiable: \( F_{i+1}(\bar{s}) := P(\bar{s}), \ \text{PUSH}(i), \ i := i + 1 \)
3. goto 2.

Procedure DFS(\( \bar{c}, i \)):

1. Solve \( F_i(\bar{s}_0) \land \neg \bar{c}(\bar{s}_0) \land T(\bar{s}_0, \bar{s}_1) \land \bar{c}(\bar{s}_1) \)
   (a) If satisfiable and \( i = 0 \): \( P \ \text{violated}, \ \text{exit} \)
   (b) If satisfiable and \( i > 0 \): extract \( \bar{c}(\bar{s}_0) \), DFS(\( \bar{c}, i - 1 \))
   (c) If unsatisfiable:
      for all \( j \in \{1, \ldots, i + 1\} : F_j(\bar{s}) := F_j(\bar{s}) \land \neg \bar{c}(\bar{s}) \)
2. Return

Procedure PUSH(\( i \)):

1. \( j := 1 \)
2. \( f := \text{true} \)
3. For each clause \( \bar{c}(\bar{s}) \) in \( F_j(\bar{s}) \) which is not in \( F_{j+1}(\bar{s}) \)
   solve \( F_j(\bar{s}_0) \land T(\bar{s}_0, \bar{s}_1) \land \neg \bar{c}(\bar{s}_1) \)
   (a) If satisfiable: \( f := \text{false} \)
   (b) If unsatisfiable: \( F_{j+1}(\bar{s}) := F_{j+1}(\bar{s}) \land \bar{c}(\bar{s}) \)
4. If \( f = \text{true} \): \( P \ \text{proven}, \ \text{exit} \)
5. If \( j < i \): \( j := j + 1, \ \text{goto} \ 2 \)
6. Return
IC3 starts in MAIN() with \( i = 1 \) and searches \( F_i(\vec{x}) \) for a state \( \vec{c}(\vec{x}) \) which is able to reach a bad state in one transition step. Such a state is called a proof obligation, because we have to prove its unreachability in order to rule out a counterexample. If such a state exists, IC3 performs a depth-first search and recursively checks whether \( \vec{c}(\vec{x}) \) has itself a predecessor that is reachable from the initial states \( F_0 \). If this is the case, a counterexample is found and IC3 concludes that the system is unsafe. Otherwise, one or more \( F_i \) are strengthened by adding a clause corresponding to a blocked cube which represents a state being unreachable in up to \( i \) transition steps. In case MAIN() does not find a predecessor, IC3 adds a new frame formula and calls PUSH() in order to push all blocked cubes to higher frames. Furthermore, PUSH() checks whether \( F_i(\vec{s}_0) \land T(\vec{s}_0, \vec{s}_1) \land \neg F_j(\vec{s}_1) \) is unsatisfiable\(^2\) to prove that \( P \) can never be violated and the system is safe.

We remark that this is just the basic workflow of IC3. Thus, we neglect details like remembering previously generated proof obligations\(^3\), pushing blocked cubes already in DFS(), and generalizing states.

Generalization however, is very important for the efficiency of IC3. Instead of enumerating complete assignments to the state variables resp. individual states, IC3 operates on minimal assignments resp. preferably expressible sets of states. This is achieved by generalizing proof obligations as well as blocked cubes. Generalized blocked cubes are done by removing literals from \( \vec{c}(\vec{x}) \) such that \( F_i(\vec{s}_0) \land \neg \vec{c}(\vec{s}_0) \land T(\vec{s}_0, \vec{s}_1) \land \vec{c}(\vec{s}_1) \) stays unsatisfiable\(^4\).

Regarding the generalization of proof obligations, common bit-level IC3 implementations apply lifting from [32] here. It was first used in the IC3 context by [10]. After the extraction of the state \( \vec{c} \) and its predecessor \( \vec{e} \), it is checked whether \( \vec{e}(\vec{s}_0) \land T(\vec{s}_0, \vec{s}_1) \land \neg \vec{c}(\vec{s}_1) \) stays unsatisfiable while removing literals from \( \vec{e}(\vec{s}_0) \). If the transition relation behaves like a function (or more precisely, is left-total), there exists a successor state for all states. Hence, the formula remaining unsatisfiable implies that each state in \( \vec{e}(\vec{s}_0) \) has a successor in \( \vec{e}(\vec{s}_0) \) – therefore, \( \vec{e}(\vec{s}_0) \) remains a valid proof obligation after removing literals.

### 2.3 iSAT3

iSAT3 is based on modern CDCL-style solvers [40]. Hence it incorporates its standard components, which are (1) a decision heuristics, (2) Boolean Constraint Propagation (BCP) – used to deduce consequences triggered by a current partial assignment to the Boolean variables – and (3) a conflict analysis which derives and learns conflict clauses from assignments unsatisfying at least one clause. The iSAT algorithm [20, 24, 41, 28, 19] lifts this scheme to SAT Modulo Theories (SMT). It does so by implementing Interval Constraint Propagation (ICP) [1] which allows for handling Boolean combinations of theory atoms. Besides Boolean variables, arithmetic reasoning requires additional variable types. iSAT3 supports bounded integer- and real-valued variables as well as integers with a fixed bit width [35] and floating-point variables [34].

During the search process the set of possible solutions for these variable types is overapproximated with intervals – i.e. iSAT3 dynamically introduces literals representing the lower and upper bounds of these intervals per variable. These literals are called simple bound literals.

For example, when introducing the literals \( l_1 \) and \( \neg l_2 \) with \( l_1 \leftrightarrow (x \geq 5) \) and \( l_2 \leftrightarrow (x > 7) \), this restricts the value range of variable \( x \) to the interval \([5, 7]\).

Therefore, the solver core of iSAT3 still operates on literals as a CDCL-style SAT solver, but additionally keeps a connection between simple bound literals and their theory variables. Hence, clauses containing such literals exclude hyper-boxes from the search space. Furthermore, the theory atoms are decomposed using a Tseitin-like transformation of arithmetic operations. The transformation introduces an auxiliary variable and assigns it to the result of each arithmetic operation.

Deduction of the consequences for each supported operation is conducted by the so called ICP-contractor. It generates new clauses and simple bound literals by overapproximating the behavior of an operator in interval arithmetic. For example, for \( x_1 \in [0, 9] \), \( x_2 \in [5, 7] \), \( x_3 \in [1, 3] \) and the primitive constraint \( x_1 = x_2 + x_3 \) the deduction \((x_2 \geq 5) \land (x_1 \geq 1) \implies (x_1 \geq 6)\) can be performed [36]. Support for new operations is just a matter of adding new ICP-contractors to iSAT3.

Therefore iSAT3 differs from standard CDCL-style solvers by adding the following modifications to the basic building blocks [36]:

1. The decision heuristics is adapted to – besides deciding existing literals – perform interval splits by dynamically generating new simple bound literals and deciding them. For instance, having \( x \in [0.9] \), iSAT3 can introduce a literal \( l \) with \( l \leftrightarrow (x > 5) \). If iSAT3 decides \( \neg l \), we have \( x \in [0, 5] \).

2. ICP supplements BCP as an additional deduction mechanism to provide currently necessary clauses and simple bound literals. Once these are determined, BCP is able to carry on. Furthermore, so called bound-implication clauses are generated lazily. These clauses encode implications between simple bound literals belonging to the same theory variable, e.g. \((\neg(x > 7) \lor (x \geq 5))\).

3. Similar to CDCL-style solvers a 1UIP [43] conflict analysis is performed by analyzing the implication graph.

### 2.4 iSAT3+IC3

iSAT3 maps the interval bounds of each theory variable to simple bound literals. Thus, it encodes the values of
the theory state variables by a set of literals. This enables iSAT3 to perform literal-based IC3 in the same manner as described in Section 2.2.

Therefore iSAT3+IC3 [36] effectively incorporates known aspects of IC3 from SAT-based bit-level model checking with new elements which adapt IC3 to interval arithmetic and and IC3.

2.4.1 Literal Rotation and Literal Dropping

For the generalization of blocked cubes we consider the following unsatisfiable formula:

\( F(\vec{s}_0) \land \neg \vec{c}(\vec{s}_0) \land T(\vec{s}_0, \vec{s}_1) \land \vec{c}(\vec{s}_1) \).

The inclusion of \( \neg \vec{c}(\vec{s}_0) \) in the formula can be interpreted as bounded inductive reasoning [16] which even allows non-monotone reductions. This means that if a solver call with a cube \( \vec{c} \) having \( n \) literals is satisfiable, it might become unsatisfiable again with \( n-1 \) literals (see Section 3.1 for further discussion). The original iSAT3+IC3 leaves \( \neg \vec{c}(\vec{s}_0) \) unchanged performing only monotone reductions [36].

Bit-level approaches like [16, 38] exploit the fact that their underlying SAT-solver provides them with a so called final conflict clause resp. some kind of unsatisfiable core under the assumption \( \vec{c}(\vec{s}_1) \). However, this core is only minimal wrt. the order by which the assumption literals from \( \vec{c}(\vec{s}_1) \) are assigned before applying BCP.

In [16] for instance, the SAT solver MiniSat [17] is used which assigns all assumption literals before applying BCP. In contrast, iSAT3+IC3 performs so-called pseudo-decisions for each unassigned assumptions literal individually executing BCP after each such decision. Thus, it might happen that MiniSat is not able to detect as many redundant literals as the pseudo-decision based approach of iSAT3+IC3.

We revisit the motivational example from [36]. We consider the list of assumption literals \((l_1, l_2)\) and the following formula (with clause numbers in superscripts):

\[
(\neg l_1 \lor l_3)^{(1)} \land (\neg l_1 \lor l_4)^{(2)} \land (\neg l_1 \lor l_5)^{(3)} \land (\neg l_5 \lor \neg l_6)^{(4)} \\
\land (\neg l_2 \lor l_6)^{(5)} \land (\neg l_2 \lor l_4 \lor l_6)^{(6)}
\]

We first consider the case that \( l_1 \) and \( l_2 \) are assigned together. Thus, the clauses (1), (2), (3) and (5) become unit. Using BCP, the literals \( l_1, l_4, l_5 \) and \( l_6 \) are deduced leading to a conflict in clause (4). Analyzing the implication graph would reveal \( l_1 \) and \( l_2 \) to be responsible for the conflict while in fact \( l_2 \) is redundant. However, if only \( l_1 \) is assigned and BCP is applied directly afterwards, clause (6) is involved to provoke a conflict in (4) which reveals \( l_2 \) to be redundant using pseudo-decisions. But the success of this approach depends on the order of the assumption literals. When using the order \((l_2, l_1)\) it again seems that \( l_2 \) is essential for the conflict. Thus, to exploit this advantage, iSAT3+IC3 rotates the (initially pseudo-randomly shuffled) order of the assumption literals and performs multiple checks.

We present the details of literal rotation regarding an unsatisfiable formula \( G \) — with \( G = F(\vec{s}_0) \land \neg \vec{c}(\vec{s}_0) \land T(\vec{s}_0, \vec{s}_1) \land \vec{c}(\vec{s}_1) \) as the initial assumption literals — using the algorithm introduced in [36]:

1. \( k := 0 \)
2. \( G \) was already solved and is known to be unsatisfiable under the assumption literals \((l_1, \ldots, l_p)\). Thus, there exists an unsatisfied \( l_j \) being assigned by BCP, i.e. the pseudo-decision of \( l_j \) failed.
3. As there might be more than one unsatisfied assumption literal, consider the decision levels of these unsatisfied literals and select a literal \( l_j \) which was assigned on the lowest decision level.
4. Traverse the implication graph backwards to determine all pseudo-decisions \((l_1', \ldots, l_m')\) which are responsible for \( l_j \) being unsatisfied (this is similar to determining the final conflict-clause).
5. Use \((l_j, l_1', \ldots, l_m')\) as new list of assumption literals.
6. \( k := k + 1 \), if \( k < m + 1 \): solve \( G \), goto 2.
7. Return \((l_j, l_1', \ldots, l_m')\)

Determining whether formula \( G \) is unsatisfiable (first solver call) might be quite expensive. In contrast, the following solver calls performed during literal rotation are much cheaper in general because no regular decisions but only up to \( m+1 \) pseudo-decisions are performed in each iteration [36].

LITERAL DROPPING — in opposition to literal rotation — means that we remove a literal \( l \) from cube \( \vec{c}(\vec{s}) \) in the assumptions and check whether the solver call is still unsatisfiable. If this is the case, we remove the literal, if not, we keep it and undo its removal. We remark, that the original iSAT3+IC3 implementation always operates on the same formula, i.e. during literal dropping, it only alters the assumptions with cube \( \vec{c}(\vec{s}) \) and not its clause \( \neg \vec{c}(\vec{s}) \).

2.4.2 Literal Rotation with Bound Generalization

While the techniques from above also apply to the Boolean case, iSAT3+IC3 is still theory-aware and is able to complement bit-level generalization techniques by so called Bound Generalization [36]. If it is not possible to simply remove a simple bound literal \( l \) from a cube \( \vec{c} \), iSAT3+IC3 tries to replace \( l \) by a simple bound literal \( l' \) which represents a weaker bound, e.g. replacing \( (x \leq 5) \) with \( (x \leq 9) \). In this case, the number of assumption literals does not change. Nevertheless, the hyper-box represented by the cube of assumption literals was still enlarged by bound generalization.

iSAT3+IC3 integrates bound generalization into literal rotation as the rotation offers the chance to generalize every essential simple bound literal. When an unsatisfied assumption literal \( l_j \) is considered and it happens to be a simple bound literal, it is checked if \( l_j \) was assigned because of a bound-implication clause (cf. Section 2.3). If this is the case and \((b \lor \neg l_j)\) is such a clause, then \( b \) represents a weaker upper (lower) bound than \( l_j \) due to the nature
of bound implications. The assumption literals \((l_1, \ldots, l_m)\) imply \(\neg b\) which is why the assumption literal \(l_j\) can be replaced with \(b\), the weakest possible bound still causing a conflicting pseudo-decision.

2.4.3 Ungeneralizing Blocked Cubes

Because of \(F_0(\overrightarrow{s}) = I(\overrightarrow{s})\) and \(F_1(\overrightarrow{s}) = F_{i+1}(\overrightarrow{s})\) it is not allowed to strengthen an \(F_i\) with a blocked cube which excludes an initial state. iSAT3+IC3 does not prevent generalization which excludes initial states\(^5\) beforehand, but much rather „repairs“ generalized cubes which intersect with the initial states.

If \(\hat{c}(\overrightarrow{s})\) is the original blocked cube and \(\hat{c}'(\overrightarrow{s})\) the generalized version of it, then by construction \(\hat{c}(\overrightarrow{s})\) never contains an initial state. Therefore, \(I(\overrightarrow{s}) \land \hat{c}(\overrightarrow{s})\) is always unsatisfiable. By applying the generalization techniques from above, iSAT3+IC3 extracts a minimal set of literals from \(\hat{c}(\overrightarrow{s})\) which are responsible for making it disjoint from \(F(\overrightarrow{s})\), appends them to \(\hat{c}(\overrightarrow{s})\) and therefore ungeneralizes it just as much such that it doesn’t violate initiation anymore.

2.4.4 Generalizing Blocked Cubes

Here, we refine step 1c) of procedure DFS from Section 2.2. The iSAT3+IC3 approach applies the following algorithm, when attempting to generalize a blocked cube \(\hat{c}\) for which \(F_1(\overrightarrow{s}_0) \land \neg \hat{c}(\overrightarrow{s}_0) \land T(\overrightarrow{s}_0, \overrightarrow{s}_1) \land \hat{c}(\overrightarrow{s}_1)\) has been unsatisfiable beforehand. The procedure tries to propagate \(\hat{c}\) into the highest frame (with the maximum index) in which it still can be proven unreachable. After each UNSAT call to the solver, we call ANALYZE to extract a minimal conflicting set of \(\hat{c}(\overrightarrow{s}_1)\) literals by using literal rotation with bound generalization (see Section 2.4.2). Optionally, iSAT3+IC3 is able to do further literal dropping (without inductive generalization).

Furthermore, \(\hat{c}(\overrightarrow{s}_1) = l_1 \land \ldots \land l_n\) and we assume that cube \(\hat{c}(\overrightarrow{s}_1)\) is passed by assumptions to the iSAT3 solver core:

Procedure \textsc{Generalize}(\(\hat{c}, i\)):

1. \(\hat{c}_\text{old}(\overrightarrow{s}) = \hat{c}(\overrightarrow{s})\)
2. While \((F_1(\overrightarrow{s}_0) \land \neg \hat{c}_\text{old}(\overrightarrow{s}_0) \land T(\overrightarrow{s}_0, \overrightarrow{s}_1) \land \hat{c}(\overrightarrow{s}_1)\) UNSAT and \(i\) smaller than the number of frames)
   
   (a) \textsc{Analyze}(\(\hat{c}\));
   (b) If \(\text{performLiteralDropping}\)
       
       For each literal \(l\) in \(\hat{c}(\overrightarrow{s}_1)\):
       
       \(\hat{c}'(\overrightarrow{s}_1) := \hat{c}(\overrightarrow{s}_1) \setminus \{l\}\);
       
       If \((F_1(\overrightarrow{s}_0) \land \neg \hat{c}_\text{old}(\overrightarrow{s}_0) \land T(\overrightarrow{s}_0, \overrightarrow{s}_1) \land \hat{c}'(\overrightarrow{s}_1))\) is UNSAT:
       
       \(\hat{c}(\overrightarrow{s}) := \hat{c}'(\overrightarrow{s});\)
   (c) \(i := i + 1;\)

Procedure \textsc{Analyze}(\(c\) (reference)):

1. Rotate and minimize assumption literals \(\hat{c}\) iteratively according to Section 2.4.1 wrt. an unsatisfied assumption literal \(l_j\) on the lowest decision level;

\(^5\)For all blocked cubes \(\hat{c}(\overrightarrow{s})\) it has to hold that \(l(\overrightarrow{s}) \implies \neg \hat{c}(\overrightarrow{s})\).

2. If \(l_j\) is a simple bound literal: Replace \(l_j\) by the weakest bound \(b\) which still causes a conflicting pseudo-decision (see Section 2.4.2);
3. If \((\hat{c} \land I)\): Ungeneralize \(\hat{c}\) according to Section 2.4.3;

We remark, that cube \(\hat{c}\) is passed to \textsc{Analyze} by reference. The Boolean flag \text{performLiteralDropping} indicates whether literal dropping is used.

2.4.5 Generalizing Proof Obligations with GeNTR

Since iSAT3+IC3 operates on general transition relations and not exclusively on functions, it is not possible to use a lifting call \(\hat{e}(\overrightarrow{s}_0) \land T(\overrightarrow{s}_0, \overrightarrow{s}_1) \land \neg \hat{c}(\overrightarrow{s}_1)\) to generalize proof obligation cube \(\hat{e}(\overrightarrow{s}_0)\). Therefore, iSAT3+IC3 uses Generalization with a Negated Transition Relation (GeNTR).

It is obvious that, if a satisfying assignment for a formula \(G\) is conjoined to \(\neg G\), the resulting formula is unsatisfiable. Therefore, since \(\hat{e}(\overrightarrow{s}_0) \land \hat{c}(\overrightarrow{s}_1)\) is a satisfying assignment for \(\hat{e}(\overrightarrow{s}_0) \land T(\overrightarrow{s}_0, \overrightarrow{s}_1) \land \hat{c}(\overrightarrow{s}_1)\), the formula \(\hat{e}(\overrightarrow{s}_0) \land \neg T(\overrightarrow{s}_0, \overrightarrow{s}_1) \land \hat{c}(\overrightarrow{s}_1)\) is unsatisfiable.

Via literal rotation iSAT3+IC3 extracts from \(\hat{e}(\overrightarrow{s}_0)\) a reduced proof obligation cube based on this unsatisfiable call.

2.4.6 A Symbiosis of \(i\)-Induction and IC3

Another specialty of iSAT3+IC3 is an extension built to counteract state enumeration in the strengthening phase of IC3.

iSAT3+IC3 uses a dynamic suffix length, which means that if the IC3 algorithm gets „stuck“ while enumerating states, iSAT3+IC3 employs a target enlargement and searches bad states with more than just a single unrolling of the transition relation [36].

iSAT3+IC3 decides the suffix length based on a heuristic approach. If too many proof obligations are encountered without any progress, i.e. the number of open time frames remains at some value \(k\), iSAT3+IC3 aborts, starts over from scratch, and searches bad states with an unrolling of \(k\) instances of the transition relation.

3 iSAT3+IC3 with Stronger Generalization

As stated in Section 2.2 the generalization of clauses resp. blocked cubes in IC3 is of utmost importance. The approach of iSAT3+IC3 maps interval bounds of non-Boolean theory variables to simple bound literals and therefore iSAT3+IC3 can use all generalization techniques which apply to the original bit-level IC3 algorithm [5]. However, iSAT3+IC3 is still theory-aware and is able to complement bit-level generalization techniques by so-called Bound Generalization (see Section 2.4.2). In [36] common and well-tryed bit-level generalization techniques for IC3 have not been implemented yet. In the following we briefly describe the most successful techniques being Inductive Generalization [16, 5] which can be further extended by the Down() algorithm [6] as well as the notion of Counterexamples To Generalization (CTG).
We extend iSAT3+IC3 by integrating these into its existing generalization procedure for blocked cubes (see Section 2.4.4).

3.1 Inductive Generalization

As discussed in Section 2.2 IC3 tries to prove that the clause $\neg \hat{c}(\vec{s})$ is inductive relative to some $F_i$ by calling the solver with

$$F_i(\vec{s}_0) \land \neg \hat{c}(\vec{s}_0) \land T(\vec{s}_0, \vec{s}_1) \land \hat{c}(\vec{s}_1)$$

(1)

The term inductive relative to $F_i$ means, that $\neg \hat{c}(\vec{s}_0) \land F_i(\vec{s}_0) \land T(\vec{s}_0, \vec{s}_1) \implies \neg \hat{c}(\vec{s}_1)$ which is exactly the case if the Formula 1 is unsatisfiable. Note that for inductiveness, $F_0 \implies \neg \hat{c}$ has to hold. However, it holds by construction and has to be respected when generalizing $\hat{c}$ further. $F_0 \implies F_i$ is an invariant of IC3.

For the correctness of IC3 it is sufficient to just call

$$F_i(\vec{s}_0) \land T(\vec{s}_0, \vec{s}_1) \land \hat{c}(\vec{s}_1)$$

(2)

because if there is no transition from $F_i$ to $\hat{c}(\vec{s}_1)$ there is also none from $F_i \land \neg \hat{c}(\vec{s}_0)$. Confining $F_i$ to just the $\neg \hat{c}(\vec{s}_0)$ states though has the advantage, that the query from Formula 1 is more likely to be UNSAT than Formula 2.

More importantly if we drop literals from cube $\hat{c}(\vec{s})$ and therefore make it weaker, the clause $\neg \hat{c}(\vec{s})$ gets stronger. If the call remains UNSAT, the literal can be removed and the cube is generalized. The formula has non-monotone behavior in terms of UNSAT resp. SAT results of the solver. If Formula 1 is SAT, it is not necessarily true that it will remain SAT if we remove further literals. However, $\neg \hat{c}(\vec{s}) \land F_i(\vec{s})$ will always remain stronger than just $F_i(\vec{s})$.

Thus, we can conclude that it is beneficial to use Formula 1 when generalizing $\hat{c}(\vec{s})$.

3.1.1 Inductive Generalization with Literal Dropping

Of course if inductive generalization is combined with iterative literal dropping (as we do here), the resulting cube is not necessarily minimum but only minimal wrt. to the chosen literal order. Literal dropping means that we remove a literal $l$ from cube $\hat{c}(\vec{s}_1)$, i.e. $\hat{c}(\vec{s}_1) = \hat{c}(\vec{s}_1) \setminus \{l\}$, and call the solver on

$$F_i(\vec{s}_0) \land \neg \hat{c}(\vec{s}_0) \land T(\vec{s}_0, \vec{s}_1) \land \hat{c}(\vec{s}_1)$$

(3)

We remark that we also remove the literal from the clause, meaning that – in opposition to prior literal dropping in iSAT3+IC3 – we use $\neg \hat{c}(\vec{s}_0)$ instead of $\neg \hat{c}(\vec{s}_0)$ as in Section 2.4.4. If the solver call is still unsatisfiable, we are allowed to remove $l$, if not, we have to undo its removal. Furthermore, when we also remove literals from the clause $\neg \hat{c}(\vec{s})$, there are exponentially many possible literal orders, because due to the aforementioned non-monotonicity, we may probe subsets of literals of $\hat{c}(\vec{s})$ in a row in order to turn SAT results to UNSAT.

It is easy to see, that the properties from above directly apply to the idea of Bound Generalization, i.e. replacing simple bound literals by one of their weaker counterparts.

3.1.2 The Down() Algorithm

The authors of [6] propose an extension to standard literal dropping which is called Down(). As in standard literal dropping from Section 3.1.1, the Down() algorithm removes a literal $\hat{c}(\vec{s}) = \hat{c}(\vec{s}) \setminus \{l\}$ and checks for predecessors by solving Formula 3. However, if the result is satisfiable, it does not just conclude that we can not remove the literal and moves on, but much rather extracts a satisfying assignment $d(\vec{s}_0)$ from Formula 3 and applies $\hat{c}(\vec{s}) = \hat{c}(\vec{s}) \cup d(\vec{s}_0)$. The $\cup$-operator computes a so called join which is an overapproximation of the union by leaving only the literals in $\hat{c}(\vec{s}_1)$ which also occur in $d(\vec{s}_1)$.

Again Formula 3 is solved until the result is unsatisfiable or the result of the join intersects with the initial states. In the unsatisfiable case, the joined cube $\hat{c}(\vec{s})$ is unreachable from $F_i(\vec{s}_0) \land \neg \hat{c}(\vec{s}_0)$ and we can proceed with literal dropping. In the case that $\hat{c}(\vec{s})$ intersects with the initial states, we discard $\hat{c}(\vec{s})$ and proceed with the removal of another literal from $\hat{c}(\vec{s})$. Thus, the Down() algorithm aggressively tries to make a non-inductive clause inductive. It does so by excluding at least one reason (here $d(\vec{s})$) for non-inductiveness from the clause $\neg \hat{c}(\vec{s})$ resp. including it into the reachable cube $\hat{c}(\vec{s})$. When we check for inductiveness again, we may succeed or eventually have increased $\hat{c}(\vec{s})$ too much, such that it intersects with the initial states – in this case, our attempt failed and we have to roll back. An algorithmic and more detailed description of the approach is presented in Section 3.4.

We remark, that joining, i.e. creating an overapproximation of the union of $\hat{c}(\vec{s})$ and $d(\vec{s})$ within bit-level IC3 is done by taking all common literals. Interval arithmetic complicates things here a bit: To „join“ we take all common literals. However, to achieve a preferably precise theory-aware approximation, we consider the theory variables as well. For all theory variables for which both cubes share simple bound literals with the same polarity, we always take the weaker bound. We remark that for simple bounds it is determined by the polarity of the literal whether we have an upper bound or a lower bound. Thus, if both cubes contain an upper bound for variable $x$, we take the weaker upper bound and vice versa for lower bounds.

3.2 Counterexamples To Generalization (CTG)

A major insight from [23] was to put even more effort into the generalization of learned clauses (blocked cubes). We again consider the solver call with formula $F_i(\vec{s}_0) \land \neg \hat{c}(\vec{s}_0) \land T(\vec{s}_0, \vec{s}_1) \land \hat{c}(\vec{s}_1)$ which resolves to UNSAT iff. cube $\hat{c}(\vec{s}_1)$ is unreachable from $F_i(\vec{s}_0) \land \neg \hat{c}(\vec{s}_0)$ within one step.

As before, we assume that we have arrived at a cube $\hat{c}(\vec{s}_1)$ by removing literal $l$ from $\hat{c}(\vec{s}_1)$, i.e. $\hat{c}(\vec{s}_1) = \hat{c}(\vec{s}_1) \setminus \{l\}$. We further assume that now $\hat{c}(\vec{s}_1)$ is reachable from $F_i(\vec{s}_0) \land \neg \hat{c}(\vec{s}_0)$, i.e. Formula 3 is SAT. This means, that either the overapproximation $F_i(\vec{s})$ is to weak to prove that $\hat{c}(\vec{s})$ is unreachable in up to $i + 1$ steps or that it really is reachable from the initial states.

[23] test whether the former is the case by applying the
following algorithm which builds on the Down() algorithm described in Section 3.1.2. We extract the \( F_d(s_0) \)-predecessor, and thus the CTG, of \( \bar{c}(s_1) \) from the solver and call it \( \bar{d}(s) \). We try to block \( \bar{d}(s) \) from the state space by calling \( F_{i-1}(\bar{s}_0) \land \neg \bar{d}(\bar{s}_0) \land T(\bar{s}_0, \bar{s}_1) \land \bar{d}(\bar{s}_1) \). If this succeeds, Formula 3 is checked again and so on, until all CTGs are ruled out – then we stop and cube \( \bar{c}(s) \) may be blocked. If not, similar to Down(), we enlarge \( \bar{c}(s) \) by joining (convex overapproximation of the union) \( \bar{c}(s) \) with its CTG \( \bar{d}(s) \) and check Formula 3 again. As in Down() we stop and roll back, if the joined cube \( \bar{c}(s) \) intersects with the initial states. An algorithmic and more detailed description of the approach is presented in Section 3.4.

It is also possible to skip the join after a CTG has been discovered and has not been proven unreachable. We can either conclude that \( l \) can not be removed from \( \bar{c}(s) \) or try to recursively block the CTG in the depth first search (DFS, see Section 2.2) manner of the standard IC3 algorithm.

### Generalizing CTGs

All CTG cubes \( \bar{d}(s) \) result from a satisfying assignment of a formula \( F_d(s_0) \land \neg \bar{c}(s_0) \land T(s_0, s_1) \land \bar{c}(s_1) \). Thus generalizing CTGs can be done with the same methods as generalizing proof obligations, specifically it is possible to apply GeNTR (see Section 2.4.5). There is a subtle difference though: Assuming cube \( \bar{d}(s) \) were a proof obligation, it is most crucial to preserve that for every \( \bar{d}(s) \)-state there is a sequence which leads the system into an unsafe \( \neg P(s) \) state. Otherwise, we could encounter a spurious counterexample. However, since \( T \) does not necessarily represent a function, we cannot apply Lifting [32, 10] to proof obligations but have to stick to less aggressive GeNTR.

For CTGs though, it is not as important to only include states into \( \bar{d}(s) \) which really are predecessors of \( \bar{c}(s) \). Having a non-predecessor state in \( \bar{d}(s) \) can only result in wrongfully disallowing the removal of a literal \( l \) from cube\(^6\) \( \bar{c}(s) \). This means that we still overapproximate the reachable states and do not violate any IC3 invariant. Also, joins could lead to larger state sets which are however independently checked for reachability from \( F_l \) afterwards and their respective clause is only learned if this check succeeds.

Hence, for generalization of CTGs we are allowed to use (more aggressive) Lifting as well as (predecessor-relation preserving) GeNTR. In Section 4 we compare both versions.

### 3.3 Including Bound Generalization

As discussed in Section 2.4.2, iSAT3+IC3 is able to relax interval bounds if it is not able to remove an entire literal. This technique hand in hand with the bitlevel generalization techniques from above. At any place, where we remove a literal and check for inductiveness, it is also sound to replace a simple bound literal with a weaker bound.

Furthermore, bound generalization also integrates with literal rotation and can therefore be applied during the search for a minimal unsatisfiable core after an unsatisfiable solver call.

### 3.4 Overall Approach

Here we present our overall approach which integrates all techniques from above to generalize an unreachable blocked cube \( \bar{c}(s) \) in time frame \( i+1 \).

We call \( \text{GENERALIZE} \) to generalize a cube \( \bar{c} \) for which \( F_d(s_0) \land \neg \bar{c}(s_0) \land T(s_0, s_1) \land \bar{c}(s_1) \) has been unsatisfiable (UNSAT) beforehand. The procedure tries to propagate \( \bar{c} \) into the highest frame (with the maximum index) in which it still can be proven unreachable. After each UNSAT call to the solver, we call \( \text{ANALYZE} \) (which remains the same as in Section 2.4.4 to extract a minimal conflicting set of \( \bar{c} \) literals by using literal rotation and optionally bound generalization). After we found the highest frame and extracted the minimal conflicting set of \( \bar{c} \) literals, we further generalize \( \bar{c} \) in the literal dropping loop 2. After dropping a literal \( l \), it is possible to just check for reachability of the reduced cube or to additionally apply Down() or even a CTG analysis if the check states that \( \bar{c} \setminus \{l\} \) is reachable. All of this can be controlled via flags inside of the CTG procedure.

Furthermore, \( \bar{c}(s_1) = l_1 \land \ldots \land l_n \) and we assume that cube \( \bar{c}(s_1) \) is passed by assumptions to iSAT3:

**Procedure**

\[
\text{GENERALIZE}(\bar{c}, i, \text{recDepth})
\]

1. While \( (F_d(s_0) \land \neg \bar{c}(s_0) \land T(s_0, s_1) \land \bar{c}(s_1)) \) UNSAT and \( i \) smaller than the number of frames

   \[
   \text{ANALYZE}(\bar{c});
   \]

   \[
   i := i + 1;
   \]

2. For each literal \( l \) in \( \bar{c} \):

   \[
   (a) \quad \bar{c} := \bar{c} \setminus \{l\};
   \]

   \[
   \text{If (CTG}(i - 1, \bar{c}’, \text{recDepth}));
   \]

   \[
   \text{ANALYZE}(\bar{c}’);
   \]

   \[
   \bar{c} := \bar{c}’;
   \]

**Procedure**

\[
\text{ANALYZE}(\bar{c} \text{ (reference)})
\]

1. Rotate and minimize assumption literals \( \bar{c} \) iteratively according to Section 2.4.1 wrt. an unsatisfied assumption literal \( l_j \) on the lowest decision level;

2. If \( (l_j \text{ is a simple bound literal}) \): Replace \( l_j \) by the weakest bound \( b \) which still causes a conflicting pseudo-decision (see Section 2.4.2);

3. If \( (\bar{c} \land I) \): Ungeneralize \( \bar{c} \) according to Section 2.4.3;

**Procedure**

\[
\text{CTG}(i, \bar{c} \text{ (reference)}, \text{recDepth}) : \text{bool}
\]

1. \( \text{ctgs} := 0; \)

2. \( \text{if (\bar{c}(s) \land I(s))}: \text{return FALSE}; \)
3. Call the solver on \( F_i(s_0) \land \neg \epsilon(s_0) \land T(s_0, s_1) \land \epsilon'(s_1) \);

Case SAT:
(a) Extract a \( F_i(s)-\)predecessor cube \( d(\hat{s}) \);
(b) if \( \text{recDepth} > \text{maxRecDepth} \): return FALSE;
(c) check for a \( F_{i-1}(\hat{s})\)-predecessor of \( d(\hat{s}) \):
   if (there is a \( F_{i-1}(\hat{s})\)-predecessor of \( d(\hat{s}) \) or \( (d(\hat{s}) \land I(\hat{s})) \) or (ctgs > maxCTGs) or only-Down):
   \[ \text{ctgs} := 0; \]
   Join \( \epsilon(\hat{s}) = \epsilon(\hat{s}) \cup \delta(\hat{s}) \) and go to 2.:
   \[ \text{ctgs} := \text{ctgs} + 1; \]
   GENERALIZE(\( \hat{d}, i - 1, \text{recDepth} + 1 \)),
   learn new clause \( d \) and go to 2.;

Case UNSAT:
(a) return TRUE;

After every unsatisfiable solver call during inductive generalization, we always analyze the conflict for further generalization potential using literal rotation and bound generalization. For simplicity of the presentation we do not explicitly state but assume that every extracted predecessor state for Down() or CTG is generalized via GeNTR (see Section 2.4.5) or Lifting (see Section 3.2) depending on the configuration.

The global parameters onlyDown, maxRecDepth, as well as maxCTGs allow us to individually control the different modules. Hereby, the Boolean variable onlyDown decides whether only the Down() algorithm is used or it is complemented by CTG analysis. The counters maxRecDepth and maxCTGs control the recursion depth during CTG analysis resp. the number of CTGs that we are allowed to enumerate and to discharge.

The algorithm differs from the prior approach in iSAT3+IC3 (see Section 2.4.4) by the fact, that literal dropping is done only after the highest frame in which the cube can be proven unreachable has been identified (see step 1. of generalize). Thus, it is not done after each propagation of \( \epsilon \) to a higher frame.

It also incorporates inductive generalization, i.e. for checking whether cube \( \epsilon \) can be generalized to cube \( \epsilon' \) in frame \( i + 1 \) we always check \( F_i(s_0) \land \neg \epsilon'(s_0) \land T(s_0, s_1) \land \epsilon'(s_1) \) instead of \( F_i(s_0) \land \neg \epsilon(s_0) \land T(s_0, s_1) \land \epsilon'(s_1) \) as in Section 2.4.4. Furthermore, we additionally implemented Down() and CTG analysis for even stronger generalization.

4 Experimental Results

For our experiments we use the same benchmarks as [34], [36] and [30]. The benchmark set contains 8778 instances originating from TargetLink-generated production C code from the automotive domain containing a fair amount of floating-point arithmetic. Each benchmark describes a goal defined by a structural code coverage metric (e.g. MC/DC) which correlates to the reachability of a certain line of code. Thus, unreachable goals correspond to dead code.

The experiments of iSAT3 were performed on a cluster – with each cluster node having 64 GB RAM and two 8-core CPUs @2.6 GHz. We applied a time limit of 300 seconds and a memory limit of 4 GB per benchmark. The results for the EP-CBMC\(^7\) experiments were achieved on a similar CPU type (also @2.6 GHz) using the same limits.

**Configuration.** To evaluate the effectiveness of the newly integrated generalization methods into iSAT3+IC3, we performed many experiments (de-)activating and combining the different available generalization schemes. As using literal rotation, bound generalization and GeNTR has proven to be effective [36], we activate them in the baseline configuration of iSAT3+IC3. Furthermore, to avoid blowing up the parameter space, we fix the following parameter values (see Section 3.4) based on [23]: maxRecDepth = 1 and maxCTGs = 3 when CTG analysis is activated, otherwise maxRecDepth = 0.

In addition to iSAT3 using BMC, k-induction and Craig Interpolation (CI), we provide different settings of iSAT3+IC3: ind\(^8\) corresponds to initial suffix length of \( i \) (see Section 2.4.6). +abort performs restarts with longer suffixes which is also discussed in Section 2.4.6. +Id performs literal dropping in GeNTR, during ungeneralization of initial cubes (see Section 2.4.3), and in each propagation step during generalization of a blocked cube (as in Section 2.4.4), +ld once performs literal dropping only once after the propagation of a blocked cube is finished (as in Section 3) and not in GeNTR or ungeneralization, the option +ig activates inductive generalization (as described in Section 3.1) with basic literal dropping from Section 3.1.1, +j applies the Down() algorithm from Section 3.1.2, +ctg applies CTG analysis from Section 3.2, and finally, +lf replaces GeNTR by Lifting for generalizing CTGs as discussed in Section 3.2. Additionally, we evaluated EP-CBMC, which bases on CBMC and uses the EP tool chain to perform an additional k-Induction check [31].

**Presentation.** The results for iSAT3 for all different configurations can be found in Table 1. Column 2 and 3 show the number of detected counterexamples (CEX) and unreachable goals (i.e. dead code (DC)), resp., while column 4 contains the number of unresolved instances within the applied time or memory limit (T/M). The following column displays the number of uniquely detected dead code. By unique, we address results which have been found by neither a portfolio (PF) consisting of the first four solvers, namely EP-CBMC and iSAT3 using BMC, k-Induction or Craig Interpolation (we call this portfolio PF1) nor the (currently) best model checker portfolio from [36] consisting of all solvers from PF1 and iSAT3+IC3\(^1\) (ind2 +abort) (PF2).

\(^{7}\)Based on CBMC version 5.12.4 [14]

\(^{8}\)We decided to perform experiments for \( i = 0 \) and \( i = 2 \) as the former is the standard IC3 algorithm and the latter has proven to be the best suffix configuration in [36].
Table 1 Experimental results over 8778 benchmark instances (time limit 300s, memory limit 4 GB per instance)

<table>
<thead>
<tr>
<th>Method</th>
<th>CEX</th>
<th>DC</th>
<th>T/M</th>
<th>uniq. DC</th>
<th>time ratio</th>
<th>red. rate</th>
<th>impr. rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>iSAT3 BMC</td>
<td>7671</td>
<td>-</td>
<td>1107</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>iSAT3 k-Induction</td>
<td>7620</td>
<td>614</td>
<td>544</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>EP-CBMk k-Induction</td>
<td>7644</td>
<td>628</td>
<td>506</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>iSAT3 Craig Interpolation</td>
<td>7653</td>
<td>977</td>
<td>148</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>iSAT3+IC3 (ind2 +abort)</td>
<td>7533</td>
<td>997</td>
<td>248</td>
<td>23 / 0</td>
<td>0.92</td>
<td>0.05</td>
<td></td>
</tr>
<tr>
<td>iSAT3+IC3 (ind0 +abort +ig)</td>
<td>7264</td>
<td>1002</td>
<td>512</td>
<td>28 / 8</td>
<td>0.12</td>
<td>0.87</td>
<td>0.05</td>
</tr>
<tr>
<td>PF1 [Portfolio 1 (w/o iSAT3+IC3)]</td>
<td>7672</td>
<td>995</td>
<td>111</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PF2 [PF1 ∪ iSAT+IC3]</td>
<td>7672</td>
<td>1018</td>
<td>88</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PF3 [PF1 ∪ iSAT+IC3]</td>
<td>7672</td>
<td>1023</td>
<td>83</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PF4 [PF2 ∪ iSAT+IC3]</td>
<td>7672</td>
<td>1026</td>
<td>80</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Discussion

We start our analysis in the second part of Table 1. Going through the results step by step, we observe that performing literal dropping during cube propagation, GeNTR and ungeneralization (setting +ld) as in Section 2.4.4 is inferior – in terms of counterexamples as well as dead code – to doing it only once after the propagation effort (setting +ldonce). Furthermore, if we apply literal dropping with inductive generalization (+ig) we can improve these results even more (namely by 62 more solved instances). Apparently the best configuration found in [36] – iSAT3+IC3 ind0 (generalization of blocked cubes only via literal rotation with bound generalization) – for standard IC3 (no suffix, no restarts) has a lot of potential for improvement when applying a more efficient literal dropping with inductive generalization.

Interestingly, the methods for further generalization (+j, +ctg) of blocked cubes which achieve very strong results on bit-level seem to be less capable in finding dead code. However, in terms of solved instances in total (163 more than stand-alone iSAT3+IC3 ind0), the best configuration for iSAT3+IC3 ind0 applies inductive generalization (+ig) and CTG analysis (+ctg) and it does not make much of a difference whether or not we apply joins from Down() in addition (+j). However, in this configuration it is necessary...
to generalize CTGs more aggressively using Lifting (+lf) instead of GeNTR to achieve best results.

For further experiments incorporating the symbiosis of IC3 and k-Induction from Section 2.4.6 [36], we stick to the best two configurations of standard iSAT3+IC3: iSAT3+IC3 ind0 +ig regarding dead code detection (1002) and iSAT3+IC3 ind0 +ig +ctg +lf regarding counterexamples (7126) and also overall solved instances. When considering these configurations using longer suffixes and aborts, the results paint a slightly different picture.

We start by focusing on the +abort configuration of iSAT3+IC3 ind0. Again, it is beneficial to apply literal dropping with inductive generalization (+ig) but only for detecting dead code. Here, with 1002 detected instances of dead code, we achieve the best result over all configurations. iSAT3+IC3 ind0 +ig (without +abort) is also able to find 1002 instances of dead code but the number of found counterexamples benefits from aborting. On the other hand, using plain iSAT3+IC3 ind0 +abort detects even more counterexamples. When we look at +ig +ctg +lf (which we found best for iSAT3+IC3 ind0) the results are inferior to the other two +abort configurations. However for iSAT3+IC3 ind2 without +abort, +ig +ctg +lf is again the best configuration solving 38 more instances in total than plain iSAT3+IC3 ind2.

The iSAT3+IC3 ind2 +abort configuration is a really close call. Having the best overall result using inductive generalization (+ig), the most dead code (by one) is still found using the standard configuration.

Our results indicate, that the impact of more sophisticated cube generalization techniques decreases the more IC3 shifts from local reasoning to global reasoning with more complex solver calls for searching ¬P predecessors with longer suffixes, i.e. longer unrollings of T. This does not necessarily come as a surprise, since this may result in less local IC3-like cube blocking.

Additional Effort for Generalization. Applying additional generalization techniques consumes additional time. With 12%, simple literal dropping with (also without) inductive generalization (+ig) seems to be the cheapest of them. Using Down() (+j) in addition consumes 6% more of the runtime while the most expensive technique is the CTG analysis (+ctg) with up to 24% consumption.

Reduction Rates. Throughout the configurations we observe very similar reduction and bound generalization rates. It seems that the differences are not too significant and exceed the expressiveness of an average value. Nevertheless, the number of solved instances indicate the benefit of using (most of) the additional generalization techniques. This is an interesting issue which should be investigated further.

Another interesting result is, that – for most configurations – bound generalization is able to relax approximately 5% of the simple bound literals of a blocked cube on average. This is a significant amount, when considering that it is only applied to complement literal removal.

Portfolio Approach. In [36], we not only had a look at the results of different iSAT3+IC3 configurations but also used the best configuration to complement and improve a portfolio of the existing four solvers. We want to achieve the same here and thus consider the (until now) neglected column 5 (uniquely solved dead code instances) as well as the upper part of Table 1. PF1 is the portfolio of the four existing solvers which was complemented and improved by 23 solved DC instances by adding iSAT3+IC3 (ind2 +abort). This led to PF2 in [36]. Looking at the left part of column 5 reveals that using additional generalization techniques lead to even more uniquely solved DC instances compared to PF1 for some configurations. We selected iSAT3+IC3 ind0 +abort +ig with 28 instances as the best iSAT3+IC3 configuration for detecting dead code (iSAT3+IC3). Adding iSAT3+IC3 to PF1 instead of iSAT3+IC3 (which leads to PF3) reduces the number of unresolved instances by additional 6% compared to PF2 (from 88 to 83) increasing the benefit of having iSAT3+IC3 in the portfolio from 21% to 25%.

Even when compared to PF2 which includes iSAT3+IC3, the right part of column 5 shows that additional generalization is still able to offer some improvements regarding dead code detection. Our selected configuration iSAT3+IC3 still provides the best improvement by additional 8 solved DC instances. Thus, adding iSAT3+IC3 to PF2 (see PF4) reduces the number of unsolved instances even further, increasing the reduction compared to PF2 to 9% and the overall benefit to 28%.

5 Conclusion

We extended an interval abstraction and ICP based SMT-implementation of IC3 (iSAT3+IC3) with different variants of the most common bit-level techniques for generalizing blocked cubes (learned clauses). We discussed the generalization of CTGs in the case of general transition relations and the applicability of more aggressive generalization techniques than GeNTR. Furthermore, we gave an intensive evaluation of their potential in the context of iSAT3. By adding inductive generalization incorporating a more efficient literal dropping approach, we could significantly improve the best standard IC3 (without target enlargement) variant from iSAT3+IC3 by reducing the number of unsolved instances by 142 (828 to 686), including 29 more solved dead code instances.

By bringing CTG analysis into the equation, we were able to improve the capability of finding counterexamples in standard IC3 significantly, although this approach was not able to improve the amount of found dead code in comparison to plain inductive generalization with literal dropping. Generalizing CTGs more aggressively and taking the risk of adding spurious CTGs (which does not affect the correctness) seems to pay off and is always slightly improving the overall performance. Interestingly, we observed, that the Down() algorithm has no significant impact on the efficiency of any of our tested iSAT3+IC3 variants. Additionally, our results indicate that iSAT3+IC3 variants with longer suffixes (ind2 and abort versions) and therefore less local reasoning benefit less from more sophisticated tech-
niques for blocked cube generalization than standard IC3. Furthermore, we determined a iSAT3+IC3 configuration – namely iSAT3+IC3 \textit{ind0 +abort, +ig} – which is able to improve the best solver portfolio from \cite{36} even more, both by complementing the portfolio as well as replacing the currently used iSAT3+IC3 configuration.

It is fair to say, that results and insights from bit-level IC3 and its generalization techniques can not be directly transferred to iSAT3+IC3. We could not observe such a significant increase in efficiency as \cite{23, 21} did for instance by using CTG and \texttt{Down()}. However, employing inductive generalization paid off in general, definitely making its integration into iSAT3+IC3 worthwhile.

In the future we plan to integrate ReverseIC3 \cite{39} into iSAT3+IC3.

6 Literature


[38] Tobias Seufert and Christoph Scholl. fbPDR: In-depth combination of forward and backward analysis in Property Directed Reachability. In DATE 2019.


