Preprint from Proceedings of GI/ITG/GMM-Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen", March 2021

ICP and IC3 with Stronger Generalization

Felix Winterer[1], Tobias Seufert[1], Karsten Scheibler[2], Tino Teige[2], Christoph Scholl[1], Bernd Becker[1]
[1] University of Freiburg, Freiburg, Germany
{winteref, seufert, scholl, becker}@informatik.uni-freiburg.de
[2] BTC Embedded Systems AG, Oldenburg, Germany
{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 Embedded*Platform*® (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 programs[1].

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

[1] For example, when comparing two large numbers like 10^{20} and $10^{20}+1$ 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. $\vec{r}$. Furthermore, we denote clauses by a tilde-decorated lower case letter, e.g. $\tilde{c}$. Similarly, we denote a cube (which is a conjunction of literals) by $\hat{c}$. A negated clause is a cube and vice versa. Hence, for simplification, we consider a negated clause $\neg\tilde{c}$ as a cube containing the negated literals of $\tilde{c}$ and the other way around. Additionally, when writing $F(\vec{s})$ we indicate that the formula F depends on the Boolean variables from the set $\vec{s}$. If the literals of clause $\tilde{c}$ and cube $\hat{c}$ belong to the Boolean variables in $\vec{q}$, we write $\tilde{c}(\vec{q})$ and $\hat{c}(\vec{q})$.

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 $\vec{s}$). Furthermore, we identify the initial states as well as the transition relation with the predicates $I(\vec{s})$ and $T(\vec{s}_i,\vec{s}_{i+1})$ while we represent the set of good safe states with the property $P(\vec{s})$. 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 $F(\vec{s}_0) \wedge T(\vec{s}_0,\vec{s}_1) \wedge \neg F(\vec{s}_1)$ is unsatisfiable.

In order to obtain such an F, IC3 [5] builds a sequence of frame formulas F_i with $F_0(\vec{s}) = I(\vec{s})$ and $F_i(\vec{s}) \implies F_{i+1}(\vec{s})$. 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 strengthens 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 $I(\vec{s}_0) \wedge \neg P(\vec{s}_0)$ and $I(\vec{s}_0) \wedge T(\vec{s}_0,\vec{s}_1) \wedge \neg P(\vec{s}_1)$ 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. $F_0(\vec{s}) := I(\vec{s}),\ F_1(\vec{s}) := P(\vec{s}),\ i := 1$
2. Solve $F_i(\vec{s}_0) \wedge T(\vec{s}_0,\vec{s}_1) \wedge \neg P(\vec{s}_1)$
   (a) If satisfiable: extract $\hat{c}(\vec{s}_0)$, DFS($\hat{c}$, $i-1$)
   (b) If unsatisfiable:
   $F_{i+1}(\vec{s}) := P(\vec{s})$, PUSH($i$), $i := i+1$
3. goto 2.

Procedure DFS($\hat{c}$, i):

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

Procedure PUSH(i):

1. $j := 1$
2. $f := \mathit{true}$
3. For each clause $\tilde{c}(\vec{s})$ in $F_j(\vec{s})$ which is not in $F_{j+1}(\vec{s})$ solve $F_j(\vec{s}_0) \wedge T(\vec{s}_0,\vec{s}_1) \wedge \neg\tilde{c}(\vec{s}_1)$
   (a) If satisfiable: $f := \mathit{false}$
   (b) If unsatisfiable: $F_{j+1}(\vec{s}) := F_{j+1}(\vec{s}) \wedge \tilde{c}(\vec{s})$
4. If $f = \mathit{true}$: P **proven, exit**
5. If $j < i$: $j := j+1$, goto 2.
6. Return

IC3 starts in MAIN() with $i = 1$ and searches $F_i(\vec{s})$ for a state $\hat{c}(\vec{s})$ 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 $\hat{c}(\vec{s})$ 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 $F_j(\vec{s}_0) \wedge T(\vec{s}_0, \vec{s}_1) \wedge \neg F_j(\vec{s}_1)$ 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 expressive sets of states. This is achieved by generalizing proof obligations as well as blocked cubes. Generalizing blocked cubes is done by removing literals from $\hat{c}(\vec{s})$ such that $F_i(\vec{s}_0) \wedge \neg\hat{c}(\vec{s}_0) \wedge T(\vec{s}_0, \vec{s}_1) \wedge \hat{c}(\vec{s}_1)$ 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 $\hat{c}$ and its predecessor $\hat{e}$, it is checked whether $\hat{e}(\vec{s}_0) \wedge T(\vec{s}_0, \vec{s}_1) \wedge \neg\hat{c}(\vec{s}_1)$ stays unsatisfiable while removing literals from $\hat{e}(\vec{s}_0)$. 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 $\hat{e}(\vec{s}_0)$ has a successor in $\hat{c}(\vec{s}_1)$ – therefore, $\hat{e}(\vec{s}_0)$ 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 $((x_2 \geq 5) \wedge (x_3 \geq 1)) \implies (x_1 \geq 6)$ 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. $(\neg(x > 7) \vee (x \geq 5))$.

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

[2] This check exploits that it is possible to rewrite $\neg F_j(\vec{s}_1)$ to a disjunction of negated clauses. Hence, $F_j(\vec{s}_0) \wedge T(\vec{s}_0, \vec{s}_1) \wedge \neg\hat{c}(\vec{s}_1)$ can be solved for each negated clause to check it individually. If all these solver calls are unsatisfiable, $F_j(\vec{s}_0) \wedge T(\vec{s}_0, \vec{s}_1) \wedge \neg F_j(\vec{s}_1)$ is unsatisfiable as well.

[3] This allows counterexamples with more transition steps than the current number of frame formulas to be found.

[4] In IC3, $F_0(\vec{s}) = I(\vec{s})$ and $F_i(\vec{s}) \implies F_{i+1}(\vec{s})$ have to be maintained. Thus, to avoid excluding states contained in $I(\vec{s})$, an *ungeneralization* might be required.

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 ICP.

2.4.1 Literal Rotation and Literal Dropping

For the generalization of blocked cubes we consider the following unsatisfiable formula: $F_i(\vec{s}_0) \wedge \neg\hat{c}(\vec{s}_0) \wedge T(\vec{s}_0,\vec{s}_1) \wedge \hat{c}(\vec{s}_1)$. The inclusion of $\neg\hat{c}(\vec{s}_0)$ 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 $\hat{c}$ 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 $\neg\hat{c}(\vec{s}_0)$ 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 $\hat{c}(\vec{s}_1)$. However, this core is only minimal wrt. the order by which the assumption literals from $\hat{c}(\vec{s}_1)$ 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):

$$(\neg l_1 \vee l_3)^{(1)} \wedge (\neg l_1 \vee l_4)^{(2)} \wedge (\neg l_1 \vee l_5)^{(3)} \wedge (\neg l_5 \vee \neg l_6)^{(4)} \wedge (\neg l_2 \vee l_6)^{(5)} \wedge (\neg l_3 \vee \neg l_4 \vee l_6)^{(6)}$$

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_3, 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 $G = F_i(\vec{s}_0) \wedge \neg\hat{c}(\vec{s}_0) \wedge T(\vec{s}_0,\vec{s}_1) \wedge \hat{c}(\vec{s}_1)$ and $\hat{c}(\vec{s}_1)$ 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 $(l_1, \ldots, l_n)$. Thus, there exists an unsatisfied l_i being assigned by BCP, i.e. the pseudo-decision of l_i 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 rotation does not necessarily find a minimum number of required assumption literals. Firstly, it leaves $\neg\hat{c}(\vec{s}_0)$ untouched (see Section 3.1 for further explanations) and secondly there might remain constellations which require literal dropping in order to gain further knowledge to detect an assumption literal as redundant [36].
Literal dropping – in opposition to literal rotation – means that we remove a literal l from cube $\hat{c}(\vec{s})$ 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 $\hat{c}(\vec{s})$ and not its clause $\neg\hat{c}(\vec{s})$.

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 $\hat{c}$, 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 $(b \vee \neg l_j)$ 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 $F_0(\vec{s}) = I(\vec{s})$ and $F_i(\vec{s}) \implies F_{i+1}(\vec{s})$ 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 $\hat{c}(\vec{s})$ is the original blocked cube and $\hat{c}'(\vec{s})$ the generalized version of it, then by construction $\hat{c}(\vec{s})$ never contains an initial state. Therefore, $I(\vec{s}) \wedge \hat{c}(\vec{s})$ is always unsatisfiable. By applying the generalization techniques from above, iSAT3+IC3 extracts a minimal set of literals from $\hat{c}(\vec{s})$ which are responsible for making it disjoint from $I(\vec{s})$, appends them to $\hat{c}'(\vec{s})$ 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 $F_i(\vec{s_0}) \wedge \neg\hat{c}(\vec{s_0}) \wedge T(\vec{s_0},\vec{s_1}) \wedge \hat{c}(\vec{s_1})$ 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 $\hat{c}(\vec{s_1})$ 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, $\hat{c}(\vec{s_1}) = l_1 \wedge \ldots \wedge l_n$ and we assume that cube $\hat{c}(\vec{s_1})$ is passed by assumptions to the iSAT3 solver core:

Procedure *GENERALIZE*($\hat{c}$, i):

1. $\hat{c}_{\text{old}}(\vec{s}) = \hat{c}(\vec{s})$
2. While $(F_i(\vec{s_0}) \wedge \neg\hat{c}_{\text{old}}(\vec{s_0}) \wedge T(\vec{s_0},\vec{s_1}) \wedge \hat{c}(\vec{s_1})$ UNSAT and i smaller than the number of frames)
   (a) *ANALYZE*($\hat{c}$);
   (b) If (*performLiteralDropping*)
   For each literal l in $\hat{c}(\vec{s_1})$:
   $\hat{c}'(\vec{s_1}) := \hat{c}(\vec{s_1}) \setminus \{l\}$;
   If $(F_i(\vec{s_0}) \wedge \neg\hat{c}_{\text{old}}(\vec{s_0}) \wedge T(\vec{s_0},\vec{s_1}) \wedge \hat{c}'(\vec{s_1}))$ is UNSAT):
   $\hat{c}(\vec{s}) := \hat{c}'(\vec{s})$;
   (c) $i := i+1$;

Procedure *ANALYZE*($\hat{c}$ (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;
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 ($\hat{c} \wedge I$): Ungeneralize $\hat{c}$ according to Section 2.4.3;

We remark, that cube $\hat{c}$ is passed to *ANALYZE* by reference. The Boolean flag *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 $\hat{e}(\vec{s_0}) \wedge T(\vec{s_0},\vec{s_1}) \wedge \neg\hat{c}(\vec{s_1})$ to generalize proof obligation cube $\hat{e}(\vec{s_0})$. 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 $\hat{e}(\vec{s_0}) \wedge \hat{c}(\vec{s_1})$ is a satisfying assignment for $\hat{e}(\vec{s_0}) \wedge T(\vec{s_0},\vec{s_1}) \wedge \hat{c}(\vec{s_1})$, the formula $\hat{e}(\vec{s_0}) \wedge \neg T(\vec{s_0},\vec{s_1}) \wedge \hat{c}(\vec{s_1})$ is unsatisfied.
Via literal rotation iSAT3+IC3 extracts from $\hat{e}(\vec{s_0})$ a reduced proof obligation cube based on this unsatisfiable call.

2.4.6 A Symbiosis of k-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-tried 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)*.

[5] For all blocked cubes $\hat{c}(\vec{s})$ it has to hold that $I(\vec{s}) \implies \neg\hat{c}(\vec{s})$.

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

$$F_i(\vec{s_0}) \wedge \neg\hat{c}(\vec{s_0}) \wedge T(\vec{s_0},\vec{s_1}) \wedge \hat{c}(\vec{s_1}) \tag{1}$$

The term *inductive relative* to F_i means, that $\neg\hat{c}(\vec{s_0}) \wedge F_i(\vec{s_0}) \wedge T(\vec{s_0},\vec{s_1}) \implies \neg\hat{c}(\vec{s_1})$ 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

$$F_i(\vec{s_0}) \wedge T(\vec{s_0},\vec{s_1}) \wedge \hat{c}(\vec{s_1}) \tag{2}$$

because if there is no transition from F_i to $\hat{c}(\vec{s_1})$ there is also none from $F_i \wedge \neg\hat{c}(\vec{s_0})$. 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, $\neg\hat{c}(\vec{s}) \wedge F_i(\vec{s})$ 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

$$F_i(\vec{s_0}) \wedge \neg\hat{c}'(\vec{s_0}) \wedge T(\vec{s_0},\vec{s_1}) \wedge \hat{c}'(\vec{s_1}) \tag{3}$$

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 $\hat{c}'(\vec{s_1}) = \hat{c}(\vec{s_1}) \setminus \{l\}$ 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 $\hat{d}(\vec{s_0})$ from Formula 3 and applies $\hat{c}'(\vec{s_1}) = \hat{c}'(\vec{s_1}) \sqcup \hat{d}(\vec{s_1})$. The $\sqcup$-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 $\hat{d}(\vec{s_1})$. 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 $\hat{c}'(\vec{s_1})$ is unreachable from $F_i(\vec{s_0}) \wedge \neg\hat{c}'(\vec{s_0})$ 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 $\hat{d}(\vec{s})$) 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 $\hat{d}(\vec{s})$ 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 $F_i(\vec{s_0}) \wedge \neg\hat{c}(\vec{s_0}) \wedge T(\vec{s_0},\vec{s_1}) \wedge \hat{c}(\vec{s_1})$ which resolves to UNSAT iff. cube $\hat{c}(\vec{s_1})$ is unreachable from $F_i(\vec{s_0}) \wedge \neg\hat{c}(\vec{s_0})$ 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 $F_i(\vec{s_0}) \wedge \neg\hat{c}'(\vec{s_0})$, 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 $F_i(\vec{s}_0)$-predecessor, and thus the CTG, of $\hat{c}'(\vec{s}_1)$ from the solver and call it $\hat{d}(\vec{s})$. We try to block $\hat{d}(\vec{s})$ from the state space by calling $F_{i-1}(\vec{s}_0) \wedge \neg\hat{d}(\vec{s}_0) \wedge T(\vec{s}_0,\vec{s}_1) \wedge \hat{d}(\vec{s}_1)$. If this succeeds, Formula 3 is checked again and so on, until all CTGs are ruled out – then we stop and cube $\hat{c}'(\vec{s})$ may be blocked. If not, similar to *Down()*, we enlarge $\hat{c}'(\vec{s})$ by joining (convex overapproximation of the union) $\hat{c}'(\vec{s})$ with its CTG $\hat{d}(\vec{s})$ and check Formula 3 again. As in *Down()* we stop and roll back, if the joined cube $\hat{c}'(\vec{s})$ 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 $\hat{c}(\vec{s})$ 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 $\hat{d}(\vec{s})$ result from a satisfying assignment of a formula $F_i(\vec{s}_0) \wedge \neg\hat{c}'(\vec{s}_0) \wedge T(\vec{s}_0,\vec{s}_1) \wedge \hat{c}'(\vec{s}_1)$. 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 $\hat{d}(\vec{s})$ were a proof obligation, it is most crucial to preserve that for every $\hat{d}(\vec{s})$-state there is a sequence which leads the system into an unsafe $\neg P(\vec{s})$ 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 $\hat{d}(\vec{s})$ which really are predecessors of $\hat{c}'(\vec{s})$. Having a non-predecessor state in $\hat{d}(\vec{s})$ can only result in wrongfully disallowing the removal of a literal l from cube[6] $\hat{c}(\vec{s})$. 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_i 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 goes hand in hand with the bit-level 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.

[6] We recall that $\hat{c}'(\vec{s}) = \hat{c}(\vec{s}) \setminus \{l\}$.

3.4 Overall Approach

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

We call *GENERALIZE* to generalize a cube $\hat{c}$ for which $F_i(\vec{s}_0) \wedge \neg\hat{c}(\vec{s}_0) \wedge T(\vec{s}_0,\vec{s}_1) \wedge \hat{c}(\vec{s}_1)$ has been unsatisfiable (UNSAT) 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 (which remains the same as in Section 2.4.4 to extract a minimal conflicting set of $\hat{c}$ literals by using literal rotation and optionally bound generalization). After we found the highest frame and extracted the minimal conflicting set of $\hat{c}$ literals, we further generalize $\hat{c}$ 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 $\hat{c} \setminus \{l\}$ is reachable. All of this can be controlled via flags inside of the *CTG* procedure.

Furthermore, $\hat{c}(s_1) = l_1 \wedge \ldots \wedge l_n$ and we assume that cube $\hat{c}(\vec{s}_1)$ is passed by assumptions to iSAT3:

Procedure
GENERALIZE($\hat{c}$, i, *recDepth*)

1. While ($F_i(\vec{s}_0) \wedge \neg\hat{c}(\vec{s}_0) \wedge T(\vec{s}_0,\vec{s}_1) \wedge \hat{c}(\vec{s}_1)$ UNSAT and i smaller than the number of frames)

   ANALYZE($\hat{c}$);
   $i := i+1$;

2. For each literal l in $\hat{c}$:

   (a) $\hat{c}' := \hat{c} \setminus \{l\}$;
   If (CTG($i-1$, $\hat{c}'$, *recDepth*)):
   ANALYZE($\hat{c}'$);
   $\hat{c} := \hat{c}'$;

Procedure
ANALYZE($\hat{c}$ (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;
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 ($\hat{c} \wedge I$): Ungeneralize $\hat{c}$ according to Section 2.4.3;

Procedure
CTG(i, $\hat{c}$ (reference), *recDepth*) : bool

1. ctgs := 0;
2. if ($\hat{c}(\vec{s}) \wedge I(\vec{s})$): return FALSE;

3. Call the solver on $F_i(\vec{s_0}) \wedge \neg\hat{c}(\vec{s_0}) \wedge T(\vec{s_0},\vec{s_1}) \wedge \hat{c}'(\vec{s_1})$;

   Case **SAT**:

   (a) Extract a $F_i(\vec{s})$-predecessor cube $\hat{d}(\vec{s})$;

   (b) if (*recDepth* > *maxRecDepth*): return FALSE;

   (c) check for a $F_{i-1}(\vec{s})$-predecessor of $\hat{d}(\vec{s})$;

   if (there is a $F_{i-1}(\vec{s})$-predecessor of $\hat{d}(\vec{s})$ or ($\hat{d}(\vec{s}) \wedge I(\vec{s})$) or (*ctgs* > *maxCTGs*) or *onlyDown*):

   ctgs := 0;

   Join $\hat{c}(\vec{s}) = \hat{c}(\vec{s}) \sqcup \hat{d}(\vec{s})$ and go to 2.;

   else:

   ctgs := *ctgs* + 1;

   GENERALIZE($\hat{d}$, i - 1, *recDepth* + 1), learn new clause $\hat{d}$ 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 $\hat{c}$ to a higher frame.

It also incorporates inductive generalization, i.e. for checking whether cube $\hat{c}$ can be generalized to cube $\hat{c}''$ in frame $i+1$ we always check $F_i(\vec{s_0}) \wedge \neg\hat{c}''(\vec{s_0}) \wedge T(\vec{s_0},\vec{s_1}) \wedge \hat{c}''(\vec{s_1})$ instead of $F_i(\vec{s_0}) \wedge \neg\hat{c}(\vec{s_0}) \wedge T(\vec{s_0},\vec{s_1}) \wedge \hat{c}''(\vec{s_1})$ 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 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: *indi*[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, *+ld* 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].

	CEX	DC	T/M	uniq. DC PF1 / PF2	∅time ratio addGen	∅red. rate litDrop	∅impr. boundGen
iSAT3 BMC	7671	-	1107	-	-	-	-
iSAT3 *k*-Induction	7620	614	544	-	-	-	-
EP-CBMC *k*-Induction	7644	628	506	-	-	-	-
iSAT3 Craig Interpolation	7653	977	148	-	-	-	-
iSAT3+IC3$_1$ (ind2 +abort)	7533	997	248	23 / 0	-	0.92	0.05
iSAT3+IC3$_2$ (ind0 +abort +ig)	7264	1002	512	28 / 8	0.12	0.87	0.05
PF1 [Portfolio 1 (w/o iSAT3+IC3$_i$)]	7672	995	111	-	-	-	-
PF2 [PF1 ∪ iSAT+IC3$_1$]	7672	1018	88	-	-	-	-
PF3 [PF1 ∪ iSAT+IC3$_2$]	**7672**	**1023**	**83**	**-**	**-**	**-**	**-**
PF4 [PF2 ∪ iSAT+IC3$_2$]	**7672**	**1026**	**80**	**-**	**-**	**-**	**-**
iSAT3+IC3 ind0	6977	973	828	14 / 2	-	0.87	0.04
iSAT3+IC3 ind0 +ld	6803	924	1051	19 / 6	-	0.87	0.03
iSAT3+IC3 ind0 +ld$_{\text{once}}$	7034	996	748	26 / 7	0.12	0.87	0.03
iSAT3+IC3 ind0 +ig	7090	**1002**	686	27 / 7	0.12	0.87	0.05
iSAT3+IC3 ind0 +ig +j	6918	983	877	16 / 4	0.18	0.87	0.05
iSAT3+IC3 ind0 +ig +ctg	6997	984	797	21 / 2	0.24	0.87	0.05
iSAT3+IC3 ind0 +ig +ctg +lf	**7126**	987	665	20 / 3	0.23	0.88	0.05
iSAT3+IC3 ind0 +ig +ctg +j	6996	983	799	21 / 2	0.24	0.87	0.05
iSAT3+IC3 ind0 +ig +ctg +j +lf	7125	987	666	20 / 3	0.23	0.88	0.05
iSAT3+IC3 ind0 +abort	7407	995	376	25 / 3	-	0.87	0.04
iSAT3+IC3 ind0 +abort +ig	**7264**	**1002**	**512**	**28 / 8**	**0.12**	**0.87**	**0.05**
iSAT3+IC3 ind0 +abort +ig +ctg + lf	7144	986	648	20 / 3	0.24	0.88	0.05
iSAT3+IC3 ind2	7480	991	307	19 / 1	-	0.92	0.05
iSAT3+IC3 ind2 +ig	7508	993	277	22 / 3	0.12	0.92	0.05
iSAT3+IC3 ind2 +ig +ctg + lf	7514	995	269	22 / 4	0.18	0.93	0.05
iSAT3+IC3 ind2 +abort	**7533**	**997**	**248**	**23 / 0**	**-**	**0.92**	**0.05**
iSAT3+IC3 ind2 +abort +ig	7549	996	233	22 / 3	0.12	0.92	0.05
iSAT3+IC3 ind2 +abort +ig +ctg + lf	7533	995	250	22 / 4	0.18	0.93	0.05

Table 1 Experimental results over 8778 benchmark instances (time limit 300s, memory limit 4 GB per instance)

The last three columns relate to the performance of the additional generalization (addGen) techniques. Firstly, we present the average time ratio which was consumed by all activated additional generalization techniques (+*ig*, +*j*, +*ctg*) followed by the average reduction rate wrt. the removed literals (litDrop) achieved by the *complete* generalization effort and the average fraction of simple bound literals which could be relaxed via bound generalization (boundGen).

We compute the average by taking all (not „trivially“) solved instances of one variant. Trivially solved instances may occur e.g. due to the initial BMC calls (with more than a single unrolling of the transition relation) of IC3, which may be quite a lot if the suffix has been prolonged – in these cases no cubes are blocked and no generalization takes place. We remark, that for *ind2* and all +*abort* settings, the number of trivially solved instances increases due to significantly more and also more complex BMC calls. Thus, these variants are able to solve benchmarks where other variants get stuck which is often the case if the benchmark does not allow for strong generalization of cubes. As a result, we can observe better average reduction rates for *ind2* configurations.

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 +ld_{once}). 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 $\neg P$ 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$_1$ (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$_2$). Adding iSAT3+IC3$_2$ to PF1 instead of iSAT3+IC3$_1$ (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$_1$, 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$_2$ still provides the best improvement by additional 8 solved DC instances. Thus, adding iSAT3+IC3$_2$ 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 *ind0 +abort, +ig* – which is able to improve the best solver portfolio from [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 [23, 21] did for instance by using CTG and *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 [39] into iSAT3+IC3.

6 Literature

[1] Frédéric Benhamou and Laurent Granvilliers. Continuous and Interval Constraints. In *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 571–603. Elsevier, 2006.

[2] Tom Bienmüller and Tino Teige. Satisfaction Meets Practice and Confidence. In *SC-square 2016*.

[3] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic Model Checking Using SAT Procedures instead of BDDs. In *DAC 1999*.

[4] Johannes Birgmeier, Aaron R. Bradley, and Georg Weissenbacher. Counterexample to Induction-Guided Abstraction-Refinement (CTIGAR). In *CAV 2014*.

[5] Aaron R. Bradley. SAT-Based Model Checking without Unrolling. In *VMCAI 2011*.

[6] Aaron R. Bradley and Zohar Manna. Checking safety by inductive generalization of counterexamples to induction. In *Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD*, 2007.

[7] Martin Brain, Vijay D'Silva, Alberto Griggio, Leopold Haller, and Daniel Kroening. Deciding Floating-Point Logic with Abstract Conflict Driven Clause Learning. *Formal Methods in System Design*, 45(2):213–245, 2014.

[8] Robert K. Brayton and Alan Mishchenko. ABC: an academic industrial-strength verification tool. In *Computer Aided Verification, 22nd International Conference, CAV*, 2010.

[9] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In *Computer Aided Verification - 26th International Conference, CAV*. Springer, 2014.

[10] Hana Chockler, Alexander Ivrii, Arie Matsliah, Shiri Moran, and Ziv Nevo. Incremental Formal Verification of Hardware. In *FMCAD 2011*.

[11] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. Smtinterpol: An interpolating SMT solver. In *Model Checking Software - 19th International Workshop, SPIN*, 2012.

[12] Alessandro Cimatti and Alberto Griggio. Software Model Checking via IC3. In *CAV 2012*.

[13] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS*.

[14] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In *TACAS 2004*.

[15] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS*, Lecture Notes in Computer Science, 2008.

[16] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. Efficient Implementation of Property Directed Reachability. In *FMCAD 2011*.

[17] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *SAT 2003*.

[18] Niklas Eén and Niklas Sörensson. Temporal Induction by Incremental SAT Solving. *Electr. Notes Theor. CS*, 89(4):543–560, 2003.

[19] Andreas Eggers. *Direct Handling of Ordinary Differential Equations in Constraint-Solving-Based Analysis of Hybrid Systems*. PhD thesis, Carl von Ossietzky University of Oldenburg, 2014.

[20] Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient Solving of Large Non-Linear Arithmetic Constraint Systems with Complex Boolean Structure. *JSAT*, 1(3-4):209–236, 2007.

[21] Alberto Griggio and Marco Roveri. Comparing different variants of the ic3 algorithm for hardware model checking. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 35(6), 2016.

[22] Zyad Hassan, Aaron R. Bradley, and Fabio Somenzi. Incremental, inductive CTL model checking. In *Computer Aided Verification - 24th International Conference, CAV*, 2012.

[23] Zyad Hassan, Aaron R. Bradley, and Fabio Somenzi. Better generalization in IC3. In *Formal Methods in Computer-Aided Design, FMCAD 2013*, 2013.

[24] Christian Herde. *Efficient Solving of Large Arithmetic Constraint Systems with Complex Boolean Structure: Proof Engines for the Analysis of Hybrid Discrete-Continuous Systems*. PhD thesis, Carl von Ossietzky University of Oldenburg, 2011.

[25] Krystof Hoder and Nikolaj Bjørner. Generalized Property Directed Reachability. In *SAT 2012*.

[26] Dejan Jovanovic and Bruno Dutertre. Property-Directed k-Induction. In *FMCAD 2016*.

[27] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-Based Model Checking for Recursive Programs. In *CAV 2014*.

[28] Stefan Kupferschmid. *Über Craigsche Interpolation und deren Anwendung in der formalen Modellprüfung*. PhD thesis, University of Freiburg, 2013.

[29] Kenneth L. McMillan. Interpolation and SAT-Based Model Checking. In *CAV 2003*.

[30] Lukas Mentel, Karsten Scheibler, Felix Winterer, Bernd Becker, and Tino Teige. Benchmarking SMT Solvers on Automotive Code. In *24. Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, MBMV 2021*.

[31] Felix Neubauer, Karsten Scheibler, Bernd Becker, Ahmed Mahdi, Martin Fränzle, Tino Teige, Tom Bienmüller, and Detlef Fehrer. Accurate Dead Code Detection in Embedded C Code by Arithmetic Constraint Solving. In *SC-square 2016*.

[32] Kavita Ravi and Fabio Somenzi. Minimal assignments for bounded model checking. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS*, 2004.

[33] Karsten Scheibler. *Applying CDCL to Verification and Test: When Laziness Pays Off*. PhD thesis, University of Freiburg, Freiburg im Breisgau, Germany, 2017.

[34] Karsten Scheibler, Felix Neubauer, Ahmed Mahdi, Martin Fränzle, Tino Teige, Tom Bienmüller, Detlef Fehrer, and Bernd Becker. Accurate ICP-based Floating-Point Reasoning. In *FMCAD 2016*.

[35] Karsten Scheibler, Felix Neubauer, Ahmed Mahdi, Martin Fränzle, Tino Teige, Tom Bienmüller, Detlef Fehrer, and Bernd Becker. Extending iSAT3 with ICP-Contractors for Bitwise Integer Operations. Technical Report 116, SFB/TR 14 AVACS, 2016.

[36] Karsten Scheibler, Felix Winterer, Tobias Seufert, Tino Teige, Christoph Scholl, and Bernd Becker. ICP and IC3. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2021*.

[37] Peter Schrammel, Daniel Kroening, Martin Brain, Ruben Martins, Tino Teige, and Tom Bienmüller. Incremental Bounded Model Checking for Embedded Software. *Formal Aspects Comput.*, 29(5):911–931, 2017.

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

[39] Tobias Seufert and Christoph Scholl. Combining PDR and reverse PDR for hardware model checking. In Jan Madsen and Ayse K. Coskun, editors, *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*, pages 49–54. IEEE, 2018.

[40] João P. Marques Silva and Karem A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *ICCAD 1996*.

[41] Tino Teige. *Stochastic Satisfiability Modulo Theories: A Symbolic Technique for The Analysis of Probabilistic Hybrid Systems*. PhD thesis, Carl von Ossietzky University of Oldenburg, 2012.

[42] Grigori S. Tseitin. On the Complexity of Derivations in Propositional Calculus. In *Studies in Constructive Mathematics and Mathematical Logics*. 1968.

[43] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *ICCAD 2001*.