Preprint from Proceedings of Design, Automation and Test in Europe (DATE), February 2021

ICP and IC3

Karsten Scheibler*, Felix Winterer†, Tobias Seufert†, Tino Teige*, Christoph Scholl†, Bernd Becker†

*BTC Embedded Systems AG
Oldenburg, Germany
{scheibler,teige}@btc-es.de

†University of Freiburg
Freiburg, Germany
{winteref,seufert,scholl,becker}@informatik.uni-freiburg.de

***Abstract*—If embedded systems are used in safety-critical environments, they need to meet several standards. For example, in the automotive domain the ISO 26262 standard requires that the software running on such systems does not contain unreachable code. Software model checking is one effective approach to automatically detect such dead code. Being used in a commercial product, iSAT3 already performs very well in this context. In this paper we integrate IC3 into iSAT3 in order to improve its dead code detection capabilities even further.**

***Index Terms*—SMT, iSAT3, ICP, IC3, PDR, software verification**

I. Introduction

Developing safety-critical embedded systems demands high-quality production code. Several standards exist for the different industrial sectors to guide the development of safety-critical software and hardware applications like the ISO 26262 standard [1] for the automotive domain.

Among many other criteria, the ISO 26262 objects to unintended functionality like unreachable code fragments (so-called *dead code*) and recommends several code coverage metrics to prove the absence of such code like statement coverage, condition coverage, decision coverage, or modified condition/decision coverage (MC/DC) [2]. A portion of code is considered as being covered if a test case is able to execute this code. Thus, with a coverage value of 100% no dead code exists according to the applied coverage metric – otherwise there is a potential risk of dead code requiring a further analysis.

One effective approach to automatically detect dead code is software model checking. Among other use cases the commercial test and verification tool suite BTC Embedded*Platform*® (EP) provides such functionality [3] by employing several model checking tools like CBMC [4], [5] and iSAT3 [6]. In a recent case study [7] several academic state-of-the-art software model checkers were compared to EP. On 179 software requirements for two large and *floating-point* dominated industrial models from Ford, the academic model checkers were able to prove at most 20% of these requirements while various competitors obtained results for 0% to 5%. In contrast, EP succeeded on 80% of the requirements. This underlines that EP has a very strong support for the automatic analysis of floating-point dominated production code becoming increasingly important in industrial safety-critical software [8].

We would like to emphasize that each single automatically derived result of dead code or each proven requirement can save several hours or even days of manual effort and is thus of utmost importance in the development and verification of safety-critical embedded software. Hence, improving the underlying model checking technology within EP is not an academic exercise – it has a strong practical relevance. Currently, iSAT3 with Craig Interpolation [9] is the strongest proof engine regarding dead code in EP [10]. In this paper we present an extended version of iSAT3 improving its performance in this context even further. Our contribution is as follows:

- We present iSAT3+IC3 which is the first ACDCL-style [11] SMT solver based on interval abstraction and ICP [12] with an IC3 [13] integration.
- We propose a new lifting scheme which also considers relationships between interval bounds.
- We present a semantic ungeneralization approach for blocked cubes and a method to generalize proof obligations in the presence of a generic transition relation.
- We show the effectiveness of a symbiotic combination of k-induction and IC3.

Related work: Previous work like [14]–[17] already considered IC3 in the SMT context. However, these approaches do not support floating-point. Instead, they build either on linear integer or linear real arithmetic (LIA or LRA) with theory-specific approximations of quantifier-elimination like interpolants or projections. The approach presented in [18] has similarities to IC3 and utilizes an MCSAT-style [19] solver – requiring the underlying theory to support quantifier-elimination. Therefore, usually arithmetic over the reals is used to approximate floating-point arithmetic. Thus, the methods of [14]–[18] are unable to directly detect dead code in floating-point programs[a]. In contrast, as iSAT3 only requires ICP-contractors for the operations of the underlying theory, iSAT3+IC3 can directly detect whether a floating-point program contains dead code.

Structure of the paper: After giving the preliminaries in Section II, the approach of iSAT3+IC3 is presented in Section III and experimentally evaluated in Section IV. Finally, Section V concludes the paper.

II. Preliminaries

A. SAT and Notations

The satisfiability problem (SAT) poses the question whether a Boolean formula F is satisfiable or not – i.e. it is checked whether an assignment for the Boolean variables of F exists such that F evaluates to true. Current state-of-the-art approaches build on Conflict-Driven Clause Learning (CDCL) [20]. CDCL-style solvers require the formula of interest to be in conjunctive normal form (CNF) – this can be achieved by applying the Tseitin-transformation [21]. A CNF is a conjunction of clauses while a clause is a disjunction of literals. A literal represents a Boolean variable or its negation.

In this paper we use upper case letters to denote formulas. Literals will be denoted by lower case letters – except i, j, k, m and n which are used for indices and x which is used for non-Boolean variables. In a slight abuse of notation lower case letters are also used for Boolean variables – in particular for sets of them, e.g. $\vec{r}$. Furthermore, a clause is denoted by a tilde-decorated lower case letter, e.g. $\tilde{c}$. Similarly, a cube (which is a conjunction of literals) is denoted by $\hat{c}$. To simplify notation a negated clause $\neg\tilde{c}$ is seen as a cube containing the negated literals of $\tilde{c}$ – and in the same manner a negated cube is seen as a clause. Additionally, the notation $F(\vec{s})$ is used to indicate that the formula F depends on the Boolean variables contained in the set $\vec{s}$. Similarly, the notations $\tilde{c}(\vec{q})$ and $\hat{c}(\vec{q})$ are used to indicate that the literals of clause $\tilde{c}$ and cube $\hat{c}$ belong to the Boolean variables in $\vec{q}$.

[a]For example, when considering an `if` condition with an expression like $10^{20} + 1 = 10^{20}$, the expression evaluates to true under 64 bit floating-point arithmetic (with round-to-nearest) while it evaluates to false under real-valued arithmetic – leading to spuriously detected dead code.

B. BMC and k-Induction

Informally, when considering the verification of a system, the system of interest is abstracted into a set of states and a relation which encodes possible state transitions – i.e. how the state of the system changes over time. In many cases the reachability of states is verified, i.e. the question is asked: if the system starts in a *good* state, is it possible to reach a *bad* state in a finite number of transition steps?

Bounded Model Checking (BMC) [22] is a way to check whether a system can reach a bad state. To achieve this, a set of Boolean variables (denoted by $\vec{s}$) is used to represent states. Furthermore, the initial states and the transition relation are encoded into the formulas $I(\vec{s})$ and $T(\vec{s}_i, \vec{s}_{i+1})$ while the set of allowed good states is encoded into the property $P(\vec{s})$. To simplify notation, we neglect that the transition relation usually considers variables representing inputs as well. Thus, P is violated if a bad state is reached. Performing BMC means solving a sequence of formulas – one formula for each transition step $0, 1, \dots, k$:

$$\begin{array}{ll} \mathrm{BMC}_0: & I(\vec{s}_0) \wedge \neg P(\vec{s}_0) \\ \mathrm{BMC}_1: & I(\vec{s}_0) \wedge T(\vec{s}_0, \vec{s}_1) \wedge \neg P(\vec{s}_1) \\ \mathrm{BMC}_2: & I(\vec{s}_0) \wedge T(\vec{s}_0, \vec{s}_1) \wedge T(\vec{s}_1, \vec{s}_2) \wedge \neg P(\vec{s}_2) \\ & \dots \\ \mathrm{BMC}_k: & I(\vec{s}_0) \wedge \left(\bigwedge_{i=0}^{k-1} T(\vec{s}_i, \vec{s}_{i+1}) \right) \wedge \neg P(\vec{s}_k) \end{array}$$

The formulas are solved in the listed order until one of them becomes satisfiable. In this case the satisfying assignment contains the values for the state variables forming a sequence of states ending in a bad state and thus violating P – such a state sequence is also called a *counterexample*. Unfortunately, if k is reached it is only known that P cannot be violated within up to k transition steps. Thus, in general, BMC is unable to prove that P is never violated. But the knowledge gained by BMC can be exploited to achieve that. For example, if $I(\vec{s}_0) \wedge \neg P(\vec{s}_0)$ is unsatisfiable $I(\vec{s}_0) \Rightarrow P(\vec{s}_0)$ is valid – i.e. whenever $I(\vec{s}_0)$ is true $P(\vec{s}_0)$ is true as well. Thus, conjoining $P(\vec{s}_0)$ to $I(\vec{s}_0)$ does not constrain the set of possible solutions. Hence, BMC can also be performed the following way:

$$\mathrm{BMC}_k: \quad I(\vec{s}_0) \wedge \left(\bigwedge_{i=0}^{k-1} (P(\vec{s}_i) \wedge T(\vec{s}_i, \vec{s}_{i+1})) \right) \wedge \neg P(\vec{s}_k)$$

This offers the possibility to prove P by considering the formula suffix – e.g. if $P(\vec{s}_0) \wedge T(\vec{s}_0, \vec{s}_1) \wedge \neg P(\vec{s}_1)$ is unsatisfiable, all BMC_i formulas with $i \geq 1$ will be unsatisfiable as well because they contain this suffix[b]. Of course this check can also be applied to suffixes of length k:

$$\mathrm{IND}_k: \quad \left(\bigwedge_{i=0}^{k-1} (P(\vec{s}_i) \wedge T(\vec{s}_i, \vec{s}_{i+1})) \right) \wedge \neg P(\vec{s}_k)$$

Hence, when alternating BMC and IND checks it is possible to prove P and find violations of P – which is the basic idea of the k-induction approach presented in [23].

C. IC3 and PDR

Informally, when evaluating $P(\vec{s}_0) \wedge T(\vec{s}_0, \vec{s}_1) \wedge \neg P(\vec{s}_1)$ the following question is asked: when starting in the set of good states is it possible to reach a bad state in one transition step? It is reasonable to assume that k-induction is likely to fail in cases with many unreachable good states which are able to reach a bad state – as the only way to strengthen this check is to increase the suffix length[c]. Hence, instead of starting in P it seems to be more advantageous to start in a subset of P (denoted by F) which overapproximates all reachable good states. In such a setting P is proven if $F(\vec{s}_0) \wedge T(\vec{s}_0, \vec{s}_1) \wedge \neg F(\vec{s}_1)$ is unsatisfiable.

In order to obtain such an F, IC3 [13] builds a sequence of frame formulas F_i with $F_0(\vec{s}) = I(\vec{s})$ and $F_i(\vec{s}) \Rightarrow F_{i+1}(\vec{s})$. Each F_i is an overapproximation of the good states reachable in up to i transition steps. While BMC and k-induction unroll the transition relation, IC3 considers only one transition relation at a time, explicitly enumerates states, and incrementally refines the F_i. Thus, compared to BMC and k-induction the IC3 approach requires much more solver calls – but each solver call in IC3 processes only a small problem.

Before IC3 starts, the formulas BMC_0 and BMC_1 have to be solved to ensure that P is not violated in up to one transition step. Basically, the workflow of IC3 is similar to the following:

```
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 $j \in \{1, \dots, i+1\} : F_j(\vec{s}) := F_j(\vec{s}) \wedge \neg\hat{c}(\vec{s})$
  2) Return
Procedure PUSH($i$):
  1) $j := 1$
  2) $f :=$ *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: $f :=$ *false*
     b) If unsatisfiable: $F_{j+1}(\vec{s}) := F_{j+1}(\vec{s}) \wedge \tilde{c}(\vec{s})$
  4) If $f =$ *true*: $P$ **proven, exit**
  5) If $j < i$: $j := j+1$, goto 2)
  6) Return
```

Starting with $i = 1$, MAIN() searches for a good state $\hat{c}(\vec{s})$ in $F_i(\vec{s})$ which is able to reach a bad state in one transition step. If such a state exists, a depth-first search is performed, i.e. it is checked whether $\hat{c}(\vec{s})$ has itself a predecessor that is reachable from F_0. If this is the case, a counterexample is found – otherwise one or more F_i are constrained by adding a *blocked cube* which represents a good state being unreachable in up to i transition steps. In IC3 a state $\hat{c}(\vec{s})$ which has to be checked for a predecessor is called a *proof obligation*, because it has to be proven whether $\hat{c}(\vec{s})$ is part of a counterexample or not. In case MAIN() does not find a predecessor for a bad state, a new frame formula is added and all existing frame formulas are processed by PUSH() in order (1) to push blocked cubes to higher frames, and (2) to check whether $F_j(\vec{s}_0) \wedge T(\vec{s}_0, \vec{s}_1) \wedge \neg F_j(\vec{s}_1)$ is unsatisfiable[d] to prove that P can never be violated.

This basic workflow neglects details like pushing blocked cubes already in DFS(), remembering previously generated proof obligations[e] and generalizing states. The last point is very important. Instead of enumerating individual states, IC3 tries to operate on sets of states by generalizing each blocked cube. This is achieved 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[f].

[b]When neglecting variable renaming due to different time steps

[c]The unique states requirement mentioned in [23] gives only minor improvements in practice.

[d]This check exploits that $\neg F_j(\vec{s}_1)$ can be rewritten to a disjunction of negated clauses. Hence, each negated clause can be checked individually by solving $F_j(\vec{s}_0) \wedge T(\vec{s}_0, \vec{s}_1) \wedge \neg\tilde{c}(\vec{s}_1)$. 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.

[e]Which allows to find counterexamples with more transition steps than the current number of frame formulas.

[f]In order to maintain $F_0(\vec{s}) = I(\vec{s})$ and $F_i(\vec{s}) \Rightarrow F_{i+1}(\vec{s})$ an *ungeneralization* might be required to avoid that states contained in $I(\vec{s})$ are excluded.

Property Directed Reachability (PDR) [24] builds on IC3. While IC3 only generalizes blocked cubes, PDR applies generalization to proof obligations as well. While in [24] ternary simulation is proposed, current approaches like [25] use a solver call as proposed in [26]. After extracting 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, i.e. there exists a successor state for all states, then this unsatisfiability implies that each state in $\hat{e}(\vec{s}_0)$ has a successor in $\hat{c}(\vec{s}_1)$ – thus, $\hat{e}(\vec{s}_0)$ remains a valid proof obligation after removing literals. For problems originating from the Hardware Model Checking Competition (HWMCC)[g] this is always the case if no invariant constraints are present. On the other hand, a different approach is required when considering transition relations in general (cf. Section III-D).

D. From SAT to iSAT3

CDCL-style solvers contain three building blocks [20]:

1) Boolean Constraint Propagation (BCP),
2) a decision heuristics, and
3) a conflict analysis.

BCP is used to deduce consequences regarding the current partial assignment by searching for a unit clause – a clause is unit if all literals but one are assigned and evaluate to false while the remaining literal l is unassigned. Thus, l has to be true to satisfy the clause and to retain a chance to satisfy the whole formula. Obviously, BCP is incomplete – i.e. it is unable to detect all consequences. Therefore, unassigned Boolean variables are decided (i.e. assigned to either true or false) and BCP is applied again. If all Boolean variables are assigned and all clauses are satisfied a solution is found. In case a clause is unsatisfied (i.e. all literals evaluate to false) the conflict analysis is triggered and derives a *conflict clause* such that BCP is now able to prevent this conflict. Such learned knowledge finally enables BCP to detect whether a formula is unsatisfiable by causing a conflict on decision level 0.

The iSAT algorithm [27] builds on this scheme and incorporates Interval Constraint Propagation (ICP) [12] as additional deduction mechanism in order to handle Boolean combinations of *theory atoms*. Thus, it is an approach to solve SAT Modulo Theories (SMT) problems. iSAT3 [6] is the third implementation of the iSAT algorithm and benefits from the knowledge gained during the development of its predecessors HySAT [28] and iSAT [29]–[31].

In order to allow arithmetic reasoning, further variable types are required – besides Boolean variables, iSAT3 currently supports bounded integer- and real-valued variables as well as integers with a fixed bit width [32] and floating-point variables [10]. During the search process the set of possible solutions for these variable types is overapproximated with intervals – i.e. there exist dynamically generated literals representing lower and upper bounds for a variable – so-called *simple bound literals*. For example, when assigning 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 $[5, 7]$. Thus, the solver core of iSAT3 still operates on literals as a CDCL-style SAT solver, but additionally keeps a mapping for simple bound literals. Hence, clauses containing such literals exclude *hyper-boxes* from the search space. Furthermore, the theory atoms are decomposed by utilizing a Tseitin-like transformation to obtain *primitive constraints* which assign the result of an arithmetic operation to an auxiliary variable. An ICP-contractor exists for each supported operation and dynamically generates new clauses and simple bound literals during deduction – thus, adding support for new operations is just a matter of adding new ICP-contractors to iSAT3. 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 following deduction can be performed $((x_2 \geq 5) \wedge (x_3 \geq 1)) \Rightarrow (x_1 \geq 6)$.

[g]http://fmv.jku.at/hwmcc/

iSAT3 builds on the three building blocks of CDCL-style solvers and extends them as follows:

1) ICP is used as additional deduction mechanism and can be understood as an oracle providing the currently needed clauses and simple bound literals to enable BCP to perform reasoning about them. Furthermore, *bound-implication clauses* are generated lazily in order to encode implications between simple bound literals belonging to the same theory variable, e.g. $(\neg(x > 7) \vee (x \geq 5))$.
2) The decision heuristics is adapted as well. Besides deciding existing literals, interval splits are performed by dynamically generating new simple bound literals and deciding them.
3) Similar to CDCL-style solvers a 1UIP [33] conflict analysis is performed by analyzing the implication graph.

While lazy SMT [34] operates on a fixed set of literals (i.e. it generates new clauses over the literals representing the theory atoms of the originating problem), iSAT3 also generates new literals which is similar to MCSAT [19]. But in contrast to MCSAT (which is able to generate completely new theory atoms), the new literals in iSAT3 originate from the fixed set of variables being the result of the Tseitin-like decomposition of the theory atoms. Hence, informally, iSAT3 and thus the iSAT algorithm lie somewhere between lazy SMT and MCSAT. On the other hand, the iSAT algorithm can be seen as the first incarnation of Abstract CDCL (ACDCL) [11] based on interval abstraction as the iSAT algorithm was proposed several years before ACDCL.

III. iSAT3+IC3

The fact that iSAT3 maps the interval bounds of each theory variable to simple bound literals, makes it amenable to perform literal-based IC3 in the same manner as described in Section II-C – because the values of the state variables are encoded by a set of literals which IC3 can directly operate on. Although being correct, the resulting approach would suffer from a suboptimal performance – as possible dependencies between simple bound literals are completely ignored. Besides the blocked cube generalization included in IC3 we also consider the generalization of proof obligations as proposed in PDR.

In IC3 most of the solver calls contain a cube in the considered formula. This can be exploited by using incremental SAT solving and passing the cube as list of assumption literals [23]. In particular, when performing a blocked cube or proof obligation generalization it is very convenient to manipulate a list of assumption literals. In iSAT3+IC3 we follow this approach and describe in Sections III-A and III-B how the number of literals in a given list of assumption literals is reduced. In Sections III-C and III-D we address the problem when I and T are arbitrary formulas – which is the case for iSAT3. Finally, a symbiosis of k-induction and IC3 is shown in Section III-E.

A. Literal Rotation

When generalizing a blocked cube the following unsatisfiable formula is considered: $F_i(\vec{s}_0) \wedge \neg\hat{c}(\vec{s}_0) \wedge T(\vec{s}_0, \vec{s}_1) \wedge \hat{c}(\vec{s}_1)$. Including $\neg\hat{c}(\vec{s}_0)$ in the formula can be viewed as a bounded inductive reasoning [24] even allowing non-monotone reductions – i.e. while the solver call with a $\hat{c}$ having n literals is satisfiable, it becomes again unsatisfiable with $n - 1$ literals. In iSAT3+IC3 we leave $\neg\hat{c}(\vec{s}_0)$ untouched and perform only monotone reductions – as non-monotone reductions give only a minor advantage according to the experiments of [24]. As mentioned, $\hat{c}(\vec{s}_1)$ can be passed as list of assumption literals. Now the task is to remove assumption literals from $\hat{c}(\vec{s}_1)$ such that the formula stays unsatisfiable, i.e. removing *redundant* literals while keeping the *essential* ones. This process can also be called *lifting* as it has similarities to [35]. The authors of [24] propose (1) to use the final conflict-clause of the SAT solver, and afterwards (2) to remove assumption literals one by one while solving again – so-called *literal dropping*. While the final conflict-clause can be quickly determined by analyzing the implication graph, literal dropping is more expensive.

In [24] the SAT solver MiniSat [36] is used. MiniSat assigns all assumption literals before applying BCP. In contrast, we assign each assumption literal as individual pseudo-decision (if the literal is still unassigned) and execute BCP after such a decision. Thus, there are constellations where MiniSat detects fewer redundant literals compared to our pseudo-decision based approach. For example, when considering the list of assumption literals (l_1, l_2) and the following formula (with clause numbers in superscripts):

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

If l_1 and l_2 are assigned first, the clauses (1), (2), (3) and (5) become unit. Thus, the literals l_3, l_4, l_5 and l_6 are deduced which leads to a conflict in clause (4). An analysis of the implication graph would not reveal that l_2 is in fact redundant. In contrast, if l_1 is assigned and BCP is applied afterwards, clause (6) is involved to provoke a conflict in (4). Thus, l_2 is detected as redundant when using pseudo-decisions. But, of course, this depends on the order of the assumption literals. With (l_2, l_1) it seems that l_2 is essential for the conflict. Therefore, we rotate the order of the assumption literals and perform multiple checks. Furthermore, before solving the formula for the first time the assumption literals are shuffled pseudo-randomly. The details of *literal rotation* regarding an unsatisfiable formula G are as follows:

1) $i := 0$
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) $i := i + 1$, if $i < m + 1$: solve G, goto 2)
7) Return $(l_j, l'_1, \ldots, l'_m)$

While the first solver call (which determines whether G is unsatisfiable) might be quite expensive, the subsequent solver calls performed during literal rotation are in general cheaper. As it is known that the assumption literals $(l'_1, \ldots, l'_m)$ imply $\neg l_j$, the solver call with $(l_j, l'_1, \ldots, l'_m)$ will either cause a conflict after pseudo-deciding an l'_i (and thus triggering the conflict analysis) or directly imply $\neg l'_i$. Thus, no regular decisions and only up to $m + 1$ pseudo-decisions are performed in each iteration.

Intuitively, literal rotation tries to get most out of the learned knowledge by rearranging the order of the pseudo-decisions. Nonetheless, there might remain constellations which require literal dropping in order to gain further knowledge to detect an assumption literal as redundant. For example, when considering the list of assumption literals (l_1, l_2) and the following formula:

$$(\neg l_2 \vee l_3)^{(7)} \wedge (\neg l_2 \vee l_4)^{(8)} \wedge (l_2 \vee l_5)^{(9)} \wedge (l_2 \vee l_6)^{(10)}$$
$$\wedge(\neg l_1 \vee \neg l_3 \vee \neg l_4)^{(11)} \wedge (\neg l_1 \vee \neg l_5 \vee \neg l_6)^{(12)}$$

After pseudo-deciding l_2 the clauses (7) and (8) become unit and imply l_3 as well as l_4 which leads to a conflict in clause (11). This conflict occurs with (l_2, l_1) as well. Thus, l_2 seems to be an essential assumption literal.

During literal rotation the conflict clause $(\neg l_1 \vee \neg l_2)$ will be learned. When dropping l_2 from the list of assumption literals, l_2 will be decided in both polarities during the search process. Thus, now the clauses (9), (10) and (12) are also considered when deriving new knowledge. Hence, the conflict clauses $(\neg l_1 \vee l_2)$ and $(\neg l_1)$ will be learned. With this knowledge, already the pseudo-decision of l_1 fails and l_2 becomes redundant. Thus, it seems beneficial to perform literal dropping in addition to literal rotation – in Section IV we will evaluate whether this pays off.

B. Literal Rotation with Bound Generalization

When processing assumption literals with literal rotation no distinction is made between simple bound literals and ordinary ones. As simple bound literals represent interval bounds, this opens a further dimension of generalization, i.e. if it is not possible to remove a simple bound literal l it might still be possible to *generalize the bound* being represented by l. This is achieved by replacing l with another simple bound literal, e.g. replacing $(x \leq 5)$ with $(x \leq 9)$. Thus, although the number of assumption literals does not change, bound generalization nevertheless enlarges the hyper-box being represented by the cube of assumption literals.

When considering an unsatisfied assumption literal l_j it is checked whether l_j is a simple bound literal and was assigned due to a bound-implication clause (cf. Section II-D). Assuming this is the case, let $(b \vee \neg l_j)$ be such a clause. Due to the nature of bound implications, b represents a weaker upper (lower) bound than l_j. As $\neg b$ is implied by the assumption literals $(l'_1, \ldots, l'_m)$, the assumption literal l_j can be replaced with b which is the weakest possible bound still causing a conflicting pseudo-decision. Interestingly, bound generalization seamlessly integrates into literal rotation – as the rotation offers the chance to generalize every essential simple bound literal.

For practical reasons we divide the list of assumption literals into two parts representing the cubes $\hat{c}_{a1}$ and $\hat{c}_{a2}$ to be conjoined to the given formula. Literal rotation is only applied to $\hat{c}_{a2}$. Thus, $\hat{c}_{a1}$ can be used for assumption literals activating or deactivating formula parts. Obviously, the assumption literals in $\hat{c}_{a1}$ should be the first pseudo-decisions being made in the solver as these literals stay unchanged during literal rotation. It should be noted, that literal rotation might remove all literals in $\hat{c}_{a2}$ and thus returning an empty list of assumption literals. This happens whenever the formula is already unsatisfiable under $\hat{c}_{a1}$. In particular this feature is used when ungeneralizing blocked cubes regarding the initial states.

C. Ungeneralizing Blocked Cubes

Because of $F_0(\vec{s}) = I(\vec{s})$ and $F_i(\vec{s}) \Rightarrow F_{i+1}(\vec{s})$ special care must be taken during the generalization of blocked cubes – i.e. it is not allowed to constrain an F_i with a blocked cube which excludes an initial state.

In contrast to HWMCC problems, iSAT3 does not restrict the formulas for I, T and P. Thus, for T it is unknown whether it behaves like a function and I might be an arbitrary formula which does not necessarily represent a single state. Hence, a syntactic literal comparison as performed in [13] and [24] is not sufficient to ungeneralize a blocked cube if required – instead a semantic approach is needed.

Let $\hat{c}(\vec{s})$ be the original blocked cube and let $\hat{c}'(\vec{s})$ be the generalized version of it. By construction $\hat{c}(\vec{s})$ never contains an initial state. Thus, $I(\vec{s}) \wedge \hat{c}(\vec{s})$ is always unsatisfiable. Now, all literals of $\hat{c}'(\vec{s})$ are put into $\hat{c}_{a1}$ while all literals of $\hat{c}(\vec{s})$ not contained in $\hat{c}'(\vec{s})$ are added to $\hat{c}_{a2}$. Hence, $I(\vec{s}) \wedge \hat{c}_{a1} \wedge \hat{c}_{a2}$ will be unsatisfiable as well – allowing literal rotation to be applied in order to remove redundant literals from $\hat{c}_{a2}$. In the best case $\hat{c}_{a2}$ contains only redundant literals, i.e. $\hat{c}'(\vec{s})$ does not contain an initial state and can be used as is. In case $\hat{c}_{a2}$ contains essential literals, a new blocked cube $\hat{c}''(\vec{s})$ is created which contains all literals of $\hat{c}'(\vec{s})$ and the essential literals of $\hat{c}_{a2}$. Furthermore, in case $\hat{c}'(\vec{s})$ contains generalized bounds $\hat{c}''(\vec{s})$ might contain the original ungeneralized bound as well as the generalized version of it. In such a case the weaker generalized bound is redundant and can be removed.

D. Generalizing Proof Obligations with GeNTR

As it is unknown whether T behaves like a function, it would be wrong to use $\hat{e}(\vec{s}_0) \wedge T(\vec{s}_0, \vec{s}_1) \wedge \neg\hat{c}(\vec{s}_1)$ for generalization (cf. Section II-C) – because this formula would also be unsatisfiable for states without successor. Thus, the generalized $\hat{e}(\vec{s}_0)$ could contain dead-end states which cannot be extended to a counterexample.

Therefore, we use *Generalization with a Negated Transition Relation* (GeNTR). Obviously, if a satisfying assignment for a formula G is conjoined to $\neg G$ the resulting formula is unsatisfiable. Thus, as $\hat{e}(\vec{s}_0) \wedge \hat{c}(\vec{s}_1)$ 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. While $\hat{c}(\vec{s}_1)$ is put into $\hat{c}_{a1}$, cube $\hat{e}(\vec{s}_0)$ is stored in $\hat{c}_{a2}$ to apply literal rotation. In case $T(\vec{s}_0, \vec{s}_1)$ has input variables, their values are also part of the satisfying assignment and are added to $\hat{c}_{a1}$ as well.

E. A Symbiosis of k-Induction and IC3

IC3 enumerates states and generalizes them. In the context of iSAT3 with its simple bound literals the generalized states can be seen as hyper-boxes over the state variables. We observed cases in which our IC3 implementation generates a large number of new hyper-boxes without making much progress. In such cases an approach seems favorable which requires fewer explicit enumerations.

As explained in Section II-C procedure MAIN() searches for a good state which is able to reach a bad state in one transition step. At this point the idea of k-induction can be incorporated – i.e. instead of considering only one transition step, multiple transition steps can be used. This can be achieved by understanding $\neg P(\vec{s}_1)$ as IND'_0[h] and replacing it with, e.g, IND'_1. A longer suffix helps to filter out partial state sequences which are known to be not part of a counterexample if more than one transition step is considered. Of course a longer suffix also requires more BMC checks being made before starting IC3 – i.e. when using IND_i all BMC formulas up to BMC_{i+1} have to be solved in advance.

To some extent the authors of [37] made a similar observation – i.e. it is beneficial to augment the local reasoning of IC3 (which considers only one transition relation at a time) with a global reasoning (which considers multiple transition relations at once). Furthermore, the idea of combining k-induction with IC3 is not new [18], [38], [39]. With *target-enlargement* a similar idea was already proposed in the original PDR paper [24]. In contrast to [24] we consider target-enlargements above IND'_1 as well. Moreover, while the experiments performed in [24] show only small performance gains for Boolean problems, our observations regarding iSAT3+IC3 are different (cf. Section IV) and are thus inline with the observations being made in [18].

Additionally, iSAT3+IC3 uses a dynamic suffix length. This means, we start with IND'_2 when searching for bad states. In order to determine whether progress is being made, we count the number of processed proof obligations. This counter is reset whenever a new frame formula is added. Furthermore, we use a limit which is initially 128. If the counter reaches this limit, we (1) double the limit, and (2) abort the solving process in order to restart with a longer suffix – if the abort happened with F_i being the last added frame formula, we restart from scratch with IND'_i and solve only BMC_{i+1} before switching to IC3.

[h]IND'_0 is similar to IND_0 from Section II-B but has an index offset of 1 in order to be compatible with $\neg P(\vec{s}_1)$.

	CEX	DC	T/M	uniq. DC	∅sec. DC
iSAT3 BMC	7674	-	1104	-	-
iSAT3 k-Induction	7640	630	508	2	41.53
EP-CBMC k-Induction	7664	628	486	2	6.28
iSAT3 Craig Interpolation	7662	983	133	5	5.44
iSAT3+IC3 (ind2 +abort)	7622	1003	153	23	10.17
Portfolio w/o iSAT3+IC3	7687	1003	88	-	8.76
Portfolio with iSAT3+IC3	7687	1026	65	-	9.78
iSAT3+IC3 ind0 -bg -gentr	5870	926	1982	9	46.00
iSAT3+IC3 ind0 -gentr	6756	952	1070	14	55.89
iSAT3+IC3 ind0 +ld	6935	925	918	14	8.42
iSAT3+IC3 ind0	7171	977	630	14	10.71
iSAT3+IC3 ind0 +abort	7535	1003	240	24	13.31
iSAT3+IC3 ind1	7435	988	355	16	19.50
iSAT3+IC3 ind1 +abort	7588	994	196	22	7.94
iSAT3+IC3 ind2	7555	996	227	20	8.58
iSAT3+IC3 ind2 +abort	7622	1003	153	23	10.17
iSAT3+IC3 ind3	7573	998	207	18	22.61
iSAT3+IC3 ind3 +abort	7625	1002	151	23	19.55
iSAT3+IC3 ind4	7572	998	208	21	28.11
iSAT3+IC3 ind4 +abort	7620	1000	158	23	24.77

TABLE I
EXPERIMENTAL RESULTS OVER 8778 BENCHMARK INSTANCES

F. Overall approach

iSAT3+IC3 follows the basic workflow described in Section II-C. Additionally, literal rotation with bound generalization is used whenever a blocked cube or proof obligation has to be generalized. Furthermore, blocked cubes are ungeneralized as described in Section III-C and proof obligations are generalized according to Section III-D. Finally, the basic workflow is wrapped by an outer loop which observes whether progress is being made in order to restart with a longer IND'_i if required.

IV. Experimental Results

We rely on the same benchmark set of 8778 instances as [10]. These benchmarks originate from TargetLink-generated production C code containing floating-point arithmetic and were provided by BTC Embedded*Platform*® (EP) users from the industrial automotive domain. Each benchmark represents a goal defined by a structural code coverage metrics like MC/DC – unreachable goals correspond to dead code. In our experiments we concentrate on the results of iSAT3 and CBMC – as both are used within EP. We refer the reader to a recent case study [7] for a comparison on a similar set of benchmarks between EP and several academic state-of-the-art software model checkers which also participate in the SV-COMP.

The iSAT3 experiments were performed on a cluster – with each cluster node having 64 GB RAM and two 8-core CPUs running at 2.6 GHz. We applied a time limit of 1 hour and a memory limit of 8 GB per benchmark. The CBMC experiments were performed with the same limits and on a computer with a similar CPU type also running at 2.6 GHz. In our experiments we used the CBMC [4] version 5.12.4.

As CBMC has no built-in support for k-induction, the EP tool chain [3] prepares different input files in order to perform BMC_i and IND_i checks. Thus, multiple CBMC calls are required for k-induction – for a better distinguishability we call this method EP-CBMC.

Table I contains the results for EP-CBMC with k-induction and for iSAT3 when using BMC, k-induction, Craig Interpolation (CI) as well as different settings of the newly integrated IC3 (*indi*: corresponds to IND'_i, *-bg*: without bound generalization, *-gentr*: without GeNTR, *+ld*: with literal dropping, *+abort*: with abort as explained in Section III-E). While column 2 lists the number of counterexamples (CEX) being found, column 3 shows the number of detected unreachable goals, i.e. dead code (DC). Column 4 contains the unsolved instances due to the applied time or memory limit (T/M). The last two columns relate to the solved DC instances: the number of uniquely solved instances (regarding all five solvers) and the average solving time in seconds.

Table I is divided in two parts. In the first part we compare the four existing solvers and the new solver iSAT3+IC3 which is used in its best configuration *ind2 +abort*. When considering the four existing solvers, iSAT3 BMC finds most of the counterexamples – i.e. it is the best solver to find reachable code fragments while iSAT3 CI detects the highest number of dead code instances. This picture changes when additionally considering iSAT3+IC3. While iSAT3 BMC remains the best solver for reachable goals, iSAT3+IC3 outperforms iSAT3 CI by solving 20 additional DC instances. As in practice a portfolio of multiple solvers is used to exploit the multi-core CPUs of today's workstations, we also evaluate two combinations of the best solver results: one portfolio with iSAT3+IC3 and one without. While the latter has 88 unsolved instances, the portfolio with iSAT3+IC3 reduces this number to 65 – due to the 23 DC instances uniquely solved by iSAT3+IC3. When considering this in practice, where each of these unsolved instances requires hours or even days of manual effort in order to meet the criteria of the ISO 26262 standard, then 23 additionally detected DC instances reduce the manual effort by 26%.

The second part of Table I underlines the effectiveness of the methods proposed in Section III. Comparing *ind0 -bg -gentr* with *ind0 -gentr* and *ind0* shows that generalizing proof obligations and performing literal rotation with bound generalization bump up the number of solved instances considerably while reducing the average DC solving time. On the other hand, when comparing *ind0* and *ind0 +ld* it can be observed that literal dropping deteriorates the performance. This is surprising as literal rotation is theoretically weaker than literal dropping. Presumably, the runtime overhead of literal dropping does not pay off in the number of additionally removed redundant literals – which emphasizes the effectiveness of literal rotation with bound generalization. Furthermore, using a longer suffix and increasing its length dynamically finally enables iSAT3+IC3 to solve the highest number of DC instances. Based on the overall performance, we selected *ind2 +abort* as the default setting.

V. Conclusion

In this paper we presented iSAT3+IC3 – which is the first ACDCL-style SMT solver with an IC3 integration. In contrast to other SMT solvers, iSAT3 allows a direct integration of the literal-based IC3 into its ICP extended CDCL framework – but requires additional techniques to make this integration performant. Therefore, we proposed a new lifting scheme which exploits dependencies between interval bounds and is in practice surprisingly strong – i.e. an additional pass of literal dropping (which is the dominant approach in Boolean IC3) just causes overhead deteriorating the overall performance. Furthermore, we presented approaches to handle arbitrary formulas for the initial states and the transition relation, i.e. a semantic ungeneralization for blocked cubes and GeNTR. While the first is a prerequisite in order to allow blocked cube generalization, the latter allows to generalize proof obligations with transition relations which do not behave like functions. Finally, we showed the effectiveness of a symbiosis of k-induction and IC3 – i.e. using the formula suffix known from k-induction and dynamically extending it. Compared to other approaches, the resulting solver iSAT3+IC3 is able to detect considerably more dead code instances in floating-point dominated C code. Furthermore, in our experiments we showed that adding iSAT3+IC3 to a portfolio of solvers reduces the number of unsolved instances by 26% from 88 down to 65 – which is a remarkable gain when considering that the instances being left unsolved by an existing solver portfolio are usually the hardest ones and require huge efforts for manual dead code inspection.

References

[1] ISO, "Road vehicles – Functional safety," 2011.
[2] H. Kelly J., V. Dan S., C. John J., and R. Leanna K., "A Practical Tutorial on Modified Condition/Decision Coverage," Tech. Rep., 2001.
[3] F. Neubauer, K. Scheibler, B. Becker, A. Mahdi, M. Fränzle, T. Teige, T. Bienmüller, and D. Fehrer, "Accurate Dead Code Detection in Embedded C Code by Arithmetic Constraint Solving," in *SC-square 2016*. [Online]. Available: http://ceur-ws.org/Vol-1804/paper-07.pdf
[4] E. M. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," in *TACAS 2004*.
[5] P. Schrammel, D. Kroening, M. Brain, R. Martins, T. Teige, and T. Bienmüller, "Incremental Bounded Model Checking for Embedded Software," *Formal Aspects Comput.*, vol. 29, no. 5, pp. 911–931, 2017.
[6] K. Scheibler, "Applying CDCL to Verification and Test: When Laziness Pays Off," Ph.D. dissertation, University of Freiburg, Freiburg im Breisgau, Germany, 2017.
[7] L. Westhofen, P. Berger, and J. Katoen, "Benchmarking Software Model Checkers on Automotive Code," in *NFM 2020*.
[8] T. Bienmüller and T. Teige, "Satisfaction Meets Practice and Confidence," in *SC-square 2016*.
[9] K. L. McMillan, "Interpolation and SAT-Based Model Checking," in *CAV 2003*.
[10] K. Scheibler, F. Neubauer, A. Mahdi, M. Fränzle, T. Teige, T. Bienmüller, D. Fehrer, and B. Becker, "Accurate ICP-based Floating-Point Reasoning," in *FMCAD 2016*.
[11] M. Brain, V. D'Silva, A. Griggio, L. Haller, and D. Kroening, "Deciding Floating-Point Logic with Abstract Conflict Driven Clause Learning," *Formal Methods in System Design*, vol. 45, no. 2, pp. 213–245, 2014.
[12] F. Benhamou and L. Granvilliers, "Continuous and Interval Constraints," in *Handbook of Constraint Programming*, ser. Foundations of Artificial Intelligence. Elsevier, 2006, vol. 2, pp. 571–603.
[13] A. R. Bradley, "SAT-Based Model Checking without Unrolling," in *VMCAI 2011*.
[14] A. Cimatti and A. Griggio, "Software Model Checking via IC3," in *CAV 2012*.
[15] K. Hoder and N. Bjørner, "Generalized Property Directed Reachability," in *SAT 2012*.
[16] A. Komuravelli, A. Gurfinkel, and S. Chaki, "SMT-Based Model Checking for Recursive Programs," in *CAV 2014*.
[17] J. Birgmeier, A. R. Bradley, and G. Weissenbacher, "Counterexample to Induction-Guided Abstraction-Refinement (CTIGAR)," in *CAV 2014*.
[18] D. Jovanovic and B. Dutertre, "Property-Directed k-Induction," in *FMCAD 2016*.
[19] L. M. de Moura and D. Jovanovic, "A Model-Constructing Satisfiability Calculus," in *VMCAI 2013*.
[20] J. P. M. Silva and K. A. Sakallah, "GRASP - A New Search Algorithm for Satisfiability," in *ICCAD 1996*.
[21] G. S. Tseitin, "On the Complexity of Derivations in Propositional Calculus," in *Studies in Constructive Mathematics and Mathematical Logics*, 1968.
[22] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic Model Checking Using SAT Procedures instead of BDDs," in *DAC 1999*.
[23] N. Eén and N. Sörensson, "Temporal Induction by Incremental SAT Solving," *Electr. Notes Theor. CS*, vol. 89, no. 4, pp. 543–560, 2003.
[24] N. Eén, A. Mishchenko, and R. K. Brayton, "Efficient Implementation of Property Directed Reachability," in *FMCAD 2011*.
[25] T. Seufert and C. Scholl, "fbPDR: In-depth combination of forward and backward analysis in Property Directed Reachability," in *DATE 2019*.
[26] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo, "Incremental Formal Verification of Hardware," in *FMCAD 2011*.
[27] M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert, "Efficient Solving of Large Non-Linear Arithmetic Constraint Systems with Complex Boolean Structure," *JSAT*, vol. 1, no. 3-4, pp. 209–236, 2007.
[28] C. Herde, "Efficient Solving of Large Arithmetic Constraint Systems with Complex Boolean Structure: Proof Engines for the Analysis of Hybrid Discrete-Continuous Systems," Ph.D. dissertation, Carl von Ossietzky University of Oldenburg, 2011.
[29] T. Teige, "Stochastic Satisfiability Modulo Theories: A Symbolic Technique for The Analysis of Probabilistic Hybrid Systems," Ph.D. dissertation, Carl von Ossietzky University of Oldenburg, 2012.
[30] S. Kupferschmid, "Über Craigsche Interpolation und deren Anwendung in der formalen Modellprüfung," Ph.D. dissertation, University of Freiburg, 2013.
[31] A. Eggers, "Direct Handling of Ordinary Differential Equations in Constraint-Solving-Based Analysis of Hybrid Systems," Ph.D. dissertation, Carl von Ossietzky University of Oldenburg, 2014.
[32] K. Scheibler, F. Neubauer, A. Mahdi, M. Fränzle, T. Teige, T. Bienmüller, D. Fehrer, and B. Becker, "Extending iSAT3 with ICP-Contractors for Bitwise Integer Operations," SFB/TR 14 AVACS, Tech. Rep. 116, 2016.
[33] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik, "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver," in *ICCAD 2001*.
[34] R. Sebastiani, "Lazy Satisfiability Modulo Theories," *JSAT*, vol. 3, no. 3-4, pp. 141–224, 2007.
[35] K. Ravi and F. Somenzi, "Minimal Assignments for Bounded Model Checking," in *TACAS 2004*.
[36] N. Eén and N. Sörensson, "An Extensible SAT-solver," in *SAT 2003*.
[37] Y. Vizel and A. Gurfinkel, "Interpolating Property Directed Reachability," in *CAV 2014*.
[38] A. Gurfinkel and A. Ivrii, "K-Induction Without Unrolling," in *FMCAD 2017*.
[39] H. G. V. Krishnan, Y. Vizel, V. Ganesh, and A. Gurfinkel, "Interpolating Strong Induction," in *CAV 2019*.