Dead-Code Detection with IC3 using SMT-LIBv2 Solvers

Lukas Mentel[1], Tobias Seufert[2], Karsten Scheibler[1], and Christoph Scholl[2]
[1]BTC Embedded Systems AG, Oldenburg, Germany, firstname.lastname@btc-embedded.com
[2]University of Freiburg, Germany, lastname@cs.uni-freiburg.de

Abstract

In this case study we evaluate an SMT-based IC3 implementation which is designed for formally proving the absence of dead-code in embedded C code. Our IC3 implementation is able to use any off-the-shelf SMT-LIBv2 solver which allows solving under assumptions and supports QF_BVFP. We extend the basic IC3 generalization techniques by splitting theory variables into intervals enabling more fine grained reasoning. We compare different state-of-the-art SMT-LIBv2 solvers and evaluate how their performance is affected by optimizations of IC3, like target enlargement or stronger generalization of proof obligations, as well as a preprocessing technique known as *constant elimination* that has proven to be very effective in our application context. We further evaluate how IC3 with SMT-LIBv2 solvers and interval-based generalization competes in comparison to iSAT3+IC3 as a part of BTC Embedded*Platform*® which we currently consider the strongest stand-alone engine in dead-code detection on floating-point dominated benchmarks.

1 Introduction

Developing safety-critical systems must adhere to high standards of quality. For instance, the ISO 26262 standard [24] for the automotive sector demands that unintended functionality like unreachable code fragments—so-called *dead-code*—is cleaned up. The standard recommends code coverage metrics like MC/DC [26] (modified condition / decision coverage): a segment of code is considered as being covered, if at least one test case is able to execute this code. A coverage of 100 % indicates that no (unreachable) dead-code exists.

An effective approach to detect dead-code or prove its absence automatically is classical safety verification. We formulate a line of code as reachability goal and use software model checking to find an execution sequence (counterexample) that executes the line or prove its *unreachability* otherwise.

Lately, the test and verification tool suite BTC Embedded*Platform*® showed unmatched performance in model checking *floating-point dominated* industrial models from the automotive domain [40]. Furthermore, in earlier work, we successfully extended a software model checker called iSAT3 [34] included in the BTC Embedded*Platform*® with IC3 [36]. IC3 [8] and its variant PDR [17] are widely considered as the strongest available hardware model checking techniques, regarding software model checking and infinite state systems in general, however, the situation is not (yet) so clear. Nevertheless, we found that IC3 (and especially its generalization routines) greatly benefits from the combination with Interval Constraint Propagation (ICP) [3] in iSAT3 even adding important yet unproven dead-code instances to the iSAT3 portfolio (consisting of industrial-strength [40] variations of bounded model checking, interpolation-based model checking [29], as well as k-induction [39]).

We use the gained insights from our earlier interval based implementation of IC3 and apply them to a *new implementation* that uses off-the-shelf *SMT-LIBv2* [2] SMT solvers which allow solving under assumptions and support QF_BVFP (quantifier free segment of the theory of bitvectors with floating point arithmetic) like Bitwuzla [32], cvc5 [1], MathSAT [14] and Z3 [16].

Our main goal is to evaluate the *strengths and weaknesses of these SMT solvers in our problem domain*. We further hope to improve on the results of IC3 in iSAT3 and find even more yet unproven dead-code instances.

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.

Our case study will mainly (among others) focus on the following key aspects.

- During the generalization of program state sets, we consider assignments of state variables as point intervals and try to profit from more fine grained generalization of interval bounds—like it deemed very helpful in iSAT3.
- We investigate how SMT-LIBv2 solvers respond to additional techniques which worked well for IC3 in iSAT3. We try target enlargement [36, 17]—which checks for multiple step predecessors (instead of one step) of the unsafe (or target) states—as well as a preprocessing technique that eliminates constants (both syntacically and semantically) from the underlying transition relation [30].
- The problem instances we consider have been generated by BTC Embedded*Platform*® and might—in general—contain dead-end states (i.e. states which falsify the transition relation). This happens when the user specifies additional restrictions. This would render classical SAT- or SMT-based *lifting* [11] for the generalization of proof obligations (state sets

which are direct or indirect predecessors of the unsafe resp. target states) useless [38]. In previous work, we therefore applied a more cautious proof obligation generalization technique which works on general transition relations (called GeNTR [36, 38]) and is strongly related to classical clause cover techniques.

However, the considered workflow from [31] produces no such restrictions and it is therefore safe to apply *lifting*. We investigate whether the application of more stronger proof obligation generalization techniques (such as lifting over GeNTR) has a positive effect on our SMT-LIBv2 approach—as it is the case in hardware verification [17, 19, 38].

Related work

IC3 has been applied to general software model checking with mixed results [4]. There are several more or less successful attempts [12, 23, 13, 25, 20, 6, 28]. General software verification tasks often struggle with representing the control flow—which might even be non-deterministic. The authors of [12] present solutions from encoding the program counter into the state space to explicitly unwinding the control flow (and only consider data symbolically). A similar approach from [28] operates on so-called control flow automata. Other solutions run IC3 on (Boolean) abstractions and apply CEGAR-like techniques for refinement [6, 13].

Our problem domain however focuses on the detection of dead-code in automatically generated C code, which is preprocessed (loops are unrolled, functions inlined, and complex data structures flattened) and translated (via a proprietary intermediate C-like language called SMI) into one unbounded feedback loop that cascades ITE (if-then-else) as well as arithmetic operators. As a result, the control flow is implicitly represented and we are (successfully) able to apply standard IC3 on top of a generic interface to arbitrary off-the-shelf SMT-LIBv2 solvers.

Technically, our intermediate format would allow for translations to constrained horn clauses (CHC) which is supported by SMT-LIBv2. Also, translation to the input format of nuXmv [10] which implements IC3 with implicit abstraction, e.g., might be manageable. However, to the best of our knowledge, nuXmv as well as CHC solvers like Spacer [23, 27, 21] or Golem [7] do not natively support floating point reasoning and, e.g., instead might overapproximate floating point using arithmetic over the reals. These approximations are not viable in our use case[1]. Therefore, we refrain from any comparison with these tools.

Structure of the paper

In Sect. 2 we provide some preliminaries including information about IC3 and its features that are relevant for our case study. The implementation of our SMT-LIBv2 based IC3 is presented in Sect. 3 and our extensive case study is presented in Sect. 4. Finally, the paper is concluded in Sect. 5.

[1]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.

2 Preliminaries

In the following, if we consider Boolean formulas over variables, we denote a variable or its negation with the term *literal*. Furthermore, we call a disjunction of literals a *clause* and a conjunction of literals a *cube*. The negation $\neg c$ of a cube c is a clause and vice versa.

We break down the formal verification of safety properties to reachability analysis of state transition systems. Informally, we assume that such a system is represented by a set of states being assignments to a set of state variables $\vec{s}$ and a *transition relation* $T(\vec{s},\vec{i},\vec{s'})$ which encodes possible state transitions between a present state over $\vec{s}$ and a next state over $\vec{s'}$ under user-controlled inputs $\vec{i}$. We identify the inital states of a system with the predicate[2] $I(\vec{s})$ and the *good* states that satisfy a given safety property P by $P(\vec{s})$. We ask whether or not there exists a path starting in an *initial* state satisfying I to a *bad* state that violates the safety property P.

2.1 IC3

IC3 [8, 17] is able to prove safety w.r.t. $P(\vec{s})$ of a system by finding an inductive invariant $F(\vec{s})$ for which it holds that $F \Rightarrow P$. IC3 does not unroll the transition relation as other SAT-based techniques such as BMC [5] do. It incrementally constructs overapproximations $F_i(\vec{s})$ of the reachable states in up to i steps—with one exception: $F_0(\vec{s})$ is represented exactly by letting $F_0(\vec{s}) = I(\vec{s})$. The F_i are gradually strengthened by removing provably unreachable states (starting from I) that contain predecessors of the *bad* states. In particular, a *bad* state s (or predecessor of a *bad* state)—theses states are called *proof obligations*—is removed from F_{i+1} if the formula $\neg s \wedge F_i(\vec{s}) \wedge T(\vec{s},\vec{i},\vec{s'}) \wedge s'$ is unsatisfiable[3]. If $\neg s \wedge F_i \wedge T \wedge s'$ is satisfiable, a new proof obligation is extracted from the satisfying assignment ($\star$). Eventually, a proof obligation is either proven unreachable at some point (because the overapproximations were only too weak beforehand) *or* it is part of an actual counterexample which disproves safety. If a proof obligation s is unreachable in up to $i+1$ steps, we strengthen all F_k with $k \in 1,\ldots,i+1$ by letting $F_k = F_k \wedge \neg s$. This allows IC3 to always maintain the invariant $F_i \Rightarrow F_{i+1}$ for all i—including $F_0 = I$. IC3 terminates if either a counterexample is found or if all proof obligations are resolved *and* two overapproximations F_i and F_{i+1} are found equivalent.

[2]For brevity, if it is clear from context, we will sometimes skip the set of state variables in parentheses on which the respective predicates depend.

[3]It is not necessary for the correctness of IC3 to add $\neg s$ to the query. Nevertheless, it is sound and improves efficiency [17].

2.2 Generalization of Proof Obligations in IC3

Proof obligations are generalized once they are introduced at $(\star)$. We assume that m is extracted as a satisfying assignment (which is represented by a cube of literals of all state variables from $\vec{s}$) from $\neg s \wedge F_i(\vec{s}) \wedge T(\vec{s},\vec{i},\vec{s'}) \wedge s'$ and thus as a predecessor in $\neg s \wedge F_i(\vec{s})$ of another proof obligation s. In particular, we want to find a subset (in terms of literals) cube $\hat{m}$ such that *for all* possible combinations of assignments to variables of the removed literals from $m \setminus \hat{m}$ the proof obligation (and therefore *bad* state or *bad* state predecessor) s can be reached. An approach to achieve this is the so-called *lifting* [33, 11].

Lifting

Lifting considers proof obligations m which are (yet) ungeneralized, i.e. single states. We further assume that m reaches s under the user-controlled input assignment i. Hence—given a transition relation which acts as a *function* [38]—the formula $m \wedge i \wedge T \wedge \neg s'$ is unsatisfiable (mind the negation in $\neg s'$). We can now make use of IC3's underlying decision procedure (usually a SAT or SMT solver) and extract a minimal subset cube $\hat{m}$ of literals from m which is still sufficient to render $\hat{m} \wedge i \wedge T \wedge \neg s'$ *unsatisfiable*. *SMT-LIBv2* offers `get-unsat-assumptions` for this purpose.

Techniques for extracting such a subset are usually not optimal because they depend on the order of m. Hence, lifting can be extended by so-called *literal dropping*, which more aggressively tries to remove additional literals from $\hat{m}$.

However, lifting is not generally applicable: if T is non-deterministic, $m \wedge i \wedge T \wedge \neg s'$ might become *satisfiable* because there might be another successor of $m \wedge i$. If T is not total, then a seemingly valid generalization of m might contain a dead end state m' for which $m' \wedge i \wedge T$ is unsatisfiable already. As a result, the generalization might add states which are no actual predecessors of the *bad* states and therefore we end up with spurious counterexamples.

GeNTR

An alternative to lifting is so-called GeNTR [36] (Generalization with Negated Transition Relation) which is—in the spirit of clause cover approaches—applicable to general transition relations [36, 38]. We consider a full satisfying assignment $A = m \wedge i \wedge c'$ of the formula $\neg s \wedge F_i(\vec{s}) \wedge T(\vec{s},\vec{i},\vec{s'}) \wedge s'$ with m over $\vec{s}$, i over $\vec{i}$, and c' over $\vec{s'}$, whereas it must hold that[4] $c' \Rightarrow s'$. Obviously, $A \wedge T(\vec{s},\vec{i},\vec{s'})$ is valid because it fixes all variables of $T(\vec{s},\vec{i},\vec{s'})$ to a valid predecessor-successor-combination. In turn, $A \wedge \neg T(\vec{s},\vec{i},\vec{s'})$ is *unsatisfiable* and therefore again reveals a *lifting* of m (removing literals from m as long as $A \wedge \neg T(\vec{s},\vec{i},\vec{s'})$ remains unsatisfiable).

[4] This means that even though s' might already be generalized, we require a single state, i.e. a full assignment to the next state variables, included in s'.

Even though GeNTR works for general transition relations, it is preferable to use lifting—if it is applicable—since lifting is very likely to achieve more general results [38].

2.3 Generalization of Unreachable State Sets

If a proof obligation s is unreachable within up to $i+1$ steps, i.e. $\neg s \wedge F_i(\vec{s}) \wedge T(\vec{s},\vec{i},\vec{s'}) \wedge s'$ is unsatisfiable, its negation $\neg s$ (clause) is added to F_{i+1}. Again, there is potential for generalization: one might generalize s to $\hat{s}$ by removing literals such that $\neg \hat{s} \wedge F_i(\vec{s}) \wedge T(\vec{s},\vec{i},\vec{s'}) \wedge \hat{s}'$ is still unsatisfiable [17, 8, 22].

3 IC3, ICP, and SMT-LIBv2 Solvers

We present an IC3 implementation which may use any off-the-shelf SMT-LIBv2 SMT solver that supports QF_BVFP theory. The solvers are queried incrementally. When evaluating formulas like $\neg s \wedge F_i \wedge T \wedge s'$, we pass the cube s' as well as an activation literal a for $\neg s$ via `check-sat-assuming`. Similar to [36] we use one formula containing all blocked cubes which are activated and deactivated via assumption literals. We also pass full assignments m for generalization with *lifting* or *GeNTR* (see Sect. 2.2) as well as the proof obligations during further generalization of unreachable state sets (see Sect. 2.3) via `check-sat-assuming`. We extract the subset of conflicting assumptions via `get-unsat-assumptions`.

3.1 Splitting Intervals

When assuming (via `check-sat-assuming`) a particular value a of a theory variable x, the natural way would be to assume a literal l with $l \Leftrightarrow x = a$ using the '=' operator of SMT-LIBv2. However, we try to profit from the advantages of Interval Constraint Propagation (ICP) without a dedicated ICP-Solver (like iSAT3 [34]) by splitting these equalities into intervals. Hence, given an equality $x = a$, we interpret it as $x \in [a,a]$ and introduce two literals l_x, u_x. Whereas l_x represents the lower bound $l_x \Leftrightarrow x \geq a$ and u_x represents an upper bound $u_x \Leftrightarrow x \leq a$.

Obviously, assuming that $x \in [a,a]$ is equivalent to assuming that $x = a$. It is very helpful during generalization though: if it is not possible to remove the equality literal l, it might still be possible to at least remove the lower bound l_x or the upper bound u_x literals.

Furthermore, generalization will be *at least* as powerful as purely with literals that represent equality. Assume that we, e.g., apply lifting with literal dropping (see Sect. 2.2). Hence we subsequently remove literals from a proof obligation as long as the solver state remains *unsatisfiable*. If the solver state remains unsatisfiable after dropping literal l, it will also remain unsatisfiable if we drop l_x *and* l_u.

We remark that for NaN (not-a-number) floating point values, we do not consider intervals but fall back to equation instead.

In the following, if we address an IC3 version without splitted intervals, we refer to it as *with equality*.

3.2 Ungeneralization

Our implementation also incorporates a technique which is not a common feature of IC3 implementations. It worked very well in the ICP context though [36] because it benefits greatly from splitting intervals. One of IC3's main invariants is that $F_i \Rightarrow F_{i+1}$ for all overapproximations of states reachable in up to i resp. $i+1$ steps. Since $F_0 = I$, we have to take special care, that for any generalization (according to Sect. 2.3) result $\hat{c}$ of an unreachable cube c (representing a state set) it always holds that $I \Rightarrow \neg\hat{c}$.
$I \Rightarrow \neg\hat{c}$ being *valid* is equivalent to $I \wedge \hat{c}$ being *unsatisfiable*. For the ungeneralized cube c, this holds by definition of IC3 [8, 17] and it also holds that $\hat{c} \subseteq c$, if we consider the cubes as sets of literals. Thus, if $I \wedge \hat{c}$ is satisfiable, we query $I \wedge c$ (which is unsatisfiable by definition) and extract exactly the literals (we pass c via `check-sat-assuming`) via `get-unsat-assumptions` of which we have to undo their removal and add them to $\hat{c}$ again ('ungeneralize' $\hat{c}$), such that $I \wedge \hat{c}$ is also *unsatisfiable*.

3.3 Target Enlargement

In the context of IC3, the states which are considered as *bad* are usually the ones that violate P and therefore satisfy $\neg P(\vec{s})$. However, it is also clear that states which satisfy $T(\vec{s},\vec{i},\vec{s'}) \wedge \neg P(\vec{s'})$ can be considered as *bad*. We call this a target enlargement by one. If we represent the *bad* states by $T(\vec{s},\vec{i},\vec{s'}) \wedge T(\vec{s'},\vec{i'},\vec{s''}) \wedge \neg P(\vec{s''})$ we call this a target enlargement by two, and so on. There are indicators that target enlargement up to some extent is beneficial in IC3 [17, 19, 36]. Our implementation can be extended by different target enlargements and we are able to evaluate how this interacts with interval splitting and different SMT-LIBv2 solvers.

3.4 Constant Elimination

The underlying transition system might contain state variables which remain constant on every possible execution path (even though they are not declared as such). It seems desireable to detect such constants and feed this information to the underlying SMT-LIBv2 solver. For instance, during the generalization of unreachable states (see Sect. 2.3) constant state variables can be immediately removed [37].
We have two versions of constant elimination from [30] at hand. One that applies a syntactic approach and another one that applies a semantic approach. It is determined, that each constant which is detected by the syntactic approach is also detected by the semantic approach (but not vice versa). The semantic approach is not guaranteed to find all constants.
We investigate how robust the SMT-LIBv2 solvers are w.r.t. more or less (un-)detected constant state variables.

4 Experimental Results

For our experiments we use the same benchmarks as [35], [36] and [31]. 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 [26]) which correlates to the reachability of a certain line of code. Thus, unreachable goals correspond to dead-code.
Unfortunately, the benchmarks are actual customer code and we are not allowed to disclose them. However, we are working on making at least parts of them public at some point.
The experiments (one benchmark per core) were performed on a cluster with each cluster node having 64 GB RAM and two 8-core CPUs @2.6 GHz. We limit CPU-time and memory to 1 h resp. 4 GB.
The used SMT-LIBv2 solvers are used—if not stated otherwise—in their latest[5] versions and in their default configurations.
Among the SMT-LIBv2 solvers which are able to handle QF_BVFP (quantifier free segment of the theory of bitvectors with floating point arithmetic) we found that *bitwuzla* as well as MATHSAT perform overwhelmingly better than *cvc5* as well as Z3 in our specific application context. Therefore, for the sake of brevity, we consider only *bitwuzla* and MATHSAT in the detailed discussion and only mention the two best configurations for *cvc5* and Z3 in our final overall comparison. This is also the reason why we have no GeNTR versions of *cvc5* and Z3 listed in Tab. 2.
We distinguish between the different versions by their respective added features: +int for interval splitting (see Sect. 3.1), +l for lifting instead of GeNTR (see Sect. 2.2), +ld for additional (bound[6]) literal dropping (see also Sect. 2.2), +ce for constant elimination (see Sect. 3.4), and +t*i* for a target enlargement by *i* time frames (see Sect. 3.3). As *plain* versions, we consider standard IC3 with equality, GeNTR, and without any target enlargement or constant elimination.
Our results are displayed in Tab. 1. We denote found counterexamples by 'CEX' (reachable code) and proofs (unreachable dead-code) by 'DC'. Timeouts resp. memouts are denoted by 'T/M'. We remark that we focus mostly on 'DC' instances, which are the most practically relevant results.
Counterexamples are internally sanity-checked by resimulation. All of the proofs are internally double-checked with another *bitwuzla* instance. That means, that if IC3 returns an inductive strengthening F of P, then we use the SMT solver to prove that

- $I \implies F \wedge P$ and
- $F \wedge P \wedge T \implies F' \wedge P'$.

[5] Experiments performed in April, 2024.
[6] If this configuration also uses interval splitting, then literal dropping is implicitly able to drop lower as well as upper bound literals.

Configuration	CEX	**DC**	T/M
IC3 *bitwuzla*	5241	444	3093
IC3 *bitwuzla* +int	6534	761	1483
IC3 *bitwuzla* +ce	5196	810	2772
IC3 *bitwuzla* +ce +int	6716	933	1129
IC3 *bitwuzla* +ce +l	6130	936	1712
IC3 *bitwuzla* +ce +int +l	7446	**1007**	325
IC3 *bitwuzla* +ce +l +ld	6087	933	1758
IC3 *bitwuzla* +ce +int +l +ld	7431	999	348
IC3 *bitwuzla* +ce +int +l +t2	7394	1003	381
IC3 *bitwuzla* +ce +int +l +t3	7515	986	277
IC3 *bitwuzla* +ce +int +l +ld +t2	7326	978	474
IC3 *bitwuzla* +ce +int +l +ld +t3	7527	956	295
IC3 *bitwuzla* **v1230d80** +ce +int +l +ld	7586	**1015**	177
IC3 MATHSAT	5807	798	1919
IC3 MATHSAT +int	6930	952	1133
IC3 MATHSAT +ce	5863	905	2010
IC3 MATHSAT +ce +int	7060	990	728
IC3 MATHSAT +ce +l	6421	943	1414
IC3 MATHSAT +ce +int +l	7544	**1007**	227
IC3 MATHSAT +ce +l +ld	6584	942	1252
IC3 MATHSAT +ce +int +l +ld	7561	**1008**	209
IC3 MATHSAT +ce +int +l +t2	7576	**1009**	197
IC3 MATHSAT +ce +int +l +t3	7577	1000	205
IC3 MATHSAT +ce +int +l +ld +t2	7562	**1007**	213
IC3 MATHSAT +ce +int +l +ld +t3	7567	1000	215
IC3 Z3 +ce +int +l	6033	834	1911
IC3 Z3 +ce +int +l +ld	6522	864	1392
IC3 *cvc5* +ce +int +l	6076	832	1870
IC3 *cvc5* +ce +int +l +ld	6517	864	1397
iSAT3+IC3	7622	1003	153

Table 1 Detailed Results. All configurations that achieved better results than the original IC3 implementation in iSAT3 in detecting dead-code are highlighted in boldface. The result of the best performing version is additionally underlined.

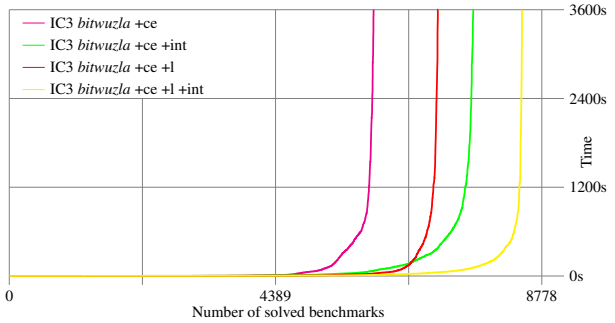

Figure 1 *bitwuzla* and the effect of interval splitting.

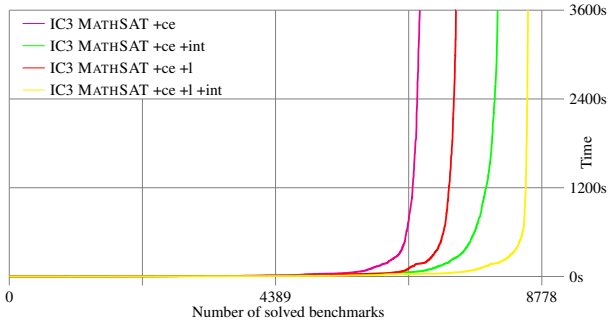

Figure 2 MATHSAT and the effect of interval splitting.

4.1 Effect of Interval Splitting

We showcase the effect that *interval splitting* (see Sect. 3.1) had on *bitwuzla* and MATHSAT. The versions which included interval splitting are marked by '+*int*'.

Besides the results in Tab. 1, we present survival plots for *bitwuzla* in Fig. 1 as well as for MATHSAT in Fig. 2. There we compare the effect of interval splitting (+int) on versions with and without lifting (+l) enabled; we consider the versions with semantic constant elimination (+ce).

Apparently, both solvers benefit greatly from interval splitting instead of simple equality. The largest effect can be observed for plain *bitwuzla* with constant elimination (+ce), where interval splitting allows for almost 1700 additionally solved instances.

The effect of better generalization via interval splits slightly weakens with better proof obligation generalization (+l, +ld).

In iSAT3 [36] we also considered *bound generalization*, which is a cheap technique as it can be performed when traversing the implication graph to collect the relevant assumption literals. With off-the-shelf SMT-LIBv2 solvers additional solve calls would be required to check whether a bound can be generalized to a weaker value. Another reason to not add bound generalization, is that previous experiments with iSAT3 showed that it does not help to increase the number of detected dead-code instances which is our main focus here.

4.2 Effect of Stronger Proof Obligation Generalization

We observe that stronger generalization of proof obligations via lifting (+l) and additional literal dropping (+ld) has great effect on the performance. Inspired by TIP [18, 15], we abort literal dropping after two failed or 32 total attempts.

	bitwuzla	MATHSAT	Z3	cvc5
GeNTR	19.41	21.20	-	-
Lifting	20.27	20.20	25.34	19.64
Lifting +ld	20.44	26.23	32.37	23.72

Table 2 Proof Obligation generalization capabilities in %.

Bitwuzla seems to profit a lot more from these techniques than MATHSAT. Whereas MATHSAT is able to detect 990 dead-code instances only with GeNTR (gain by lifting is 17 'DC' instances), *bitwuzla* improves from 933 (GeNTR) to 1007 with lifting.

Results for additional literal dropping are mixed: MATHSAT seems to profit only slightly (one more 'DC' instance), *bitwuzla* not. Z3 and *cvc5*, however, gain *a lot more* instances by adding literal dropping.

In Tab. 2 we present the average reduction ratio (original size in terms of literals, divided by generalized size) over all generalization attempts on jointly solved benchmarks for each solver. Interestingly, the generalization capabilities (e.g., via `get-unsat-assumptions`) of the solvers do not seem to correlate with their overall performance. Nevertheless, the rather good GeNTR reduction ratio achieved by MATHSAT might be a cause for it only solving slightly less benchmarks than with lifting. Z3 is able to find the best proof obligation generalizations on average.

4.3 Effect of Target Enlargement

In iSAT3+IC3 [36] we observed a significant gain in solved problem instances by adding an adaptive target enlargement (in case IC3 gets 'stuck'). In this case study however, we observed that *adaptive* target enlargement is not beneficial. These were observations that we made prior to our case study, therefore there are *no* statistics regarding *adaptive* target enlargement in Tab. 1. Target enlargement in general only slightly improves the performance of MATHSAT. There are some slight improvements in the 'CEX' instances (only marginal from t2 to t3).

We expected that at least the 'CEX' instances would *always* improve under target enlargements—*bitwuzla* though even performed worse in configuration +ce +int +l **+t2** compared to the same without target enlargement +ce +int +l.

4.4 Effect of Constant Elimination

We compare the effect of constant elimination (+ce) on versions with and without lifting (+l) enabled. Furthermore, we compare semantic constant elimination (+ce) against syntactic constant elimination (+ce (syntactic)).

We present survival plots for *bitwuzla* in Fig. 3 as well as for MATHSAT in Fig. 4.

Apparently, *semantic* constant elimination provides us

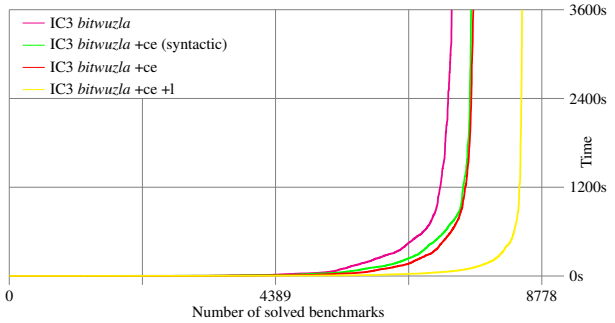

Figure 3 *bitwuzla* and the effect of constant elimination.

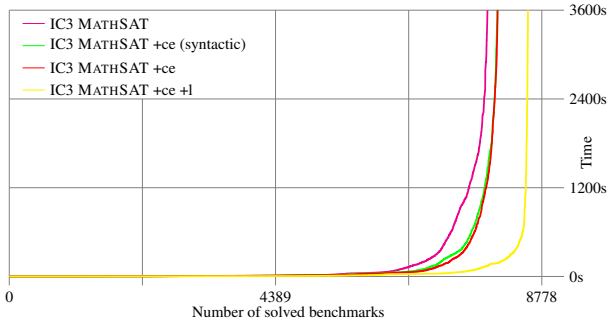

Figure 4 MATHSAT and the effect of constant elimination.

with better results than the *syntactic* variant. In fact, we observed that the results of semantic constant elimination almost always subsumed the ones with syntactic constant elimination. Therefore we only consider *semantic* constant elimination (+ce) in Tab. 1.

It seems like constant elimination is beneficial throughout all IC3 configurations. This matches previous results we had for *k-induction* in [30].

4.5 Overall Comparison

If we consider the overall performance of the best configurations for each solver, *Bitwuzla* and MATHSAT perform similarly well, whereas both Z3 and *cvc5* also perform similarly (but not so well).

Furthermore, we made the very interesting observation, that *bitwuzla* in an *older* version (v1230d80) performs better than in the current version—even outperforming MATHSAT.

Compared to the currently strongest IC3 implementation in BTC Embedded*Platform*® which is denoted by 'iSAT3+IC3' in Tab. 1, the here considered configurations are able to detect at most *twelve* more (very valuable[7]) instances of dead-code.

Lastly, in Fig. 5 we present a survival plot which compares the configuration with semantic constant elimination, interval splitting, and lifting (+ce +int +l) of all SMT-LIBv2 solvers. We remark that here we depict the latest version of *bitwuzla* (not v1230d80).

[7]While 12 out of 8778 sounds insignificant at first sight, we want to point out, that only a minority (we suspect around 1100) are actual dead-code. Furthermore, we consider only around 100 to be *really* hard for IC3. Thus, solving twelve more of these hard instances is pretty valuable for us.

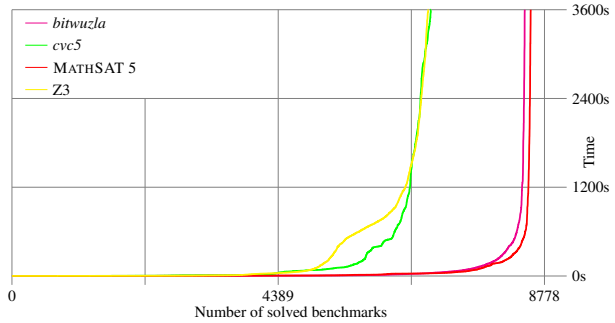

Figure 5 Overall comparison in +ce + int +l configuration.

5 Conclusion and Future Work

We presented an IC3 implementation that uses an SMT-LIBv2 interface and is therefore compatible with any SMT solver which supports SMT-LIBv2 as well as the theory QF_BVFP. For generalization queries handed to the SMT solvers, we split equalities into a lower and an upper bound of a point interval, enabling more fine grained reasoning. We evaluated different state-of-the-art SMT solvers in our application context of dead-code detection in embedded C code. We further investigated how the SMT solvers are affected by different techniques for the generalization of proof obligations, target enlargement, as well as a preprocessing technique that detects constant variables.

We observed that *bitwuzla* and MATHSAT outperform Z3 and *cvc5* on our benchmark set. Apparently, all of the configurations we considered benefit greatly from generalizing with lower and upper bounds instead of equalities—especially, if a weaker proof obligation technique (like GeNTR) has to be used. Furthermore, preprocessing benchmarks with constant elimination from [30] also pays off in the IC3 context (with all of the considered SMT-LIBv2 solvers).

Finally, we were able to improve upon previous results of our native ICP implementation in iSAT3+IC3. It would be very interesting to further elaborate on the reasons for the (sometimes vast) differences between our selection of SMT-LIBv2 solvers.

Considering future work, it might be interesting to bit blast into AIGER and compare the word-level SMT approach to a bit-level SAT approach with tools like ABC's PDR [9].

6 Literature

[1] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms*

for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[2] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, volume 13, page 14, 2010.

[3] 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.

[4] Dirk Beyer and Matthias Dangl. Software verification with PDR: an implementation of the state of the art. In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part I*, volume 12078 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 2020.

[5] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic model checking using SAT procedures instead of bdds. In Mary Jane Irwin, editor, *Proceedings of the 36th Conference on Design Automation, New Orleans, LA, USA, June 21-25, 1999*, pages 317–320. ACM Press, 1999.

[6] Johannes Birgmeier, Aaron R. Bradley, and Georg Weissenbacher. Counterexample to induction-guided abstraction-refinement (CTIGAR). In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 831–848. Springer, 2014.

[7] Martin Blicha, Konstantin Britikov, and Natasha Sharygina. The golem horn solver. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*, volume 13965 of *Lecture Notes in Computer Science*, pages 209–223. Springer, 2023.

[8] Aaron R. Bradley. Sat-based model checking without unrolling. In Ranjit Jhala and David A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011.

[9] Robert K. Brayton and Alan Mishchenko. ABC: an academic industrial-strength verification tool. In Tayssir Touili, Byron Cook, and Paul B. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 24–40. Springer, 2010.

[10] 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.

[11] Hana Chockler, Alexander Ivrii, Arie Matsliah, Shiri Moran, and Ziv Nevo. Incremental formal verification of hardware. In Per Bjesse and Anna Slobodová, editors, *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 135–143. FMCAD Inc., 2011.

[12] Alessandro Cimatti and Alberto Griggio. Software model checking via IC3. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 277–293. Springer, 2012.

[13] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. IC3 modulo theories via implicit predicate abstraction. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 46–61. Springer, 2014.

[14] 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*, 2013.

[15] Koen Claessen and Niklas Sörensson. A liveness checking algorithm that counts. In Gianpiero Cabodi and Satnam Singh, editors, *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*, pages 52–59. IEEE, 2012.

[16] 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.

[17] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. Efficient implementation of property directed reachability. In Per Bjesse and Anna Slobodová, editors, *International Conference*

on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011, pages 125–134. FMCAD Inc., 2011.

[18] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[19] 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):1026–1039, 2016.

[20] Henning Günther, Alfons Laarman, and Georg Weissenbacher. Vienna verification tool: IC3 for parallel software - (competition contribution). In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9636 of *Lecture Notes in Computer Science*, pages 954–957. Springer, 2016.

[21] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The seahorn verification framework. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 343–361. Springer, 2015.

[22] Zyad Hassan, Aaron R. Bradley, and Fabio Somenzi. Better generalization in IC3. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 157–164. IEEE, 2013.

[23] Krystof Hoder and Nikolaj S. Bjørner. Generalized property directed reachability. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2012.

[24] ISO. Road vehicles – Functional safety, 2011.

[25] Dejan Jovanovic and Bruno Dutertre. Property-directed k-induction. In Ruzica Piskac and Muralidhar Talupur, editors, *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, pages 85–92. IEEE, 2016.

[26] Hayhurst Kelly J., Veerhusen Dan S., Chilenski John J., and Rierson Leanna K. A Practical Tutorial on Modified Condition/Decision Coverage. Technical report, 2001.

[27] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. Smt-based model checking for recursive programs. *Formal Methods Syst. Des.*, 48(3):175–205, 2016.

[28] Tim Lange, Martin R. Neuhäußer, Thomas Noll, and Joost-Pieter Katoen. IC3 software model checking. *Int. J. Softw. Tools Technol. Transf.*, 22(2):135–161, 2020.

[29] Kenneth L. McMillan. Interpolation and sat-based model checking. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

[30] Lukas Mentel, Karsten Scheibler, and Tino Teige. Detection and elimination of constants to strengthen k-induction. In *Methods and Description Languages for Modelling and Verification of Circuits and Systems, MBMV 2022, 25th Workshop, Virtual Event, Germany, February 17-18, 2022*, pages 1–10. VDE/IEEE, 2022.

[31] Lukas Mentel, Karsten Scheibler, Felix Winterer, Bernd Becker, and Tino Teige. Benchmarking SMT solvers on automotive code. In *Methods and Description Languages for Modelling and Verification of Circuits and Systems, MBMV 2021, 24th Workshop, Virtual Event, Germany, March 18-19, 2021*, pages 1–10. VDE/IEEE, 2021.

[32] Aina Niemetz and Mathias Preiner. Bitwuzla. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*, volume 13965 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2023.

[33] 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.

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

[35] Karsten Scheibler, Felix Neubauer, Ahmed Mahdi, Martin Fränzle, Tino Teige, Tom Bienmüller, Detlef Fehrer, and Bernd Becker. Accurate icp-based floating-point reasoning. In Ruzica Piskac and Muralidhar Talupur, editors, *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, pages 177–184. IEEE, 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, Grenoble,*

France, February 1-5, 2021, pages 1116–1121. IEEE, 2021.

[37] Tobias Seufert, Christoph Scholl, Arun Chandrasekharan, Sven Reimer, and Tobias Welp. Making PROGRESS in property directed reachability. In Bernd Finkbeiner and Thomas Wies, editors, *Verification, Model Checking, and Abstract Interpretation - 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16-18, 2022, Proceedings*, volume 13182 of *Lecture Notes in Computer Science*, pages 355–377. Springer, 2022.

[38] Tobias Seufert, Felix Winterer, Christoph Scholl, Karsten Scheibler, Tobias Paxian, and Bernd Becker. Everything you always wanted to know about generalization of proof obligations in PDR. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 42(4):1351–1364, 2023.

[39] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000.

[40] Lukas Westhofen, Philipp Berger, and Joost-Pieter Katoen. Benchmarking software model checkers on automotive code. In Ritchie Lee, Susmit Jha, and Anastasia Mavridou, editors, *NASA Formal Methods - 12th International Symposium, NFM 2020, Moffett Field, CA, USA, May 11-15, 2020, Proceedings*, volume 12229 of *Lecture Notes in Computer Science*, pages 133–150. Springer, 2020.