Better Generalization in IC3

Zyad Hassan, Aaron R. Bradley, and Fabio Somenzi
Dept. of Electrical, Computer, and Energy Engineering
University of Colorado at Boulder
Email: zyad.hassan@colorado.edu, bradleya@colorado.edu, fabio@colorado.edu

***Abstract*—An improved clause generalization procedure for IC3 is presented. Whereas standard generalization extracts a relatively inductive clause from a single state, called a counterexample to induction (CTI), the new procedure also extracts such clauses from other states, called counterexamples to generalization (CTG), that interfere with the primary generalization attempt. The motivation is to enable IC3 to explore states farther from the error states than are CTIs while remaining property-focused. CTGs are strong candidates for being farther but still backward reachable. Significant reductions in the maximum depth reached by IC3's priority queue-directed explicit backward search indicate that this intention is achieved in practice. The effectiveness of the new procedure is established in two independent implementations of IC3, which demonstrate an increase of 17 and 27, respectively, in the number of solved HWMCC benchmarks.**

I. Introduction

IC3 [1], [2] is an incremental, inductive model checking algorithm for invariance properties. It operates in a demand-driven manner, generating relatively inductive lemmas in response to states that interfere with the inductiveness of the property. Lemma generation proceeds incrementally until an inductive strengthening is discovered or the lemmas guide the backward search to a counterexample trace. IC3 is SAT-based but, in contrast to other SAT-based approaches, poses relatively easy but numerous SAT queries that arise from considering single steps of a transition relation. This style of using a SAT solver keeps its memory footprint small.

One of the key components of IC3 is inductive generalization. While IC3 has an element of explicit state model checking in that it examines individual states, called counterexamples to induction (CTIs), inductive generalization makes it symbolic, allowing it to handle huge state spaces. IC3's success on a model thus hinges on its ability to generalize facts that it discovers from considering specific states.

The effectiveness of generalization depends on the connectivity of a model's state graph and its encoding. Some encodings and some models, independent of encoding, coupled with the overapproximate nature of inductive generalization, require IC3 to examine more individual states. Consider the state graph in Figure 1, where 000 is the initial state and 001 is the bad state. This model has two counterexamples to the inductiveness of the property: 110 and 100, two good states with a bad successor.

Suppose state 100 is the first CTI that IC3 finds. Since this state does not have predecessors, its negation is inductive, so that IC3 concludes it is unreachable. The unreachability of this state is a specific fact that IC3 next tries to generalize in order to prove that other states are unreachable as well. It does so by attempting to drop as many literals as possible. However, in this case no literals can be dropped. For example, if IC3 attempts to drop the third literal, the negation of the resulting cube $10-$, where $-$ indicates a don't care, is not inductive because of the predecessor 011 to state 101. If there is a cube whose negation is inductive and excludes both 100 and 101, that cube must also include 101's predecessor, 011. However, the smallest cube that includes all three states is $---$, which includes the initial state and whose negation is therefore not inductive. Similar reasoning shows that IC3 also cannot drop the first and second literals. Thus the strongest clause that can be derived through generalization only blocks the CTI itself. IC3 then has to prove that the other CTI (110) is unreachable without having learned much from the first CTI.

A state that hinders a generalization attempt (011 in the example) is called a *counterexample to generalization* (CTG): it prevents dropping a literal (the third in the example), i.e., generalizing to a larger cube. Despite being itself unreachable, state 011 causes the inclusion of an initial state into the cube that covers both it and $10-$, which in turn causes generalization to fail. In this case, it is useful to focus some effort on the CTG rather than only on the CTI. Since the negation of the CTG is inductive, IC3 can block it. Then, with its predecessor blocked, dropping the third literal of 101 succeeds. Indeed, the second literal can be dropped as well, as all predecessors of the cube $1--$ are blocked. This further expansion takes care of state 110 as well, ending the analysis.

This example motivates the improved generalization procedure described in this paper. The proposed procedure addresses CTGs that appear during the generalization of some CTI-derived relatively inductive clause. CTGs are often deep back-

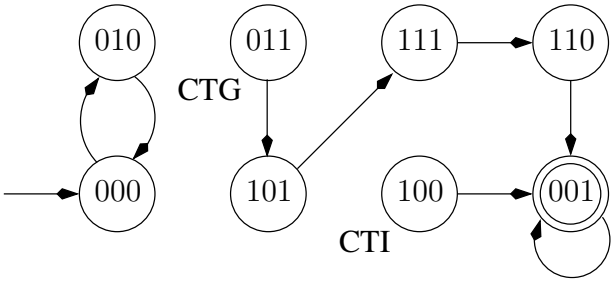


Fig. 1. Failure to generalize a clause.

ISBN 978-0-9835678-3-7/13.

ward reachable states. Addressing them reduces the depth of the explicit backward search IC3 performs and allows stronger inductive generalizations.

The proposed generalization procedure is evaluated within the implementations of IC3 in the model checkers IImc [3] and ABC [4]. Both show considerable improvement on Hardware Model Checking Competition (HWMCC) benchmarks [5].

Preliminaries are in Section II. Section III describes the proposed generalization procedure. Section IV presents the results of improved generalization on the HWMCC 2010–2012 benchmark suites. The behavior of IC3 with the improved procedure is studied in detail in Section V. Section VI discusses related work. Finally, conclusions are in Section VII.

II. Preliminaries

A. Transition Systems and Induction

Following standard practice, a *finite-state system* is represented as a tuple $S : (\bar{i},\ \overline{x},\ I(\overline{x}),\ T(\bar{i},\overline{x},\overline{x}'))$ consisting of primary inputs $\bar{i}$, state variables $\overline{x}$, a propositional formula $I(\overline{x})$ describing the initial configurations of the system, and a propositional formula $T(\bar{i},\overline{x},\overline{x}')$ describing the transition relation. Primed state variables $\overline{x}'$ represent the next state.

A state of the system is an assignment of Boolean values to all variables $\overline{x}$ and is described by a *cube* over $\overline{x}$, which is a conjunction of literals, each *literal* a variable or its negation. An assignment s to all variables of a formula F either satisfies the formula, $s \models F$, or falsifies it, $s \not\models F$. If s is interpreted as a state and $s \models F$, then s is called an F*-state*. A formula F *implies* another formula G, written $F \Rightarrow G$, if every satisfying assignment of F satisfies G. The (in)validity of $F \Rightarrow G$ is established by querying a SAT solver for the unsatisfiability of $F \wedge \neg G$.

A *clause* is a disjunction of literals. A subclause $d \subseteq c$ is a clause d whose literals are a subset of c's literals.

A *run* of S, $s_0, s_1, s_2, \ldots$, which may be finite or infinite in length, is a sequence of states such that $s_0 \models I$ and for each adjacent pair (s_j, s_{j+1}) in the sequence, $\exists \bar{i}.(\bar{i}, s_j, s'_{j+1}) \models T$. That is, a run is the sequence of assignments in an execution of the transition system. A state that appears in some run of the system is *reachable*.

Checking a *safety property* of S is reducible to checking an invariance property [6]. An *invariance property* $P(\overline{x})$, a propositional formula, asserts that only P-states are reachable. P is *invariant* for the system S (that is, S-invariant) if indeed only P-states are reachable. If P is not invariant, then there exists a finite *counterexample* run $s_0, s_1, \ldots, s_k$ such that $s_k \not\models P$. An invariance property $P(\overline{x})$ is *inductive* if

1) (*initiation*) every initial state satisfies the property: $I(\overline{x}) \Rightarrow P(\overline{x})$; and
2) (*consecution*) every transition from a P-state leads to a P-state: $P(\overline{x}) \wedge T(\bar{i},\overline{x},\overline{x}') \Rightarrow P(\overline{x}')$.

While an inductive property P is invariant, the converse is not necessarily true. In this case, it is customary to seek an *inductive strengthening* of P, which is a formula F such that $F \wedge P$ is inductive.

An assertion F is *inductive relative to* another assertion G, possibly containing primed variables, if

1) every initial state satisfies F: $I(\overline{x}) \Rightarrow F(\overline{x})$; and
2) F satisfies consecution under assumption G: $G(\overline{x},\overline{x}') \wedge F(\overline{x}) \wedge T(\bar{i},\overline{x},\overline{x}') \Rightarrow F(\overline{x}')$.

B. An Overview of IC3

IC3 maintains a sequence of overapproximations F_i to sets of states reachable within i steps, for $0 \leq i \leq k$, where F_k is the frontier. Each F_i is a conjunction of the property P with an initially empty set of clauses. For each $k > 0$, IC3 refines the F_i's for $i \leq k$ as needed to prove inductiveness of P relative to F_k. This refinement is property-driven: a counterexample to the inductiveness (CTI) of the property, which is an F_k-state with a $\neg P$-successor, triggers IC3 to derive a clause to block it. If successful, it applies induction to generalize the clause to block many more states than the CTI alone. It then adds the generalized clause to F_i for all $i \leq k$.

Otherwise, it explores (transitive) predecessors of the CTI to derive supporting strengthening clauses until the original CTI can itself be addressed relative to F_k. This exploration of concrete predecessors is guided by a priority queue of pairs of states and frame indices: (s, i) represents the obligation that state s must be inductively excluded relative to F_i, i.e., proved unreachable for at least $i + 1$ steps. Obligations are handled in lowest-index-first order, guaranteeing termination. IC3 aggressively generalizes from states: once it addresses (s, i) by finding a clause $c \subseteq \neg s$ that is inductive relative to some F_j, $j \geq i$, IC3 adds obligation $(s, j+1)$ to the queue if $j < k$. This aggressive strategy not only facilitates early discovery of mutually inductive clauses; it also allows IC3 to find deep counterexamples even when k is small.

When no CTIs remain (for F_k), IC3 checks each clause of each F_i to determine if it can be propagated forward, i.e., if it has become inductive relative to F_i since its creation because of subsequent strengthening of F_i. In the process, IC3 determines whether any F_i has become an inductive strengthening of the property, in which case the property is declared to hold. If not, it increments k and "bootstraps" the new frontier F_k with all clauses that are inductive relative to F_{k-1}. This process continues until IC3 finds an inductive strengthening of the property or finds a counterexample by following a sequence of CTIs back to an initial state.

IC3 generalizes a clause by using induction to guide the dropping of literals. IC3's generalization procedure is described in Listing 1. Notice, when reading the pseudocode, that cubes are passed by reference. The minimum-inductive clause procedure (MIC) attempts to drop each literal in turn from q, calling down to validate each potential strengthening of the clause (and, as a side effect, to further strengthen the clause). If down reports that the literal cannot be dropped, MIC returns it to the clause.

Given a cube q, the down procedure seeks the maximal inductive subclause of $\neg q$. It returns true if found and false if no inductive subclause exists. The down procedure effectively computes an overapproximation of states backward reachable

Listing 1. IC3 generalization procedure.

```
void MIC(q: cube ref, i: level):                          1
  foreach literal l in q:                                 2
    q̂ := q \ l                                            3
    if down(q̂, i):                                        4
      q := q̂                                              5
                                                          6
bool down(q: cube ref, i: level):                         7
  while true                                              8
    if I ⇏ ¬q:                                            9
      return false                                        10
    if F_i ∧ ¬q ∧ T ⇒ ¬q′:                                 11
      return true                                         12
    with (F_i ∧ ¬q)−state s:                              13
      q := q ⊔ s                                          14
```

from a given state set. It limits the cost by maintaining sets of states in the form of a single cube. If upon reaching a fixpoint, the cube does not include any of the initial states, the cube represents a set of states that are unreachable in $i+1$ steps. Denoting the cube at iteration j by q_j, each fixpoint iteration queries the SAT solver for the existence of an $(F_i \wedge \neg q_j)$-predecessor to some q_j-state. The absence of such a predecessor indicates the inductiveness of $\neg q_j$ relative to F_i. Otherwise, there is an $(F_i \wedge \neg q_j)$-state s that is a predecessor to some q_j-state. A new cube q_{j+1} is formed by taking the common literals in q_j and s (denoted by $q_j \sqcup s$). The number of literals in the cube thus strictly decreases in every iteration, effectively expanding the set of states in q_j.

The MIC procedure can be optimized using the up procedure [7], which is outside the scope of this paper.

III. Addressing Counterexamples-to-Generalization

A. Presentation of the Procedure

Keeping only the common literals of q_j and s provides an overapproximating union over state sets—a *join* in the cube lattice. While this operation responds to the need to include the q_j-predecessor s in the state set described by q_{j+1}, it also typically brings in other F_i-states. Therefore, even when all q_0-states are unreachable, down (eventually) fails if, through overapproximation, it incorporates a reachable state.

State s is called a *counterexample to generalization* (CTG) since it is encountered in the context of dropping a literal (in MIC) in order to generalize a cube. Unlike CTIs, states brought in as a result of dropping a literal or joining are not necessarily backward reachable from the error. On one hand, if s is backward reachable—and it represents a set of deep backward reachable states—then addressing it could save IC3 from having to explicitly traverse the state graph from the error state to s. On the other hand, if s is neither backward nor forward reachable, it could still obstruct generalization: when it is joined with q_j to form q_{j+1}, it could cause the inclusion of a reachable state. As described so far, IC3 would never attempt directly to block s since it only generalizes from backward reachable states. Yet blocking s, rather than joining with it, could enable finding an inductive subclause, thereby helping the generalization procedure produce stronger clauses and potentially shortening the way to a proof.

Listing 2. Proposed generalization procedure.

```
void MIC(q: cube ref, i: level):                          1
  MIC(q, i, 1)                                            2
                                                          3
void MIC(q: cube ref, i: level, d: recDepth):             4
  foreach literal l in q:                                 5
    q̂ := q \ l                                            6
    if ctgDown(q̂, i, d):                                  7
      q := q̂                                              8
                                                          9
bool ctgDown(q: cube ref, i: level,                       10
                          d: recDepth):                   11
  ctgs := 0                                               12
  while true:                                             13
    if I ⇏ ¬q:                                            14
      return false                                        15
    if F_i ∧ ¬q ∧ T ⇒ ¬q′:                                 16
      return true                                         17
    with (F_i ∧ ¬q)−state s:                              18
      if d > maxDepth:                                    19
        return false                                      20
      if ctgs < maxCTGs and i > 0 and                     21
            (I ⇒ ¬s) and (F_{i−1} ∧ ¬s ∧ T ⇒ ¬s′):         22
        ctgs := ctgs + 1                                  23
        for j := i to k do:                               24
          if F_j ∧ ¬s ∧ T ⇏ ¬s′:                           25
            break                                         26
        MIC(s, j − 1, d + 1)                              27
        clauses(F_j) := clauses(F_j) ∪ ¬s                 28
      else:                                               29
        ctgs := 0                                         30
        q := q ⊔ s                                        31
```

A generalization procedure that addresses CTGs is presented in Listing 2. Similarly to down, ctgDown first checks whether $\neg q$ is inductive (lines 14–17). However, if it is not inductive, it does not immediately join q with the discovered predecessor s. Rather, it attempts to block s at level i by proving it inductive relative to F_{i-1} (line 22). If this attempt succeeds, it tries to block it at higher levels (lines 24–26). It then strengthens the clause at the highest level relative to which it was found to be inductive by applying MIC (line 27). (Again, notice that cubes are passed by reference, so that when MIC returns, the cube s may be significantly expanded.) Having addressed one cause for the non-inductiveness of $\neg q$, ctgDown returns its attention to q.

To maintain its focus on the main goal of strengthening $\neg q$, ctgDown considers at most maxCTGs CTGs between joins (line 21). If the limit is exceeded or a CTG is not found to be inductive, the CTG is joined with q (line 31). New states brought in as a result of the join present an opportunity to explore behaviors farther from the error, so ctgDown resets the number of allowable CTGs to maxCTGs (line 30).

Since ctgDown calls MIC, the version of MIC associated with ctgDown monitors the recursion depth through its d parameter. The recursion depth is initialized by the wrapper function

to 1 (line 2) and updated by the call to MIC from ctgDown (line 27). At a recursion depth of 1, ctgDown examines CTGs that are encountered during generalization of a CTI-induced clause. For larger depths, an encountered CTG is one that interferes with the generalization of a CTG-induced clause. A limit, maxDepth, limits the effort spent on addressing CTGs of CTG-induced clauses. When this limit is exceeded (line 19), CTGs are not examined, and joins are disabled; instead, ctgDown fails immediately if $\neg q$ is not inductive (line 20).

B. Discussion

The recycled limit on handling CTGs results in an interesting pattern of state exploration. IC3 itself explores, through its priority queue, the state space in an explicit manner backward from the error. Let s be such a state: s can reach the error in a relatively few number of steps. If IC3 is forced to consider a predecessor of s, then it is known that the predecessor, too, can reach the error. In contrast, when MIC is applied to s, the first step is to drop a literal, enlarging the represented state set. In ctgDown, up to maxCTGs times, predecessors of the enlarged cube are then considered as CTGs. They are likely to be backward reachable; they are also likely to be about as close to an error as s is[1].

Eventually maxCTGs is exhausted, forcing a join. Predecessors to the enlarged cube are then considered as CTGs. These predecessors are less likely to be backward reachable but more likely to be "farther" from an error than s. Deep backward reachable states may be particularly valuable. This cycle can continue for several iterations, each iteration exploring states that are increasingly far from the error but at the cost of being increasingly likely not to be able to reach the error. Further iterations of dropping literals by MIC add layers of likelihoods of depth and backward reachability to the state exploration.

Another behavior worth noting is that ctgDown can fail more softly than down. When down fails, the only gained information is that the dropped literal is actually required. In contrast, ctgDown may successfully handle some CTGs on the way to failing to prove the inductiveness of the given cube. These CTG-derived lemmas could well prove useful in addressing the overall model checking problem.

In early attempts at considering CTGs, a scheme that delayed the handling of CTGs as much as possible was investigated. Rather than prioritizing the direct handling of CTGs over joining with them, it aggressively joined and only handled CTGs upon failure. If $\neg q_j$ failed initiation, the last-encountered CTG s was addressed directly. Successful elimination of s would enable the reconsideration of q_{j-1}; failure would cause the CTG leading from q_{j-2} to q_{j-1} to be addressed instead, and so on. This version was inferior to ctgDown, possibly because too much effort was put into addressing states that were either not actually backward reachable or too removed from the original CTI to be relevant to the generalization effort. ctgDown explores CTGs in an outwardly expanding set from the error; the unsuccessful variant explored CTGs in an inwardly contracting set.

While these explanations assume characteristics of a state space that need not hold for a given model, they offer an intuitive motivation for using ctgDown instead of down: with some trade-offs, it jumps to deep states, complementing IC3's conservative top-level behavior. Section V compares, empirically, the behavior of IC3 with down versus with ctgDown.

IV. Results

In this section IC3 with ctgDown is compared empirically to existing standard implementations of IC3. The new procedure was implemented within the IC3 engines in IImc v1.3 (upcoming release) [3] and in ABC `vbb0deac` (Apr 3, 2013) [4]. The implementations of ctgDown differ from the pseudocode of Listing 2 in the following respects:

- In the IImc implementation, the consecution call in line 26 was implemented as a call to down. This change enables blocking a CTG at a (higher) level at which its negation is not inductive but contains an inductive subclause. The experiments are inconclusive with regards to which version is better.
- In the ABC implementation, the CTG cube is expanded through ternary simulation before it is checked for inductiveness (line 18) [8].

ABC's standard implementation of IC3 does not employ down in its generalization procedure; in particular, it never joins. However, the implementation of ctgDown includes joining. Experiments (whose details are not reported here) show that a variant of ctgDown in which joining was disabled is inferior to full ctgDown.

Hence, experiments with IImc compare the effects of ctgDown against down, while experiments with ABC compare the effects of ctgDown against ABC's generalization procedure.

The following parameter values were used in the experiments for both implementations of ctgDown: maxDepth $= 1$ and maxCTGs $= 3$[2].

The benchmark suite was gathered from the HWMCC 2010–2012 benchmarks—with one exception. Backward-encoded BEEM models (distinguished by the names `beem*ibj`) were replaced with their corresponding "functional" versions, also available from [5]. The backward encoding of these models involves two features[3]:

1) Serial exists-step transition relation [9]: this feature adds "shortcut" transitions to the state graph.
2) Reverse relational encoding: the transition relation is inverted, and the initial states are swapped with the bad states. The latch updates are directly from primary inputs, and a `valid` bit is added to track whether a state is backward reachable in the original design.

[1] While there are models for which this assumption is invalid, the fraction of state bits of a large digital system that changes at each clock cycle is often less than one tenth. This fraction supports the view that similarity between states decreases with their distance in the state graph.

[2] Generally, small values for maxCTGs (2–5) gave the best performance. For higher values, IC3 tended to derive too many clauses.

[3] See http://fmv.jku.at/aiger/README.beemaigs for details.

IImc has a "reverse" option to invert the transition relation and exchange the initial and error states. With this option, IImc typically works better on reverse-encoded models and worse on forward-encoded ones. A possible explanation is that a clause is a natural logical means of describing a design intention; moreover, conjunctions of clauses capture local arguments. In contrast, disjunctions of cubes—which is what IC3 produces from the forward perspective on reverse-encoded designs—do not. With both the functional and the backward encodings of these models available, one would never choose to use the backward encoding with IC3. Conclusions drawn from data based on such benchmarks are misguided.

As a preprocessing step, IImc's `sr` simplification tactic was applied to each benchmark. Then, IImc and ABC with and without the new generalization procedure were run on the simplified benchmarks only invoking their IC3 engines. No other features of IImc or ABC—e.g., multi-threading, other proof engines, or more powerful simplification techniques—were used. Each benchmark was run for up to 900 seconds. To account for variability, each benchmark was run five times with different random seeds. The experiments were performed on two identical machines with four 2.80 GHz Intel cores and 9 GBs of memory. The full results can be found at http://vlsi.colorado.edu/fmcad13.

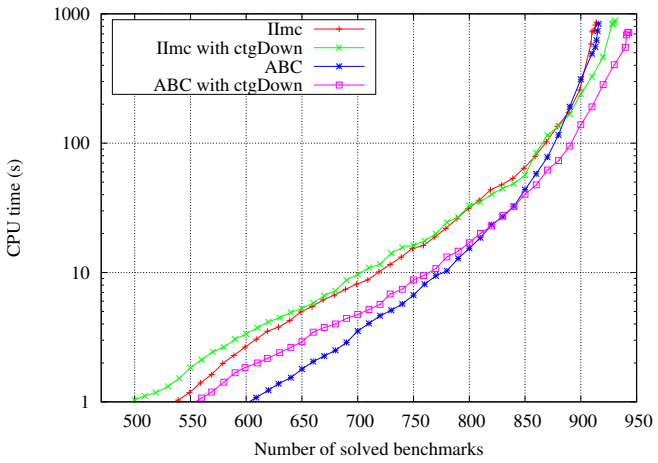


Fig. 2. Cactus plot comparing the performance of IImc and ABC with and without ctgDown.

A comparison between the performance of IC3 with and without ctgDown is presented in Figures 2–4. Figure 2 shows cactus plots for IImc and ABC and Figures 3 and 4 show scatter plots. All the plots use the results of the median runs.

For the easier models, the use of ctgDown does not typically reduce CPU time (Figures 3 and 4). An exception is the effect on the run times of failing properties with IImc.

Detailed results by benchmark family are presented in Table I. Benchmark families with at least 60 benchmarks are listed separately. The remaining benchmarks are included in the "other" category. The "Solved" columns show the minimum, median, and maximum number of solved instances over the five runs. The "Time" columns reports the median CPU time in

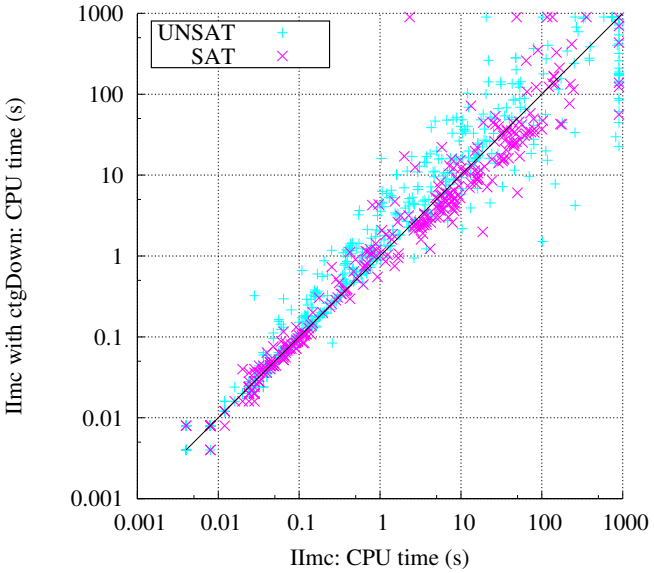


Fig. 3. IImc scatter plot

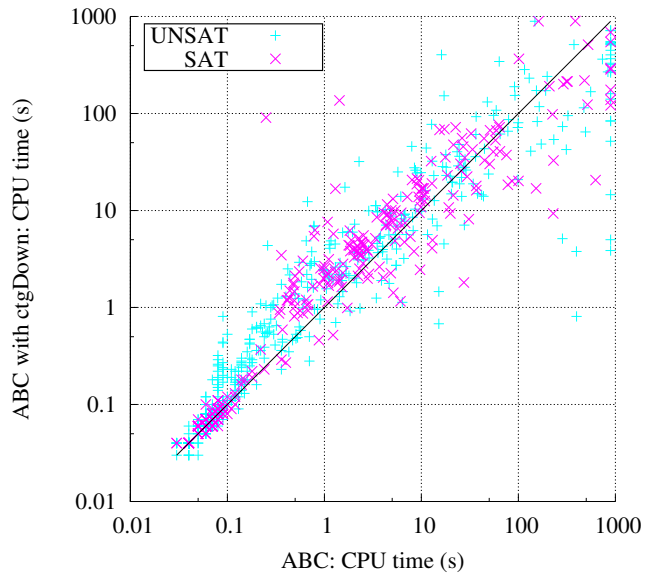


Fig. 4. ABC scatter plot

seconds. Overall, IImc and ABC with ctgDown solve 17 and 27 more instances, respectively, than their standard counterparts[4]. The same trend was observed when the timeout was increased to one hour: IImc and ABC solved 17 and 24 more instances respectively.

V. Analysis of IC3's Behavior

An observed weakness in IC3 with down is that on some models, it handles long chains of states explicitly rather than symbolically. ctgDown is intended to address this weakness

[4]Since the median is used, the sum of the gains for the individual families is not necessarily equal to the overall gain.

TABLE I
DETAILED RESULTS BY BENCHMARK FAMILY.

		IImc					**ABC**				
		Standard		With ctgDown			Standard		With ctgDown		
Family	Size	Solved	Time (s)	Solved	Gain	Time (s)	Solved	Time (s)	Solved	Gain	Time (s)
139	99	98/99/99	2524	99/99/99	0	1230	99/99/99	701	99/99/99	0	754
6s	120	18/19/22	93466	19/21/22	2	94211	19/23/24	88401	27/30/31	7	82941
beem	86	46/48/49	38149	47/50/51	2	39594	50/51/53	34098	54/56/57	5	31191
bob	149	121/122/125	25804	120/120/122	*(2)*	28679	122/123/124	24292	122/124/127	1	24083
intel	60	22/23/23	35004	29/30/31	7	31153	23/23/23	35665	25/26/27	3	34249
pdt	350	330/331/332	19291	336/336/337	5	15469	327/329/329	22162	333/333/333	4	18120
other	280	270/271/272	11947	272/274/275	3	11463	269/270/271	12591	272/274/274	4	10359
Total	1144	910/913/917	226790	924/930/932	**17**	222460	914/916/919	218906	936/943/944	**27**	201417

by accelerating IC3's exploration of deep backward reachable states while still maintaining its characteristic focus on the property. It attempts to achieve this objective by considering CTGs. As discussed, CTGs interfere with generalizing from CTIs and so are worthwhile candidates for blocking with generalization—although they need not be backward reachable. This section presents an analysis that, through measuring several metrics, suggests that ctgDown achieves its intended behavior. It highlights differences in the behavior of IC3 with the standard (down) and improved (ctgDown) generalization procedures. The data in this section were collected from IImc's IC3 runs. Data collected from ABC's runs also support the observations made.

Data points for scatter plots in this section are divided into two categories: those for which IC3 performs better with ctgDown, marked by a $\times$ in the plots, and those for which IC3 performs better with down, marked by a $+$.

The first experiment compares the average distances of CTGs and CTIs from an error. To measure the depths of CTGs, exact BDD-based backward reachability is performed; the resulting "onion rings" can be used to compute the depth of a given state. Of the CTGs handled in these experiments, 42% were backward reachable. For the depths of the CTIs, the length of the chain through which a CTI was found provides an upper bound on its actual backward depth. Figure 5 shows a plot for the average CTI depth against the average CTG depth for the 294 benchmarks for which the preliminary BDD-analysis managed to complete within 12 hours. The plot confirms that CTGs are typically deeper than CTIs—sometimes by several orders of magnitude. The plot also indicates that ctgDown helps in the cases where IC3 is forced to explore deep CTIs.

Next, several metrics of IC3 runs were analyzed to understand when the proposed generalization procedure helps or harms the performance of IC3. The metrics are the maximum length of traces from states in the priority queue to an error; the average size of derived clauses; the convergence level, i.e., the level at which a proof or a counterexample is found; and the average number of clauses derived per level.

Plots comparing IC3 with and without the proposed generalization procedure on the four metrics are shown in Figures 6a–6d. The same information is presented with box-and-whisker plots in Figure 6e with the ratio of each metric with ctgDown

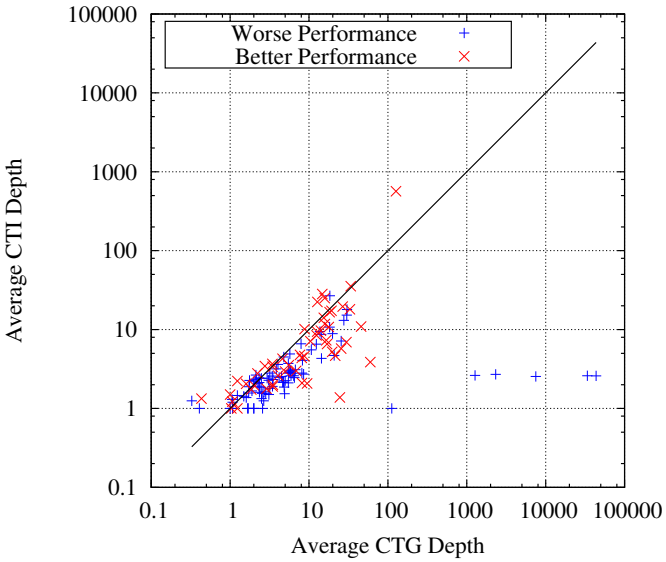


Fig. 5. A comparison between the depths of CTIs and CTGs.

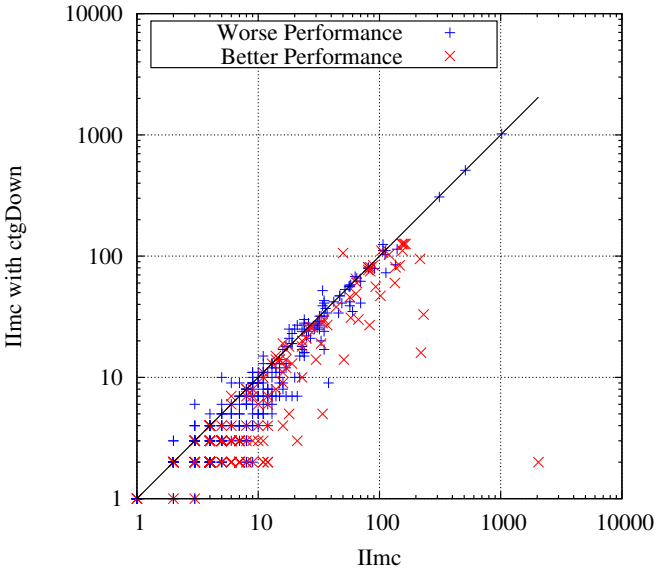
to without.

Figure 6a indicates a significant reduction in the depth of the explicit search performed by IC3 when ctgDown is used. Statistics indicate an average reduction of 22.3% in the depth of IC3's explicit search over all benchmarks. A higher reduction in the depth of the search often indicates better performance for IC3. This is confirmed by the non-overlapping notches in the box plot, which indicate a significant difference in the median depth ratios between cases with better and those with worse performance.

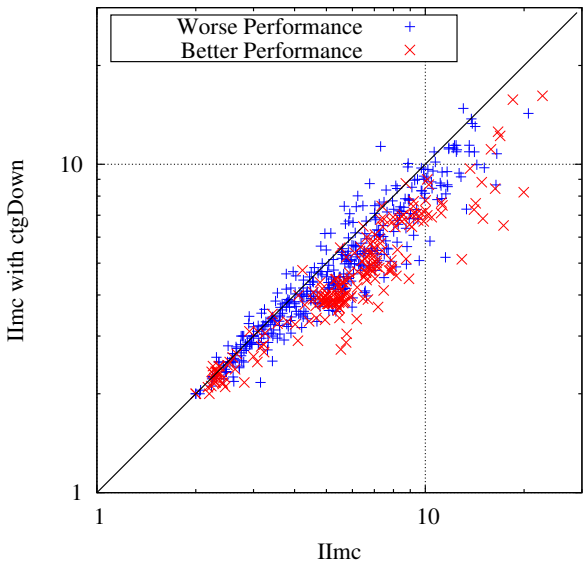
The point in the lower right corner of Figure 6a represents an extreme case in which IC3 with ctgDown proved the property with very little explicit backward search; with down, the depth of the priority queue went up to 2049.

Figure 6b points out ctgDown's ability to produce stronger CTI-induced clauses. Again, a stronger clause indicates improved performance. On average, ctgDown drops 14% more literals than down, which is statistically significant as indicated by the box plot.

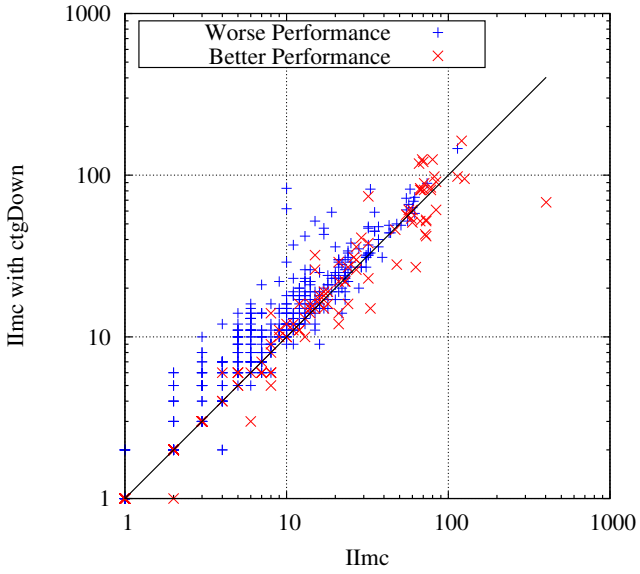
A characteristic of the new procedure is that it often increases the convergence level of IC3, as indicated in Figure 6c.



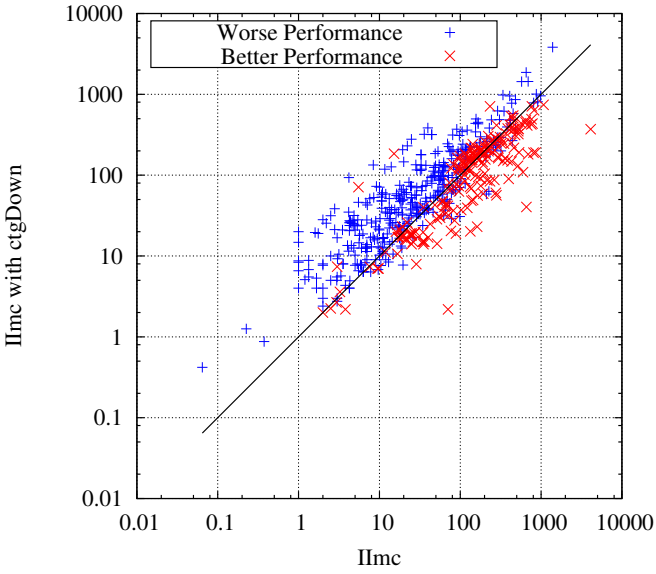
(a) Maximum depth of priority queue.



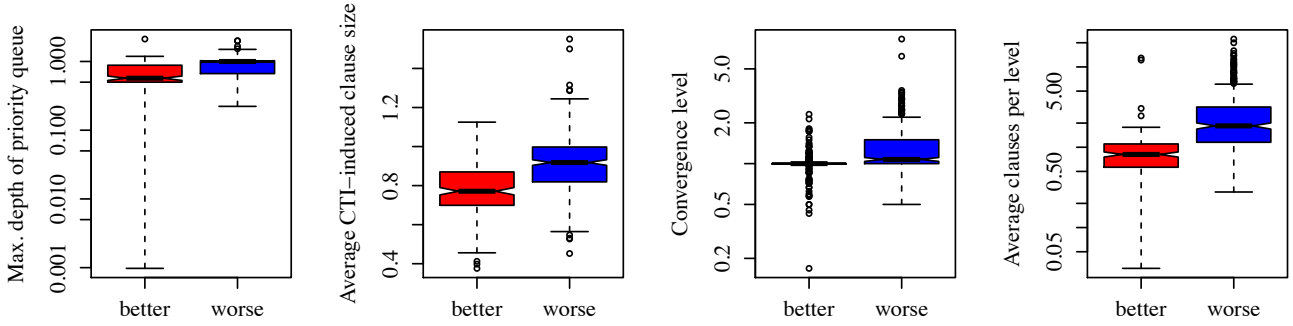
(b) Average CTI-induced clause size.



(c) Convergence level.



(d) Average clauses per level.



(e) Box plots for the ratios of the metrics shown in parts a–d.

Fig. 6. Analyzing the effects of ctgDown on the IImc runs.

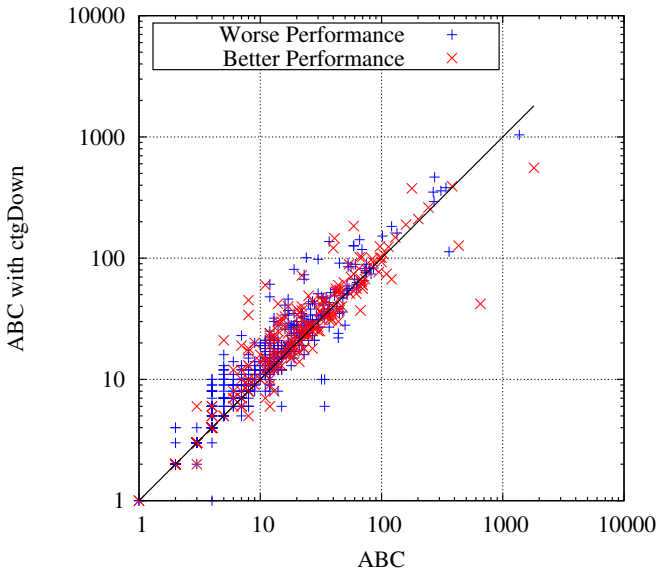


Fig. 7. Convergence level.

This potentially undesirable side effect is probably attributable to the aggressiveness of ctgDown in deriving clauses to block CTGs—which, again, need not actually be backward reachable. In contrast, the standard procedure only derives clauses in response to truly backward reachable states. A clause that blocks a forward reachable state is certainly not inductive and thus cannot appear in the final inductive strengthening. Such clauses can cause overstrengthening of the F_i's; then IC3 must propagate to higher levels in order to drop the clauses. Points to the far right in Figure 5 represent cases in which such behavior is exhibited. Although CTGs are much deeper than CTIs, the percentage of handled CTGs that are forward reachable is higher than average causing overstrengthening. Also, as Figure 6c shows, a higher convergence level is significantly correlated with worse performance. Similar observations hold for ABC with ctgDown as Figure 7 indicates. The box plot in Figure 6e shows that 75% of the runs in which ctgDown was beneficial did not increase the convergence level. In contrast, for 75% of the runs that did not benefit from ctgDown, the convergence level was higher. On the other hand, statistics indicate that the increase in convergence level only occurs for passing properties; for 75% of the failing properties, the convergence level isn't affected.

Points on the y-axis in Figure 6c correspond to benchmarks for which IC3 with down converges at level 1 while IC3 with ctgDown converges at higher levels. A characteristic behavior of IC3 with down is that clauses generated at level 1 are globally inductive until IC3 is forced to step back to level 0. Subsequently, generated clauses have the support of clauses generated relative to F_0 and thus need not be globally inductive. Aggressive handling of CTGs interferes with this initial behavior. A variant implementation was tried in which CTG handling was disabled until IC3 was forced to step back to level 0. IC3 with this variant ctgDown then converged at level 1 on these benchmarks; however, the performance difference across the benchmark suite was insignificant.

Finally, Figure 6d and the corresponding box plot indicate a clear correlation between the performance difference and the average number of clauses derived per level. An excessive number of clauses derived to block CTGs is often accompanied by longer runtimes.

VI. Related Work

Several improvements orthogonal to the generalization method presented here have been described for IC3. Ternary simulation [8] and SAT-based [10] methods of enlarging CTI cubes significantly improve running time. A scheme for integrating lazy abstraction with IC3 has also been developed [11].

VII. Conclusion

This paper presents an improved generalization procedure for IC3. Generalization is a key operation that lifts IC3 from explicit to symbolic analysis. Addressing states that impede generalization allows IC3 to deal with deep counterexamples to induction with less effort. The proposed procedure has been shown to significantly improve the performance of two independent implementations of IC3. While ctgDown achieves the objective of decreasing the depth of the explicit search, the impact on convergence level is mixed. Ongoing investigations seek to explain the interplay between the strength of lemmas, the convergence level, and the overall performance of IC3.

Acknowledgment

This work utilized the Janus supercomputer, which is supported by the National Science Foundation (award number CNS-0821794) and the University of Colorado Boulder. The Janus supercomputer is a joint effort of the University of Colorado Boulder, the University of Colorado Denver and the National Center for Atmospheric Research. Janus is operated by the University of Colorado Boulder.

References

[1] A. R. Bradley, "k-step relative inductive generalization," CU Boulder, Tech. Rep., Mar. 2010, http://arxiv.org/abs/1003.3649.
[2] ——, "SAT-based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation (VMCAI'11)*, Austin, TX, Jan. 2011, pp. 70–87, lNCS 6538.
[3] "URL: http://iimc.colorado.edu."
[4] "URL: http://www.eecs.berkeley.edu/~alanmi/abc/."
[5] "Hardware model checking competition. http://fmv.jku.at/hwmcc."
[6] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.
[7] A. R. Bradley and Z. Manna, "Checking safety by inductive generalization of counterexamples to induction," in *Formal Methods in Computer Aided Design (FMCAD'07)*, Austin, TX, Nov. 2007, pp. 173–180.
[8] N. Een, A. Mishchenko, and R. Brayton, "Efficient implementation of property-directed reachability," in *Formal Methods in Computer Aided Design (FMCAD'11)*, Austin, TX, Nov. 2011, pp. 125–134.
[9] J. Dubrovin, T. Junttila, and K. Heljanko, "Exploiting step semantics for efficient bounded model checking of asynchronous systems," *Science of Computer Programming*, vol. 77, no. 10-11, pp. 1095–1121, 2012.
[10] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo, "Incremental formal verification of hardware," in *Formal Methods in Computer Aided Design (FMCAD'11)*, Austin, TX, Nov. 2011, pp. 135–143.
[11] Y. Vizel, O. Grumberg, and S. Shoham, "Lazy abstraction and SAT-based reachability for hardware model checking," in *Formal Methods in Computer-Aided Design (FMCAD'12)*, Oct. 2012.