Trading-off incrementality and dynamic restart of multiple solvers in IC3

G. Cabodi (*), A. Mishchenko (**), M. Palena (*)
(*) Dip. di Automatica ed Informatica
Politecnico di Torino - Torino, Italy
(**) Dept. of EECS, University of California, Berkeley, CA, USA

***Abstract*—This paper[1] addresses the problem of SAT solver performance in IC3, one of the major recent breakthroughs in Model Checking algorithms. Unlike other Bounded and Unbounded Model Checking algorithms, IC3 is characterized by numerous SAT solver queries on small sets of problem clauses. Besides algorithmic issues, the above scenario poses serious performance challenges for SAT solver configuration and tuning. As well known in other application fields, finding a good compromise between learning and overhead is key to performance. We address solver cleanup and restart heuristics, as well as clause database minimality, based on on-demand clause loading: transition relation clauses are loaded in solver based on structural dependency and phase analysis. We also compare different solutions for multiple specialized solvers, and we provide an experimental evaluation on benchmarks from the HWMCC suite. Though not finding a clear winner, the work outlines several potential improvements for a portfolio-based verification tool with multiple engines and tunings.**

I. Introduction

IC3 [1] is a SAT-based invariant verification algorithm for bit-level Unbounded Model Checking (UMC). Since its introduction, IC3 has immediately generated strong interest, and is now considered one of the major recent breakthroughs in Model Checking. IC3 proved to be impressively effective on solving industrial verification problems. Our experience with the algorithm shows that IC3 is the single invariant verification algorithm capable of solving the largest number of instances among the benchmarks of the last editions of the Hardware Model Checking Competition (HWMCC).

A. Motivations

IC3 heavily relies on SAT solvers to drive several parts of the verification algorithm: a typical run of IC3 is characterized by a huge amount of SAT queries. As stated by Bradley in [2], the queries posed by IC3 to SAT solvers differ significantly in character from those posed by other SAT-based invariant verification algorithms (such as Bounded Model Checking [3], k-induction [4] [5] or interpolation [6]). Most notably, SAT queries posed by IC3 don't involve the unrolling of the transition relation for more than one step and are thus characterized by small-sized formulas.

IC3 can be thought as composed of two different layers: at the top level, the algorithm itself drives the verification process by constantly refining a set of over-approximations to forward reachable states with new inductive clauses; at the bottom level, a SAT solving framework is exploited by the top-level algorithm to respond to queries about the system. As shown in [7], these two layers can be separated by means of a clean interface.

Performance of IC3 turns out to be both highly sensitive to the various internal behaviours of SAT solvers, and strictly dependent on the way the top-level algorithm is integrated with the underlying SAT solving framework.

The peculiar characteristics exposed by the SAT queries of IC3 can thus be exploited to improve the overall performance of the algorithm in two different manners:

1) Tuning the internal behaviours of the particular SAT solver employed to better fit IC3 needs.
2) Defining better strategies to manage the SAT solving work required by IC3.

In this paper we address this second issue, proposing and comparing different implementation strategies for handling SAT queries in IC3. The aim of this paper is to identify the most efficient way to manage SAT solving in IC3. To achieve this goal we experimentally compare a number of different implementation strategies over a selected set of benchmarks from the recent HWMCC.

The experimental work has been done by two different research groups, on two different state-of-the-art verification tools, ABC [8] and PdTRAV [9], that share similar architectural and algorithmnic choices in their implementation of IC3.

The focus of this paper is neither on the IC3 algorithm itself nor on the internal details of the SAT solving procedures employed, but rather on the implementation details of the integration between IC3 and the underlying SAT solving framework.

B. Contributions

The main contributions of this paper are:

- A characterization of SAT queries posed by IC3.
- Novel approaches to solver allocation, loading and clean up in IC3.
- An experimental evaluation of performance using two verification tools.

[1]This work was supported in part by SRC Contracts No. 2012-TJ-2328 and No. 2265.001

C. Outline

First in Section II we introduce the notation used and give some background on IC3. Then, in Section III we present a systematic characterization of the SAT solving work required by IC3. Section IV introduces the problem of handling SAT queries posed by IC3 efficiently. Both commonly used and novel approaches to the allocation, loading and cleaning up of SAT solvers in IC3 are discussed in Sections V, VI and VII respectively. Experimental data comparing these approaches are presented in Section VIII. Finally, in Section IX we draw some conclusions and give summarizing remarks.

II. Background

A. Notation

Definition 1. *A transition system is the triple $S = \langle \mathbf{M}, I, T \rangle$ where M is a set of boolean variables called state variables of the system, $I(M)$ is a boolean predicate over M representing the set of initial states of the system and $T(M, M')$ is a predicate over M, M' that represents the transition relation of the system.*

Definition 2. *A state of the system is represented by a complete assignment s to the state variables M. A set of states of the system is represented by a boolean predicate over M. Given a boolean predicate F over M, a complete assignement s such that s satisfies F (i.e. $s \models F$) represents a state contained in F and is called an F-state. Primed state variables M' are used to represent future states and, accordingly, a boolean predicate over M' represent a set of future states.*

Definition 3. *A boolean predicate F is said to be stronger than another boolean predicate G if $F \rightarrow G$, i.e. every F-state is also a G-state.*

Definition 4. *A literal is a boolean variable or the negation of a boolean variable. A disjunction of literals is called a clause while a conjunction of literals is called a cube. A formula is said to be in Conjunctive Normal Form (CNF) if it is a conjunction of clauses.*

Definition 5. *Given a transition system $S = \langle \mathbf{M}, I, T \rangle$, if an assignment s, t' satisfies the transition relation T (i.e. if $s, t' \models T$) then s is said to be a predecessor of t and t is said to be a successor of s. A sequence of states $s_0, s_1, \ldots, s_n$ is said to be a path in S if every couple of adjacent states s_i, s_{i+1}, $i \leq 0 < n$ satisfies the transition relation (i.e. if $s_i, s'_{i+1} \models T$).*

Definition 6. *Given a transition system $S = \langle \mathbf{M}, I, T \rangle$, a state s is said to be reachable in S if there exists a path $s_0, s_1, \ldots, s$, such that s_0 is an initial state (i.e. $s_0 \models I$). We denote with $R_n(S)$ the set of states that are reachable in S in at most n steps. We denote with $R(S)$ the overall set of states that are reachable in S. Note that $R(S) = \bigcup_{i \geq 0} R_i(S)$.*

Definition 7. *Given a transition system $S = \langle \mathbf{M}, I, T \rangle$ and a boolean predicate P over M (called A safety property), the invariant verification problem is the problem of determining if P holds for every reachable state in S: $\forall s \in R(S) : s \models P$. An algorithm used to solve the invariant verification problem is called an invariant verificatio algorithm.*

Definition 8. *Given a transition system $S = \langle \mathbf{M}, I, T \rangle$, a boolean predicate F over M is called an inductive invariant for S if the following two conditions hold:*

- *Base case: $I \rightarrow F$*
- *Inductive case: $F \wedge T \rightarrow F'$*

A boolean predicate F over M is called an inductive invariant for S relative to another boolean predicate G if the following two conditions hold:

- *Base case: $I \rightarrow F$*
- *Relative inductive case: $G \wedge F \wedge T \rightarrow F'$*

Lemma 1. *Given a transition system $S = \langle \mathbf{M}, I, T \rangle$, an inductive invariant F for S is an over-approximation to the set of reachable states $R(S)$.*

Definition 9. *Given a transition system $S = \langle \mathbf{M}, I, T \rangle$ and a boolean predicate P over M, an inductive strengthening of P for S is an inductive invariant F for S such that F is stronger than P.*

Lemma 2. *Given a transition system $S = \langle \mathbf{M}, I, T \rangle$ and a boolean predicate P over M, if an inductive strengthening of P can be found, then the property P holds for every reachable state of S. The invariant verification problem can be seen as the problem to find an inductive strengthening of P for S.*

B. IC3

Given a transition system $S = \langle \mathbf{M}, I, T \rangle$ and a safety property P over $\mathbf{M}$, IC3 aims to find an inductive strengthening of P for S. For this purpose, it maintains a sequence of formulas $\mathbf{F_k} = F_0, F_1, \ldots F_k$ such that, for every $0 \leq i < k$, F_i is an over-approximation of the set of states reachable in at most i steps in S. Each of these over-approximations is called a *time frame* and is represented by a set of clauses, denoted by $\mathrm{clauses}(F_i)$. The sequence of time frames $\mathbf{F_k}$ is called *trace* and is maintained by IC3 in such a way that the following conditions hold throughout the algorithm:

(1) $F_0 = I$
(2) $F_i \rightarrow F_{i+1}$, for all $0 \leq i < k$
(3) $F_i \wedge T \rightarrow F'_{i+1}$, for all $0 \leq i < k$
(4) $F_i \rightarrow P$, for all $0 \leq i < k$

Condition (1) states that the first time frame of the trace is special and is simply assigned to the set of initial states of S. The remaining conditions, claim that for every time frame F_i but the last one: (2) every F_i-state is also a F_{i+1}-state, (3) every successor of an F_i-state is an F_{i+1}-state and (4) every F_i-state is safe. Condition (2) is maintained syntactically, enforcing the condition (2') $\mathrm{clauses}(F_{i+1}) \subseteq \mathrm{clauses}(F_i)$.

Lemma 3. *Let $S = \langle \mathbf{M}, I, T \rangle$ be a transition system, $\mathbf{F_k} = F_0, F_1, \ldots F_k$ a sequence of boolean formulas over $\mathbf{M}$ and let conditions (1-3) hold for $\mathbf{F_k}$. Then each F_i, with $0 \leq i < k$, is an over-approximation to the set of states reachable within i steps in S.*

Lemma 4. *Let $S = \langle \mathbf{M}, I, T \rangle$ be a transition system, P a safety property over $\mathbf{M}$, $\mathbf{F_k} = F_0, F_1, \ldots F_k$ a sequence of boolean formulas over $\mathbf{M}$ and let conditions (1-4) hold for $\mathbf{F_k}$. Then P is satisfied up to $k-1$ steps in S (i.e. there doesn't exist any counter-example to P of length less or equal than $k-1$).*

The main procedure of IC3 is described in Algorithm 1 and is composed of two nested iterations. Major iterations (lines 3-16) try to prove that P is satisfied up to k steps in S, for increasing values of k. To prove so, in minor iterations (lines 4-9), IC3 refines the trace $\mathbf{F_k}$ computed so far, by adding new relative inductive clauses to some of its time frames. The algorithm iterates until either (i) an inductive strengthening of the property is produced (line 4), or (ii) a counter-example to the property is found (line 7).

Input: $S = \langle \mathbf{M}, I, T \rangle ; P(\mathbf{M})$
Output: *SUCCESS or FAIL(σ), with σ counter-example*
1: $k \leftarrow 0$
2: $F_k \leftarrow I$
3: **repeat**
4: **while** $\exists t : t \models F_k \wedge \neg P$ **do**
5: $s \leftarrow$ *Extend(t)*
6: **if** *BlockCube*(s, Q, $\boldsymbol{F_k}$) = FAIL(σ) **then**
7: **return** FAIL(σ)
8: **end if**
9: **end while**
10: $F_{k+1} \leftarrow \emptyset$
11: $k \leftarrow k + 1$
12: $\boldsymbol{F_k} \leftarrow$ *Propagate*($\boldsymbol{F_k}$)
13: **if** $F_i = F_{i+1}$ *for some* $0 \leq i < k$ **then**
14: **return** SUCCESS
15: **end if**
16: **until** forever

Algorithm 1. *IC3(S, P)*

At major iteration k, the algorithm has computed a trace $\mathbf{F_k}$ such that conditions (1-4) hold. From Lemma 4, it follows that P is satisfied up to $k-1$ steps in S. IC3 then tries to prove that P is satisfied up to k steps as well. This is done by enumerating F_k-states that violate P and then trying to block them in F_k.

Definition 10. *Blocking a state (or, more generally, a cube) s in a time frame F_k means proving s unreachable within k steps in S, and consequently refine F_k to exclude it.*

To enumerate each state of F_k that violates P (line 4), the algorithm poses SAT queries to the underlying SAT solving framework in the following form:

$$\text{SAT ?}(F_k \wedge \neg P) \qquad (Q_1)$$

If Q_1 is SAT, a bad state t in F_k can be extracted from the satisfying assignment. That state must be blocked in F_k. To increase performance of the algorithm, as suggested in [7], the bad state t generated this way is first (possibly) extended to a bad cube s. This is done by means of the *Extend*(t) procedure (line 5), not reported here, that employs ternary simulation to remove some literals from t. The resulting cube s includes t and violates the property P, it is thus a *bad cube*. The algorithm then tries to block the whole bad cube s rather than t. It is showed in [7] that extending bad states into bad cubes before blocking them dramatically improves IC3 performance.

Once a bad cube s is found, it is blocked in F_k calling the *BlockCube*(s, Q, $\boldsymbol{F_k}$) procedure (line 6). This procedure is described in Algorithm 2.

Input: *s: bad cube in F_k; Q: priority queue; $\boldsymbol{F_k}$: trace*
Output: *SUCCESS or FAIL(σ), with σ counter-example*
1: add a proof-obligation (s, k) to the queue Q
2: **while** Q is not empty **do**
3: extract (s, j) with minimal j from Q
4: **if** $j > k$ *or* $t \not\models F_j$ **then continue**;
5: **if** $j = 0$ **then return** FAIL(σ)
6: **if** $\exists t, v' : t, v' \models F_{j-1} \wedge T \wedge \neg s \wedge s'$ **then**
7: $p \leftarrow$ *Extend(t)*
8: add $(p, j-1)$ and (s, j) to Q
9: **else**
10: $c \leftarrow$ *Generalize*($j, s, \boldsymbol{F_k}$)
11: $F_i \leftarrow F_i \cup c$ for $0 < i \leq j$
12: add $(j+1, c)$ to Q
13: **end if**
14: **end while**
15: **return** SUCCESS

Algorithm 2. *BlockCube(s, Q, $\boldsymbol{F_k}$)*

Otherwise, if Q_1 is UNSAT, every bad state of F_k has been blocked so far, conditions (1-4) hold for $k+1$ and IC3 can safely move to the next major iteration, trying to prove that P is satisfied up to $k+1$ steps. Before moving to the next iteration, a new empty time frame F_{k+1} is created (line 10). Initially, $\text{clauses}(F_{k+1}) = \emptyset$ and such time frame represent the entire state space, i.e. $F_{k+1} = \text{Space}(S)$. Note that $\text{Space}(S)$ is a valid over-approximation to the set of states reachable within $k+1$ steps in S. Then a phase called *propagation* takes place (line 12). The procedure *Propagate*($\boldsymbol{F_k}$) (Algorithm 4), which is discussed later, handles that phase. During propagation, IC3 tries to refine every time frame F_i, with $i < 0 \leq k$, by checking if some clauses of one time frame can be pushed forward to the following time frame. Possibly, propagation refines the outmost timeframe F_k so that $F_k \subset \text{Space}(S)$. Propagation can lead to two adjacent time frames becoming equivalent. If that happens, the algorithm has found an inductive strengthening of P S (equal to those time frames), so the property P holds for for every reachable state of S and IC3 return success (line 13).

The procedure *BlockCube*(s, Q, $\boldsymbol{F_k}$) (Algorithm 2) is responsible for refining the trace $\boldsymbol{F_k}$ in order to block a bad cube s found in F_k. To preserve condition (3), prior to blocking a cube in a certain time frame, IC3 has to recursively block its

predecessor states in the preceding time frames. To keep track of the states (or cubes) that must be blocked in certain time frames, IC3 uses the formalism of *proof-obligations*.

Definition 11. *Given a cube s and a time frame F_j, a proof-obligation is a couple (s, j) formalizing the fact that s must be blocked in F_j.*

Given a proof obligation (s, j), the cube s can either represent a set of bad states or a set of states that can all reach a bad state in one or more transitions. The number j indicates the position in the trace where s must be proved unreachable, or else the property fails.

Definition 12. *A proof obligation (s, j) is said to be discharged when s becomes blocked in F_j.*

IC3 maintains a priority queue Q of proof-obligations. During the blocking of a cube, proof-obligations (s, j) are extracted from Q and discharged for increasing values of j, ensuring that every predecessor of a bad cube s will be blocked in F_j ($j < k$) before s will be blocked in F_k. In the *BlockCube*(s, Q, $\boldsymbol{F_k}$) procedure, first a proof-obligation encoding the fact that s must be blocked in F_k is added to Q (line 1). Then proof-obligations are iteratively extracted from the queue and discharged (lines 2-14).

Prior to discharge the proof-obligation (s, j) extracted, IC3 checks if that proof-obligation still needs to be discharged. It is in fact possible for an enqueued proof-obligation to become discharged as a result of the discharging of some previously extracted proof-obligations. To perform this check, the following SAT query is posed (line 4):

$$SAT?(F_j \wedge s) \qquad (Q_2)$$

If the result of Q_2 is SAT, then the cube s is still in F_j and (s, j) still needs to be discharged. Otherwise, s has already been blocked in F_j and the procedure can move on to the next iteration.

If the proof-obligation (s, j) still needs to be discharged, then IC3 checks if the time frame identified by j is the initial time frame (line 5). If so, the states represented by s are initial states that can reach a violation of the property P. A counter-example σ to P can be constructed going up the chain of proof-obligations that led to $(s, 0)$. In that case, the procedure terminates with failure returning the counter-example found.

To discharge a proof-obligation (s, j), i.e. to block a cube s in F_j, IC3 tries to derive a clause c such that $c \subseteq \neg s$ and c is inductive relative to F_{j-1}. This is done by posing the following SAT query (line 6):

$$SAT?(F_{j-1} \wedge \neg s \wedge T \wedge s') \qquad (Q_3)$$

If Q_3 is UNSAT (lines 10-12), then the clause $\neg s$ is inductive relative to F_{j-1} and can be used to refine F_j, ruling out s. To pursue a stronger refinement of F_j, the inductive clause found undergoes a process called *inductive generalization* (line 10) prior to being added to F_i. Inductive generalization is carried out by the procedure *Generalize*($j, s, \boldsymbol{F_k}$) (Algorithm 3), which tries to minimize the number of literals in clause $c = \neg s$ while maintaining its inductiveness relative to F_{j-1}, in order to preserve condition (2). The resulting clause is added not only to F_j, but also to every time frame F_i, $0 < i < j$ (line 11). Doing so discharges the proof-obligation (s, j). In fact, this refinement rule out s from every F_i with $0 < i \leq j$. Since the sets F_i with $i > j$ are larger than F_j, s may still be present in one of them and $(s, j + 1)$ may become a new proof-obligation. To address this issue, Algorithm 2 adds $(s, j + 1)$ to the priority queue (line 12).

Otherwise, if Q_3 is SAT (lines 7-8), a predecessor t of s in F_{j-1} can be extracted from the satisfying assignment. To preserve condition (3), before blocking a cube s in a time frame F_j, every predecessor of s must be blocked in F_{j-1}. So, the predecessor t is extended with ternary simulation (line 7) into the cube p, and then both proof-obligations $(p, j - 1)$ and (s, j) are added to the queue (line 8).

Input: *j: time frame index; s: cube such that $\neg s$ is inductive relative to F_{j-1}; $\boldsymbol{F_k}$: trace*
Output: *c : a sub-clause of $\neg s$*

```
1:  c ← ¬s
2:  for all literals l in c do
3:      try ← the clause obtained by deleting l from c
4:      if ∄t, v' : t, v' ⊨ F_{j-1} ∧ T ∧ try ∧ ¬try' then
5:          if ∄t ⊨ I ∧ ¬try then
6:              c ← try
7:          end if
8:      end if
9:  end for
10: return c
```

Algorithm 3. *Generalize($j, s, \boldsymbol{F_k}$)*

The *Generalize*($j, s, \boldsymbol{F_k}$) procedure (Algorithm 3) tries to remove literals from $\neg s$ while keeping it relatively inductive to F_{j-1}. To do so, a clause c intialized with $\neg s$ (line 1) is used to represent the current minimal inductive clause. For every literal of c, the clause try obtained by dropping that literal from c (line 3), is checked for inductiveness relative to F_{j-1} by means of the following SAT query (line 4):

$$SAT?(F_{j-1} \wedge try \wedge T \wedge \neg try') \qquad (Q_4)$$

If Q_4 is UNSAT, the iductive case for the reduced formula still holds. Since dropping literals from a relatively inductive clause can break both the inductive case and the base case, the latter must be explicilty checked too for the reduced clause try (line 5). This is done by posing the following SAT query:

$$SAT?(I \wedge \neg try) \qquad (Q_5)$$

If both the inductive case and the base case hold for the reduced clause try, the currently minimal inductive clause c is updated with try (line 6).

The *Propagate*($\boldsymbol{F_k}$) procedure (Algorithm 4) handles the propagation phase. For every clause c of each time frame F_j, with $0 \leq j < k - 1$, the procedure checks if c can be pushed

Input: $\boldsymbol{F_k}$*: trace*
Output: $\boldsymbol{F_k}$*: updated trace*
1: **for** $j = 0$ to $k - 1$ **do**
2: **for all** $c \in F_j$ **do**
3: **if** $\exists t, v' : t, v' \models F_j \wedge T \wedge c'$ **then**
4: $F_{j+1} \leftarrow F_{j+1} \cup \{c\}$
5: **end if**
6: **end for**
7: **end for**
8: **return** $\boldsymbol{F_k}$

Algorithm 4. *Propagate(*$\boldsymbol{F_k}$*)*

forward to F_{j+1} (line 3). To do so, it poses the following SAT query:

$$SAT?(F_j \wedge T \wedge c') \qquad (Q_6)$$

If the result of Q_6 is SAT, then it is safe, with respect to condition (3), to push clause c forward to F_{i+1}. Otherwise, c can't be pushed and the procedure iterates to the next clause.

C. Related works

In [2], Aaron Bradley outlined the opportunity for SAT and SMT researchers to directly address the problem of improving IC3's performance by exploiting the peculiar character of the SAT queries it poses. A description of the IC3 algorithm, specifically addressing implementation issues, is given in [7].

III. SAT SOLVING IN IC3

SAT queries posed by IC3 have some specific characteristics:

- *Small-sized formulas*: they employ no more than a single instance of the transition relation;
- *Large number of calls*: reasoning during the verification process is highly localized and takes place at relatively-small-cubes granularity;
- *Separated contexts*: each SAT query is relative to a single time frame;
- *Related calls*: subsequent calls may expose a certain correlation (for instance, inductive generalization calls take place on progressively reduced formulas).

We performed an analysis of the implementation of IC3 within the academic model checking suite PdTRAV, closely following the description of IC3 given in [7] (there called PDR: Property Directed Reachability). The experimental analysis lead us to identify six different types of SAT queries that the algorithm poses during its execution. These queries are the ones already outlined in Section II-B. The type of these queries is reported in Table I.

For each of the queries identified, we have measured the average number of calls and the average solving time. These statistics are reported in Table II. The results were collected by running PdTRAV's implementation of IC3 on the complete set of 310 single property benchmarks of the HWMCC'12, using time and memory limits of 900 s and 2 GB, respectively.

Such statistics can be summarized as follows:

Name	Query Type	Query
Q_1	Target Intersection	$SAT?(F_k \wedge \neg P)$
Q_2	Blocked Cube	$SAT?(F_i \wedge s)$
Q_3	Relative Induction	$SAT?(F_i \wedge \neg s \wedge T \wedge s')$
Q_4	Inductive Generalization	$SAT?(F_i \wedge c \wedge T \wedge \neg c')$
Q_5	Base of Induction	$SAT?(I \wedge \neg c)$
Q_6	Clause Propagation	$SAT?(F_i \wedge T \wedge \neg c')$

Table I: SAT Queries Breakdown in IC3

- SAT calls involved in inductive generalization are by far the most frequent ones. These are in fact the calls that appears at the finest granularity. In the worst case scenario, one call is posed for every literal of every inductive clause found.
- Inductive generalization and propagation checks are the most expensive queries in terms of average SAT solving time required.
- Target intersection calls are very infrequent and don't take much time to be solved.
- Blocked cube and relative induction checks are close in the number of calls and solving time.

Query	Calls		Avg Time
	[Number]	[%]	[ms]
Q_1	483	0.1	81
Q_2	27891	6.8	219
Q_3	31172	7.6	334
Q_4	142327	34.7	575
Q_5	147248	35.9	112
Q_6	61114	14.9	681

Table II: SAT queries statistics in IC3: Number of calls, percentage, and average time spent in different SAT queries during an IC3 run.

IV. HANDLE SAT SOLVING IN IC3

Subsequent SAT calls in IC3 are often applied to highly different formulas. In the general case, two subsequent calls can in fact be posed in the context of different time frames, thus involving different sets of clauses. In addition, one of them may require the use of the transition relation, while the other may not, and each query can involve the use of temporary clauses/cubes that are needed only to respond to that particular query (e.g. the candidate inductive clause used to check relative inductiveness during cube blocking or inductive generalization).

In the general case, whenever a new query is posed by IC3 to the underlying SAT solving framework, the formula currently loaded in the solver must be modified to accomodate the new query. For this reason, IC3 requires the use of SAT solvers that expose an incremental SAT interface. An incremental SAT interface for IC3 must support the following features:

- Pushing and popping clauses to or from the formula.

- Specifying literal assumptions.

Many state-of-the-art SAT solvers, like MiniSAT [10], feature an incremental interface capable of pushing new clauses into the formula and allowing literal assumptions. Removing clauses from the current formula is a more difficult task since one have to remove not only the single clause specified, but also every learned clause that has been derived from it. Although solvers such as *zchaff* [11] directly support clause removal, the majority of the state-of-the-art SAT solvers feature an interface like the one of MiniSAT, in which clause removal can only be simulated. This is done through the use of literal assumptions and the introduction of auxiliary variables known as *activation variables*, as described in [12]. In such solvers, clauses aren't actually removed from the formula but only made redundant for the purpose of determining the satisfiability of the rest of the formula. Since such clauses are not removed from the formula, they still participate in the Boolean Constraint Propagation (BCP) and, thus, degrade the overall SAT solving performance. In order to minimize this degradation, each solver employed by IC3 must be periodically cleaned up, i.e. emptied and re-loaded with only relevant clauses. In this work we assume the use of a SAT solver exposing an incremental interface similar to the one of MiniSAT.

Once an efficient incremental SAT solving tool has been chosen, any implementation of IC3 must face the problem of deciding how to integrate the top-level algorithm with the underlying SAT solving layer. Such problem can be divided into the following three sub-problems:

- *SAT solvers allocation*: decide how many different SAT solvers to employ and how to distribute workload among them.
- *SAT solvers loading*: decide which clauses must be loaded in each solver to make them capable of responding correctly to the SAT queries posed by IC3.
- *SAT solvers clean up*: decide when and how often the algorithm must clean up each solver, in order to avoid performance degradation.

V. SAT SOLVERS ALLOCATION

We assume in this work the use of multiple SAT solvers, one for each different time frame. Using multiple solvers, we observed that performance is highly related to:

- *Solver cleanup frequency*: cleaning up the solver means removal of incrementally loaded problem clauses and learned clauses
- *Clause loading strategy*: on-demand loading of transition relation clauses based on topological dependency

A. Specialized solvers

From the statistical results of reported in Table II it's easy to see that inductive generalization and clause propagation queries are by far the most expensive ones in terms of average SAT solving time. Inductive generalization queries, in addition of being expensive, are also the most frequent type of query posed.

The reason why inductive generalization calls are so expensive can be due to the fact that during the inductive generalization of a clause, at every iteration a slightly changing formula is queried for a satisfying assignment in increasingly larger subspaces. Given two subsequent queries in inductive generalization, in fact, it can be noticed that their formulas can differ only for one literal of the present state clause try and one literal of the next state cube $\neg try$. As the subspace becomes larger solving times for those queries increases. The average expensiveness of clause propagation calls can be intuitively motivated by noticing that they are posed one time for every clause of every time frame. The innermost time frames are the ones with the largest number of clauses, and thus require the largest number of propagation calls. Unfortunately, given the high number of clauses in those time frames, the CNF formulas upon which such calls act are highly constrained and usually harder to solve. So during propagation there are, in general, more hard queries than simple queries, making the average SAT solving time for those queries high.

In an attempt to reduce the burden of each time frame's SAT solver, we have experimented the use of specialized solvers for handling such queries. For every time frame, a second solver is instantiated and used to respond a particular type of query (Q_4 or Q_6). Table III summarize the results of such experiment.

VI. SOLVER LOADING

To minimize the size of the formula to be loaded into each solver, a common approach is to load, for every SAT call that queries the dynamic behavior of the system, only the necessary part of the transition relation.

It is easy to observe that every SAT call that uses the transition relation involves a constraint on the next state variables of the system in the form of a cube c'. Such queries ask if there is a state of the system satisfying some constraints on the present state, which can reach in one transition states represented by c'. Since c' is a cube, the desired state must have a successor p such that p correspond to c' for the value of every variable of c'. It's easy to see that the only present state variables that are relevant in determining if such a state exists, are those in the structural support of the next state variables of c'. It follows that the only portions of the transition relation required to answer such queries are the logic cones of the next state variables of c'.

Such loading strategy, known as *lazy loading of transition relation*, is commonly employed in various implementations of IC3, as the ones of PdTRAV and ABC. We observed in average 50% reduction in the size of the CNF formula for the transition relation using lazy loading of transition relation.

We noticed that, for these queries, the portions of the transition relation loaded can be further minimized employing a CNF encoding technique, called Plaisted-Greenbaum CNF encoding [13] (henceforth simply called PG encoding). The AIG representation of the transition relation together with the assumptions specified by the next state cube c' can be viewed as a *constrained boolean circuit* [14], i.e. a boolean circuit in which some of the outputs are constrained to assume

certain values. The Plaisted-Greenbaum encoding is a special encoding that can be applied in the translation of a constrained boolean circuit into a CNF formula.

Below we give an informal description of the PG encoding. For a complete description refer to [13] or [14].

Given an AIG representation of a circuit, a CNF encoding first subdivides that circuit into a set of functional blocks, i.e. gates or group of connected gates representing certain boolean functions, and introduces a new CNF variable a for each of these blocks. For each functional block representing a function f on the input variables $x_1, x_2, \ldots, x_n$, a set of clauses is derived translating into CNF the formula:

$$a \leftrightarrow f(x_1, x_2, \ldots, x_n) \tag{1}$$

The final CNF formula is obtained by the conjunction of these sets of clauses. Different CNF encodings differ in the way the gates are grouped together to form functional blocks and in the way these blocks are translated into clauses. The idea of PG encoding is to start from a base CNF encoding, and then use both output assumptions and topological information of the circuit to get rid of some of the derived clauses, while still producing an equi-satisfiable CNF formula. Based on the output constraints and the number of negations that can be found in every path from a gate to an output node, it can be shown that, for some gates of the circuit, an equi-satisfiable encoding can be produced by only translating one of the two sides of the bi-implication (1). The CNF formula produced by PG encoding will be a subset of the one produced by the traditional encoding.

PG encoding proves to be effective in reducing the size of loaded formulas, but it is not certain whether it is more efficient for SAT solving, since it may have worst propagation behaviour [15].

We observed a 10-20% reduction in the size of the CNF formula for the transition relation.

VII. Solvers clean up

A natural question arises regarding how frequently and at what conditions SAT solvers cleanups should be scheduled. Cleaning up a SAT solver, in fact, introduces a certain overhead. This is because:

- Relevant clauses for the current query must be reloaded.
- Relevant inferences previously derived must be derived again from those clauses.

A heuristic cleanup strategy is needed in order to achieve a trade-off between the overhead introduced and the slowdown in BCP avoided. The purpose of that heuristic is to determine when the number of irrelevant clauses (w.r.t. the current query) loaded in a solver becomes large enough to justify a cleanup. To do so, a heuristic measure representing an estimate of the number of currently loaded irrelevant clauses is compared to a certain threshold. If that measure exceeds the threshold, then the solver is cleaned up.

Clean up heuristics currently used in state-of-the-art tools, like ABC and PdTRAV, rely on loose estimates of the size of irrelevant portions of the formula loaded into each solver. These heuristics clean up each solver as soon as the computed estimate meets some, often static, threshold.

Our experiments with IC3 prove that the frequency of the cleanups play a crucial role in determining the overall verification performance. We explored the use of new cleanup heuristics based on more precise measures of the number of irrelevant clauses loaded and able to exploit correlation among different SAT calls to dynamically adjust the frequency of cleanups.

For SAT calls in IC3, notice that there are two sources of irrelevant clauses loaded in a solver:

1) Deactivated clauses loaded for previous inductive checks (Q_3 and Q_4 queries).
2) Portions of logic cones loaded for previous calls querying the dynamic behavour of the system.

Cleanup heuristics commonly used, such as the ones used in baseline versions of ABC and PdTRAV, typically take into account only the number of deactivated clauses in the solver to compute an estimate of the number of irrelevant clauses. We investigated the use of a new heuristic measure taking into account the second source of irrelevant clauses, i.e. irrelevant portions of previously loaded cones.

Every time a new query requires to load a new portion of the transition relation, to compute a measure of the number of the irrelevant transition relation's clauses, the following quantities are computed (assuming that c' is the cube constraining the next state variables for that query):

- A: the number of transition relation clauses already loaded into the solver (before loading the logic cones required by the current query);
- $S(c')$: the size (number of clauses) of the logic cones required for solving the current query;
- $L(c')$: the number of clauses in the required logic cones to be loaded into the solver (equal to the size of the required logic cone minus the number of clauses that such cone shares with previously loaded cones);

A measure of the number of irrelevant transition relation's clauses loaded for c', denoted by $u(c')$, can be computed as follows:

$$u(c') = A - (S(c') - L(c')) \tag{2}$$

Such a heuristic measure, divided by the number of clauses loaded into the solver, indicates the percentage of irrelevant clauses loaded in the solver w.r.t the current query. In Section VIII we investigate the use new cleanup strategies based on this measure. In order to take into account correlation between subsequent SAT calls, we consider such measure averaged on a time window of the last SAT calls.

VIII. Experimental Results

We have compared different cleanup and clause loading heuristics in both PdTRAV and ABC. In this section, we briefly summarize the obtained results.

A. PG encoding

A first set of experiments was done on the full set of 310 HWMCC'12 benchmarks [16], with a lower timeout, of 900 seconds, in order to evaluate the use of PG encoding. Results were controversial. A run in PdTRAV, with 900 seconds timeout, showed a reduction in the number of solved instances from 79 to 63 (3 of which previously unsolved). The percentage reduction of loaded transition relation clauses was 21.32%.

A similar run on ABC, showed a more stable comparison, from 80 to 79 solved instances (3 of which previously unsolved). So a first conclusion is that, PG encoding was not able to win over the standard encoding, suggesting it can indeed suffer of worst propagation behaviour. Nonetheless, it was very interesting to observe that the overall number of solved problems grew from 79 to 82 in PdTRAV and from 80 to 85 in ABC.

Different results between the two tools in this experimentation could be partially due to different light-weight preprocessing done by default within them. We started a more detailed comparison, in order to better understand the partially controversial results.

B. Experiments with PdTRAV

We then focused on a subset of 70 selected circuits, for which preprocessing was done off-line, and the tools were run on the same problem instances. In the following tables, the $P0$ rows always represent the default setting of PdTRAV. We report number of solver instances (out of 70) and average execution time (time limit 900 seconds). Column labeled New, when present, shows the number of instances not solved by $P0$.

Configuration	Solved [#]	New [#]	Avg Time [s]
P0 (baseline)	64		137.00
P1 (Q_4 spec.)	66	4	144.18
P2 (Q_6 spec.)	60	3	134.25

Table III: Tests on specialized solvers.

Table III shows two different implementations with specialized solvers (so two solvers per time frame): in the first one ($P1$) the second solver handles generalization calls while in the second one ($P2$) the it handles propagation calls. Overall, we see a little improvement by $P1$, with two more instances solved w.r.t $P0$, including 4 instances not solved by $P0$.

The next two tables show an evaluation of different solver cleanup heuristics. Let a be the number of activation variables in the solver, $|vars|$ the total number of variables in the solver, $|clauses|$ the total number of clauses in the solver, $u(c')$ the heuristic measure discussed in Section VII and $W(x, n)$ a sliding window containing the values of x for the last n SAT calls. The heuristics compared are the following:

- H1: $a > 300$;
- H2: $a > \frac{1}{2}|vars|$;
- H3: $avg\big(W\big(\frac{u(c')}{|clause|}, 1000\big)\big) > 0.5$
- H4: $H2 \mid\mid H3$;

Heuristic H1 is the cleanup heuristic used in the baseline versions of both PdTRAV and ABC. The static threshold of 300 activation literals was determined experimentally. Heuristic H2 cleans up each solver as soon as half of its variables are used for activation literals. It can be seen as a first simple attempt to adopt a dynamic cleanup threshold. Heuristic H3 is the first heuristic proposed to take into account the second source of irrelevant clauses described in Section VII, i.e. irrelevant portions of previously loaded cones. In H3 a solver is cleaned up as soon as the percentage of irrelevant transition relation's clauses loaded, averaged on a window of the last 1000 calls, reaches 50%. Heuristic H4 takes into account both sources of irrelevant clauses, combining H2 and H3.

Configuration	Solved [#]	New [#]	Avg Time [s]
P0 (H1)	64		137.00
P3 (H2)	60	1	122.19
P4 (H3)	44		116.28
P5 (H4)	62	3	125.94

Table IV: Tests on clean up heuristics.

Table IV shows a comparison among H1 (row $P0$), H2, H3, and H4, in rows $P3$, $P4$, and $P5$, respectively. No PG encoding, nor specialized solvers were used. Heuristic H1, that employs a static threshold, was able to solve the largest number of instances. Among dynamic threshold heuristics, both H2 and H3 take into account a single source of irrelevant loaded clauses, respectively the deactivated clauses in H2 and the unused portions of transition relation in H3. Data clearly indicates that H3 has worse performance. This suggests that deactivated clauses play a bigger role in degrading SAT solvers' performance than irrelevant transition relation's clauses do. Taking into account only the latter source of irrelevant clauses it's not sufficient. It can be noticed that heuristic H4, that takes into account both sources, outperforms both H2 and H3. This proves that considering irrelevant clauses arising from previoulsy loaded cones in addition to deactivated clauses can be beneficial. In addition Table IV shows that dynamic heuristics were able solve some instances that can't be solved by the static heuristic H1 and viceversa. In terms of overall number of solved instances, the static heuristic H1 outperformes our best dynamic heuristic H4. This can be due to the fact that the threshold parameter of H1 results from extensive experimentation while to determine the parameters of H4 (window size and percentage thresholds) a narrower experimentation has been performed.

We then focused on H4, and benchmarked it in different setups. Results are showed in table V.

Here PG encoding was used in configurations $P6$ and $P8$, single solver per time frame in $P6$, additional specialized solver for generalization in $P7$ and $P8$. We see that the specialized solver configuration appears to perform worse with PG encoding. Also, adding a specialized solver for generalization to the dynamic heuristic H4 doesn't seem to be as effective as it is when using the static heuristic H1.

Configuration	Solved [#]	New [#]	Avg Time [s]
P0 (baseline)	64		137.00
P6 (PG)	59	3	208.85
P7 (Q_4 spec)	58	1	111.59
P8 (PG+Q_4 spec)	50	1	178.56

Table V: Tests on mixed strategies for cleanup heuristic H4.

This can be due to the fact that irrelevant clauses arising from generalization calls are not taken into account to schedule the clean up of the main solver that, in turn, is cleaned up less frequently.

C. Experiments with ABC

The 70 selected circuits were also benchmarked with ABC, with same pre-processing used in PdTRAV: Table VI report in row $A0$ the default setting of ABC. Row $A1$ shows the variant with PG encoding, row $A2$ shows a run without dynamic TR clause loading. Row $A3$ finally shows a different period for solver cleanup (1000 variables instead of 300).

Configuration	Solved [#]	New [#]	Avg Time [s]
A0	64		138.66
A1	63	1	152.18
A2	63	2	158.75
A3	64		138.45

Table VI: Tests on ABC with different strategies.

Overall, results show little variance among different settings, which could suggest lesser sensitivity of ABC to different tunings. Nonetheless, a further experimentation with ABC on the full set of 310 benchmarks (with 300 seconds time limit), showed a 14% improvement in the number of solved problems (from 71 to 81), which indicate a potential improvement for a portfolio-based tool, able to concurrently exploit multiple settings.

IX. Conclusions

The paper shows a detailed analysis and characterization of SAT queries posed by IC3. We also discuss new ideas for solver allocation/loading/restarting. The experimental evaluation done on two different state-of-the-art academic tools, shows lights and shadows, as no breakthrough or clear winner emerges from the new ideas.

PG encoding showed to be less effective than expected. This is probably because the benefits introduced in terms of loaded formula size will be overwhelmed by the supposed worst propagation behaviour of that formula.

The use of specialized solvers seems to be effective when a static cleanup heuristic is used, less effective when combined with PG encoding or a dynamic heuristic.

Our experiments showed that, when a dynamic cleanup heuristic is used, IC3's performance can be improved by taking into account both deactivated clauses and irrelevant portions of previously loaded cones. Even if a parameter configuration for H4 that is able to outperform the currently used, well-rounded static heuristic H1 hasn't been found yet, we believe that a more extensive experimentation could lead to better results.

Nonetheless, the results are more interesting if we consider them from the standpoint of a portfolio-based tool, since the overall coverage (by the union of all setups) is definitely higher.

So we believe that more effort in implementation, experimentation, and detailed analysis of case sudies, needs to be done. We also deem that this work contributes to the discussion of new developments in the research related to IC3.

References

[1] A. R. Bradley, "SAT-based model checking without unrolling," in *VMCAI*, Austin, Texas, Jan. 2011, pp. 70–87.

[2] A. R. Bradley, "Understanding IC3," in *SAT*, ser. Lecture Notes in Computer Science, A. Cimatti and R. Sebastiani, Eds., vol. 7317. Springer, 2012, pp. 1–14.

[3] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic Model Checking using SAT procedures instead of BDDs," in *Proc. 36th Design Automation Conf.* New Orleans, Louisiana: IEEE Computer Society, Jun. 1999, pp. 317–320.

[4] M. Sheeran, S. Singh, and G. Stålmarck, "Checking Safety Properties Using Induction and a SAT Solver," in *Proc. Formal Methods in Computer-Aided Design*, ser. LNCS, W. A. Hunt and S. D. Johnson, Eds., vol. 1954. Austin, Texas, USA: Springer, Nov. 2000, pp. 108–125.

[5] P. Bjesse and K. Claessen, "SAT–Based Verification without State Space Traversal," in *Proc. Formal Methods in Computer-Aided Design*, ser. LNCS, vol. 1954. Austin, TX, USA: Springer, 2000.

[6] K. L. McMillan, "Interpolation and SAT-Based Model Checking," in *Proc. Computer Aided Verification*, ser. LNCS, vol. 2725. Boulder, CO, USA: Springer, 2003, pp. 1–13.

[7] N. Eén, A. Mishchenko, and R. K. Brayton, "Efficient implementation of property directed reachability," in *FMCAD*, 2011, pp. 125–134.

[8] A. Mishchenko, "ABC: A System for Sequential Synthesis and Verification, http://www.eecs.berkeley.edu/∼alanmi/abc/," 2005.

[9] G. Cabodi, S. Nocco, and S. Quer, "Benchmarking a model checker for algorithmic improvements and tuning for performance," *Formal Methods in System Design*, vol. 39, no. 2, pp. 205–227, 2011.

[10] N. Eén and N. Sörensson, "The Minisat SAT Solver, http://minisat.se," Apr. 2009.

[11] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proc. 38th Design Automation Conf.* Las Vegas, Nevada: IEEE Computer Society, Jun. 2001.

[12] N. Eén and N. Sörensson, "Temporal induction by incremental SAT solving," *Electr. Notes Theor. Comput. Sci.*, vol. 89, no. 4, pp. 543–560, 2003.

[13] D. A. Plaisted and S. Greenbaum, "A structure-preserving clause form translation," *J. Symb. Comput.*, vol. 2, no. 3, pp. 293–304, 1986.

[14] M. Järvisalo, A. Biere, and M. Heule, "Simulating circuit-level simplifications on CNF," *J. Autom. Reasoning*, vol. 49, no. 4, pp. 583–619, 2012.

[15] N. Eén, "Practical SAT: a tutorial on applied satisfiability solving," *Slides of invited talk at FMCAD*, 2007.

[16] A. Biere and T. Jussila, "The Model Checking Competition Web Page, http://fmv.jku.at/hwmcc."