Form Methods Syst Des (2017) 50:39–74
DOI 10.1007/s10703-017-0272-0

SAT solver management strategies in IC3: an experimental approach

G. Cabodi[1] · P. E. Camurati[1] · A. Mishchenko[2] · M. Palena[1] · P. Pasini[1]

Published online: 25 February 2017

Abstract This paper addresses the problem of handling SAT solving in IC3. SAT queries posed by IC3 significantly differ in both character and number from those posed by other SAT-based model checking algorithms. In addition, IC3 has proven to be highly sensitive to the way its SAT solving requirements are handled at the implementation level. The scenario pictured above poses serious challenges for any implementation of the algorithm. Deciding how to manage the SAT solving work required by the algorithm is key to IC3 performance. The purpose of this paper is to determine the best way to handle SAT solving in IC3. First we provide an in-depth characterization of the SAT solving work required by IC3 in order to gain useful insights into how to best handle its queries. Then we propose an experimental comparison of different strategies for the allocation, loading and clean-up of SAT solvers in IC3. Among the compared strategies we include the ones typically used in state-of-the-art model checking tools as well as some novel ones. Alongside comparing multiple versus single SAT solver implementations of IC3, we propose the use of secondary SAT solvers dedicated to handling certain types of queries. Different heuristics for SAT solver clean-up are evaluated, including new ones that follow the locality of the verification process. We also address clause database minimality, comparing different CNF encoding techniques. Though not finding a clear winner among the different sets of strategies compared, we outline several potential improvements for portfolio-based verification tools with multiple engines and tunings.

Keywords Formal verification · Hardware model checking · IC3 · Sat solving

A preliminary version [1] of this paper was presented at DIFTS'13 workshop http://www.cmpe.boun.edu.tr/difts13/.

✉ M. Palena
marco.palena@polito.it

1 Dip. di Automatica Ed Informatica, Politecnico di Torino, Turin, Italy

2 Department of EECS, University of California, Berkeley, CA, USA

1 Introduction

IC3 is a SAT-based invariant verification algorithm for bit-level Unbounded Model Checking (UMC), first proposed by Bradley [2]. Since its introduction in 2010, IC3 has immediately generated strong interest in the model checking community and is now considered to be one of the major recent breakthroughs in the field. The algorithm has proved to be impressively effective in solving industrial-size verification problems. In our experience, in fact, IC3 is the invariant verification algorithm able to solve the largest number of instances by itself, among the benchmarks of the last Hardware Model Checking Competitions (HWMCC) [3].

1.1 Motivations

Like other SAT-based invariant verification algorithms, IC3 operates by expressing queries about the reachable state space of the system under verification in the form of SAT problems. Solving those problems with the aid of a SAT solver provides answers to the original queries, that are then used to drive the verification process. Any implementation of IC3 can thus be thought of as composed of two layers. At the top layer, the verification algorithm drives the traversal of the reachable state space by posing a series of SAT queries. At the bottom layer, the SAT solving framework interacts with a SAT solver in order to provide answers to those queries. From the implementation point of view, it is important to separate these layers by means of a clean interface, as shown in [4], so that both can be independently improved.

The SAT solving work required by IC3 presents some peculiar characteristics. First of all, unlike other SAT-based model checking algorithms (such as bounded model checking [5], k-induction [6,7] or interpolation [8]), IC3 typically requires to solve a very large number of SAT problems per verification instance (in the order of the tens of thousands SAT queries per run). Furthermore, as first noted by Bradley [9], the SAT queries posed by IC3 significantly differ in character from those posed by other SAT-based algorithms as they do not involve the use of transition relation unrollings. These queries are thus significantly smaller and easier to solve than the ones posed by other SAT-based algorithms. In addition, given the highly local nature of its verification process, IC3 has proved to be highly sensitive both to the internal behaviours of SAT solvers and to the way its SAT solving requirements are handled at the implementation level. Different SAT solver set-ups and/or configurations may, in fact, lead to widely varying results in terms of performance.

Besides algorithmic issues, the scenario pictured above poses serious challenges for any implementation of the algorithm. Choosing which SAT solver to use and how to manage the SAT solving work required are both critical aspects of IC3 implementation, directly affecting its performance. In such a scenario, a natural question arises, regarding whether or not the specific characteristics of its SAT queries can be exploited to improve the overall performance of the algorithm. This can be done by either modifying the SAT solving procedures to better fit IC3 needs or by defining better strategies to manage the SAT solving requirements of the algorithm. Identifying the desirable characteristic of a SAT solver intended for IC3 and determining which is the most profitable way to handle SAT queries in IC3 are still open problems. In this paper we address the latter problem, proposing and comparing different implementation-level strategies for SAT queries management in IC3. Furthermore, we focus only on CNF-based SAT solvers as the algorithm naturally lends to a CNF representation of SAT problems and these are the solvers used in mainstream state-of-the-art IC3 implementations.

In this context, the IC3 implementations available in many state-of-the-art model checking tools differ in several aspects, most notably: how many SAT solvers to use and which queries

are to be solved by each one, how to load SAT formulas in the different solvers and how long the lifespan of each solver must be. We refer to these aspects as SAT solvers *allocation*, *loading* and *clean-up* respectively. Our aim is to identify the most efficient set of strategies for each of these aspects.

1.2 Contributions

Our first contribution is to provide an in-depth characterization of the SAT solving work required by IC3. First, we identified several types of SAT queries based on their role inside the verification process of the algorithm, as implemented in our model checking suite PdTRAV [10]. Then, we collected many useful statistics on each of them, running our implementation with default settings over a large set of benchmarks. The purpose of such an evaluation is to gain useful insights into how best to handle SAT queries in IC3. A second contribution of this paper is to propose novel strategies for SAT solver allocation and clean-up based on such insights. In particular, regarding SAT solver allocation, we propose the use of *specialized SAT solvers*, i.e., secondary solvers dedicated to handling certain types of queries. As far as SAT solvers clean-up is concerned, we propose new strategies for triggering clean-ups based on the locality of the verification process. A third contribution of the paper is to provide a thorough experimental comparison of different strategies for the allocation, loading and clean-up of SAT solvers in IC3. Such a comparison was conducted over a large set of benchmarks from the HWMCC suite using three different state-of-the-art model checkers: PdTRAV [10], ABC [11] and nuXmv [12]. Among the compared strategies we included the ones commonly used by many state-of-the-art model checking tools as well as the newly proposed ones. In particular, we compare multiple versus single SAT solver implementations of IC3, specialized SAT solvers strategies for various types of queries, several heuristics for SAT solvers clean-up, different CNF encoding techniques and on-demand transition relation loading. Although not finding a clear winner among the different sets of strategies considered, we outline several potential improvements for a portfolio-based verification tool with multiple engines and tunings.

1.3 Related works

In recent years, many works improving on the original IC3 algorithm have been published, e.g. [13–15]. These works aim at improving IC3 at the algorithmic level by proposing different variants and/or optimizations for many steps of the algorithm. In this paper we aim at improving performance of IC3 at the implementation level, addressing the way its SAT solving queries are handled by the underlying SAT solving framework.

A fundamental work on IC3, both from the algorithmic and the implementation standpoint, is [4], in which the authors introduce a more efficient implementation of the original algorithm alongside proposing numerous high-level optimizations. Another significant contribution of the paper is to provide a clean interface separating the top-level verification algorithm from the underlying SAT solver. The authors of [4] call their variant of the algorithm Property Directed Reachability (PDR). Many IC3 implementations in state-of-the-art model checkers, including the ones available in PdTRAV, ABC and nuXmv, are based on PDR.

The work closest to the one presented in this paper is Griggio and Roveri [16], in which different variants of IC3 are implemented in the same tool and empirically compared. In the paper, the authors mainly focus on algorithmic variations of IC3, evaluating different optimizations, data structures and/or alternative procedures for many of its steps in order to assess their impact on performance. Alongside these high-level variants of IC3, many low-

level modifications are evaluated as well. From the perspective of SAT solving management, the paper compares single versus multiple SAT solver configurations, three different clean-up strategies based on the number of incremental SAT calls and two different CNF encoding techniques. The experimental evaluation we provide in this paper is complementary to the one presented in [16]. Instead of considering different algorithmic variants of IC3, we focus on different strategies for handling its SAT solving work. In this sense, we provide an independent evaluation of some of the low-level variants of IC3 compared by Griggio and Roveri, but we also extend the scope of such a comparison to many other strategies not considered in [16].

Our work is largely motivated by the considerations outlined by Bradley [9]. In particular, Bradley recognizes the high sensitivity of IC3 w.r.t. its SAT solving requirements and highlights the opportunity for SAT and SMT researchers to directly address the problem of improving IC3 performance by exploiting the peculiar character of the SAT queries it poses. Based on our experience with the algorithm, we believe this can be done by either tackling the internal behaviours of SAT solvers or the way SAT solving requirements of IC3 are handled.

1.4 Outline

In Sect. 2 we introduce the notation used and give some background on invariant verification, CNF encodings and IC3. Section 3 describes the SAT solving requirements of IC3. In Sect. 4 we provide a characterization of the SAT queries posed by our implementation on the algorithm. Both common and novel approaches to the allocation, loading and clean-up of SAT solvers in IC3 are discussed in Sects. 5, 6 and 7 respectively. Experimental data comparing these approaches are presented in Sect. 8. Finally, in Sect. 9 we draw some conclusions and give summarizing remarks.

2 Background

This section reviews the needed background related to invariant verification, CNF encodings[1] and IC3. We address invariant verification of Boolean circuits represented as And-Inverter Graphs (AIGs). These circuits can be modeled as state transition systems that are implicitly represented by Boolean formulas.

2.1 Boolean formulas and circuits

Definition 1 A *literal* is a Boolean variable or the negation of a Boolean variable. A *clause* is a disjunction of literals whereas a *cube* is a conjunction of literals. A Boolean formula is said to be in *Conjunctive Normal Form* (CNF) iff it is a conjunction of clauses.

Definition 2 A truth assignment for a Boolean formula F is a function $\tau : \mathbb{B}^n \rightarrow \{\top, \bot\}$ that maps variables in F to truth values. A truth assignment τ for F is *complete* iff every variable in F is assigned a truth value by τ, otherwise τ is *partial*.

Definition 3 A truth assignment τ satisfies a literal x, written $\tau \models x$, iff $\tau(x) = \top$. Conversely, a truth assignment τ satisfies a literal $\neg x$ iff $\tau(x) = \bot$. A truth assignment τ satisfies a clause C, written $\tau \models C$, iff at least a literal in C is satisfied by τ. A truth assignment τ satisfies a CNF formula F, written $\tau \models F$, iff each clause in F is satisfied by τ.

[1] We provide in this section an informal and high-level description of CNF encoding techniques, we refer the interested reader to the more rigorous and detailed description provided in [17].

Definition 4 A Boolean formula F is *satisfiable* iff there exists a truth assignment τ for F so that $\tau \models F$. Otherwise F is *unsatisfiable*. Two Boolean formulas F and G are *equi-satisfiable* iff either both F and G are satisfiable or both are unsatisfiable.

With abuse of notation we sometimes represent a truth assignment as a set of literals of different variables. A truth assignment represented this way assigns each variable to the truth value satisfying the corresponding literal in the set. We also represent a clause (cube) as a set of literals, leaving the disjunction (conjunction) implicit when clear from the context.

Definition 5 A Boolean circuit is a directed acyclic graph $\mathcal{C} = (G, E)$ where each node $g \in G$ is called a *gate* and can either be a Boolean variable $x \in \mathbb{B}$, a Boolean constant $c \in \{\top, \bot\}$ or a Boolean n-ary function $f : \mathbb{B}^n \to \mathbb{B}$ of its n incoming gates:

$$g := \begin{cases} x & \textit{with } x \in \mathbb{B} \\ c & \textit{with } c \in \{\top, \bot\} \\ f(g_1, \ldots, g_n) & \textit{with } \forall g_i \in \{g_1, \ldots, g_n\} : \exists\, (g_i, g) \in E \end{cases}$$

If $(g, g') \in E$, then g is a *parent* of g' and g' is a *child* of g. Each gate $g \in G$ for which $g := f(g_1, \ldots, g_n)$ is called a f-gate or a gate of type f. The *fan-out* (*fan-in*) of a gate is the number of antecedents (descendants, respectively) the gate has. A Boolean circuit $\mathcal{C}$ with n inputs and m outputs can be used to represent a Boolean function $f_{\mathcal{C}} : \mathbb{B}^n \to \mathbb{B}^m$, according to the definition of its gates.

Boolean functions that typically occur as gate types are logical negation ($\neg$), logical conjunction ($\wedge$), logical disjunction ($\vee$), logical exclusive disjunction ($\oplus$) and if-then-else (ITE). As typical, we consider gates of type logical negation to be embedded in the edges of the graph. Each edge $(g, g') \in E$ can be marked as negated. If that is the case, then each occurrence of g in the definition of g' should be replaced with $\neg g$.

Definition 6 An AND-Inverter Graph (AIG) is a Boolean circuit $\mathcal{C}$ in which only binary logical conjunctions and logical negations may appear as gate functions. Negations are considered embedded in the edges of the graph.

Definition 7 A constrained Boolean circuit is a pair $\mathcal{C}^\tau = \langle \mathcal{C}, \tau \rangle$, where $\mathcal{C}$ is a Boolean circuit and τ is a partial truth assignment over the outputs of $\mathcal{C}$. The truth assignment τ represents a set of constraints that force some of the outputs of $\mathcal{C}$ to assume certain truth values.

2.2 CNF encoding techniques

Definition 8 A CNF encoding technique is a transformation that takes a Boolean circuit $\mathcal{C}$ as input and produces a CNF formula F as output such that F is equi-satisfiable to $f_{\mathcal{C}}$.

A CNF encoding technique typically operates by first subdividing $\mathcal{C}$ into a set of functional blocks, i.e., gates or groups of connected gates representing certain Boolean sub-functions. Then, for each of the functional blocks identified, a new Boolean variable y_i is introduced to represent its output. Finally, for each functional block representing a function $y_i := f(y_1, \ldots, y_n)$, a set of clauses is derived by translating into CNF the formula:

$$y_i \leftrightarrow f(y_1, \ldots, y_n)$$

The final CNF formula F is obtained as the conjunction of these sets of clauses. Various CNF encoding techniques are available in literature. Different CNF encoding techniques may

identify different types of functional blocks (representing different Boolean sub-functions) as well as translating a given functional block into different sets of clauses. The number of functional blocks identified and the way each of them is translated into clauses, directly affect the size of the CNF formula obtained, both in terms of variables and clauses.

The standard CNF encoding technique is called *Tseitin encoding* [18]. Such a technique translates an AIG circuit $\mathcal{C}$ into a CNF formula F whose size is linear in the number of gates of $\mathcal{C}$. A new variable y is introduced for every AND gate of the circuit (i.e., no functional blocks are recognized) and F is derived by translating each gate formula as follows:

$$y \leftrightarrow x_1 \wedge x_2 \Longleftrightarrow (y \vee \neg x_1 \vee \neg x_2) \wedge (\neg y \vee x_1) \wedge (\neg y \vee x_2)$$

A simple variant of Tseitin encoding, recognizing some simple functional blocks, can also be used. Given an AIG circuit, basic functional blocks realizing n-ary conjunction, n-ary disjunction, binary exclusive disjunction and if-then-else can be easily detected. Such functional blocks can then be translated into clauses as follows:

$$y \leftrightarrow x_1 \wedge \cdots \wedge x_n \Longleftrightarrow (y \vee \neg x_1 \vee \cdots \vee \neg x_n) \wedge (\neg y \vee x_1) \wedge \cdots \wedge (\neg y \vee x_n)$$

$$y \leftrightarrow x_1 \vee \cdots \vee x_n \Longleftrightarrow (\neg y \vee x_1 \vee \cdots \vee x_n) \wedge (y \vee \neg x_1) \wedge \cdots \wedge (y \vee \neg x_n)$$

$$y \leftrightarrow x_1 \oplus x_2 \Longleftrightarrow (\neg y \vee \neg x_1 \vee \neg x_2) \wedge (\neg y \vee x_1 \vee x_2) \wedge (y \vee \neg x_1 \vee x_2) \wedge (y \vee x_1 \vee \neg x_2)$$

$$y \leftrightarrow ITE(s, x_1, x_2) \Longleftrightarrow (\neg y \vee \neg s \vee x_1) \wedge (\neg y \vee s \vee x_2) \wedge (y \vee \neg s \vee \neg x_1) \wedge (y \vee s \vee \neg x_2)$$

Technology mapping encoding [19], pushes functional blocks identification even further. The circuit is partitioned into cells with k-inputs and a single output that fits the LUTs of FPGA hardware. A new variable is then associated with each cell and conversion to clauses is performed resorting to a pre-computed library of CNF conversions for k-input LUT functions.

2.3 Transition systems

Definition 9 A *transition system* $\mathcal{S}$ is a triple $\langle X, I, T \rangle$, where X is a set of Boolean variables representing the states of the system, $I : \mathbb{B}^n \rightarrow \mathbb{B}$, n being the cardinality of X, is a Boolean formula over X representing the set of *initial states* of the system and $T : \mathbb{B}^n \times \mathbb{B}^n \rightarrow \mathbb{B}$ is a Boolean formula over $X \times X'$ that represents the *transition relation* of the system.

Variables of X are called *state variables* of $\mathcal{S}$. A state of $\mathcal{S}$ is thus represented by a complete truth assignment s to its state variables. Boolean formulas over X represent sets of system states. We denote as Space($\mathcal{S}$) the state space of $\mathcal{S}$. Given a Boolean formula F over X and a complete truth assignment s such that $s \models F$, then s is a state contained in the set represented by F and is thus called an F-*state*. Primed state variables X' represent future states of $\mathcal{S}$, i.e., states reached after a transition. Accordingly, Boolean formulas over X' represent sets of future states. We denote as s, s' a complete truth assignment to $X \times X'$ obtained by combining a complete truth assignment s to X and a complete truth assignment s' to X'.

Definition 10 A Boolean formula F is said to be *stronger* than another Boolean formula G iff $F \rightarrow G$, i.e., every F-state is also a G-state.

Definition 11 Given a transition system $\mathcal{S} = \langle X, I, T \rangle$, and a complete truth assignment s, s' to $X \times X'$, if $s, s' \models T$ then s is said to be a *predecessor* of s' and s' is said to be a *successor* of s. A sequence of states $(s_0, \ldots, s_n)$ is said to be a *path* in $\mathcal{S}$ iff $s_i, s'_{i+1} \models T$ for every couple of adjacent states in the sequence (s_i, s_{i+1}), $0 \leq i < n$.

Definition 12 Given a transition system $\mathcal{S} = \langle X, I, T \rangle$, a state s is said to be *reachable* (in n steps) in $\mathcal{S}$ iff there exists a path $(s_0, \ldots, s_n)$, such that $s_n = s$ and s_0 is an initial state (i.e., $s_0 \models I$). We denote with $R_n(\mathcal{S})$ the set of states that are reachable in $\mathcal{S}$ in at most n steps. We denote with $R(\mathcal{S})$ the overall set of states that are reachable in $\mathcal{S}$, i.e., $R(\mathcal{S}) = \bigcup_{i \geq 0} R_i(\mathcal{S})$.

We use transition systems to model the behaviour of hardware sequential circuits, where each state variable $x_i \in X$ corresponds to a latch, the set of initial states I is defined by reset values of latches and the transition relation T is the conjunction latches next-state functions δ_i (represented as AIGs or CNFs).

2.4 Invariant verification and induction

Definition 13 Given a transition system $\mathcal{S} = \langle X, I, T \rangle$ and a Boolean formula P over X (called safety property or invariant), the *invariant verification problem* is the problem of determining if P holds for every reachable state in $\mathcal{S}$, i.e., $\forall s \in R(\mathcal{S}) : s \models P$. An algorithm used to solve the invariant verification problem is called an *invariant verification algorithm*.

Definition 14 Given a transition system $\mathcal{S} = \langle X, I, T \rangle$, a Boolean formula F over X is said to be an inductive invariant for $\mathcal{S}$ if the following two conditions hold:

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

A Boolean formula F over X is called an inductive invariant for $\mathcal{S}$ relative to another Boolean formula 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 $\mathcal{S} = \langle X, I, T \rangle$, an inductive invariant F for $\mathcal{S}$ is an over-approximation to the set of reachable states $R(\mathcal{S})$.*

Definition 15 Given a transition system $\mathcal{S} = \langle X, I, T \rangle$, a Boolean formula P over X and an inductive invariant F for $\mathcal{S}$, F is an *inductive strengthening* of P for $\mathcal{S}$ iff F is stronger than P.

Lemma 2 *Given a transition system $\mathcal{S} = \langle X, I, T \rangle$ and a Boolean formula P over X, if an inductive strengthening of P can be found, then the property P holds for every reachable state of $\mathcal{S}$. The invariant verification problem of P for $\mathcal{S}$ can be solved by finding an inductive strengthening of P for $\mathcal{S}$.*

2.5 IC3

Given a transition system $\mathcal{S} = \langle X, I, T \rangle$ and a safety property P over X, IC3 aims at finding an inductive strengthening of P for $\mathcal{S}$. To this end, 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 $\mathcal{S}$. Each of these over-approximations is called a *time frame* and is represented by a set of clauses, denoted by 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:

(C1) $F_0 = I$
(C2) $F_i \rightarrow F_{i+1}$, for all $0 \leq i < k$

(C3) $F_i \wedge T \rightarrow F'_{i+1}$, for all $0 \leq i < k$
(C4) $F_i \rightarrow P$, for all $0 \leq i < k$

Condition (C1) states that the first time frame of the trace is simply assigned to the set of initial states of $\mathcal{S}$. The remaining conditions state that, for every time frame F_i but the last one: every F_i-state is also a F_{i+1}-state (C2), every successor of an F_i-state is an F_{i+1}-state (C3) and every F_i-state is safe (4), i.e., does not violate P. Condition (C2) is maintained syntactically, enforcing the condition clauses$(F_{i+1}) \subseteq$ clauses(F_i).

From these conditions, it is possible to derive the following lemmas:

Lemma 3 *Let $\mathcal{S} = \langle X, I, T \rangle$ be a transition system, $\mathbf{F_k} = F_0, F_1, \ldots F_k$ be a sequence of Boolean formulas over X and let conditions (C1–C3) hold for $\mathbf{F_k}$, then each F_i, with $0 \leq i < k$, is an over-approximation to $R_i(\mathcal{S})$, the set of states reachable within i steps in $\mathcal{S}$.*

Lemma 4 *Let $\mathcal{S} = \langle X, I, T \rangle$ be a transition system, P be a safety property over X, $\mathbf{F_k} = F_0, F_1, \ldots F_k$ be a sequence of Boolean formulas over X and let conditions (C1–C4) hold for $\mathbf{F_k}$, then P is satisfied up to $k-1$ steps in $\mathcal{S}$, i.e., there does not exist any counterexample to P of length less or equal than $k-1$ in $\mathcal{S}$.*

The main procedure of IC3, described in Algorithm 1 is composed of an initialization phase followed by two nested iterations. During initialization, the algorithm initializes the frame and ensures initial states are safe (lines 1–4). If that is not the case, a counterexample of length 0 is found and the procedure outputs a failure. During each outer iteration (lines 6–19), the algorithm tries to prove that P is satisfied up to k steps in $\mathcal{S}$, for increasing values of k. To this end, inner iterations of the algorithm refine the trace $\mathbf{F_k}$ computed so far, by adding new relative inductive clauses to some of its time frames (lines 7–12). The algorithm iterates until either an inductive strengthening of the property is produced (line 17) or a counterexample to the property is found (line 10).

```
Input: S = ⟨X, I, T⟩ ; P(X)
Output: SUCCESS or FAIL(σ), with σ counterexample
 1: k ← 0
 2: F_0 ← I
 3: if ∃t : t ⊨ F_0 ∧ ¬P then
 4:    return FAIL(σ)
 5: end if
 6: repeat
 7:    while ∃t : t ⊨ F_k ∧ ¬P do
 8:       s ← Extend(t)
 9:       if BlockCube(s, Q, F_k) = FAIL(σ) then
10:          return FAIL(σ)
11:       end if
12:    end while
13:    F_{k+1} ← ∅
14:    k ← k + 1
15:    F_k ← Propagate(F_k)
16:    if F_i = F_{i+1} for some 0 ≤ i < k then
17:       return SUCCESS
18:    end if
19: until forever
```

Algorithm 1. *High-level description of the algorithm IC3($\mathcal{S}$, P)*

At outer iteration k, the algorithm has already computed a trace $\mathbf{F_k}$ such that conditions (C1–C4) hold. From Lemma 4, it follows that P is satisfied up to $k-1$ steps in $\mathcal{S}$. IC3 then

tries to prove that P is satisfied up to k steps as well, by enumerating F_k-states that violate P and trying to block them in F_k.

Definition 16 Blocking a state (or, more generally, a cube) s in a time frame F_k means to prove s unreachable within k steps in $\mathcal{S}$ and, consequently, to refine F_k so that s is excluded from it.

To enumerate each state of F_k that violates P (line 7), the algorithm looks for states t in $F_k \wedge \neg P$. Such states are called *bad states* and can be found as satisfying assignments for the following SAT query:

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

If a bad state t can be found (i.e., Q_{target} is SAT), the algorithm tries to block it in F_k. To increase performance of the algorithm, as suggested in [4], the bad state t found is first extended to a bad cube s by removing, if possible, some of its literals. The *Extend*(t) procedure (line 8), not reported here, performs this operation via either ternary simulations [4] or other SAT-based procedures [15]. The resulting cube s is stronger than t and it still violates P, it is thus called a *bad cube*. The algorithm then tries to block the bad cube s rather than t. It is shown in [4] that, extending bad states into bad cubes before blocking them dramatically improves IC3 performance. The bad cube s is blocked in F_k calling the *BlockCube*(s, Q, $\boldsymbol{F_k}$) procedure (line 9), 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 *counterexample*
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 $s \not\models F_j$ **then continue**;
5: **if** $j = 0$ **then return** FAIL(σ)
6: **if** $\exists t, v' : t, v' \models F_{j-1} \wedge \neg s \wedge T \wedge s'$ **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: **if** doInductivePush **then**
12: **while** $\nexists t, v' : t, v' \models F_j \wedge c \wedge T \wedge \neg c'$ **do**
13: $j \leftarrow j + 1$
14: **end while**
15: **end if**
16: $F_i \leftarrow F_i \cup c$ for $0 < i \leq j$
17: add $(j + 1, c)$ to Q
18: **end if**
19: **end while**
20: **return** SUCCESS

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

When no further bad states can be found, conditions (C1–C4) hold for $k + 1$ and IC3 can safely move to the next outer iteration, i.e., 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 13). Initially clauses$(F_{k+1}) = \emptyset$, so that $F_{k+1} = \text{Space}(\mathcal{S})$. Note that Space($\mathcal{S}$) is a valid over-approximation to the set of states reachable within $k + 1$ steps in $\mathcal{S}$.

After creating a new time frame, a phase called *clause propagation* takes place (line 15). During such a phase, IC3 tries to refine every time frame F_i, with $0 < i \leq k$, by checking if some of its clauses can be pushed forward to the following time frame. Possibly, clause propagation refines the outermost time frame F_k so that $F_k \subset \text{Space}(\mathcal{S})$. The propagation phase can lead to two adjacent time frames becoming equivalent. If that happens, the algorithm has found an inductive strengthening of P for $\mathcal{S}$ (equal to those time frames). Therefore, following Lemma 2, property P holds for for every reachable state of $\mathcal{S}$ and IC3 returns a successful result (line 16). Procedure *Propagate*($\boldsymbol{F_k}$), described in Algorithm 4 and discussed later, handles the clause propagation phase.

The purpose of procedure *BlockCube*(s, Q, $\boldsymbol{F_k}$) is to refine the trace $\boldsymbol{F_k}$ in order to block a bad cube s in F_k. To preserve condition (C3), prior to blocking a cube in a certain time frame, IC3 has to recursively block predecessors of that cube 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 17 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 reach a bad state in some number of transitions. The index j indicates the position in the trace where s must be proved unreachable, or else the property fails.

Definition 18 A proof obligation (s, j) is said to be discharged when s becomes blocked in F_j.

In order to discharge a proof obligation, new ones may have to be recursively discharged. This can be done through a recursive implementation of the cube blocking procedure. However, in practice, handling proof obligations using a priority queue Q proved to be more efficient [4] and it is thus the commonly used approach in most state-of-the-art IC3 implementations. While blocking 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, described in Algorithm 2, the queue of proof obligations is initialized with (s, k), encoding the fact that s must be blocked in F_k (line 1). Then, proof obligations are iteratively extracted from the queue and discharged (lines 2–19).

Prior to discharging a proof obligation (s, j), 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 some previous proof obligation discharging. To perform this check, the algorithm tests if s is still included in F_j (line 4). This can be done by posing the following SAT query:

$$SAT?(F_j \wedge s) \qquad (Q_{blocked})$$

If s is in F_j (i.e., $Q_{blocked}$ is SAT), the proof obligation (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 F_j is the initial time frame (line 5). If so, the states represented by s are initial states that can reach a violation of property P. Thus, a counterexample σ to P can be constructed by following

the chain of proof obligations that led to $(s, 0)$. In that case, the procedure terminates with a failure and returns the counterexample 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}. The base case of induction $(I \rightarrow \neg s)$ holds by construction, whereas the algorithm must check whether or not the relative inductive case $(F_{j-1} \wedge \neg s \wedge T \rightarrow \neg s')$ holds. This can be done by proving that the following SAT query is unsatisfiable (line 6):

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

If the inductive case holds (i.e., Q_{relind} is UNSAT), then the clause $\neg s$ is inductive relative to F_{j-1} and can be used to refine F_j, ruling out s (lines 10–17). 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 the time frame. Inductive generalization is carried out by the *Generalize*$(j, s, \boldsymbol{F_k})$ procedure, described in Algorithm 3, which tries to minimize the number of literals in a clause $c = \neg s$ while maintaining its inductiveness relative to F_{j-1}, in order to preserve condition (C2).

After an inductive clause is found and generalized, the algorithm can optionally try to push it forward to the following time frame as far as relative induction continues to hold (lines 11–15). This step, called *inductive push*, is not necessary for the algorithm to converge, but it may be useful in some cases. The resulting clause is added not only to F_j, but also to every time frame F_i, $0 < i < j$ (line 16). Doing so discharges the proof obligation (s, j) ruling 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 17).

Otherwise, if the inductive case does not hold, there is a predecessor t of s in $F_{j-1} \wedge \neg s$. Such predecessors are called *counterexample to the inductiveness* (CTIs). To preserve condition (C3), before blocking a cube s in a time frame F_j, every CTI of s must be blocked in F_{j-1}. Therefore, the CTI t is first extended into a cube p (line 7), 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: $c \leftarrow \neg s$
2: **for all** literals l in c **do**
3: $try \leftarrow$ the clause obtained by deleting l from c
4: **if** $\nexists t, v' : t, v' \models F_{j-1} \wedge try \wedge T \wedge \neg try'$ **then**
5: **if** $\nexists t \models I \wedge \neg try$ **then**
6: $c \leftarrow try$
7: **end if**
8: **end if**
9: **end for**
10: **return** c

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

The *Generalize*$(j, s, \boldsymbol{F_k})$ procedure (Algorithm 3) is used to perform inductive generalization. Inductive generalization is a pivotal step of IC3 in which, given a clause $\neg s$ relative inductive to F_{j-1}, the algorithm tries to compute a subset c of $\neg s$ such that c is still inductive relative to F_{j-1}. Using c to refine the time frames blocks not only the original bad cube s

but potentially also other states, thus allowing a faster convergence of the algorithm. Conceptually, inductive generalization works by dropping literals from the input clause while maintaining relative inductiveness w.r.t. F_{j-1}. Ideally, the procedure computes a minimal inductive sub-clause, i.e., a sub-clause that is inductive relative to F_{j-1} and no further literals can be dropped while preserving inductiveness [20]. However, as noted in [2], finding a minimal inductive sub-clause is often inefficient. Most implementations of IC3, therefore, fall back to an approximate version of such a procedure which is significantly less expensive, yet still able to drop a reasonable number of literals.

In the *Generalize*$(j, s, \boldsymbol{F_k})$ procedure, a clause c initialized with $\neg s$ (line 1) is used to represent the current inductive sub-clause. For every literal of c, the candidate clause try is obtained by dropping that literal from c (line 3). Dropping literals from a relative inductive clause can violate both the base and the inductive case. Therefore, the candidate clause try must be checked for inductiveness relative to F_{j-1}. The algorithm checks if the relative inductive case $(F_{j-1} \wedge try \wedge T \rightarrow try')$ still holds for try by posing the following SAT query:

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

If the relative inductive case holds (i.e., Q_{gen} is UNSAT), the algorithm also needs to prove the base of induction $(I \rightarrow try)$. This check (line 5) can either be done in a syntactic or semantic way. If the set of initial states I can be described as a cube, it is sufficient to check whether at least one of the literals of I appears in try with opposite polarity. This ensures that $\neg try$ does not intersect I and, thus the base case holds. Otherwise, the base of induction must be checked explicitly by answering the following SAT query:

$$SAT?(I \wedge \neg try) \qquad (Q_{baseInd})$$

If $Q_{baseInd}$ is UNSAT than the base case holds for try. If both the inductive case and the base case still hold for the candidate clause try, the current inductive sub-clause c is updated with try (line 6).

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** $\exists t, v' : t, v' \models F_j \wedge c \wedge T \wedge \neg c'$ **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. *Clause propagation procedure Propagate(*$\boldsymbol{F_k}$*)*

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 forward to F_{j+1} by checking if c is inductive relative to F_j (line 3). To do so, the algorithm must check whether the inductive case $F_j \wedge c \wedge T \rightarrow c'$ holds, by answering the following

SAT query:

$$SAT?(F_j \wedge c \wedge T \wedge \neg c') \qquad (Q_{push})$$

If the relative inductive case holds (i.e., Q_{push} is UNSAT), then c is relative inductive to F_j and can thus be pushed forward to F_{i+1}. Otherwise, c can't be pushed forward and the procedure moves to the next iteration.

3 SAT solving requirements of IC3

In this section we analyse the SAT solving requirements of IC3 from the perspective of its underlying SAT framework. From the standpoint of the SAT solving layer, the top-level verification algorithm poses a sequence of SAT calls, each querying the satisfiability of a different CNF formula. A natural question arises regarding how different the CNF formulas of two subsequent SAT calls posed by IC3 are. Subsequent SAT calls may act on different time frames, thus involving different sets of frame clauses, may require the use of the property or the transition relation and may include some additional clauses and/or cubes. In general, thus, no strictly incremental or decremental pattern in the sequence of CNF formulas that IC3 presents to the SAT framework can be identified. Some of the sub-formulas that compose IC3 SAT queries are determined by the transition system and may appear unaltered in different SAT calls. It is the case, for instance, of the property and transition relation clauses. Some other sub-formulas, such as time frames clauses, may appear in different SAT queries but they grow incrementally as verification proceeds. Finally, some other sub-formulas are only required to answer a specific query. Examples of these query-specific clauses and cubes are those appearing in relative induction checks.

A naïve approach to handle this sequence of SAT calls would be to use a new SAT solver instance to handle each query, initializing it from scratch with the CNF formula of that query. Unfortunately, initializing a new SAT solver instance from scratch implies a certain overhead. Given the huge amount of SAT queries posed by a typical run of IC3, this approach would be prohibitively expensive for the algorithm. For this reason, state-of-the-art implementations of IC3 resort to the use of SAT solvers exposing an incremental interface. Incremental SAT interfaces allow a sequence of SAT calls to be performed on a single solver, accommodating the loaded CNF formula between calls. This approach, besides reducing the overhead, also allows the SAT framework to reuse previously learned clauses when solving SAT queries.

Analysing the sequence of SAT calls posed by IC3, we can identify a set of features for an incremental SAT interface that are desirable for IC3. In particular, an incremental SAT interface for IC3 should support:

- Adding clauses to the CNF formula currently loaded in the solver;
- Removing clauses from the CNF formula currently loaded in the solver;
- Specifying variable assumptions to be used when solving the next SAT query.

Adding and removing clauses from the formula allow the SAT framework to use a single incremental SAT solver instance to manage the whole sequence of SAT calls. In particular, the ability to remove clauses from the current CNF formula (alongside inferences deriving from such clauses) is required to handle query-specific clauses introduced by previous SAT calls (such as the candidate inductive clause in relative induction checks). Similarly, variable assumptions are needed as a way to efficiently add query-specific cubes to the current formula.

Each literal of these cubes is treated as a variable assumption, enforced while solving the upcoming query and then forgotten by the solver.

Many SAT solvers, like MiniSAT [21], feature an incremental interface capable of adding new clauses to the formula and to support variable assumptions. Removing clauses from the formula, however, is a complicated task and many state-of-the-art SAT solvers do not directly support such a feature. This is because, when a clause C is being removed from the current CNF formula, the solver must also track down and remove every learned clause deriving from C. Although some solvers, such as *zchaff* [22], directly support clause removal, incremental interfaces such as the one exposed by MiniSAT are by far more common. In this work we assume the use of a SAT solver exposing a MiniSAT-like incremental interface.

Clause removal can be simulated through the use of variable assumptions and the introduction of additional variables, called *activation variables*, as described in [23]. The idea is that clauses that may have to be removed from the solver are loaded augmented with a literal, called *activation literal*. Such a literal is obtained as the negation of a new variable, called *activation variable*. Activation variables are additional variables, introduced in the solver for the purpose of clause removal and that should appear only in activation literals. Given a CNF formula F loaded into a SAT solver and $a \in \mathbb{B}$ an activation variable, we denote by F^a the set of clauses loaded with activation literal $\neg a$ into the solver, i.e., the clauses of F in the form $(C \vee \neg a)$, where C is an original problem clause. The following lemma holds:

Lemma 5 *Clauses in F^a do not influence the satisfiability of the formula F unless their activation variable a is asserted, i.e., assumed positive in the solver.*

Considering the case in which a is not asserted, if a satisfying assignment τ for $F \setminus F^a$ exists, then the assignment $\tau \wedge a$ satisfies F. Vice-versa, if no satisfying assignment exists for $F \setminus F^a$, then the same is true for F. Otherwise, if the activation variable a is asserted, activation literals in the clauses of F^a become falsified. Each clause of F^a, in the form $(C \vee \neg a)$, thus reduces to the original clause C and contributes with the rest of F to determine the satisfiability of the loaded CNF formula.

Definition 19 Given a SAT solver S, a clause C and an activation variable a, we say that C is *a-activable* in S iff the clause $(C \vee \neg a)$ is loaded into S. A clause C that is a-activable in a SAT solver S is said to be *activated* in S iff the literal a is assumed in S. A clause C that is a-activable in a SAT solver S is said to be *deactivated* in S iff the unitary clause $(\neg a)$ is added to S. Otherwise, C is said to be *inactive*.

Activated clauses become relevant for determining the satisfiability of the formula loaded in S. Deactivated clauses, instead, become satisfied (and are therefore made logically redundant) by the addition of a unitary clause asserting their activation literal. Note that every learned clause derived by resolution from a deactivated clause will share its activation literal and therefore be deactivated as well. Whenever a SAT query has to be solved by a SAT solver S, if the query includes an a-activable clause C in S, then C has to be activated by adding a to its set of variable assumptions. Clause C remains activated as long as the query is being solved, then it returns to the inactive state. Whenever a clause that is a-activable in S is not needed in the formula anymore, it must be deactivated. Deactivated clauses persist in the clause database of the solver until its periodical clause database cleaning procedure is executed. Inactive clauses are also redundant w.r.t. the loaded formula but they will not be deleted by the solver during its periodical clause cleaning. Once a clause is deactivated it cannot be made inactive again.

The described approach to clause removal has the disadvantage of introducing one fresh variable in the solver per clause (or group of clauses) that must be removed. After a clause

is deactivated, depending on the solver implementation, its activation variable may still occupy some space in the internal data structures of the solver. Over time, the presence of such variables can significantly degrade the SAT solver performance. Inactive clauses, albeit redundant w.r.t. the loaded formula, slow down the solver by occupying space in the clause database and in other data structures (the watch lists in particular). Also deactivated clauses will continue to linger in the solver for some time, contributing to the solver slow down meanwhile.

Given the highly local nature of its verification process, IC3 has proved to be highly sensitive both to the internal behaviours of SAT solvers and to the way its SAT solving requirements are handled at the implementation level. Different SAT solver setups and/or configurations may, in fact, lead to widely varying results in terms of performance. Altering the SAT solver configuration in any IC3 implementation may lead, for instance, to find different CTIs during the cube blocking phase of the algorithm. These, in turn, may lead to whole different chains of proof obligations to be discharged. As a consequence, different inductive lemmas can be derived and the algorithm may either converge faster or find itself dwelling excessively on some part of the state space, depending on the nature of those lemmas.

The scenario pictured above poses serious challenges for any implementation of the algorithm. Given a SAT solver exposing the required incremental interface, any implementation of IC3 must face the problem of deciding how to integrate the top-level algorithm with the underlying SAT solving layer. In order to do so, three aspects must be considered:

– *SAT solver allocation*: decide how many SAT solver instances to use and how to distribute SAT calls among them;
– *SAT solver loading*: decide which formula to incrementally load in each solver to correctly answer its SAT queries;
– *SAT solver clean-up*: decide when and how often each solver should be cleaned up to avoid performance degradation.

4 SAT queries in IC3

In order to gain some insight into how best to handle the SAT solving requirements of IC3, we analysed the sequence of SAT queries it poses. Depending on the variant of IC3 considered, different sets of SAT queries may need to be answered. We base our analysis on the version of the algorithm presented in Sect. 2.5, which is also the one implemented in our model checking tool PdTRAV. Note that, although some of these queries can be considered structural to every variant of the algorithm, different variants of IC3 may involve different subsets of queries.

Three kinds of SAT queries can be identified:

– Queries that test the intersection between a set of states and a time frame;
– Queries that check the inductive case for relative induction of a clause w.r.t. a time frame;
– Queries that check the base case for relative induction of a clause w.r.t. a time frame.

We characterize each query both in terms of its kind and the role it plays in the verification process. For instance, queries that check the relative inductive case for a clause play different roles in the algorithm as they are used in several of its steps for different purposes: to find, generalize or push inductive lemmas. Following the description of the algorithm presented in Sect. 2.5, we identify the queries reported in Table 1.

As previously stated, the base case for relative induction can be checked in either a syntactical or semantic way. Since the syntactic check is cheaper to perform, it is prioritized in our implementation of the algorithm. In the problems we consider the set of initial states can

Table 1 SAT queries breakdown in IC3

Name	Role	SAT query
Q_{target}	Target intersection	$SAT?(F_k \wedge \neg P)$
$Q_{blocked}$	Blocked cube	$SAT?(F_i \wedge s)$
Q_{relind}	Relative induction	$SAT?(F_i \wedge \neg s \wedge T \wedge s')$
Q_{gen}	Inductive generalization	$SAT?(F_i \wedge try \wedge T \wedge \neg try')$
$Q_{baseInd}$	Base of induction	$SAT?(I \wedge \neg c)$
$Q_{indPush}$	Inductive push	$SAT?(F_j \wedge c \wedge T \wedge \neg c')$
Q_{push}	Clause propagation	$SAT?(F_i \wedge c \wedge T \wedge \neg c')$

always be represented as a cube. Therefore, the query $Q_{baseInd}$ will be excluded from the rest of the analysis. From preliminary experiments, the inductive push optimization appears to be detrimental to algorithm performance in the general case. Therefore we consider inductive push disabled and exclude $Q_{indPush}$ from the analysis. We also assume that the $Extend$ procedure described in Sect. 2.5 reduces proof obligation cubes using ternary simulation.

The following characteristics can be recognized for the different types of SAT queries posed by IC3:

- *Small-sized formulas* each query includes at most a single instance of the transition relation;
- *Large number of calls* as the verification process of IC3 is highly localized, a large number of reachability checks are required;
- *Separate time frame contexts* each SAT query focuses on a single time frame;
- *Related sequence of calls* subsequent SAT calls arising from a given step of the algorithm may expose a certain correlation.

On the one hand, given their small size, SAT queries posed by IC3 can be considered trivial to solve w.r.t. the ones posed by other SAT-based invariant verification algorithms. On the other hand, such a triviality is largely compensated by their number. This suggests that IC3 implementations should prefer faster SAT solvers to more powerful ones. SAT queries posed by IC3 operate on separate time frame contexts. This suggests that the most natural strategy for SAT solvers allocation would be to instantiate one SAT solver instance for each time frame and then use that instance to handle every SAT query regarding the particular time frame. As mentioned is Sect. 3, in the general case, no strictly incremental or decremental pattern in the sequence of SAT formulas posed by IC3 can be identified. However, restricting the focus only to some steps of the algorithm, some patterns can be observed. For instance, SAT calls performed during inductive generalization act on very similar formulas, where each call differs from the previous one only by one literal in a clause and one variable assumption.

We conducted an experimental characterization of IC3 SAT queries, running our implementation of IC3 in its default configuration on the complete set of solved single property benchmarks of HWMCC 2014 and HWMCC 2015. The benchmark set is composed of 525 different instances overall, of which 337 are UNSAT and 188 are SAT. In line with the competition settings, time and memory limits of 900 s and 8 GB respectively were enforced. In Table 2 we report the statistics collected. Data are presented differentiating between SAT and UNSAT calls. For each type of SAT query we consider an average IC3 run and provide the number of calls performed and the total solving time. The first three columns report the

Table 2 Total number of calls and solving times of SAT queries in an average IC3 run

Query	Avg. total number of calls				Avg. total solving time (ms)			
	SAT	UNSAT	TOT	%	SAT	UNSAT	TOT	%
Q_{target}	324	20	344	0.61	1495	11	1506	0.41
$Q_{blocked}$	2978	24	3002	5.31	2204	2	2206	0.59
Q_{relind}	1046	1933	2979	5.26	21,747	2532	24,279	6.55
Q_{gen}	17,884	8473	26,357	46.57	70,104	5677	75,781	20.44
Q_{push}	15,781	8133	23,914	42.25	14,361	7889	22,250	6.00
Total	38,013	18,583	56,596	100	10,9912	16,110	126,021	33.99

average number of calls per query type and their percentage over the total number of calls. The last four columns report the total time spent, in an average run of IC3, solving queries of each type as well as the percentage of execution time dedicated to each of them. Averages are computed over the 525 instances of the benchmark set. Percentages of execution time are computed over the average execution time of the runs, equal to 370.75 s.

Concerning the volume of SAT calls for each query, we can derive the following considerations. A typical run of IC3 executes tens of thousands of SAT calls. SAT calls involved in inductive generalization and clause propagation are by far the most frequent ones, representing almost 90% of the queries posed by an average execution of the algorithm. This is because, Q_{gen} is called multiple times for each inductive clause found, potentially one time for each of its literals. Also the query Q_{push} can be performed multiple times for every inductive clause, potentially once or more for every propagation phase. As it may be expected considering the procedure described in Sect. 2.5, the numbers of $Q_{blocked}$ and Q_{relind} calls are very close. Their disparity is mainly due to the benchmarks in which a counterexample to the property is found. The number of target intersection calls is very small if compared to the one of other queries. This suggests that IC3 does not need to explicitly enumerate a very large number of bad cubes before a time frame is proved safe.

Considering the proportion of SATs and UNSATs for each query, we can notice that for $Q_{blocked}$ the number of UNSATs is extremely low. This suggests that proof obligation rarely becomes discharged as a result of discharging a previously extracted one. Regarding Q_{relind}, the higher number of UNSATs compared to SATs suggests that, when discharging a proof obligation, bad cubes are often found already relative inductive w.r.t. the previous time frame, i.e., the case in which a CTI is found is less frequent. In addition, it is interesting to notice that both number and proportion between SATs and UNSATs for Q_{gen} and Q_{push} are roughly the same, suggesting that those queries are very similar in nature. For both query types, a SAT result is far more frequent than an UNSAT one. This means that many of the relative inductive case checks performed during inductive generalization and clause propagation fail.

Analysing total solving times in Table 2, we can notice that, in an average run of IC3, about one third (34%) of the execution time is spent answering SAT queries. Most of this time is spent answering inductive generalization queries, in particular the ones producing a SAT result. This suggests that, speeding up the solving time for this type of queries would greatly improve overall algorithm performance. Despite presenting similar volumes of calls, queries Q_{gen} and Q_{push} consume significantly different percentages of the execution time.

Table 3 Average solving time, number of decisions and CNF size of a single call for each query type

Query	Avg. solving time (ms)		Avg. number of decisions		Avg. CNF size		
	SAT	UNSAT	SAT	UNSAT	Clauses	Literals	Variables
Q_{target}	4.61	0.53	13,669	257	2205	5497	1187
$Q_{blocked}$	0.74	0.07	1118	91	1047	2450	415
Q_{relind}	20.79	1.31	129,301	420	4396	11,736	2255
Q_{gen}	3.92	0.67	3580	636	2573	7187	813
Q_{push}	0.91	0.97	1226	759	1074	2780	299
Average	6.19	0.71	29,779	432	2259	5930	994

In order to assess how difficult to solve the different queries are, in Table 3 we present statistics regarding the average solving time, number of decisions and size of single calls for each query type. The size of each query is expressed in terms of the average number of clauses, variables and literals included in its CNF formula. Regarding average solving times of single calls, our measures suggest that solving satisfiable instances is by far more time consuming than solving unsatisfiable ones. Solving time is tightly related to the number of decisions required by each SAT query. Data reported in the third and fourth column of Table 3 show that only a few decisions are needed to detect unsatisfiability, whereas recognizing satisfiable instances requires a considerably higher number of decisions. Among the different types of queries, Q_{relind} appears to be the most difficult one to solve, both in terms of time and number of decisions. This can be due in part to its size and in part to the fact that, differently from Q_{gen} and Q_{push}, such a query often checks the relative induction of a certain clause w.r.t. a time frame for the first time. It is thus less likely for this type of queries to benefit from previous learning available in the solver.

In conclusion, Q_{relind} queries are the most difficult ones to solve but, given their limited number of calls, they do not burden the solver too much. Queries Q_{gen} and Q_{push} are the most frequent ones, but Q_{gen} seems to be significantly more difficult to solve than Q_{push}, resulting in a much more substantial load for the solver.

5 SAT solver allocation

The problem of SAT solver allocation in IC3 consists in deciding how many SAT solver instances to employ and how to distribute the SAT solving work required by the algorithm among them. As mentioned in Sect. 3, the naïve approach comprising a throwaway SAT solver instance for each query is not feasible. In order to limit the overhead due to SAT solver loading and to exploit clause learning, we consider an incremental SAT solver exposing a MiniSAT-like interface. As noted in Sect. 4, many of the queries posed by IC3 share sub-formulas, primarily time frame clauses. On the one hand, solving queries that share a sub-formula using the same incremental SAT solver instance allows us to exploit clause learning and thus to avoid fruitless repetition of work. On the other hand, given the limitations of the incremental SAT interface considered regarding clause removal, solving unrelated SAT queries with the same solver instance can significantly degrade its performance over time. We aim at finding an

allocation of SAT queries to incremental SAT solver instances that achieves a good trade-off between these two opposing trends, allowing learning of useful clauses to be shared among different SAT calls but at the same time limiting performance degradation due to the presence of redundant clauses and unused variables in the solver.

We distinguish two main approaches to SAT solver allocation:

– *Monolithic solver* a single solver is used to answer all queries;
– *One solver per time frame* each time frame has its own dedicated solver.

In the monolithic solver approach, the set of clauses of each time frame must be loaded into a single solver. Since SAT queries posed by IC3 act on separate time frame contexts, each time frame is assigned a different activation variable and its clauses are loaded inactive into the solver. To answer each SAT query, the appropriate time frame has to be activated through literal assumptions. Similarly, the property is loaded inactive into the solver with its own activation variable. If one solver per time frame is used, each solver is loaded only with the set of clauses of its corresponding time frame and the inactive property clauses.

Both approaches have their advantages and shortcomings. On the one hand, using a monolithic solver allows learned clauses to be shared among every SAT query but the presence of a large number of inactive clauses in the formula may degrade performance. On the other hand, using one solver per time frame leads to symmetrical considerations. Deciding which approach is the best one is not trivial, as their performance results from a trade-off between the aforementioned opposing trends and depends on the particular instance under verification.

Among state-of-the-art model checking tools, ABC and PdTRAV adopt the one solver per time frame approach, whereas nuXmv supports both approaches as well as an hybrid approach in which the first n time frames have their dedicated SAT solver and the remaining ones share a single solver.

We propose the use of dedicated solvers to handle specific queries. For every solver, a secondary solver can be instantiated to handle all the queries of a specific type. Using dedicated solvers to answer queries of a given type, will guarantee that any unnecessary clause arising from them will not impact the performance of future queries of different types. Therefore, the main purpose of using dedicated solvers is not to directly affect solving time of the queries being specialized, but rather to improve solving time of the other queries by preventing many irrelevant clauses to accumulate in the solver, hindering its performance.

As noted in Sect. 4, inductive generalization and clause propagation are by far the most frequent SAT queries in IC3. Both types of query introduce a query-specific clause in the solver, as well as some logic cones if lazy loading of the transition relation is used (see Sect. 6). Furthermore, some patterns can be recognized in the sequence of SAT calls performed during inductive generalization and clause propagation. SAT calls performed during an execution of inductive generalization act on very similar formulas, where each call differs from the previous one only by a literal in a clause and a variable assumption. Regarding clause propagation, we can identify correlations among the SAT checks performed in subsequent propagation phases. When a clause fails to be propagated, the same clause will be checked for propagation w.r.t. the same time frame in the next propagation phase. These SAT calls only differ by the inductive lemmas that were added to the given time frame between the two propagation phases.

We suppose it would be beneficial for the algorithm to solve those queries using dedicated solvers. This way, the large number of deactivated clauses and unused variables arising from these queries would not impact on solving other queries. Note that, althought Q_{relind} queries have a higher average solving time than Q_{push} ones, the formers are much less frequent than the latters and therefore the are not solved by dedicated solvers as they represent a lesser

source of irrelevant clauses. Furthermore, these related SAT queries would be solved by a solver whose internal state is not altered by the answering of other (unrelated) SAT calls in between. We call solvers dedicated to the handling of a particular type of query *specialized solvers*. For each SAT solver used, a specialized solver is instantiated and maintained by the SAT solving framework. Depending of the SAT solver allocation strategy used, either a single specialized solver or one specialized solver per time frame are instantiated. We propose three different specialized SAT solvers strategies:

- *Specialized generalization* each query of type Q_{gen} is performed on the specialized solver;
- *Specialized push* each query of type Q_{push} is performed on the specialized solver;
- *Specialized generalization+push* both queries of type Q_{gen} and Q_{push} are performed on the specialized solver.

6 SAT solver loading

Prior to solving a SAT query, all the relevant clauses of its formula that are not already loaded in the solver must be added to it. In order to minimize the number of loaded clauses to deactivate, we analyse the queries to identify what is the minimal subset of clauses that are needed to answer each one. Many of the SAT queries posed by IC3 involve the use of the transition relation. As mentioned in Sect. 2, the transition relation of the system is composed of many next-state function δ_i, one for each state variable x_i, so that:

$$x_i' \equiv \delta_i(X) \qquad \forall x_i \in X$$

Each next-state function δ_i determines the next-state value of a state variable x_i. We assume these functions are represented as AIGs and that they can be transformed into CNF formulas by applying a CNF encoding technique. We refer to the CNF formula derived from the next-state function δ_i of a state variable x_i as the *logic cone* of x_i. Next-state functions of different state variables may share sub-formulas, i.e., their AIGs may share some gates and, as a consequence, their logic cones may share some clauses.

Analysing the SAT queries posed by IC3, we observe that, when a SAT query involving the use of the transition relation is posed, the next state variables of the system are always constrained by some cube c'. Such queries ask if there exists a state s of the system that satisfies some present-state constraints (such as the intersection of a time frame and a clause) and which can reach one of the states c' in one transition. Being s a state that satisfies one of these queries, since c' is a cube, s must have a successor p' such that $p' \in c'$. Only the present state variables that appear in the logic cones of variables in c' are relevant in determining if such a state s exists. Therefore, only the logic cones of the variables in c' are needed to determine the satisfiability of the query. In order to minimize the size of the formula to be loaded into each solver, a common approach is to load, for every SAT call that involves the transition relation, only the necessary logic cones. This loading strategy, called as *lazy loading of transition relation*, is used in various implementations of IC3, as the ones of PdTRAV and ABC. Since different logic cones may share part of their clauses, the algorithm implementation must keep track of the portions of logic cones loaded in each solver. When a new logic cone must be loaded into a solver, only the clauses that it does not share with any of the logic cones already loaded in the solver should be added.

In the general case, each query involving the transition relation may require different logic cones to be loaded into the solver. As verification proceeds, portion of unused logic cones accumulate in the solver, degrading its performance as inactive clauses, deactivated clauses

and unused activation variables do. The size of logic cones to load depends on the particular CNF encoding technique employed to translate next-state functions of the transition relation into CNF formulas. As mentioned in Sect. 2, different CNF encoding techniques may, in fact, detect different functional blocks in the AIGs representing those functions as well as translating a given functional block into different sets of clauses. Besides influencing the size of the logic cones, choosing a CNF encoding technique also impacts on the propagation behaviour inside the SAT solver [24].

We noticed that, disregarding the particular CNF encoding technique employed, the number of logic cones clauses that need to be loaded into the solver can further be reduced by using a particular CNF encoding scheme due to Plaisted and Greenbaum [25], henceforth called *PG encoding*. Given a next state cube c' and the AIG representation $\mathcal{C}$ of the next-state functions of its variables, c' can be seen as a set of constraints on the outputs of $\mathcal{C}$. Therefore, the pair $(\mathcal{C}, c')$ can be seen as a constrained Boolean circuit. PG encoding is a CNF encoding scheme that may be applied when translating a constrained Boolean circuit into a CNF formula with a given CNF encoding technique. The CNF formula obtained when applying PG encoding is a subset of the one obtained when applying the original CNF encoding technique by itself. Given a constrained circuit $(\mathcal{C}, c')$ and a CNF encoding technique, translating $\mathcal{C}$ into a CNF formula F, PG encoding works by exploiting constraints on the outputs and topological information on the circuit to remove some of the clauses of F, producing a CNF formula $PG(F)$ that is equi-satisfiable to F. For some of the functional blocks detected by the given CNF encoding technique, in fact, it is proved that an equi-satisfiable encoding can be produced by only translating one of the two sides of their defining bi-implication (see [25] or [17]). PG encoding effectively reduces the size of CNF formulas, but it is not clear whether or not it leads to worse propagation behaviour [26].

7 SAT solver clean-up

Inevitably, as verification proceeds, some redundant clauses and unused variables accumulate in each solver degrading its performance. A solution to this problem is to periodically destroy each solver and to replace it with a fresh SAT solver instance. We refer to such an operation as a *SAT solver clean-up*. Besides recollecting wasted memory in the SAT solver data structures, this procedure has the added benefit to reset the internal state of the solver which over time might accumulate too much bias towards certain (possibly poor) choices. As a consequence, different bad cubes can be derived and blocked by IC3, leading the time frames to be differently refined and, in turn, steering the overall verification process toward a different direction.

To clean up a SAT solver instance, a new instance is allocated and loaded with only those clauses that are actually pertinent to answer the upcoming SAT query. Local data of the solver, such as learned clauses, saved variable phases and score values of variables and clauses, are lost in the process. Therefore, cleaning up a SAT solver instance introduces a certain overhead, mainly because clauses pertinent to answer the upcoming SAT query must be loaded again in the solver and any useful inference previously derived by the solver must be derived again while solving the upcoming SAT call.

Deciding how frequently SAT solver instances must be cleaned up is an important aspect of IC3 implementation and can greatly influence the performance of the algorithm. Any IC3 implementation must adopt a clean-up strategy, based on some heuristic measure, in order to achieve a trade-off between avoidance of performance degradation and clean-up overhead. The purpose of a clean-up strategy is to determine whether the number of irrelevant clauses

and/or variables (w.r.t. the upcoming query) currently loaded into a solver has become large enough to justify a clean-up. To do so, we define a heuristic measure representing an estimate of the number of irrelevant clauses/variables currently loaded into a solver. Such a measure is then compared to some threshold and, if it exceeds it, the solver is cleaned up.

Clean-up strategies used in state-of-the-art model checking tools, such as ABC, PdTRAV and nuXmv, usually rely on loose estimates of the size of the irrelevant portion of the formula loaded into each solver. Furthermore, these tools often use static thresholds to decide whether the computed estimates justify a SAT solver clean-up. Both PdTRAV and ABC use the number of deactivated clauses as a measure of the number of irrelevant clauses in the solver (without taking into account unused logic cones), whereas nuXmv only considers the number of incremental SAT calls.

We explore the use of novel clean-up heuristics, based on different measures of the number of irrelevant clauses loaded into each solver and able to dynamically adjust the frequency of clean-ups based on the locality of the verification process.

As stated before, upon receiving a new SAT call, two kinds of irrelevant clauses can be present into a SAT solver instance:

1. Deactivated and inactive clauses from previous inductive checks (such as Q_{relind} and Q_{gen} queries);
2. Portions of logic cones from previous SAT calls involving T that fall outside the logic cones of the upcoming query.

The actual number of irrelevant clauses in the solver depends both on the formula F currently loaded into the solver and on the nature of the upcoming SAT query Q. We investigate the use of a heuristic measure taking into account both sources of irrelevant clauses.

Given a SAT solver instance loaded with the formula F, and a SAT query $Q(a)$ in which a cube of activation literals a is assumed, the number of deactivated clauses $d(F, a)$ in F is simply computed by adding a counter for each activation variable to the solver. Every time a clause with a given activation literal is added to the solver, the counter of the corresponding variable is incremented. The number of clauses of F currently deactivated for $Q(a)$ is computed as the sum of the counters for each activation variable not in a.

Given a SAT solver instance loaded with the formula F and a SAT query $Q(c')$ whose solution requires the logic cones of a cube c' to be loaded in the solver, the number of irrelevant transition relation clauses of F w.r.t. $Q(c')$, depends on F and the logic cones of c'. Logic cones of different state variables may, in fact, share some of their clauses, corresponding to the common gates in the AIG representation of their next-state function. The number of logic cones clauses of F that are irrelevant to answer Q, denoted by $u(F, c')$, is computed as follows:

$$u(F, c') = |T(F)| - |S(F, c')|$$

where $|T(F)|$ is the number of transition relation clauses already loaded into the solver and $|S(F, c')|$ is the number of clauses that the logic cones of c' share with the logic cones previously loaded in the solver. In order to compute such a measure, the algorithm implementation must keep track of the logic cones currently loaded in each solver and must be able to compute the number of shared clauses among different logic cones through an appropriate data structure.

Given F and $Q(c')$, dividing $u(F, c')$ by the number of transition relation clauses in F we obtain the ratio of transition relation clauses in F that are irrelevant to solve the current query, denoted by $U(F, c')$:

$$U(F, c') = \frac{u(F, c')}{|T(F)|}$$

We consider such a measure averaged on a sliding window $W(U, n)$ of the last n SAT calls, in order to to dynamically adjust the frequency of clean-ups based on the locality of the verification process. As soon as the ratio of irrelevant transition relation clauses loaded in the solver, averaged on the last n SAT calls, exceeds some predetermined threshold we assume that the locality of the verification process has varied enough to deem the loaded logic cones futile.

Given a solver instance loaded with F and a SAT query $Q(c')$, let $|vars|$ be the total number of variables in F, $|act|$ be the number of activation variables in F, $U(F, c')$ be the ratio of transition relation clauses in F that are irrelevant for c' and $W(X, n)$ be a sliding window containing the measures of X collected for the last n SAT calls of a solver. We consider the following four clean-up strategies:

$$|act| > 300 \tag{H1}$$

$$|act| > \frac{1}{2}|vars| \tag{H2}$$

$$avg\big(W\big(U(F, c'), 1000\big)\big) > 0.5 \tag{H3}$$

$$H2 \;||\; H3 \tag{H4}$$

Clean-up strategy $H1$ is the default strategy implemented in PdTRAV and ABC. Each solver is cleaned up after a fixed number of variables are reserved for activation. Strategy $H2$, first proposed in [4], triggers SAT solvers clean-ups after half their variables are reserved for activation. $H3$ is based on the heuristic measure proposed before and cleans up each solver after the average number of irrelevant transition relation clauses, computed over the last thousand of calls, exceeds 50% of the total number of transition relation clauses. Finally, strategy $H4$ combines the preceding two in order to take into account both sources of irrelevant clauses.

8 Experimental results

We conducted an experimental evaluation of different SAT solver allocation, loading and clean-up strategies in IC3, considering the same set up described in Sect. 4.

Two sets of experiments were conducted on the IC3 implementation of three state-of-the-art model checking tools: PdTRAV, nuXmv and ABC. Section 8.1 shows results attained with PdTRAV on the full set of single property benchmarks of HWMCC 2014 and 2015. As it turns out that most of the benchmarks can be considered *easy* model checking problems, Sect. 8.2 focuses on a *core* subset of experiments, on which we present data for all three model checkers. All experiments were performed by comparison with a *baseline* configuration of the tool. For PdTRAV and ABC, the default configuration was used as baseline; for nuXmv, the same baseline configuration considered in [16] was used. As most model checkers operate some preprocessing (e.g., circuit reduction, signal and latch correspondence, phase abstraction) before activating model checking engines, we performed off-line preprocessing with PdTRAV, stored netlists on file, for fair comparison among the three model checkers. All of the three model checking tools considered make use of MiniSAT 2.2.0 as a back-end SAT solver. This helps us making more fair comparisons across the tools, excluding bias due to the specific SAT solver implementation.

Table 4 Number of solved instances using specialized solvers w.r.t. baseline

Configuration	Solved	UNSAT	SAT	New UNSAT	New SAT	Lost UNSAT	Lost SAT	$\Delta_{baseline}$
Q_{gen} spec.	311	225	86	4	1	4	3	−2
Q_{push} spec.	312	226	86	4	0	3	2	−1
Q_{gen} + Q_{push} spec.	315	226	89	4	3	3	2	+2
Baseline	313	225	88	–	–	–	–	–
Virtual best	323	232	91	7	3	–	–	+10

Experimental results are reported by comparing different groups of techniques against the respective baseline. For each group, results are provided by means of a table, a survival plot and a Venn's diagram. Tables report the total number of solved instances and the number of instances that are gained and lost by each technique w.r.t. the baseline. For each group, the corresponding table also reports the results of a *virtual best* configuration, obtained combining the results of all the compared techniques. Survival plots compare the number of solved instances of each technique in a group over time. Venn's diagrams show how the compared techniques relate in terms of mutually solved instances. For each technique, within each group, we also provide a scatter plot to assess its impact on performance w.r.t. the baseline, using different graphical artefacts to represent different benchmark families.

8.1 Experiments with PdTRAV on full benchmark set

We have run a full set of experiments over the 525 HWMCC 2014 and 2015 circuits, with a 900 s time limit and a 8 GB memory limit. Experiments were run on an Intel Core $i7-3770$, with 8 CPUs running at 3.40 GHz, 16 GB of main memory DDR III 1333, and hosting a Ubuntu 12.04 LTS Linux distribution.

8.1.1 Experiments with specialized SAT solvers

We report the number of solver instances (differentiating SAT and UNSAT results) alongside gains and losses and overall delta for each configuration w.r.t. our baseline.

As described before, we decided to specialize solvers in the context of Q_{gen} and Q_{push}, in three different configurations: only the former, only the latter and both at the same time. We focused our experiments on these two queries as they were the most frequent kind, as shown in Table 3.

Table 4 illustrates the role of specialized solvers, i.e., additional solvers per frame dedicated to a specific query, during IC3 runs. Such a table compares the three aforementioned specialization configurations against our baseline, which instead uses a single unspecialised solver for each time frame. All the variants are rather close in terms of overall solved instances, but each of them is characterized by the presence of some unique solves, thus providing a slight contribution to the portfolio.

Figure 1 illustrates data distribution of solved instances for all the instances w.r.t. to the baseline. Each sub-figure is dedicated to a specific variant, and each scatter plot represents the main circuit families with a different graphical style. Points that lie beneath the diagonal rep-

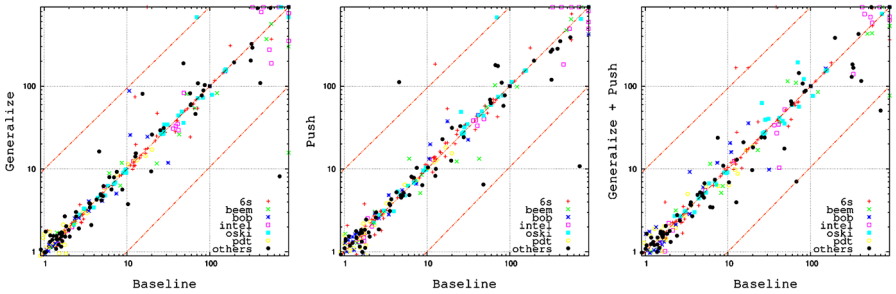

Fig. 1 Detailed comparison of different SAT solver specialization strategies versus the baseline. The baseline is always on the x-axis. Points below the diagonal characterize runs with a better performance. Points on borders represent time-outs. Different graphical artefacts visualize different benchmark families

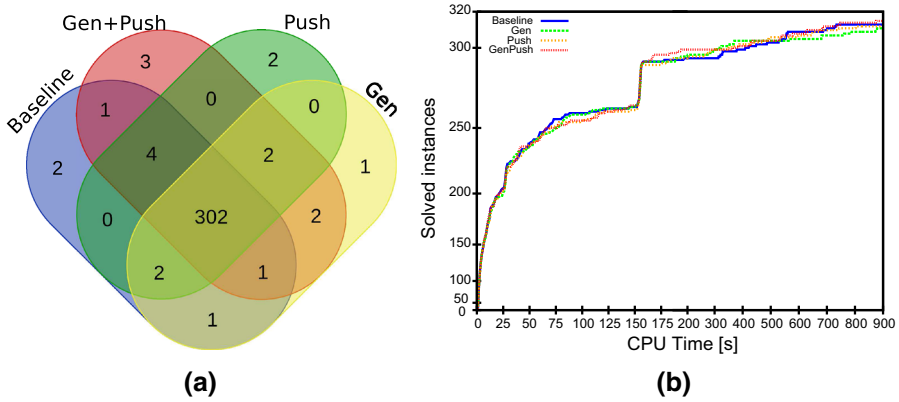

Fig. 2 Detailed comparison of different specialized SAT solvers strategies on the complete benchmark set. **a** is a Venn's diagram illustrating mutual coverage of solved instances among the compared strategies, **b** is a survival plot illustrating distribution of solved instances over CPU time for each compared strategy

resent instances for which the alternative method performs better, in term of mere execution time, than our baseline one. It is possible to notice that most of the points are rather close to the diagonal, with few obvious exceptions. Analysing the full logs produced by PdTRAV, we noticed that execution times for solved instances are rather close, regardless of the technique used. The main differences come from verification runs that end in time-out/memory-out for some of the considered variants or, on the other hand, from unique solves.

Figure 2a presents the same results of Table 4 in a graphical way. Such a figure adopts Venn's diagrams to better illustrate the overlap among different techniques and to instantly visualize the subset on unique instances that each variant is capable of proving. Overlapping regions describe instances that are solved in common by different techniques, whereas outer lobes underline unique solves for each strategy.

Figure 2b illustrates data distribution of the number of solved instances w.r.t. increasing execution time. All the variants are plotted on the same survival chart to facilitate data comparison. Once again, as already noticed while analysing scatter plots, it is rather straightforward to remark how close all the variants are to one another. All the distributions presented follow somehow the same pattern. We can identify a large subset of easy instances that all the proposed variants are capable of solving in a short amount of time, before encountering harder instances.

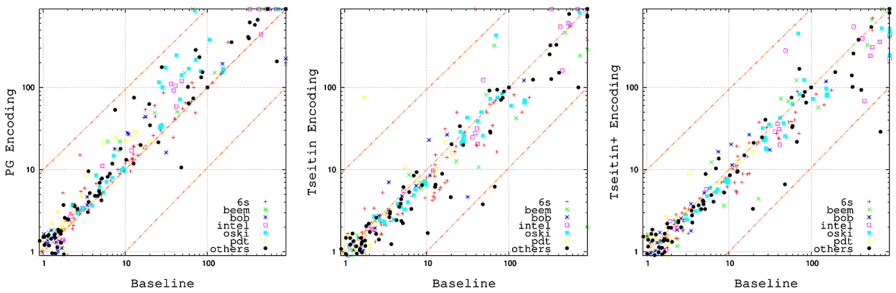

Fig. 3 Detailed comparison of different CNF encodings versus the baseline

8.1.2 Experiments with CNF encodings

A second set of experiments evaluates the use of different encodings. Results were partly controversial, at least if compared with [16].

The baseline version of PdTRAV uses technology mapping encoding as proposed in [19]. A run in PdTRAV using PG, with 900 s time-out, showed a reduction in the number of solved instances from 313 to 302 (2 of which, a SAT instance and an UNSAT instance, previously unsolved). Overall, a decrease of performance, despite a reduction of dynamically loaded transition relation clauses in the order of 20% for most of the considered circuits.

A first conclusion could be that PG encoding was not able to improve performance, suggesting it can indeed suffer of worst propagation behaviour. Nonetheless, it is interesting to observe that, combining the results of our baseline implementation with those obtained with PG encoding, the overall number of solved problems grows, albeit slightly, as the latter was able to solve two instances that the former could not.

An interesting result was obtained using the Tseitin encoding and an improved version with clauses for simple gate blocks (T and T+ for shortening, respectively). The two encodings proved better than our baseline one, both in terms of sheer number of solved instances and in terms of solving time.

Figure 3 illustrates data distribution of solved instances w.r.t. to the baseline. On the one hand we can see that PG encoding tends to result in worse execution time that the baseline, with most of the points lying above the diagonal. On the other hand, T+, manages to improve the baseline results for several instances.

Figure 4a presents the same results of Table 5 in a graphical way.

Figure 4b illustrates data distribution of the number of solved instance w.r.t. increasing execution time. All the variants are plotted on the same survival chart to facilitate data comparison. Once again, it is surprising to observe the faster rate of convergence for the T+ encoding, whereas PG stays constantly beneath the other two methods. Both axes feature a non-linear scale in order to better visualize the almost vertical slope on the left-hand side of the plot as well as the portion where the techniques start to plateau on its right-hand side. In this plot (and in all the following ones of the same kind) it is possible to notice a significant increase of solved instances in the middle of the curve. This is due to a number of *oski** instances, deriving from the same design, which are all solved roughly in the same amount of time.

We can conclude that T and T+ encodings seem to play better with incremental load of transition relation clauses, as, though using more clauses that the baseline, due to a finer granularity. A different conclusion is driven in [16], probably due to the fact that the baseline

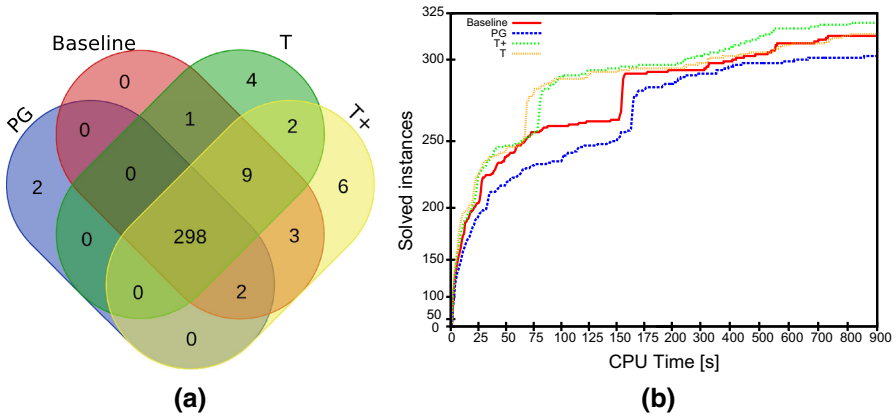

Fig. 4 Detailed comparison of different specialized SAT solvers strategies on the complete benchmark set. **a** is a Venn's diagram illustrating mutual coverage of solved instances among the compared strategies, **b** is a survival plot illustrating distribution of solved instances over CPU time for each compared strategy

Table 5 Number of solved instances using different CNF encodings w.r.t. baseline

Configuration	Solved	UNSAT	SAT	New UNSAT	New SAT	Lost UNSAT	Lost SAT	$\Delta_{baseline}$
PG encoding	302	219	83	1	1	7	6	−11
Tseitin encoding	314	226	88	4	2	3	2	+1
Tseitin+encoding	320	233	87	8	0	1	1	+7
Baseline	313	225	88	–	–	–	–	–
Virtual best	327	236	91	10	3	–	–	+14

IC3 implementation in nuXmv does not use dynamic loading of the transition relation, thus providing an ideal ground for a minimal clause set encoding.

8.1.3 Experiments with SAT solver clean-up strategies

The same set of experiments has been run taking into account different clean-up heuristics.

Heuristic H1 is the clean-up heuristic used in the baseline version of PdTRAV. The static threshold of 300 activation literals was determined experimentally. Heuristic H2 cleans up each solver as soon as half its variables are used for activation literals. It can be seen as a first simple attempt to adopt a dynamic clean-up threshold. Heuristic H3 is the first heuristic proposed to take into account the second source of irrelevant clauses described in Sect. 7, i.e., irrelevant portions of previously loaded cones. In H3 a solver is cleaned up as soon as the percentage of irrelevant transition relation 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.

Table 6 shows a comparison among H1 (our baseline), H2, H3, and H4 respectively. 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

Table 6 Test or restart strategies w.r.t. baseline

Configuration	Solved	UNSAT	SAT	New UNSAT	New SAT	Lost UNSAT	Lost SAT	$\Delta_{baseline}$
Variable activation (H2)	314	227	87	4	3	2	4	+1
Windowed (H3)	304	221	83	2	0	6	5	−9
Double threshold (H4)	310	226	84	4	1	3	5	−3
Baseline (H1)	313	225	88	–	–	–	–	–
Virtual best	321	230	91	5	3	–	–	+8

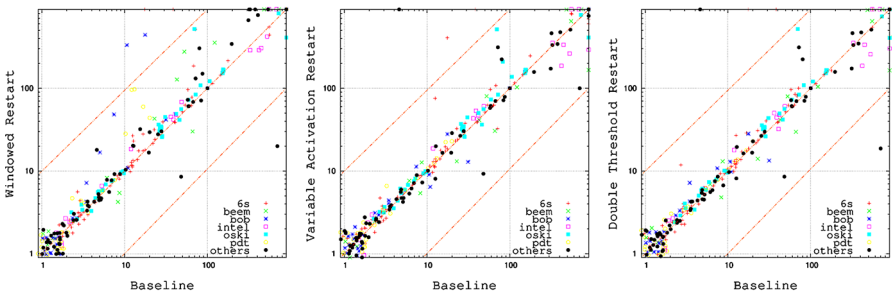

Fig. 5 Detailed comparison of different SAT solver restart strategies versus the baseline

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 clauses do. Taking into account only the latter source of irrelevant clauses is not sufficient.

Figure 5 illustrates data distribution of solved instances for all the instances w.r.t. to the baseline. In this figure, note that the points are still fairly close the diagonal, like in Fig. 1, but in this case data variance is a bit larger as restart strategies may significantly impact on overall solving time.

Figure 6a presents the same results of Table 6 in a graphical way.

Figure 6b illustrates data distribution of the number of solved instance w.r.t. increasing execution time. All the variants are plotted on the same survival chart to facilitate data comparison. In the context of restart strategies, despite a similar progression among data distributions, it is possible to notice a faster convergence of our baseline technique, which lies above all the other lines for roughly the first third of execution times. After this, all the techniques, with the exception of the windowed approach, fight for the lead, with the variable activation one proving the best, in terms of overall number of solved instances, in the end.

8.2 Experiments on core benchmark set

We noticed that, among the solved instances, a significant number could be handled in less than 10% of the allotted time (i.e., 90 s) by the baseline and all the variants alike. Such subset of benchmarks, that we consider a *core* set for experimental evaluation, is composed of 223 instances, of which 188 are UNSAT and 35 are SAT. All experimental data in this subsection,

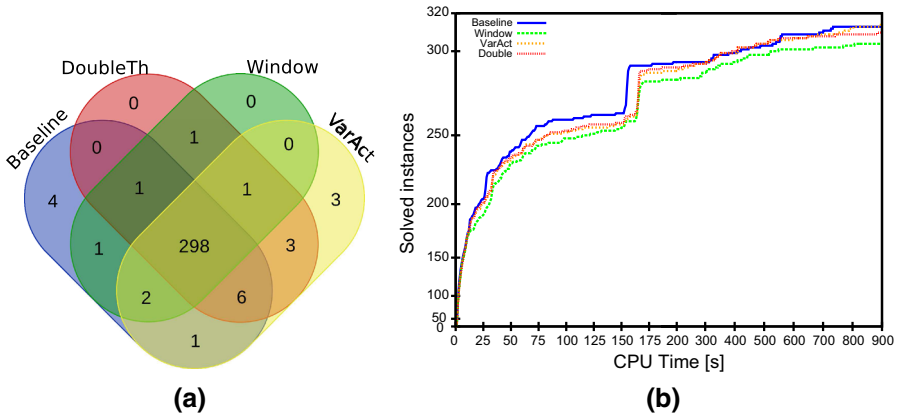

Fig. 6 Detailed comparison of different SAT solver cleanup strategies on the complete benchmark set. **a** is a Venn's diagram illustrating mutual coverage of solved instances among the compared strategies, **b** is a survival plot illustrating distribution of solved instances over CPU time for each compared strategy

Table 7 Comparison of all the tested techniques purged of easy instances w.r.t. to baseline

Configuration	Solved	UNSAT	SAT	New UNSAT	New SAT	Lost UNSAT	Lost SAT	$\Delta_{baseline}$
Generalize spec.	88	37	51	4	1	4	3	−2
Push spec.	89	38	51	4	0	3	2	−1
Gen+Push spec.	92	38	54	4	3	3	2	+2
Variable activation	91	39	52	4	3	2	4	+1
Windowed	81	33	48	2	0	6	5	−9
Double threshold	87	38	49	4	1	3	5	−3
PG	79	31	48	1	1	7	6	−11
T+	97	45	52	8	0	0	1	+7
T	91	38	53	4	2	3	2	+1
Baseline	90	37	53	–	–	–	–	–
Virtual best	108	50	58	13	5	–	–	+18

obtained with PdTRAV, nuXmv and ABC, refer to this set. We propose the same kind of visual representation and data analysis presented in the previous section. Overall, pruning away *easy* problems gives a better evaluation of the improvements obtained.

8.2.1 Experiments with PdTRAV

Table 7 compares the results of all the techniques we experimented with, w.r.t. the baseline, underlying gains and losses for each one. We deem that excluding the bulk of easy instances from the presented results allows for a better representation of the impact of the different techniques in terms of overall achievable coverage.

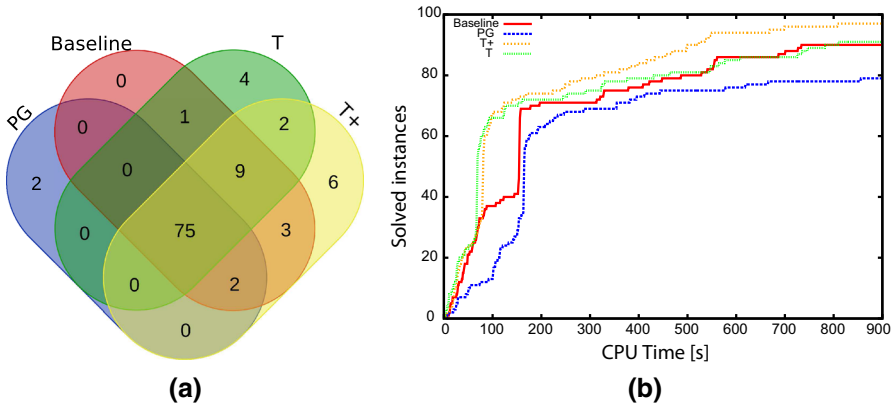

Fig. 7 Detailed comparison of different CNF encodings strategies on non-trivial instances. **a** is a Venn's diagram illustrating mutual coverage of solved instances among the compared strategies, **b** is a survival plot illustrating distribution of solved instances over CPU time for each compared strategy

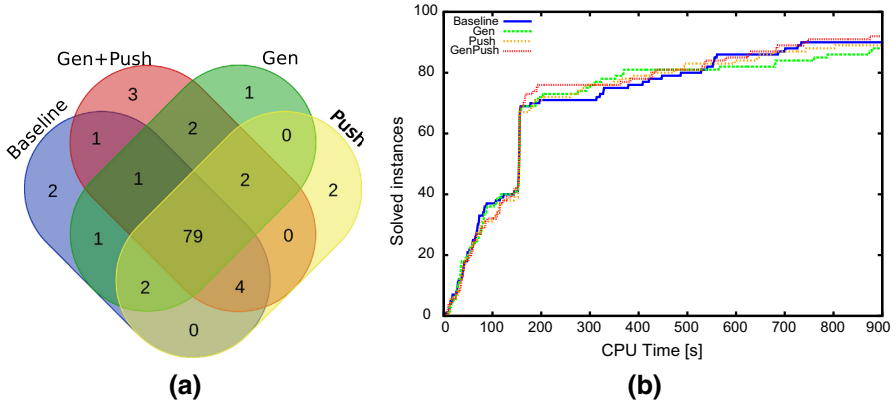

Fig. 8 Detailed comparison of different specialized SAT solvers strategies on non-trivial instances. **a** is a Venn's diagram illustrating mutual coverage of solved instances among the compared strategies, **b** is a survival plot illustrating distribution of solved instances over CPU time for each compared strategy

Figure 7a, b visually represent as a Venn's diagram and as a survival plot, respectively, the data reported in Table 7 for CNF encodings strategies. In this case, the survival plot is created with linear axis given the slope of the plotted curves. This holds for the following survival plots as well.

Figure 8a, b visually represent as a Venn's diagram and as a survival plot, respectively, the data reported in Table 7 for specialized SAT solvers strategies.

Figure 9a, b visually represent as a Venn's diagram and as a survival plot, respectively, the data reported in Table 7 for SAT solver clean-up strategies.

8.2.2 Experiments with different SAT solver tunings

As stated before, IC3 is highly sensitive to the way its SAT solving requirements are handled. Besides choosing among different SAT solvers management strategies, another determining factor for the algorithm's performance is deciding how to tune the internal behaviour of the

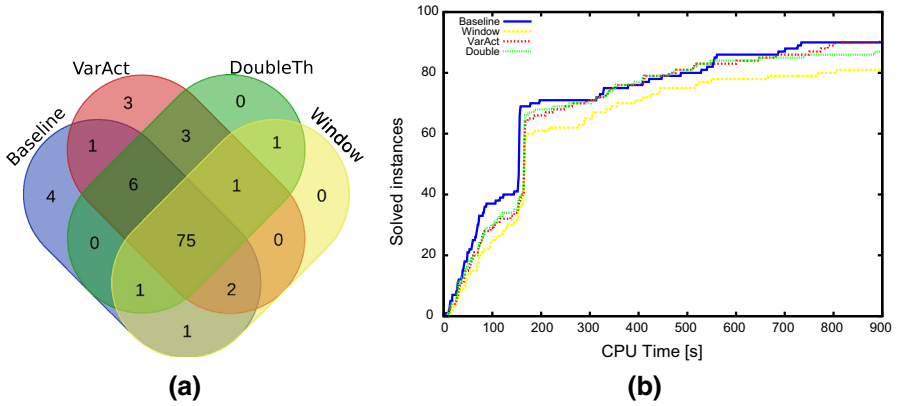

Fig. 9 Detailed comparison of different SAT solver clean-up strategies on non-trivial instances. **a** is a Venn's diagram illustrating mutual coverage of solved instances among the compared strategies, **b** is a survival plot illustrating distribution of solved instances over CPU time for each compared strategy

SAT solver. In order to evaluate the impact of different SAT solver tunings, we experimented with different settings for the clause deletion strategy of MiniSAT. In particular, we are interested in assessing whether or not varying the aggressiveness of the clause deletion strategy, instead or in conjunction with SAT solver restart strategies, could lead to a difference in terms of performance and/or coverage.

We isolated two parameters regulating the aggressiveness of the clause deletion strategy of MiniSAT: the deletion trigger threshold and the deletion ratio. The former defines the percentage of learnt clauses, w.r.t. to the original number of problem clauses, required to trigger a cleanup of the clause database. The latter defines the minimum percentage of learnt clauses to forget at each cleanup. Clauses are removed according to an activation-based heuristics. In MiniSAT the default values for these thresholds are 33 and 50% respectively.

We have run a first set of experiments using different values for those thresholds, on a subset of the harder instances. Our goal was to evaluate the impact or more aggressive and less aggressive deletion strategies on the overall procedure. We selected 20, 33 and 100% as cleanup thresholds and 25, 50 and 75% as deletion ratio, for a total of nine different experimental configuration. Experiments were run on PdTRAV's baseline configuration after changing the settings of the underlying MiniSAT solver.

Preliminary results proved to be rather consistent with our baseline. Taking into account solved instances, the execution times improved of more than 10 for 4.91% of the test cases, whereas they worsened of more than 10 for 5.72% of the cases. Almost 15% of previously solved instances, though, resulted in execution timeout. Nevertheless, the most favourable results derived from anticipated cleanups (using cleanup triggers of 20 or 33% instead of 50%) but less aggressive deletion (just 50% instead of 75%). Overall, the best combination proved to be 20% as cleanup trigger and 50% deletion ratio, which increased the coverage of our baseline configuration by 2. This seems to suggest that non useful clauses quickly accumulate in the solver's database, but, at the same time, the activity-based heuristic of MiniSAT is effective in recognizing and deleting them.

On these premises, we conducted a second set of experiments, testing our winning combination of cleanup thresholds on the whole set of hard instances. We tested this more aggressive clause deletion strategy, called *early clause deletion* (*ECD*), in conjunction with PdTRAV's baseline configuration as well as the most performing ones among our specialization and restarting strategies. Results are summarized in Table 8.

Table 8 Comparison of different configurations using early clause deletion w.r.t. to baseline

Configuration	Solved	UNSAT	SAT	New UNSAT	New SAT	Lost UNSAT	Lost SAT	$\Delta_{baseline}$
Baseline+ECD	77	29	48	3	0	11	5	−13
Gen+Push spec.+ECD	81	31	50	3	0	9	3	−9
Variable activation+ECD	77	29	48	3	0	11	5	−13
Baseline	90	37	53	–	–	–	–	–

Table 9 Comparison of a subset of SAT solvers management techniques run in nuXmv

Configuration	Solved	UNSAT	SAT	New UNSAT	New SAT	Lost UNSAT	Lost SAT	$\Delta_{baseline}$
Single+spec	102	48	54	1	1	0	0	+2
Multi+spec	96	43	53	3	2	7	2	−4
VarAct	95	42	53	3	3	8	3	−5
Multi	97	42	55	3	4	8	2	−3
Baseline	100	47	53	–	–	–	–	–
Virtual best	109	52	57	5	4	–	–	+9

Introducing early clause deletion it was possible to increase PdTRAV coverage by three more circuits. However, at the same time, the specific variants seemed to perform worse, on average, at least in terms of sheer number of solved instances. Nevertheless, it seems plausible that a more careful tuning of the solver parameters (e.g., devising clause deletion strategies specifically tailored for IC3) may lead to performance improvements. We leave such an investigation as future work.

8.2.3 Experiments with NuXmv

We have conducted a subset of the aforementioned experimental analysis on another state-of-the-art model checking tool, nuXmv, in order to gather more data. Furthermore, nuXmv natively supports both single solver and multiple solvers approaches, whereas the former is not currently available in PdTRAV. This allowed us to inquire further in this direction as well.

We have run nuXmv in both its single and multiple solvers variants, as well as introducing some tweaks to the code to test some of our proposed techniques. We extended its capability to support specialized solvers, in order to test the best configuration we found: Q_{gen} and Q_{push} specialized. We also introduced a different way to handle SAT solvers restart, introducing our variable activation strategies. Implementing also the windowed approach just to be able to test our double threshold strategy, seemed somewhat unnecessarily expensive. Furthermore, as the windowed approach impacts on the way the transition relation is loaded, it could have led us to significantly alter the nature of the tools. This is something we wanted to avoid.

We took into account only a subset of the benchmarks, running experiments only on the subset of non-trivial aforementioned instances.

Table 9 represents a summary of nuXmv runs against its own possible configurations and the virtual best results. It is possible to notice how the single solver implementation of

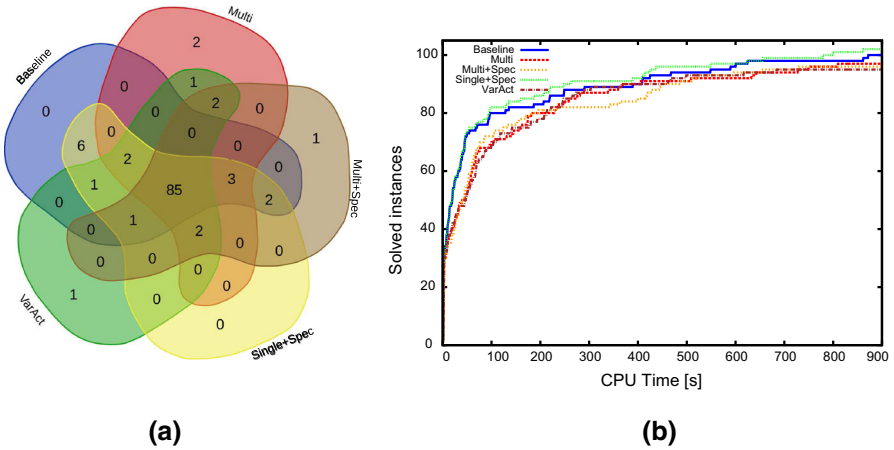

Fig. 10 Experimental analysis with nuXmv. **a** Detailed comparison of different SAT solver management strategies, represented as Venn's diagram, **b** detailed comparison of different SAT solver management strategies, represented as survival plot

Table 10 Comparison of a subset of SAT solvers management techniques run in ABC

Configuration	Solved	UNSAT	SAT	New UNSAT	New SAT	Lost UNSAT	Lost SAT	$\Delta_{baseline}$
Monolithic	95	42	53	5	0	7	4	−6
R1	101	46	55	6	0	4	2	+0
R2	96	42	54	2	1	4	4	−5
Baseline	101	44	57	–	–	–	–	–
Virtual best	110	52	58	8	1	–	–	+9

nuXmv is the most performing one, with a further increase in solved instances when we enable specialized solvers. The differences among all runs are minimal, at least in terms of sheer number of solved instances, but is still possible to notice a significant increase in term of overall coverage.

To better characterize the peculiarity of each tool and configuration, we use once more Venn's diagrams. At the same time, we plot as a survival plot the number of solved instances over increasing solving time, to provide a feedback concerning techniques convergence.

Figure 10a graphically illustrates the regions of overlap among techniques as well as the more interesting unique solves. Furthermore, it is possible to infer the grade of orthogonality among tools and configurations, stemming from the subsets of mutually unsolved instances.

Figure 10b graphically illustrates as a survival plot the number of solved instances over increasing verification time.

8.2.4 Experiments with ABC

We have conducted a subset of the aforementioned experimental analysis on ABC, as well. Following the same strategies used for nuXmv, we have excluded from further experimentation the subset of trivial instances, focusing on the more interesting ones.

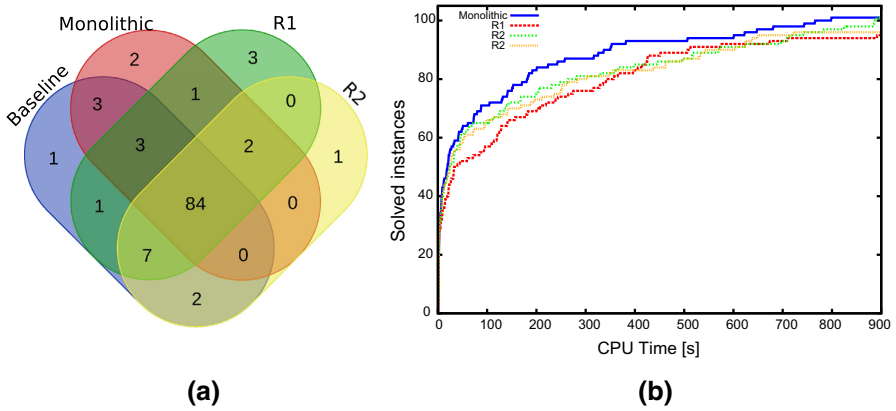

Fig. 11 Experimental analysis with ABC. **a** Detailed comparison of different SAT solver management strategies, represented as Venn's diagram, **b** detailed comparison of different SAT solver management strategies, represented as survival plot

Given the nature and structure of ABC, it was possible to run just a subset of the intended experiments, mainly focusing on restart strategies and transition relation loading.

Table 10 reports in row *Baseline* the default setting of ABC. Row *Mono* shows a run without dynamic transition relation clause loading (i.e., ABC with a monolithic approach). Row *R1* shows a different period for solver clean-up (1000 variables instead of 300) whereas row *R2* shows a different restart strategy, mirroring the aforementioned variable activation approach.

Figure 11a graphically illustrates the regions of overlap among techniques as well as the more interesting unique solves. Furthermore, it is possible to infer the grade of orthogonality among tools and configurations, stemming from the subsets of mutually unsolved instances.

Figure 11b graphically illustrates as a survival plot the number of solved instances over increasing verification time.

Overall, results are in line with previous tools, showing that an increase in the number of solved problems (from 101 to 110), which indicate a potential improvement for a portfolio-based tool, able to concurrently exploit multiple settings.

9 Conclusions

The paper presents a detailed characterization and analysis of SAT queries posed by IC3, taking into account a subset of the most recent HWMCC benchmarks available. We also discuss new ideas for solver allocation, loading and/or restarting. Experimental evaluation, performed considering three different state-of-the-art academic tools, shows lights and shadows as no breakthrough or clear winner emerges from the new ideas. Still, it is possible to appreciate the contribution each technique could bring to a portfolio-based model checking environment.

PG encoding have shown to be less effective than expected. This is probably because the benefits introduced in terms of loaded formula size are overwhelmed by the supposed worst propagation behaviour of that formula, whereas finer grained encodings surpassed initial expectations.

The use of specialized solvers seems to be effective when a static clean-up heuristic is used.

Our experiments showed that, when a dynamic clean-up heuristic is used, IC3 performance can be improved by taking into account both deactivated clauses and irrelevant portions of previously loaded cones.

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 set ups) is definitely higher. Evermore so if we focus on harder instances, rather that let the bulk of trivial ones obfuscate interesting cases with their number.

So we believe that, despite the need for more effort in implementation, experimentation, and detailed analysis of case studies, this work contributes to the discussion of new developments in research related to IC3. The lack of a winner-take-all technique is in line with many related works and engineering efforts in the area of SAT-based model checking, and specifically IC3. As a matter of fact, any specific optimization can improve on a set (or family) of model checking problems, while losing ground on other ones (as clearly confirmed by our experiments). Overall, given the intrinsic difficulty of characterizing families of benchmarks and relating them to best working techniques and setups, the improved portfolio can be considered as the real final outcome.

Whereas other papers explore the area of algorithmic improvements within IC3 algorithms, or within SAT solvers, we focused on managing IC3 SAT queries in order to improve performance, given a state-of-the-art general purpose SAT solver.

As future works we plan to follow two directions: devise better SAT solving management strategies and shift the focus on internal details of SAT solvers. The former requires to keep investigating in the same direction of this paper, in order to identify a subset of winning SAT solver management schemes while considering solvers as black boxes. A possible development in this direction would be to consider a more flexible incremental SAT interface capable of deleting clauses without the need of activation variables. This can be done by taking advantage of resolution-based solvers as shown in [27], in the context of MUS extraction. The latter requires to investigate in more depth how to tune a SAT solver or how to tailor its internal procedures to best fit IC3 needs. A possible follow-up work in this direction could be to take into account different SAT solver implementations or solvers based on different forms of presenting the satisfiability problem. In particular it would be interesting to investigate the use of hybrid SAT solvers such as CirCUs [28] in this context.

References

1. Cabodi G, Mishchenko A, Palena M (2013) Trading-off incrementality and dynamic restart of multiple solvers in IC3. DIFTS, Portland
2. Bradley AR (2011) SAT-based model checking without unrolling. VMCAI, Austin
3. Biere A, Jussila T (2008) The model checking competition web page. http://fmv.jku.at/hwmcc. Accessed Feb 2017
4. Eén N, Mishchenko A, Brayton RK (2011) Efficient implementation of property directed reachability. FMCAD, Austin, pp. 125–134. ISBN:978-0-9835678-1-3
5. Biere A, Cimatti A, Clarke EM, Fujita M, Zhu Y (1999) Symbolic Model Checking using SAT procedures instead of BDDs. In: Proceedings of the 36th design automation conference on New Orleans, Louisiana. IEEE Computer Society, June 1999, pp 317–320. DOI:10.1145/309847.309942
6. Sheeran M, Singh S, Stålmarck G (2000) Checking safety properties using induction and a SAT solver. In: Hunt WA, Johnson SD (eds) Proceedings of the formal methods in computer-aided design series, LNCS, vol 1954. Springer, Austin, November 2000, pp 108–125. ISBN:3-540-41219-0

7. Bjesse P, Claessen K (2000) SAT-based verification without state space traversal. In: Proceedings of the formal methods in computer-aided design, series, LNCS, vol 1954. Springer, Austin, pp 372–389. ISBN:3-540-41219-0
8. McMillan KL (2003) Interpolation and SAT-based model checking. In: Proceedings of the computer aided verification, series. LNCS, vol 2725. Springer, Boulder, pp 1–13. doi:10.1007/978-3-540-45069-6_1
9. Bradley AR (2012) Understanding IC3. In: Cimatti A, Sebastiani R (eds) SAT series Lecture Notes in Computer Science, vol 7317. Springer, pp. 1–14. doi:10.1007/978-3-642-31612-8_1
10. Cabodi G, Nocco S, Quer S (2011) Benchmarking a model checker for algorithmic improvements and tuning for performance. Form Methods Syst Des 39(2):205–227. doi:10.1007/s10703-011-0123-3
11. Mishchenko A (2007) ABC: a system for sequential synthesis and verification. http://www.eecs.berkeley.edu/~alanmi/abc/. Accessed Feb 2017
12. Cavada R, Cimatti A, Dorigatti M, Griggio A, Mariotti A, Micheli A, Mover S, Roveri M, Tonetta S (2014) The nuxmv symbolic model checker. In: Biere A, Bloem R (eds) Proceedings of the computer aided verification, series. Lecture Notes in Computer Science, vol 8559. Springer International Publishing, pp 334–342. doi:10.1007/978-3-319-08867-9_22
13. Hassan Z, Bradley AR, Somenzi F (2013) Better generalization in IC3. FMCAD, Portland, pp 57–164
14. Vizel Y, Grumberg O, Shoham S (2012) Lazy abstraction and SAT-based reachability in hardware model checking. FMCAD, Cambridge, pp 173–181
15. Chockler H, Ivrii A, Matsliah A, Moran S, Nevo Z (2011) Incremental formal verification of hardware. FMCAD, Austin, pp 135–143
16. Griggio A, Roveri M (2015) Comparing different variants of the IC3 algorithm for hardware model checking. IEEE Trans Comput Aided Des. doi:10.1109/TCAD.2015.2481869
17. Järvisalo M, Biere A, Heule M (2012) Simulating circuit-level simplifications on CNF. J Autom Reason 49(4):583–619. doi:10.1007/s10817-011-9239-9
18. Tseitin GS (1983) On the complexity of derivation in propositional calculus. In: Automation of reasoning: 2: classical papers on computational logic 1967–1970. pp 466–483. doi:10.1007/978-3-642-81955-1_28
19. Eén N, Mishchenko A, Sörensson N (2007) Applying logic synthesis for speeding up SAT. Theory and applications of satisfiability testing, series, LNCS, vol 4501. Springer, Lisbon, May 28–31 2007, pp 272–286. doi:10.1007/978-3-540-72788-0_26
20. Bradley AR, Manna Z (2007) Checking safety by inductive generalization of counterexamples to induction. FMCAD, Austin, pp 173–180. doi:10.1109/FAMCAD.2007.15
21. Eén N, Sörensson N (2016) The minisat SAT solver. http://minisat.se. Accessed Feb 2017
22. Moskewicz M, Madigan C, Zhao Y, Zhang L, Malik S (2001) Chaff: engineering an efficient SAT solver. In: Proceedings of the 38th design automation Conference on Las Vegas, Nevada. IEEE Computer Society, June 2001. doi:10.1145/378239.379017
23. Eén N, Sörensson N (2003) Temporal induction by incremental sat solving. Electron Notes Theor Comput Sci 89(4):543–560. doi:10.1016/S1571-0661(05)82542-3
24. Björk M (2009) Successful SAT encoding techniques. In: Journal on satisability, boolean modeling and computation. Addendum, IOS Press
25. Plaisted DA, Greenbaum S (1986) A structure-preserving clause form translation. J Symb Comput 2(3):293–304. doi:10.1016/S0747-7171(86)80028-1
26. Eén N (2007) Practical sat: a tutorial on applied satisfiability solving. Slides of invited talk at FMCAD, 2007. www.cs.utexas.edu/users/hunt/FMCAD/2007/presentations/practicalsat.html. Accessed 02 May 2016
27. Nadel A, Ryvchin V, Strichman O (2013) Efficient MUS extraction with resolution. FMCAD, Portland, pp 197–200
28. Jin H, Somenzi F (2005) CirCUs: a hybrid satisfiability solver. In: Theory and applications of satisfiability testing. Vancouver, BC, Canada, 2005. pp 211–223. doi:10.1007/11527695_17