Comparing Different Variants of the IC3 Algorithm for Hardware Model Checking

Alberto Griggio and Marco Roveri
Fondazione Bruno Kessler
Via Sommarive 18, I-38050, Trento, Italy
{griggio,roveri}@fbk.eu

***Abstract*—IC3 is one of the most successful algorithms for hardware model checking. Since its invention in 2010, several variants of the original algorithm have been published, proposing optimizations and/or alternative procedures for the many different steps of the algorithm. In this paper, we present a thorough empirical comparison of a large set of optimizations and procedures for the steps of IC3, considering "high-level" variants/extensions to the basic algorithm, as well as "low-level" optimizations/configuration settings. We implemented each of them in the same tool, optimizing the implementations to the best of our knowledge. This enabled for a flexible experimentation in a controlled environment, and to gain new insights about their most important differences and commonalities, as well as about their performance characteristics. We conducted the experiments using as benchmarks the problems used in the last four editions of the hardware model checking competition. The analysis helped us to identify several settings leading to significant improvements wrt. a basic implementation of IC3.**

I. Introduction

THE results of the last four Hardware Model Checking Competitions (HWMCC) [1] clearly show that IC3 [2], a bit-level SAT-based symbolic model checking algorithm invented by Bradley in 2010, is very efficient on analyzing (both satisfiable and unsatisfiable) problems of industrial size.

IC3 is a sophisticated algorithm, and its performance can be significantly affected by a number of parameters and implementation choices for its main components. Because of this, since its introduction in [2], several variants of its different main components have been proposed, implemented and integrated independently in the many available model checkers (e.g. ABC [3], [4], IIMC [5], PDTRAV [6]).

Each implementation differs from the others by the possible use of different SAT solvers, different programming languages, different low-level algorithms and different data structures. The majority of these differences may seem insignificant at a first sight. However, they may spoil some insights about the efficiency of the variants, and make the comparison among the different variants difficult.

This paper provides the first systematic independent comparison of the different variants of the IC3 algorithm, considering many possible configurations that may affect its efficiency, including some that have not been thoroughly analyzed before. The expectation is to better understand the impact of each variant and thus to gather further insights about the algorithm.

We implemented the most important variants of the main components of IC3 within the NUXMV verification platform [7]. Our implementation allows to control the various configuration parameters by making it possible to enable or disable the considered variants and optimizations without causing any overhead.

We performed a thorough experimental evaluation using as benchmarks for the comparison all the single track benchmarks of the last four editions of the HWMCC [1]. As baseline for conceptually identifying the differences and the impact of the different variants we have chosen a configuration close to the description of the algorithm given in [3].

The experimental evaluation produced several outcomes. First, our results provide an independent and publicly available confirmation of the results discussed in the papers where the considered variants were first introduced. Second, the analysis of the results identified several unexpected impacts, like e.g. the surprising importance of the CNF conversion. Third, we identified a set of best candidate configurations that lead to solve the largest number of problems in the considered resource constraints. Finally, the results show that quite often there is not a clear winner among the different configurations. Indeed, enabling one configuration may result in differences in the number of solved instances wrt. the baseline, as well as in differences in the number of instances gained (i.e. solved by the given configuration but not by the baseline) and lost. Moreover, our analysis shows that the effectiveness of many of the considered parameters varies significantly across the different families of instances in our benchmark set. Considering the virtual best, it becomes evident that a portfolio approach is the one that leads to the best performances in terms of number of problems solved in the given resource bounds.

Finally, we compared the identified best candidate configurations against our baseline, against the "reference" IC3 implementation available at [8], and against the version of IC3 implemented within ABC [3], [4]. The results show that each of our best candidate configurations results in substantial improvements over the "basic" version of the algorithm.

A. Related Work

The work closest to the work presented in this paper is [3], where different configurations are compared for the PDR variants of IC3 implemented within ABC [3], [4]. This paper,

A preliminary version of this paper was presented at the DIFTS'14 workshop (http://fmgroup.polito.it/cabodi/difts2014/).

besides providing an independent evaluation of the findings presented in [9] and in other papers introducing variants to the basic IC3 algorithm, extends the scope of the evaluation along two main directions. First, we consider a larger benchmark suite, including all the single track instances used for the 2011, 2012, 2013 and 2014 editions of the HWMCC. Second, we consider a larger set of configurations, including some not considered before (e.g. different SAT solvers and CNF conversion algorithms, or use of approximated SAT checks).

B. Structure of the paper

This paper is structured as follows. In Sect. II we provide some background concepts. In Sect. III we describe the space of IC3 variants. In Sect. IV we briefly describe our parameterized implementation of the IC3 variants. In Sect. V we describe the evaluation methodology we adopted. In Sect. VI we discuss the results of the analysis and we discuss the lessons learned. Finally, in Sect. VII we draw conclusions and we outline possible future directions.

II. Background

A. Notation

Our setting is standard propositional logic. We denote Boolean formulas with φ, ψ, I, T, P, Boolean variables with x, y, and sets of Boolean variables with X, Y. A literal is a variable or its negation. A *clause* is a disjunction of literals, whereas a *cube* is a conjunction of literals. If s is a cube $l_1 \wedge \ldots \wedge l_n$, with $\neg s$ we denote the clause $\neg l_1 \vee \ldots \vee \neg l_n$, and vice versa. With a little abuse of notation, we sometimes interpret both cubes and clauses as sets of literals (and vice versa). If l is a literal, we denote its corresponding variable with var(l). If $X_1, \ldots, X_n$ are sets of variables and φ is a formula, we might write $\varphi(X_1, \ldots, X_n)$ to indicate that all the variables occurring in φ are elements of $\bigcup_i X_i$. For each variable x, we assume that there exists a corresponding variable x' (the *primed version* of x). If X is a set of variables, X' is the set obtained by replacing each element x with its primed version. Given a formula φ, φ' is the formula obtained by replacing each variable occurring in φ with the corresponding primed variable, whereas $\varphi^{\langle i \rangle}$ denotes the formula obtained by i consecutive applications of priming(that is, $\varphi^{\langle 0 \rangle} \equiv \varphi$ and $\varphi^{\langle i \rangle} \equiv (\varphi^{\langle i-1 \rangle})'$). A *model* μ for a formula φ is an assignment to (possibly a subset of) the variables of φ that makes the formula true. Given two formulas φ and ψ, we denote entailment with $\varphi \models \psi$, meaning that all the models of φ are also models of ψ.

B. Symbolic transition systems

Given a set X of state variables and a set Y of primary input variables, a *transition system* S over X can be described symbolically with two formulas: $I(X)$, representing the initial states of the system, and $T(Y, X, X')$, representing its transition relation. In this paper, unless otherwise specified, we assume that the transition relation T is written as $\bigwedge_{x_j \in X}(x'_j = \tau_j(Y, X))$, as is typically the case for hardware designs.[1] A *state* of S is a cube over X. A *path* of S is a sequence of states $s_0, \ldots, s_n$ such that $s_0 \models I$ and $\exists Y.\ s_{i-1}(X) \wedge T(Y, X, X') \models s_i(X')$ for all i in $1 \ldots n$. That is, s_0 is an initial state of S, and s_i is the result of performing one transition step in S starting from s_{i-1} (for some values of the primary inputs Y).

C. SAT solving

A *SAT solver* is a procedure that can decide the satisfiability of a propositional formula φ (typically assumed to be in Conjunctive Normal Form – CNF), i.e. it returns true iff φ has at least one model. Modern SAT solver implementations typically follow the CDCL (Conflict-Driven-Clause-Learning) architecture [11]. In the following, we abstract from the implementation details of the SAT solver, and we only assume to have the following API[2]:

- is_sat(φ) checks the satisfiability of the input formula φ;
- get_model() retrieves the model computed by the previous is_sat call (only if the result of the last call was true);
- is_sat_assuming(φ, *assumptions*) checks the satisfiability of φ under the given additional assumptions (a list of literals) [13]. Semantically, this is equivalent to is_sat($\varphi \wedge$ *assumptions*), but the implementation is typically more efficient. Moreover, this function also allows to compute an *unsatisfiable core* of the assumptions, i.e. a subset of the assumption literals that is enough to determine the unsatisfiability of the input;
- get_unsat_assumptions() retrieves an unsatisfiable core of the assumption literals of the previous is_sat_assuming call (only if the result of the last call was false).

III. Review of IC3 techniques

A. High-level description of IC3

We follow the formulation of IC3 given in [3], which is known as PDR. Let S be a given transition system described symbolically by $I(X)$ and $T(Y, X, X')$. Let $P(X)$ describe a set of good states. The objective is to prove that all the reachable states of S are good. The IC3 algorithm tries to prove that S satisfies P by finding an inductive invariant $F(X)$ such that:

(i) $I(X) \models F(X)$;
(ii) $F(X) \wedge T(Y, X, X') \models F(X')$; and
(iii) $F(X) \models P(X)$.

To construct F IC3 maintains a sequence of formulas (called *trace*, following [3]) $F_0(X), \ldots, F_k(X)$ such that:

- $F_0 = I$;
- for all $i > 0$, F_i is a set of clauses;
- $F_{i+1} \subseteq F_i$ (thus, $F_i \models F_{i+1}$);
- $F_i(X) \wedge T(Y, X, X') \models F_{i+1}(X')$;
- for all $i < k$, $F_i \models P$.

For $i > 0$, each element F_i of a trace (called *frame*) represents an over-approximation of the states of S reachable in i transition steps or less.

[1]This is e.g. how transition systems are represented in the Aiger [10] standard format, used in the hardware model checking competitions [1].

[2]Notice that, this API is the one typically provided by state-of-the-art SAT solvers such as MINISAT [12].

```
bool IC3($I$, $T$, $P$):
1.  if is_sat($I \wedge \neg P$): return False
2.  $F[0] = I$  # first elem of trace is init formula
3.  $k = 1, F[k] = \top$  # add a new frame to the trace
4.  while True:
        # blocking phase
5.      while is_sat($F[k] \wedge \neg P$):
6.          $c$ = get_state($F[k] \wedge \neg P$)  # $c \models F[k] \wedge \neg P$
7.          if not rec_block($c, k$):
8.              return False  # counterexample found

        # propagation phase
9.      $k = k + 1,\ F[k] = \top$
10.     for $i = 1$ to $k - 1$:
11.         for each clause $c \in F[i]$:
12.             if not is_sat($F[i] \wedge c \wedge T \wedge \neg c'$):
13.                 add $c$ to $F[i+1]$
14.         if $F[i] == F[i+1]$: return True  # property proved

# simplified recursive description (see §III-D)
bool rec_block($s, i$):
1.  if $i == 0$: return False  # reached initial states
2.  while is_sat($F[i-1] \wedge \neg s \wedge T \wedge s'$):
3.      $c$ = get_predecessor($i - 1$, $s'$)  # see §III-C
4.      if not rec_block($c, i - 1$): return False
5.  $g$ = generalize($\neg s$, i)  # see §III-B
6.  add $g$ to $F[1] \ldots F[i]$
7.  return True
```

Fig. 1. High-level description of IC3 (following [3] but using a stack-based cube blocking procedure as opposed to a priority-queue-based one).

```
void generalize-MIC(ref clause $c$, int $i$):
1.  required = {}, fail = 0
2.  for each $l$ in $c$:
3.      cand = $c \setminus \{l\}$
4.      if down(cand, $i$, required):
5.          if size(cand) > min_up_size: $c$ = up($cand, i$)  # see [14]
6.          else: $c$ = cand
7.          fail = 0
8.      else:
9.          if ++fail > max_fail: break
10.         required = required $\cup \{l\}$

bool down(ref clause $c$, int i, set required):
1.  while True:
2.      if is_sat($I \wedge \neg c$): return False  # $I \not\models c$
3.      if not is_sat_assuming($F[i] \wedge T \wedge c$, $\neg c'$):
4.          $c_c = \{l \mid l' \in$ get_unsat_assumptions()$\}$
5.          while is_sat($c_c \wedge I$): pick $l \in c \setminus c_c$, set $c_c = c_c \cup \{l\}$
6.          $c = c_c$
7.          return True
8.      else:
9.          $s$ = get_predecessor($i, \neg c'$)
10.         if $(c \setminus \neg s) \cap$ required $\neq \emptyset$: return False
11.         $c = c \cap \neg s$
```

Fig. 2. MIC-based inductive generalization [14] (max_fail and min_up_size are configuration parameters).

The algorithm proceeds incrementally, by alternating two phases: a blocking phase, and a propagation phase. In the *blocking* phase, the trace is analyzed to prove that no intersection between F_k and $\neg P$ is possible. If such intersection cannot be disproved on the current trace, the property is violated and a counterexample can be reconstructed. During the blocking phase, the trace is enriched with additional clauses, that can be seen as strengthening the approximation of the reachable state space. At the end of the blocking phase, if no violation is found, $F_k \models P$.

The *propagation* phase tries to extend the trace with a new formula F_{k+1}, moving forward the clauses from preceding F_i. If, during this process, two consecutive frames become identical (i.e. $F_i = F_{i+1}$), then a fix-point is reached, and IC3 terminates with F_i being an inductive invariant proving the property.

Let us now consider the lower level details of IC3. The distinguishing feature of IC3 is that the sets of clauses F_i are constructed incrementally, starting from cubes representing sets of states that can reach a bad state in zero or more transition steps. More specifically, in the blocking phase, IC3 maintains a set of *proof obligations* (s, i), where s is a *counterexample to induction (CTI)*, i.e. a cube representing a set of states that can lead to a bad state, and $i > 0$ is a position in the current trace. New clauses to be added to (some of the frames in) the current trace are derived by (recursively) proving that the set s of a pair (s, i) is unreachable starting from the formula F_{i-1}. This is done by checking the satisfiability of the formula:

$$F_{i-1} \wedge \neg s \wedge T \wedge s'. \qquad (1)$$

If (1) is unsatisfiable, and s does not intersect the initial states I of the system, then $\neg s$ is *inductive relative to* F_{i-1}, and it can be used to *strengthen* F_i in order to block the bad state s at i. This is done by first *generalizing* $\neg s$ to a stronger clause g such that $g \models \neg s$ and g is still inductive relative to F_{i-1}, and then by adding g to F_i, thus blocking s at i. If, instead, (1) is satisfiable, then the over-approximation F_{i-1} is not strong enough to show that s is unreachable. In this case, let p be a cube representing a subset of the states in $F_{i-1} \wedge \neg s$ such that all the states in p lead to a state in s' in one transition step, for some values of the inputs Y (i.e. $\exists Y.\ p(X) \wedge T(Y, X, X') \models s(X')$). Then, IC3 continues by trying to show that p is not reachable in one step from F_{i-2} (that is, it tries to block the pair $(p, i-1)$). This procedure continues recursively, possibly generating other pairs to block at earlier points in the trace, until either IC3 generates a pair $(q, 0)$, meaning that the system does not satisfy the property, or the trace is eventually strengthened so that the original pair (s, i) can be blocked. Fig. 1 reports the pseudo-code for IC3. In the rest of the section, we describe in more detail the most important components of the algorithm, illustrating the different variants proposed in the literature.

B. Inductive clause generalization

Inductive generalization is a central step of IC3, that is crucial for the performance of the algorithm. Given a successfully blocked cube s at step i, inductive generalization tries to compute a subset c of s such that $\neg c$ is still inductive relative to F_{i-1}. Adding $\neg c$ to F_i blocks not only the bad cube s, but possibly also many others, thus allowing for a faster convergence of the algorithm.

At a high level, the algorithm for performing inductive generalization works by dropping some literals from the input

```
void generalize-iter(ref clause $c$, int $i$):
1.  done = False
2.  for iter = 1 to max_iter:
3.      if done: break
4.      done = True
5.      for each $l$ in $c$:
6.          $g = c \setminus \{l\}$
7.          if not is_sat($I \wedge \neg g$) and
                not is_sat_assuming($F[i] \wedge T \wedge g$, $\neg g'$):
8.              $c_c = \{l \mid l' \in$ get_unsat_assumptions()$\}$
9.              while is_sat($c_c \wedge I$): pick $l \in g \setminus c_c$, set $c_c = c_c \cup \{l\}$
10.             $c = c_c$
11.             done = False
12.             break
```

Fig. 3. Iterative inductive generalization algorithm (max_iter is a configuration parameter).

```
void generalize-CTG(ref clause $c$, int $i$, int rec_lvl=1):
1.  required = {}, fail = 0
2.  for each $l$ in $c$:
3.      cand = $c \setminus \{l\}$
4.      if down-CTG(cand, $i$, rec_lvl, required):
5.          $c =$ cand, fail = 0
6.      else:
7.          if ++fail > max_fail: break
8.          required = required $\cup \{l\}$

bool down-CTG(ref clause $c$, int i, int rec_lvl, set required):
1.  ctgs = 0
2.  while True:
3.      if is_sat($I \wedge \neg c$): return False
4.      if not is_sat_assuming($F[i] \wedge T \wedge c$, $\neg c'$):
5.          $c_c = \{l \mid l' \in$ get_unsat_assumptions()$\}$
6.          while is_sat($c_c \wedge I$): pick $l \in c \setminus c_c$, set $c_c = c_c \cup \{l\}$
7.          $c = c_c$
8.          return True
9.      else if rec_lvl > max_lvl: return False
10.     else:
11.         $s =$ get_predecessor($i, \neg c'$)
12.         if ctgs < max_ctg and $i > 0$ and not is_sat($I \wedge s$) and
              not is_sat($F[i-1] \wedge T \wedge \neg s \wedge s'$):
13.             ++ctgs
14.             j = $i$
15.             while not is_sat($F[j] \wedge T \wedge \neg s \wedge s'$): ++j
16.             generalize-CTG($\neg s$, j-1, rec_lvl+1)
17.             add $\neg s$ to $F[j]$
18.         else:
19.             ctgs = 0
20.             if $(c \setminus \neg s) \cap$ required $\neq \emptyset$: return False
21.             $c = c \cap \neg s$
```

Fig. 4. CTG-based inductive generalization [9] (max_fail, max_ctg and max_lvl are configuration parameters).

clause $\neg s$ and testing whether the result is still inductive (by checking the satisfiability of (1)), until a stopping criterion is reached (e.g. a fix-point or a resource bound). In the literature, several variants of this basic approach have been proposed. An effective algorithm for computing a *minimal* inductive sub-clause of a given clause was originally proposed in [14]. The algorithm is based on a smart exploration of the lattice of sub-clauses of the input clause. The original IC3 implementation [2] uses an approximated version of such procedure, trading effectiveness for computational efficiency. The pseudo-code of this algorithm is shown in Fig. 2.[3]

```
cube get_predecessor(int $i$, cube $s$):
1.  assert is_sat($F[i-1] \wedge T \wedge \neg s \wedge s'$)
2.  $\mu =$ get_model()
3.  inputs $= \{l \in \mu \mid$ var($l$) $\in Y\}$  # primary inputs
4.  $p = \{l \in \mu \mid$ var($l$) $\in X\}$  # state variables
5.  for iter = 1 to max_iter:
6.      $b$ = is_sat_assuming($T \wedge$ inputs $\wedge \neg s'$, $p$)
7.      assert not $b$
8.      $s =$ get_unsat_assumptions()
9.      if $s == p$: break
10.     else: $p = s$
11. return $p$
```

Fig. 5. SAT-based algorithm for generalization of predecessors (i.e. CTIs) [16] (max_iter is a configuration parameter).

An even cheaper (and conceptually simpler) variant of the procedure was proposed in [3] and is used e.g. by the PDR implementation in ABC [4] and in TIP [15]. The pseudo-code is shown in Fig. 3.

A third variant has been recently proposed in [9]. In this approach, relatively inductive sub-clauses are computed not only from successfully blocked CTIs, but also from other cubes, called *counterexamples to generalization (CTGs)*, that are generated from failed attempts at generalizing some CTIs. The pseudo-code is shown in Fig. 4. In [9], CTG-based generalization was shown to significantly improve the performance of IC3 compared to both the original IC3 procedure and the one of ABC.

C. Predecessors computation

When blocking of a bad cube c fails, a predecessor p of c (wrt. the transition relation T) must be computed. p can be computed simply by taking the values of the state variables X from the model produced by the SAT solver for formula (1). p can then be generalized to represent a set of bad states, rather than a single bad state. In the original IC3 implementation [2], only a simple syntactic generalization based on cone of influence is performed. A significant improvement was proposed in [3], where ternary simulation is used to drop as many literals as possible from p (by setting them to "don't cares") as long as all the states encoded by p are predecessors of c. An algorithm to obtain the same effect using a SAT solver instead of ternary simulation was proposed in [16]. Its pseudo-code is shown in Fig. 5.

D. Proof obligations handling

The pseudo-code of Fig. 1 describes a simple recursive implementation of the management of proof obligations for CTIs. In practice, however, it is more efficient to use a priority queue, ordered by the depth i of proof obligations (s, i). When a CTI (s, i) is successfully blocked, a new proof obligation $(s, i + 1)$ is inserted in the queue, so that IC3 attempts to block s even at later positions in the trace. This not only

[3]A detailed description of the pseudo-code of this and other algorithms shown later is outside the scope of this paper. We refer the reader to the original publications for more information.

allows IC3 to generate counterexamples longer than the length of the trace when disproving properties, but it also generally improves performance for proving properties [2], [3].

E. Target-enlargement of P

The IC3 invariants and pseudo-code described above follow the PDR description of [3], which is slightly different from the original IC3 of [2]. In particular, in [2], *all* the elements of the trace (including the last one) always entail the property P, and the blocking phase continues as long as P is not inductive relative to the last element of the trace. In [3], it is shown that this behavior can be emulated by PDR by preprocessing the input system using a one-step target-enlargement of P, and that this leads to a (small) performance benefit. In principle, the same target-enlargement idea can be generalized and applied for any number $k \geq 1$ of steps.

F. Combination with Lazy Abstraction

The work of [17] has investigated the combination of IC3 with a form of *lazy abstraction*, showing very positive results on industrial hardware verification problems. The main idea of the algorithm is to associate each frame F_i of the IC3 trace with a set $V_i \subseteq X$ of *visible* state variables, such that all the variables not in V (i.e. the invisible ones) are treated as primary inputs. Given the transition relation T written as $\bigwedge_{x_j \in X}(x'_j = \tau_j(Y, X))$, the algorithm considers a different *abstract* transition relation $T_i^\alpha \overset{\text{def}}{=} \bigwedge_{x_j \in V_i}(x'_j = \tau_j(Y, X))$ (over-approximating T) at each position i in the IC3 trace.[4] When a counterexample trace (of length k) is found, it is analyzed to to determine whether it is spurious (i.e. due to the abstraction) or not. In the former case, a *refinement* step is performed which increases the precision of the abstractions T_i^α by enlarging the sets V_i of visible variables, ensuring that no spurious counterexamples of length k exist.

A high-level view of the algorithm of [17] is shown in Fig. 6. In the pseudo-code, the function rec_block_abstract is the same as the rec_block function of IC3 (see Fig. 1 and §III-D), except that the abstract transition relation T_i^α is used instead of T, and state variables not in V_i are considered inputs, whereas IC3_concrete_blocking_phase is the blocking phase of the concrete IC3 (see Fig. 1).

The counterexample analysis and refinement is based on a variation of the blocking phase of IC3. When successful, the function strengthens the current trace until all the counterexamples of length k are blocked. The sets of visible variables V are then refined by identifying, for each position j in the trace, the subset of conjuncts of T of the form $x'_i = \tau_i(Y, X)$ that are needed to ensure that F_j is an over-approximation of the image of F_{j-1} also in the abstract space. The identification of the needed variables is based on the computation of unsatisfiable cores of assumptions [17].

Besides the original IC3-based refinement procedure of [17], here we consider also another variant, based on Bounded Model Checking (BMC), which to the best of our knowledge

[4]For technical reasons (see [17]), the sets of visible variables are such that $V_{i+1} \supseteq V_i$ for all i.

```
**bool** IC3_lazy_abstraction($I$, $T$, $P$):
  **if** is_sat($I \wedge \neg P$): **return** False
  $F[0] = I,\ k = 1,\ F[k] = \top$
  $V[0] = \{x \mid x \in P\}$  # *only variables in $P$ are visible initially*
  **while** True:
      **while** is_sat($F[k] \wedge \neg P$):
          $c$ = get_state($F[k] \wedge \neg P$)
          **if not** rec_block_abstract($c$, $k$, $V[k-1]$):
              **if** refine_abstraction($k$): **break**
              **else**: **return** False
      $k = k + 1,\ F[k] = \top$
      $V[k-1] = V[k-2]$
      **for** $i = 1$ **to** $k - 1$:
          **for each** clause $c \in F[i]$:
              **if not** is_sat($F[i] \wedge \neg c \wedge T_{i-1}^\alpha \wedge c'$):
                  add $c$ to $F[i+1]$
          **if** $F[i]$ == $F[i+1]$: **return** True

**bool** refine_abstraction($k$):
  old$F$ = $F$
  **if not** IC3_concrete_blocking_phase($F$, $k$):
      **return** False  # *found concrete counterexample*
  $\varphi = \bigwedge_{x_i \in X}(l_i \leftrightarrow (x'_i = \tau_i(Y, X))$
  $A = \{l_i \mid l_i \in \varphi\}$
  **for** $j = 0$ **to** $k$:
      **if** $F[j]$.size() > old$F[j]$.size():  # *refinement at i*
          b = is_sat_assuming($\varphi \wedge F[j-1] \wedge \neg F[j]$, $A$)
          **assert not** b
          **for each** $x_i$ **in** $X$:
              **if** $l_i \in$ get_unsat_assumptions():
                  add $x_i$ to $V[j] \ldots V[k-1]$
   **return** True
```

Fig. 6. High-level description of IC3 with lazy abstraction [17].

```
**bool** refine_abstraction_BMC($k$):
  $\varphi = \bigwedge_{x_i \in X}(l_i \leftrightarrow (x'_i = \tau_i(Y, X)))$
  $A = \{l_i \mid l_i \in \varphi\}$
  **if** is_sat_assuming($I^{\langle 0 \rangle} \wedge \bigwedge_{j=0}^{k-1} \varphi^{\langle j \rangle} \wedge \neg P^{\langle k \rangle}$, $\bigwedge_{j=0}^{k-1} A^{\langle j \rangle}$):
      **return** False  # *found concrete counterexample*
  **else**:
      **for each** $x_i$ **in** $X$:
          **for** $j = 0$ **to** $k - 1$:
              **if** $l_i^{\langle j \rangle} \in$ get_unsat_assumptions():
                  add $x_i$ to $V[j] \ldots V[k-1]$
   **return** True
```

Fig. 7. BMC-based refinement for IC3 with lazy abstraction.

has not been considered before. The pseudo-code is reported in Fig. 7. Similarly to the original refinement of [17], also the BMC-based algorithm uses unsatisfiable cores in order to identify the state variables that must be made visible at each position j of the trace. However, differently from the original algorithm, the check for counterexamples of length k is performed with BMC, rather than with a sub-IC3 call.

IV. A PARAMETERIZED IC3 IMPLEMENTATION

It is well-known that comparing separate implementations of similar algorithms within different model checking tools is somewhat problematic: different tools typically differ in many ways (e.g. programming language, data structures and basic routines used, front-ends) which can have a significant impact on their relative performance, and can make it very difficult to perform a fair analysis of the effectiveness of a given variant of an algorithm. Thus, in order to conceptually and effectively compare the characteristics and the impact of the possible variants of IC3, we implemented all of them in the same tool. As a basis for our implementation we took the NUXMV verification platform [7]. Our implementation is in C++, and its source code is available at https://nuxmv.fbk.eu/tests/ic3-eval. We have implemented all the variants of the high-level components of IC3 described in the previous section (§III-B–§III-E) to the best of our understanding, using both the literature describing them and (when available) the original source code as a reference. We have distinguished the algorithm configuration parameters in two main categories: *high-level* and *low-level*.

The high-level parameters are those corresponding to the techniques described in Sections III-B–III-F:

- We consider six different variants for inductive generalization, namely:
  - the original IC3 procedure of [2] (*indgen-ic3*), using the MIC algorithm of Fig. 2 with the following parameter values: $\mathsf{min\_up\_size} = 25, \mathsf{max\_fail} = 3$;
  - the simple iterative algorithm of Fig. 3, with $\mathsf{max\_iter} = +\infty$ (*indgen-iter*);
  - the iterative algorithm of Fig. 3 with $\mathsf{max\_iter} = 1$, as done in the original PDR [3] (*indgen-pdr*);
  - the CTG-based algorithm of Fig. 4 [9] with the settings used also in [9], namely $\mathsf{max\_fail} = +\infty, \mathsf{max\_ctg} = 3, \mathsf{max\_lvl} = 1$ (*indgen-ctg*);
  - the MIC-based algorithm of Fig. 2 with $\mathsf{min\_up\_size} = +\infty, \mathsf{max\_fail} = +\infty$ (*indgen-down*), which uses the same strategy of *indgen-ctg* for exploring the lattice of subclauses of the input clause c, but without using CTGs;
  - a configuration in which relative induction is not used at all (*norelind*), and a simple implication check of the form $F_{i-1} \wedge T \wedge s'$ is used instead of (1) for blocking CTIs (generalization is performed with the iterative algorithm with $\mathsf{max\_iter} = +\infty$).
- For the computation of predecessors, we consider the SAT-based generalization procedure of Fig. 5 [16] with $\mathsf{max\_iter} = +\infty$ (*pre-gen*), and the cone of influence procedure of [2] (*pre-basic*).
- For the management of proof obligations, we use either a priority queue (*queue*) or a stack (*stack*).
- We consider three variants of target-enlargement for the property P, namely no enlargement (*unroll-0*), like in the original PDR [3], 1-step enlargement (*unroll-1*), and a more aggressive 4-step enlargement (*unroll-4*), inspired by the implementation of the TIP model checker [15].
- Regarding the combination of IC3 with lazy abstraction, besides the original IC3 performing no abstraction (*noabs*), we consider the two variants described in §III-F, namely the one of [17] which uses a sub-IC3 for performing abstraction refinement (*absref-ic3*, Fig. 6), and a variant that uses BMC for refinement (*absref-bmc*, Fig. 7).

Finally, as a further high-level parameter, we consider also the impact of preprocessing the transition system using sequential simplification techniques, commonly used by state-of-the-art model checkers, before invoking IC3. In particular, in our preprocessing configuration (*preproc*) we apply two simple techniques used (in various forms) by several tools, namely 2-step temporal decomposition [18] and detection of equivalent latches using ternary simulation [19], [20]. Overall, we have 432 possible configurations for the high-level parameters (assuming that they are all independent).

The low-level parameters that we consider and that may affect the performance of IC3 are:

- *SAT solver*: our implementation is based on a generic SAT solver interface that can be instantiated using back-end solvers. Here, we use the latest version of MINISAT [12] available from Github [21], both with (*minisat-simp*) and without (*minisat*) SAT preprocessing enabled, and the latest version of PICOSAT [22] (*picosat*). When using *minisat-simp*, we apply SAT preprocessing once every time the SAT solver is reset (see also below).
- we consider two different algorithms for *CNF conversion*: the standard Tseitin encoding (*cnf-simple*) and the more sophisticated one presented in [23], as implemented in ABC (*cnf-abc*).
- *number of SAT solver instances*: we considered having either a single SAT solver instance for all the frames in IC3 (*onesolver*), or a separate SAT solver instance for each frame (*manysolvers*), as suggested in [3].
- *literal activity*: we can turn on (*activity*) or off (*noactivity*) the ordering of literals based on their activity when performing inductive generalization, as suggested in [2].[5]
- *SAT solver reset*: IC3 implementations make heavy use of *incremental* SAT checks. This feature is not always available in modern SAT solvers, but it can be emulated using solving under assumptions (see e.g. [3]). However, this has the disadvantage of introducing one fresh variable per incremental call in the SAT solver. Over time, such variables can cause a significant degrade in performance. As a solution to this problem, the authors of [3] suggest to destroy and create a fresh SAT solver instance every few hundred incremental SAT checks.
  Destroying the SAT solver periodically might still be beneficial even when the solver provides a built-in incremental interface,[6] because it also resets the internal state of the solver (scores of variables and clauses, saved variable phase, database of learned clauses), which over time might accumulate too much bias towards certain (possibly poor) choices.
  In this paper, we evaluate three different reset strategies: every 200 incremental calls (*sat-reset-200*, a value similar to what suggested in [3]), a less aggressive strategy that destroys the solver every 5000 calls (*sat-reset-5000*), and a strategy that never resets the solver (*no-sat-reset*).
- finally, we also investigate the possibility to perform *approximated* calls to the SAT solver when possible.

[5]The *activity* heuristic works by preferring literals occurring less frequently in the F_i's for removal from a clause c during inductive generalization.

[6]For example, the latest version of MINISAT available from Github [21] (our default SAT solver) supports the retraction of some variables, which can be used to implement an efficient incremental interface without the need of resetting the solver periodically. In our implementation we exploit this feature.

From our measurements (consistent with what reported in [3]), it turns out that satisfiable queries to the SAT solver are much more expensive than unsatisfiable ones. Moreover, only very few decisions are needed to detect unsatisfiability in most cases. Therefore, we introduced an option (*sat-approx*) to use approximated calls to the SAT solver during inductive generalization queries (on formulas of the form (1)), by setting a bound on the number of decisions. If unsatisfiability is not detected before the bound is hit, we treat the result as a failed generalization.[7] This allows to trade the effectiveness of the generalization for the run-time spent in the SAT solver. In our current implementation, we use a static limit of 100 decisions as upper bound.

Overall, we have 144 possible configurations for the low-level parameters.

V. Setup of the comparison

We consider as *baseline* for the experimental evaluation the configuration with the following settings: *indgen-iter*, *pre-gen*, *queue*, *unroll-0*, *noabs*, *minisat*, *cnf-simple*, *onesolver*, *noactivity*, *sat-reset-5000*. The rationale for using this configuration as baseline is that it corresponds to a basic implementation of the algorithm following the description of [3].

In the experimental analysis, we assume all the algorithm configuration parameters as being independent from each other. We activate each of them separately and we analyze its impact on the performance against the baseline. While the assumption of independence might not be true in some cases, it is however necessary in order to avoid the combinatorial explosion of configurations to test.

For the comparison we considered all the 873 single track benchmarks used in the hardware model checking competitions from 2011 to 2014 [1]. These benchmarks come in *families* of related problems. The three main ones, in terms of number of instances, are:

- *6s* are benchmarks generated by the SixthSense verification platform of IBM [24]; this family consists of 318 instances;
- *Beem* are benchmarks derived from the BEEM explicit state model checking benchmark set [25]; the family contains 103 instances;
- *Intel* are benchmarks generated by Intel's formal verification flow; the set contains 60 instances.

We ran the experiments on a cluster running Scientific Linux, equipped with 2.5Ghz Intel Xeon CPUs with 96Gb of RAM. We set up a time limit of 900 seconds, and a memory limit of 6Gb. For the comparison with other implementations we used a time limit of 2 hours.

We concentrate mainly on two metrics for the comparison: the number of problems solved in the given resource limits; the time needed to provide an answer. We choose these metrics since they are the most intuitive to analyze given the assumption of independence of the options. For certain configurations, we also consider other metrics, in order to better explain the results.

The data to reproduce the executed experiments, and all the log files can also be downloaded from https://nuxmv.fbk.eu/tests/ic3-eval.

[7]Note that neither the correctness nor the completeness of IC3 is affected by this, because we still use a complete SAT call for checking whether a CTI can be blocked. We also use complete calls whenever a model must be extracted from the SAT solver, in order to ensure that get_model always produces correct results.

VI. Results

In this section, we first analyze the impact of the high-level parameters, then we analyze the impact of the low-level parameters, and finally we compare our implementation with other ones and we provide an overall discussion.

A. Inductive generalization

We start from inductive generalization, one of the distinguishing features of IC3. Fig. 8 shows the survival plots for the different inductive generalization options, plotting the number (#) of solved instances (y-axis) in the given timeout (x-axis).[8] More information is provided in Table I, where for each configuration we show the # of solved instances, the difference in # of solved instances wrt. the baseline, the number of instances gained (i.e. solved by the given configuration but not by the baseline) and lost, and the total execution time taken on solved instances. Finally, Fig. 9 shows scatter plots comparing each configuration against the baseline. (Each configuration is indicated by the name of the parameter that is changed wrt. the baseline.) In the scatter plots, we distinguish instances from different families using points of different colors and shapes.

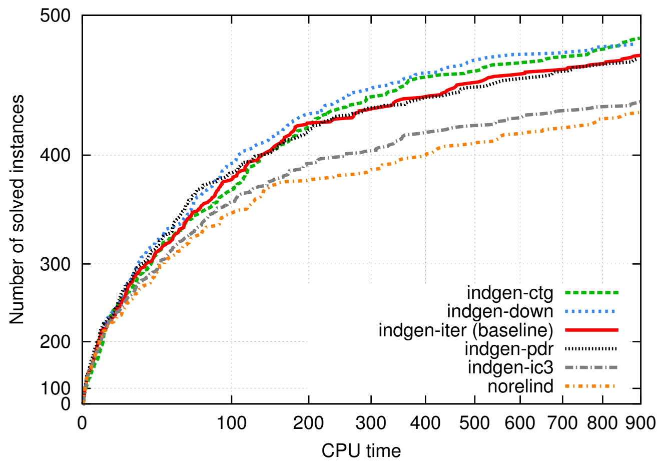

Fig. 8. Survival plots comparing the baseline configuration with configurations using different inductive generalization strategies.

Looking at the results, we can make the following observations:

- The CTG-based technique proposed in [9] (*indgen-ctg*) is the best performing one, solving 9 more instances than our baseline. The survival plots show that CTG-based generalization introduces a overhead for easy instances,

[8]Notice that, in order to improve readability of the plots, we use non-linear scales in both axes that amplify the differences in the curves.

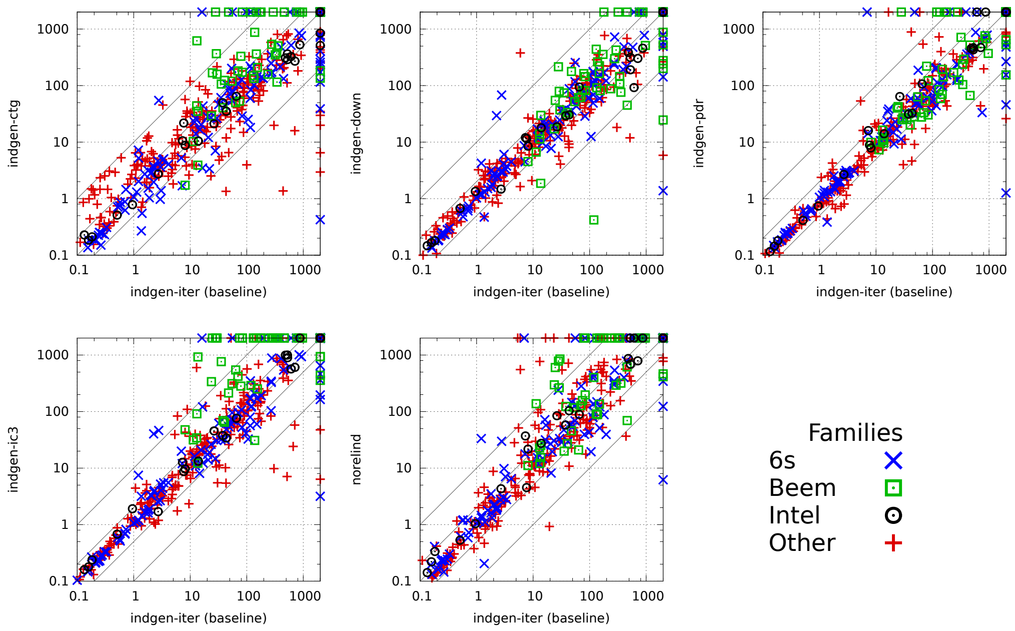

Fig. 9. Detailed comparison of different inductive generalization strategies vs the baseline. The baseline is always on the x-axis. Points above the diagonal indicate better performance of the baseline. Points on the borders indicate timeouts (900 s). Different point types denote different benchmark families.

TABLE I
SUMMARY OF RESULTS FOR THE DIFFERENT INDUCTIVE GENERALIZATION CONFIGURATIONS

Configuration	# Solved	$\Delta_{baseline}$	Gained	Lost	Cumulative time (sec)
indgen-ctg	486	+9	31	22	40729
indgen-down	482	+5	16	11	31716
indgen-iter (baseline)	477	0	0	0	38197
indgen-pdr	472	-5	13	18	34054
indgen-ic3	449	-28	13	41	38860
norelind	434	-43	7	50	35764

but pays off for harder problems. These results are in line with the findings of [9]. It is interesting however to observe that *indgen-ctg* and *indgen-iter* (used in *baseline*) seem to have somewhat complementary strengths, as can be seen also by the scatter plots: there are 22 instances that can be solved with *indgen-iter* but not with *indgen-ctg*. This is mainly due (14 out of 22 instances) to the *Beem* family, on which in general *indgen-iter* performs better.

- The performance of *indgen-down* is quite close to that of *indgen-ctg*. Although the latter solves 4 more instances, *indgen-down* is generally faster, as shown by the survival plots. It is also interesting to observe that, unlike *indgen-ctg*, there is no specific family of benchmarks for which *indgen-down* performs significantly worse than *indgen-iter*.
- The original IC3 generalization strategy (*indgen-ic3*) [2] performs significantly worse than the baseline. As for *indgen-ctg*, this is mostly due to the *Beem* family (26 instances out of the 41 lost are in this set). This is due in part to the cost of applying the up algorithm of [14], but more importantly to the use of max_fail $= 3$ in generalize-MIC (Fig. 2). The purpose of the parameter is to limit the time spent in inductive generalization. It seems that this cutoff value however is too low for our benchmark set, resulting in a significant loss of effectiveness of the whole inductive generalization algorithm.[9] Using max_fail $= +\infty$, as done in *indgen-down*, allows MIC-based generalization to perform at its full potential.
- Using a fixpoint strategy in the iterative algorithm gives only a small advantage compared to the single-round strategy used in PDR: *indgen-iter* solves 5 more instances than *indgen-pdr*, although both the survival plot and the scatter plot show that the two behave very similarly.
- Finally, as expected disabling relative induction (*norelind*) significantly hurts performance.

B. Other high-level parameters

Results for the other high-level parameters we considered are showing in Fig. 10, Fig. 11, and Table II. We make the following observations.

1) Predecessor computation: our results confirm that generalizing predecessors of CTIs (either via ternary simulation [3] or via SAT [16]) is crucial for performance: this is the single most important parameter among those we considered, while evaluating the whole benchmark set. However, different families show quite different behaviors. The *6s* family is the one for which *pre-basic* performs worse: out of the 103 instances lost, 45 belong to this family. On the other hand, *pre-basic* seems to have basically no effect on the *Intel* family.

[9]In our first implementation, we were using the same cutoff value also in *indgen-ctg*. With this setting, also the CTG-based generalization performed worse than *indgen-iter*. We are grateful to Aaron Bradley for fruitful discussions about this.

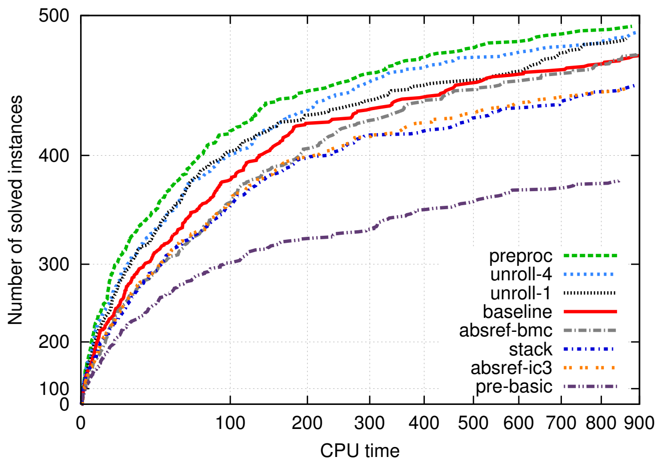

Fig. 10. Survival plots comparing the baseline configuration with configurations changing the high-level parameters.

TABLE II
SUMMARY OF RESULTS FOR THE OTHER HIGH-LEVEL PARAMETERS

Configuration	# Solved	$\Delta_{baseline}$	Gained	Lost	Cumulative time (sec)
preproc	494	+17	34	17	28943
unroll-4	490	+13	33	20	34109
unroll-1	486	+9	24	15	37022
baseline	477	0	0	0	38197
absref-bmc	476	-1	35	36	44239
stack	454	-23	13	36	38241
absref-ic3	452	-25	9	34	34011
pre-basic	381	-96	7	103	31739

2) Proof obligations management: our results confirm that using a simple stack instead of a priority queue for managing proof obligations leads to a visible degrade in performance. The performance degradation happens for both safe and unsafe instances alike, and in fact out of the 36 instances lost by *stack*, 17 are safe and 19 are unsafe. The *Beem* family is the most sensitive to this parameter (as can be seen from Fig.11): there are 11 *Beem* instances solved by *baseline* but not by *stack*, whereas there is only 1 that *stack* could solve but *baseline* could not. Moreover, for the 33 *Beem* instances that both configurations could solve, *stack* is on average 3.13 times slower than *baseline* (with median 2.16 and 9th percentile 5.5).

3) Target-enlargement: Both *unroll-1* and *unroll-4* perform better than *baseline*, both in # of solved instances (see Table II) and in execution time (see Fig. 10). *unroll-4* provides the best performance, solving 13 instances more than *baseline* and reducing the runtime (on instances solved by both configurations) of a factor of 1.4 on average (with median 1.2 and 9th percentile 5.5). However, *unroll-4* results also in a non-negligible number of lost instances (13), and more in general in a significant performance degradation for many problems: more specifically, there are 89 instances for which *baseline* is at least 2 times faster than *unroll-4*, and for 31 of them *baseline* is at least 4 times faster than *unroll-4*. In contrast, *unroll-1* provides a “more stable” (although less significant) improvement, as can be seen from the scatter plots of Fig. 11.

4) Preprocessing: Turning on *preproc* helps significantly both in increasing the # of solved instances and in reducing the run-time. This is typically due to a reduction in the number of state variables in the transition system. Preprocessing is crucial for the *6s* family, as can be seen in the scatter plot of Fig. 11: of the 34 instances gained by *preproc*, 25 are of the *6s* set; moreover, for 26 of the 102 instances solved by both configurations, *preproc* is is at least one order of magnitude faster than *baseline*. However, it might happen that our preprocessor actually increases the number of variables in some cases (when applying temporal decomposition does not allow to discover further simplifications). This happens on 11 of the 17 lost instances (and in 6 of them the increase is $\geq$2x). We remark that our preprocessor only implements two commonly applied techniques (temporal decomposition and detection of equivalent latches); given the importance of preprocessing for some families, experimenting with other techniques proposed in the literature is an interesing direction for future work.

TABLE III
SUMMARY OF RESULTS FOR THE LOW-LEVEL PARAMETERS

Configuration	# Solved	$\Delta_{baseline}$	Gained	Lost	Cumulative time (sec)
cnf-abc	504	+27	30	3	30872
sat-approx	501	+24	34	10	33601
minisat-simp	485	+8	20	12	30920
activity	484	+7	19	12	33703
baseline	477	0	0	0	38197
manysolvers	474	-3	11	14	35820
sat-reset-200	466	-11	4	15	35159
no-sat-reset	462	-15	4	19	32792
picosat	456	-21	10	31	35886

5) Lazy abstraction: The data in Table II seems to suggest that using lazy abstraction does not pay off. This is particularly evident for the *absref-ic3* configuration using (our implementation of) the IC3-based refinement suggested in [17], which solves 25 instances less than *baseline*. The situation is more interesting for the configuration with the BMC-based refinement of Fig. 7 (*absref-bmc*): although this configuration is comparable to *baseline* in terms of # of solved instances, their behavior is quite different on different benchmark families (see Fig. 11). Lazy abstraction (with both refinement algorithms) performs significantly worse than *baseline* on the *Beem* family, losing 14 instances and gaining only 2. In contrast, *absref-bmc* is very effective on the *Intel* family, solving 12 instances of this set that *baseline* could not solve (and losing only 1). In particular, 9 of such 12 instances cannot be solved by any other of the configurations we tried (including the other state-of-the-art implementations we tested, see §VI-D).

C. Low-level parameters

In this section we analyze the impact of the low-level parameters. Fig. 12 shows the “survival plots” for the different configurations. Table III provides a summary of the information and Fig. 13 shows scatter plots comparing each low-level parameter against the baseline. These results lead to the following observations.

1) CNF Conversion: the results show that *cnf-abc* is a clear winner. Enabling this parameter leads to a noticeable general improvement in run-time (see Fig. 13), with (almost)

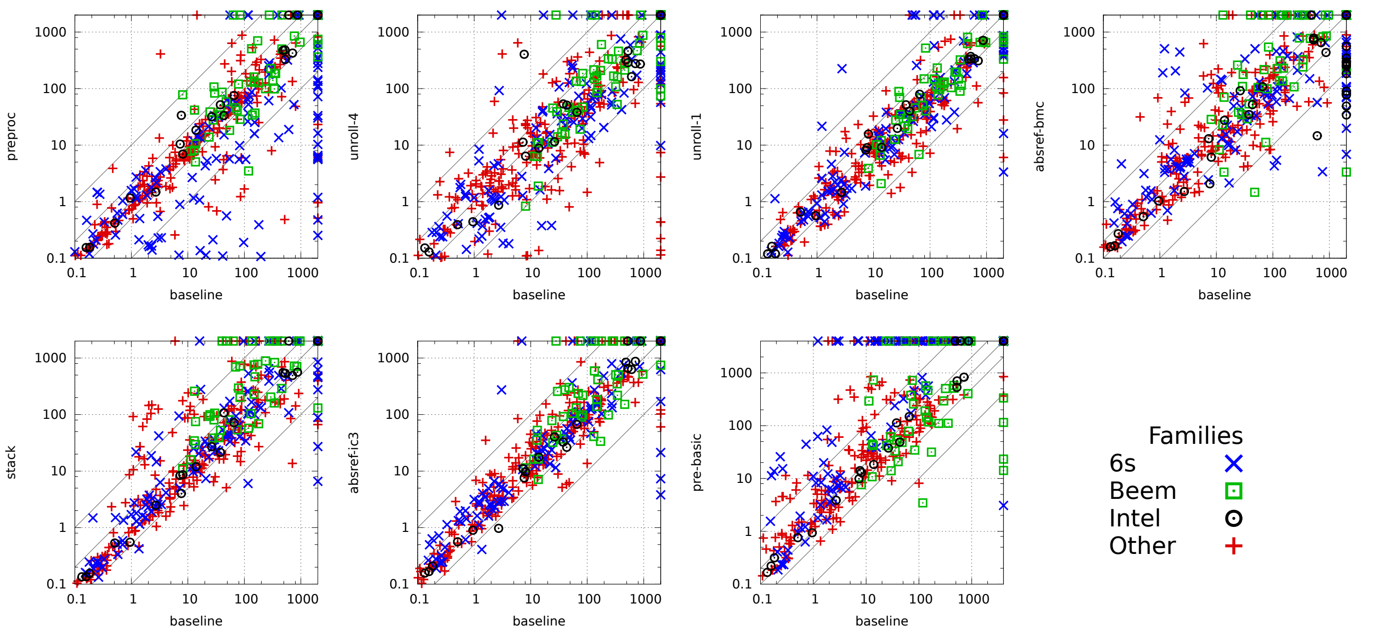

Fig. 11. Detailed comparison of high-level configurations vs the baseline. The baseline is always on the x-axis. Points above the diagonal indicate better performance of the baseline. Points on the borders indicate timeouts (900 s). Different point types denote different benchmark families.

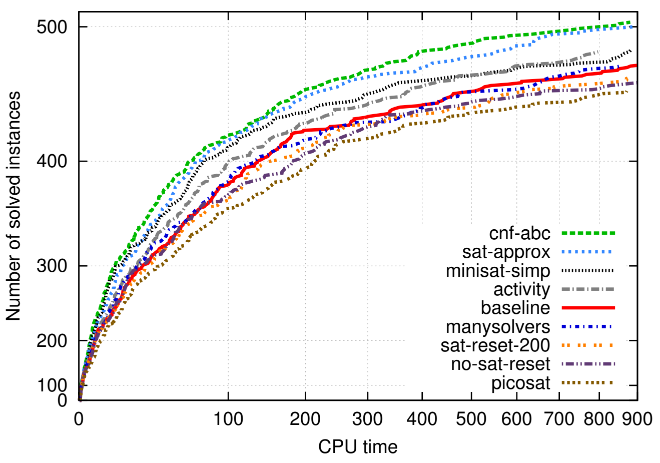

Fig. 12. Survival plots comparing the baseline configuration with configurations changing the low-level parameters.

no instance lost. This is due mostly to a reduction in the SAT solving time: the median of the ratio between the solving time of *baseline* and *cnf-abc* (on instances solved by both) is 2.07, the mean 4.66 and the 9th percentile 4.63. However, also the number of generated cubes decreases slightly (median 1.0, mean 1.30). This is a somewhat surprising result. Indeed, in all the analysis performed so far (to the best of our knowledge) the importance of the CNF conversion for the performance of IC3 was not clearly highlighted.

2) *Approximated SAT:* the results show that enabling *sat-approx* leads to general improvements. The logs show a general decrease in the time spent in the SAT solver for satisfiable queries, and (as a consequence) in the overall SAT solving time: the median of the ratio between the solving time of *baseline* and *sat-approx* for satisfiable queries is 1.71, the mean 3.37 and the 9th percentile 5.67; a similar trend is also shown for for the overall SAT solving time, with median 1.65, mean 2.84, and 9th percentile 4.13. Yet, on most instances using approximated checks does not seem to have a negative impact on inductive generalization: both the # of generated cubes and the length of the IC3 trace do not vary much on average between *baseline* and *sat-approx*. There are however cases in which the approximated queries hurt: in the 10 instances lost by *sat-approx*, on 6 cases the # of generated cubes is at least 3 times more than for the *baseline* configuration. Considering the significant gains obtained and the very small amount of code for its implementation, *sat-approx* is probably the most effective of the optimizations we tried. Given the simplicity of our heuristic (use a static limit of 100 decisions, irrespective of the problem size), we believe that these results show that using a more clever form of approximated SAT checks is a promising direction.

3) *Activity:* enabling *activity* gives a small benefit in terms of execution time and # of instances (see Fig. 12), although the scatter plot of Fig. 13 shows no clear trend.

4) *SAT preprocessor:* similarly to the *activity* case, the use of SAT preprocessing (*minisat-simp*) leads to a small improvement in performance (see Fig. 12), but several instances are lost. Surprisingly, using a specialized CNF conversion (*cnf-abc*) seems to be much more effective.

5) *# SAT solvers:* the use of one SAT solver instance per frame shows no benefit; moreover, as expected it leads in general to a significant increase in memory consumption (>3x median, >6x average, almost 100x max).

6) *SAT solver:* the use of PICOSAT as SAT solver leads to a general increase in solving time (median 1.45, mean 3.26), with no significant change in trace length, number of added cubes, or invariant size. We believe that this behavior might

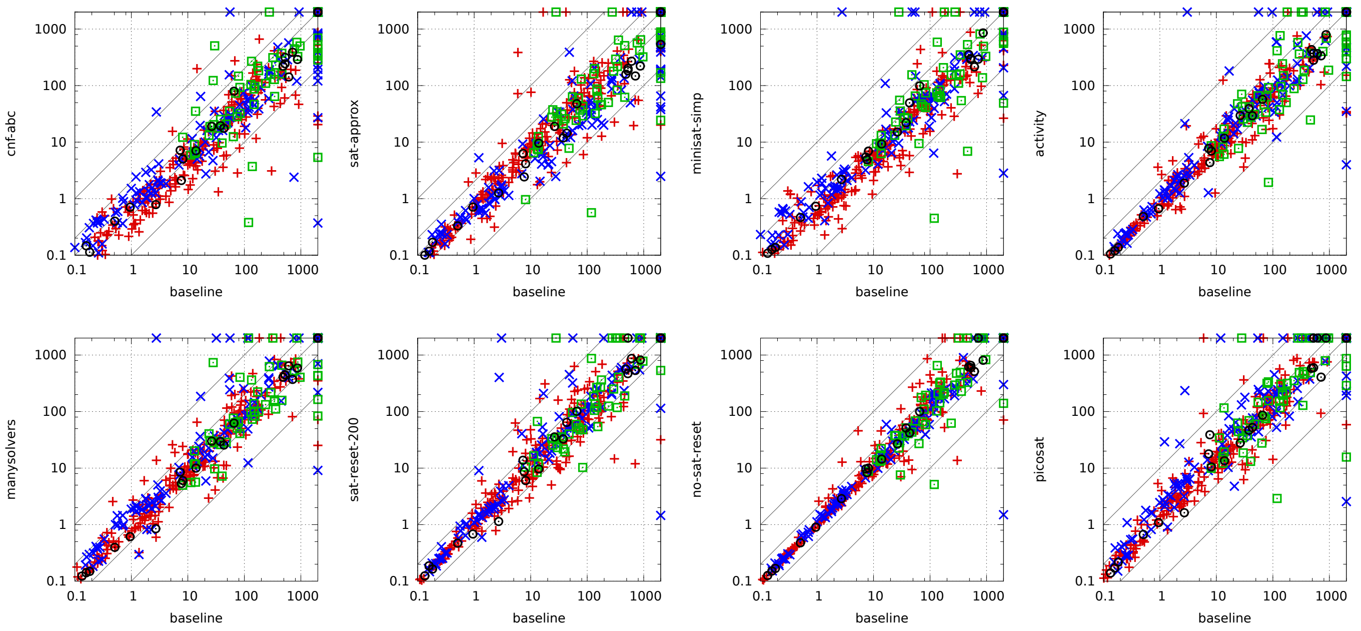

Fig. 13. Detailed comparison of low-level configurations vs the baseline. The baseline is always on the x-axis. Points above the diagonal indicate better performance of the baseline. Points on the borders indicate timeouts (900 s). Different point types denote different benchmark families.

be due to a not as good support to incremental solving as MINISAT. However, in this respect, we envisage to perform a deeper experimentation with different thresholds/heuristics for resetting the solver that may lead to better performance.

7) SAT solver reset: the frequency with which the SAT solver is reset has a visible impact on performance. Both resetting too often (*sat-reset-200*) and not resetting at all (*no-sat-reset*) results in a similar degradation in performance. In both cases, the degradation is more evident for safe instances (10 out of 15 lost for *sat-reset-200*, 14 out of 19 for *no-sat-reset*). Our default strategy (*sat-reset-5000*, used in *baseline*) seems to be a "sweet spot". We should remark however that we could not observe any convincing correlation between the SAT solving time and the performance differences of the tested configurations in this case.

8) Randomness: in order to verify the stability of the results wrt. randomness in the algorithm, we performed an experiment in which we compared different runs of the same configuration with different random seeds. We used the PICOSAT solver, since MINISAT by default does not perform random decisions, and we used a random ordering of literals when performing inductive generalization. Although on individual instances there is indeed a visible impact, the survival plots for the various runs are essentially identical, and the gap in term of # of solved instances is at most 2. Similar results hold also if we only change the random seed in PICOSAT, but use the same fixed ordering for literals (We don't include such plots for lack of space).

D. Comparison with other implementations

We also compared our implementation against other state-of-the-art implementations, namely the "reference" IC3 implementation provided by Bradley [8] (*ic3ref*) and the version of PDR implemented within ABC [3] (*abcpdr*). We also considered four new "best candidate" configurations, obtained by combining the various parameters that lead to an improvement in the # of solved problems wrt. *baseline*. More specifically:

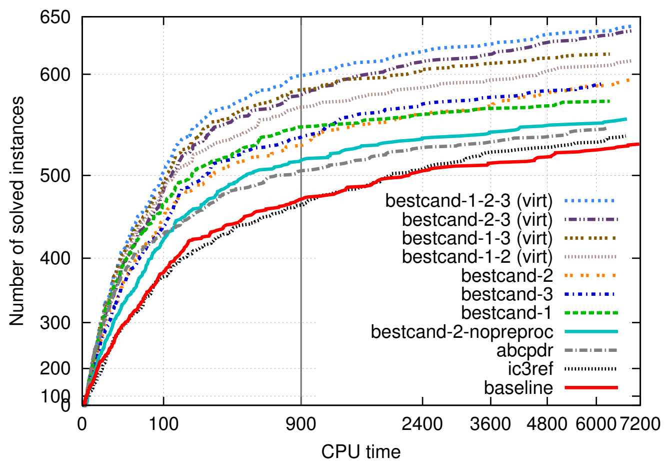

Fig. 14. Survival plots comparing the baseline configurations with the best candidate configuration, virtual best and other implementations.

- *bestcand-1* consists of *baseline* plus *preproc*, *unroll-4*, *cnf-abc*, *activity*, *sat-approx*, and *minisat-simp*;
- *bestcand-2* is obtained from *bestcand-1* by using *indgen-ctg* instead of *indgen-iter*; and
- *bestcand-3* consists of *bestcand-1* plus *absref-bmc*.
- *bestcand-2-nopreproc* is obtained from *bestcand-2* by turning off preprocessing of the input system (*preproc*). This is intended to provide a fair comparison with *abcpdr* and *ic3ref* which do not use any preprocessing.

We also compared against various "virtual best" configura-

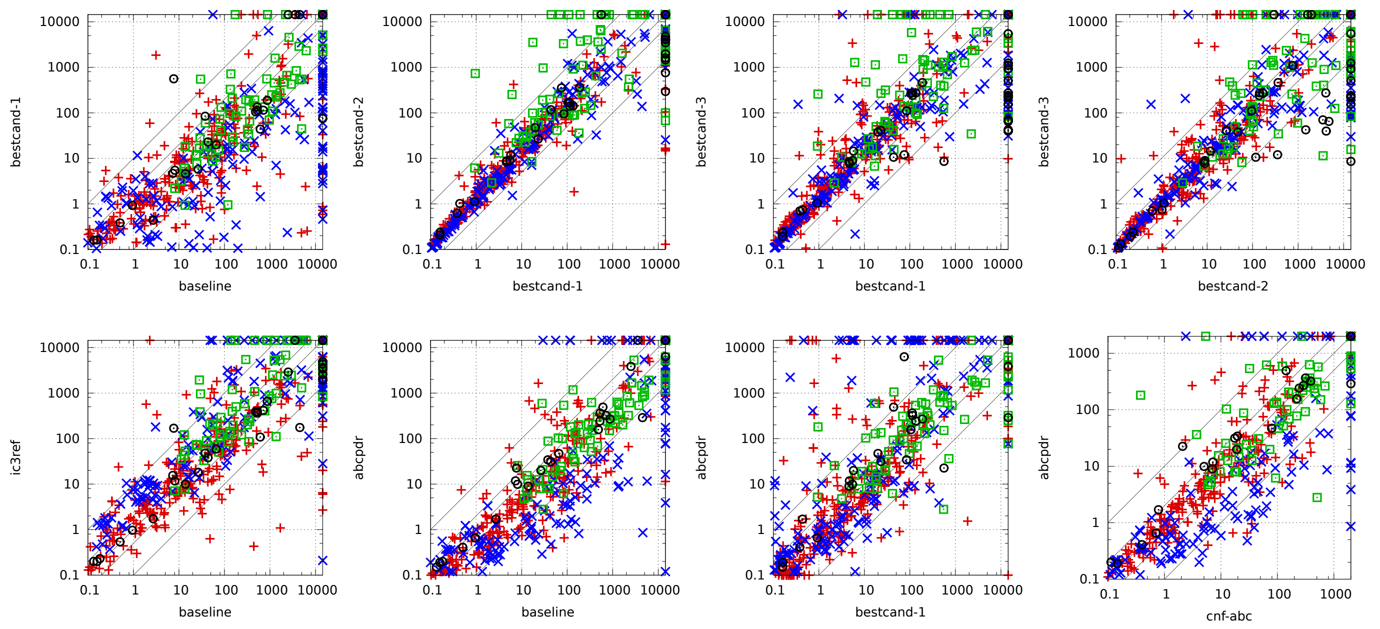

Fig. 15. Comparison of our baseline and best configurations vs ABC and IC3REF. Points on the borders indicate timeouts (2 h, except for *cnf-abc* vs *abcpdr* which is 900 s). Different point types denote different benchmark families.

TABLE IV
SUMMARY OF RESULTS: BEST VS OTHER

Configuration	# Solved	$\Delta_{baseline}$	Gained	Lost	Cumulative time (sec)
bestcand-1-2-3 (virtual)	643	+109	116	7	169220
bestcand-2-3 (virtual)	639	+105	116	11	206418
bestcand-1-3 (virtual)	619	+85	95	10	121598
bestcand-1-2 (virtual)	613	+79	88	9	175423
bestcand-2	596	+62	85	23	223335
bestcand-3	592	+58	91	33	162063
bestcand-1	576	+42	56	14	106056
bestcand-2-nopreproc	559	+25	55	30	157383
abcpdr	550	+16	36	20	156140
ic3ref	542	+8	40	32	240867
baseline	534	0	0	0	219738

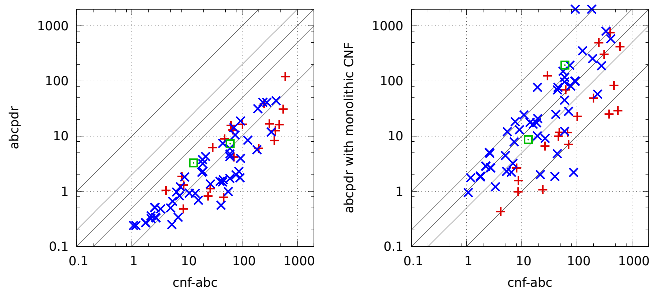

Fig. 16. Comparison of *cnf-abc* and *abcpdr* with and without incremental CNF conversion (on selected instances).

tions, obtained by taking the best result for each individual instance when considering all three best candidates (*bestcand-1-2-3*) or a combination of two of them (*bestcand-1-2*, *bestcand-1-3*, *bestcand-2-3*), in order to evaluate the (potential) benefits of running a portfolio of different configurations in parallel. Moreover, in order to better represents the use of IC3 in real-world settings, for this comparison we used a longer timeout of 2 hours rather than 900 seconds as in the previous cases (and as done in the HWMCC competitions). The results are reported in Fig. 14 (survival plots), in Fig. 15 (scatter plots), and summarized in Table IV. The following observations arise from these results.

- *ic3ref* is slightly ahead of our *baseline* in terms of # of solved problems, especially when considering longer timeouts (as can be clearly seen from the curves in Fig. 14). However, the scatter plot (Fig. 15) and the relatively large numbers of gained (40) and lost (32) instances show that the two implementations appear to be quite complementary.
- The IC3 implementation within ABC is significantly more efficient than our *baseline* implementation. However, in terms of # of solved instances, it is comparable to *cnf-abc*. This seems to indicate that the efficiency of *abcpdr* is due at least in part to the CNF conversion algorithm. From the scatter plot that compares *abcpdr* vs *cnf-abc* (Fig. 15), we see that *abcpdr* is still significantly faster than *cnf-abc*, particularly on instances of the *6s* family. An explanation for this might be the use of a smart incremental CNF conversion algorithm in ABC, which generates clauses on demand, only when needed [3]. Our implementation currently lacks such feature, and always adds all the clauses at once (in a "monolithic" fashion). In order to test the importance of this optimization, we selected the subset of benchmarks for which *cnf-abc* was at least 4 times slower than *abcpdr*, and reran *abcpdr* with incremental CNF conversion turned off. The results, reported in Fig. 16, show that when using the monolithic CNF algorithm the performance gap between *abcpdr* and

cnf-abc gets almost completely closed.

- All our best candidate configurations are a significant improvement wrt. the *baseline* configuration, both in terms of the # of solved problems and in solving time. The configuration solving the largest # of instances is *bestcand-2*, but all the three configurations show different strengths in different benchmark families. For instance, *bestcand-3* is particularly effective on *Intel* benchmarks, thanks to its use of lazy abstraction (see §VI-B5). Similarly, *bestcand-2* generally performs worse than *bestcand-1* on instances of the *Beem* family, for which the CTG-based inductive generalization algorithm does not seem to perform well (see §VI-A). However, both *bestcand-2* and *bestcand-3* result in a non-negligible number of lost instances (23 and 33 respectively) wrt. *baseline*.
- Even without considering the benefits of preprocessing, which are crucial for several instances, our best performing configuration (*bestcand-2-nopreproc*) is quite competitive with other state-of-the-art implementations, solving 9 instances more than *abcpdr* and 17 more than *ic3ref*.
- The three best candidate configurations are different enough that their combination in a (virtual) portfolio gives significant advantages for this set of benchmarks. As can be seen from Fig. 14 and Table IV, all three configurations (*bestcand-1-2-3*) contribute to the performance of the virtual portfolio. However, it is also interesting to see that even with the combination of all three best candidate configurations there are still some (7) lost instances wrt. *baseline*.
- It is interesting to observe that, if we consider the short timeout of 900 seconds used in the HWMCC competitions, the ranking for the three best candidate configurations changes substantially (in fact, it becomes the opposite).

E. Discussion

The results clearly show that the different parameters are not independent. Indeed, by enabling all the most promising options we obtain the best performance in term of # of solved problems and in term of search time. However, the # of solved problems is lower than the sum of additional problems solved if enabled individually wrt. the baseline. In order to better characterize the dependencies, a deeper analysis is needed, possibly considering additional new metrics. The results also show that some "low-level" parameters can have a bigger impact on performance than more sophisticated "high-level" ones. This is indeed the case for the CNF conversion algorithm, or for the new approximated SAT solving heuristic. If we disregard the results for *cnf-abc*, it is evident from the analysis that there is no clear winner among the above approaches, and a given configuration can allow to solve problems that another configuration may fail to solve. The results obtained considering the best configurations confirm that a portfolio approach, with many configurations run in parallel, can give very significant performance advantages for this kind of problems.

VII. Conclusions

We have presented a first systematic comparison of different variants of the IC3 algorithm. We implemented all the variants in a unique tool, to the best of our understanding from the literature and reference implementations (whenever available), and we carried out a thorough experimental evaluation on all the benchmarks used for the latest hardware model checking competitions. In the analysis we considered well-known and state-of-the-art optimizations, and also parameters not yet argument of comparison in existing papers (e.g. the underlying SAT solver, the CNF conversion algorithm). The results showed that the CNF conversion algorithm has a non-negligible impact on the performance of IC3 algorithm, sometimes more evident than other higher-level parameters (e.g. inductive generalization or model preprocessing). We also identified various sets of parameters that allow our implementation to compare very favorably with existing well-optimized implementations. Finally, our results highlight that in most cases different configurations have different and complementary strengths, suggesting that a portfolio approach might give significant advantages.

As future work, we aim at better investigating the relation existing among the different parameters, in order to better understand their respective impact. We also plan to consider other configuration settings that have been omitted from the current analysis, (such as further preprocessing techniques and advanced CNF conversion algorithms, or specialized "IC3-aware" SAT heuristics), and identify an even better set of parameter configurations, possibly with the help of automatic parameter tuning procedures as in [26].

References

[1] "Hardware model checking competition," http://fmv.jku.at/hwmcc/.
[2] A. R. Bradley, "Sat-based model checking without unrolling," in *VMCAI*, ser. LNCS, R. Jhala and D. A. Schmidt, Eds., vol. 6538. Springer, 2011.
[3] N. Eén, A. Mishchenko, and R. K. Brayton, "Efficient implementation of property directed reachability," in *FMCAD*, P. Bjesse and A. Slobodová, Eds. FMCAD Inc., 2011.
[4] R. K. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *CAV*, ser. LNCS, T. Touili, B. Cook, and P. Jackson, Eds., vol. 6174. Springer, 2010.
[5] A. Bradley, F. Somenzi, and Z. Hassan, "IIMC – Incremental Inductive Model Checker," http://ecee.colorado.edu/wpmu/iimc.
[6] G. Cabodi, S. Nocco, and S. Quer, "PdTRAV – politecnico di torino reachability analysis & verification."
[7] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The NUXMV symbolic model checker," in *CAV*, ser. LNCS, A. Biere and R. Bloem, Eds., vol. 8559. Springer, 2014.
[8] A. Bradley, "IC3ref," https://github.com/arbrad/IC3ref.
[9] Z. Hassan, A. R. Bradley, and F. Somenzi, "Better generalization in ic3," in *FMCAD*. IEEE, 2013.
[10] "Aiger," http://fmv.jku.at/aiger/.
[11] J. Marques-Silva, I. Lynce, and S. Malik, "Conflict-driven clause learning sat solvers," *Handbook of satisfiability*, vol. 185, 2009.
[12] N. Eén and N. Sörensson, "An extensible sat-solver," in *SAT*, ser. LNCS, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Springer, 2003.
[13] N. Eén and N. Sörensson, "Temporal induction by incremental SAT solving," *Electr. Notes Theor. Comput. Sci.*, vol. 89, no. 4, 2003. [Online]. Available: http://dx.doi.org/10.1016/S1571-0661(05)82542-3
[14] A. R. Bradley and Z. Manna, "Checking safety by inductive generalization of counterexamples to induction," in *FMCAD*, 2007.
[15] K. Claessen and N. Sörensson, "A liveness checking algorithm that counts," in *FMCAD*. IEEE, 2012.

[16] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo, "Incremental formal verification of hardware," in *FMCAD*, P. Bjesse and A. Slobodová, Eds. FMCAD Inc., 2011.
[17] Y. Vizel, O. Grumberg, and S. Shoham, "Lazy abstraction and SAT-based reachability in hardware model checking," in *FMCAD*, G. Cabodi and S. Singh, Eds. IEEE, 2012.
[18] M. L. Case, H. Mony, J. Baumgartner, and R. Kanzelman, "Enhanced verification by temporal decomposition," in *FMCAD*. IEEE, 2009.
[19] P. Bjesse and J. H. Kukula, "Automatic generalized phase abstraction for formal verification," in *ICCAD*. IEEE Computer Society, 2005, pp. 1076–1082.
[20] M. L. Case, J. Baumgartner, H. Mony, and R. Kanzelman, in *FMCAD*, P. Bjesse and A. Slobodová, Eds. FMCAD Inc., 2011, pp. 109–115.
[21] N. Sörensson, "MINISAT," https://github.com/niklasso/minisat.
[22] A. Biere, "Picosat essentials," *JSAT*, vol. 4, no. 2-4, 2008.
[23] N. Eén, A. Mishchenko, and N. Sörensson, "Applying logic synthesis for speeding up sat," in *SAT*, ser. LNCS, J. Marques-Silva and K. A. Sakallah, Eds., vol. 4501. Springer, 2007.
[24] "Sixthsense," http://researcher.watson.ibm.com/researcher/view_group.php?id=2987.
[25] "Beem," http://anna.fi.muni.cz/models/index.html.
[26] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "Paramils: An automatic algorithm configuration framework," *J. Artif. Intell. Res. (JAIR)*, vol. 36, 2009.

Alberto Griggio is a researcher in the Embedded Systems Unit of the Fondazione Bruno Kessler, Trento, Italy. He received a Ph.D. degree in Computer Science from the University of Trento, Italy in 2009. He joined the Embedded Systems Unit of the Fondazione Bruno Kessler in 2010. His research interests include automated reasoning, SAT and SMT solving, and automated formal verification of hardware and software systems.

Marco Roveri is a senior researcher in the Embedded Systems Unit of the Fondazione Bruno Kessler, Trento, Italy. He received a Ph.D. degree in Computer Science from the University of Milano, Italy in 2002. He joined the Automated Reasoning Division of the Istituto Trentino di Cultura, Trento, Italy, in 1998, first as a consultant, and in 2002 as a full time researcher. In 2008 he was appointed as a senior researcher in the Embedded Systems Unit of the Fondazione Bruno Kessler, Trento, Italy. His research interests include automated formal verification of hardware and software systems, formal requirements validation of embedded systems, and automated model based planning.