Comparing Different Variants of the IC3 Algorithm for Hardware Model Checking

Alberto Griggio Marco Roveri
Fondazione Bruno Kessler, Trento, Italy
{griggio,roveri}@fbk.eu

***Abstract*—IC3 is one of the most successful algorithms for hardware model checking. Several variants of the original IC3 algorithm have been proposed in the literature: each differing from the other for the approach used in implementing the different steps of the algorithm, e.g. pre-processing, clause generalization, the SAT solver used, simplification. In this paper we perform a thorough practical comparison of the different variants by implementing each of them in the same tool pushing as much as possible the corresponding implementations to achieve the best performance. This enabled for a flexible experimentation and to gain new insights, both about their most important differences and commonalities, as well as about their performance characteristics. We performed the experimentation using as bench set the problems used for the last three hardware model checking competitions. The analysis helped to identify which is the best variant.**

I. Introduction

In 2010 Bradley presented a novel bit-level symbolic model checking method, named IC3 [1], that turned out to be quite efficient on analyzing industrial problems. This appears to be quite evident from the results of the Hardware Model Checking Competitions (HWMMC) [2]. Since its original presentation [1], several optimizations and variants have been proposed and have been integrated and implemented independently in the many available model checkers, relying on different SAT solvers, different programming languages, different low-level algorithms and data structures. These differences may spoil some insights about the efficiency of the optimization, and make the comparison among the different variants difficult.

The contribution of this paper is to study further the approach by systematically comparing different variants of IC3, considering many possible parameters that may affect its efficiency, including some that have not been thoroughly analyzed before. First, we implemented the most important variants of the main components of IC3 and the most important optimizations in the same tool, namely the NUXMV verification platform [3]. Our implementation allows for controlling various configuration parameters, enabling/disabling the considered variants without causing any overhead. Second, we performed a thorough experimental evaluation using as benchmarks for the comparison all the single track benchmarks for the hardware model checking competitions of the 2011, 2012, and 2013 editions [2]. We have chosen a configuration, close to the description of the algorithm given in [4], that we used as baseline for conceptually identifying the differences and the impact of the different options/variants. The analysis identified several unexpected impacts, like e.g. the importance of the CNF conversion. Moreover, our results provide an independent confirmation of the results discussed in the papers where the considered options/variants were first introduced. As an outcome of the analysis, we also identified a best candidate set of options that leads to solve the largest number of problems in the considered resource constraints, and we compared this final best candidate configuration against two state-of-the-art implementations: the "reference" IC3 implementation available at [5], and the version of IC3 implemented within ABC [4], [6]. The results show that our best candidate configuration outperforms the two other considered versions. Moreover, the results show that quite often there is not a clear winner among different configurations. Indeed, enabling one option/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. Considering the virtual best, it turns out to be evident that a portfolio approach is the one that leads to the best performance in terms of number of problems solved in the given resource bounds.

Related Work. There have been several works that compared different verification algorithms within the same framework (e.g. [7] for software model checking algorithms). As far as IC3 is concerned, the closest work

is [4], where different options are compared within the PDR variant of IC3. Besides providing an independent evaluation of the findings of that and other papers introducing options/variants to the basic IC3 (like e.g. [8]), here we extend the scope of the evaluation, by considering both a larger benchmark suite, including the instances used in the most recent editions of HWMCC, and a larger set of options/configurations, including some not considered before (e.g. different SAT solvers and CNF conversion algorithms, or use of approximated SAT checks).

Structure of the paper. This paper is structured as follows. In Sect. II we briefly provide some background concepts. In Sect. III we briefly provide basic notions about the different variants. In Sect. IV we describe the evaluation methodology we adopted. In Sect. V we discuss the results of the analysis and we discuss the lessons learned. Finally, in Sect. VI we draw conclusions and we outline possible future works.

II. Background

Our setting is standard propositional logic. We denote formulas with φ, ψ, I, T, P, variables with x, y, and sets of variables with X, Y. A literal is an atom 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. 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. Given a set X of state variables, a *transition system* S over X can be described symbolically with two formulas: $I_S(X)$, representing the initial states of the system, and $T_S(Y, X, X')$, representing its transition relation, where Y is a set of input variables.

III. Review of IC3 techniques

A. High-level description of IC3

We follow the formulation of IC3 given in [4], which is known as PDR. Let S be a given transition system described symbolically by $I_S(X)$ and $T_S(Y, X, X')$. 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 a formula $F(X)$ such that: (i) $I_S(X) \models F(X)$; (ii) $F(X) \wedge T_S(Y, X, X') \models F(X')$; and (iii) $F(X) \models P(X)$.

In order to construct F, which is an inductive invariant, IC3 maintains a sequence of formulas (called *trace*, following [4]) $F_0(X), \ldots, F_k(X)$ such that:

- $F_0 = I_S$;
- for all $i > 0$, F_i is a set of clauses;
- $F_{i+1} \subseteq F_i$ (thus, $F_i \models F_{i+1}$);
- $F_i(X) \wedge T_S(Y, X, X') \models F_{i+1}(X')$;
- for all $i < k$, $F_i \models P$;

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 $\neg P(X)$ 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 elements of the trace (called *frames*) become identical (i.e. $F_i = F_{i+1}$), then a fix-point is reached, and IC3 can terminate with F_i being an inductive invariant proving the property.

Let us now consider the lower level details of IC3. For $i > 0$, F_i represents an over-approximation of the states of S reachable in i transition steps or less. The distinguishing feature of IC3 is that such sets of clauses 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 a 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:

$$F_{i-1} \wedge \neg s \wedge T_S \wedge s'. \tag{1}$$

If (1) is unsatisfiable, and s does not intersect the initial states I_S of the system, then $\neg s$ is *inductive relative to* F_{i-1}, and it can be used to *strengthen* F_i in order

```
bool IC3 (I, T, P):
 1.  trace = [I_S]  # first elem of trace is init formula
 2.  trace.push()  # add a new frame to the trace
 3.  while True:
         # blocking phase
 4.      while there exists a cube c s.t. c ⊨ trace.last() ∧ ¬P:
 5.          if not recBlock(c, trace.size() − 1):
 6.              return False  # counterexample found
         # propagation phase
 7.      trace.push()
 8.      for i = 1 to trace.size() − 1:
 9.          for each clause c ∈ trace[i]:
10.              if trace[i] ∧ ¬c ∧ T_S ∧ c' ⊨ ⊥:
11.                  add c to trace[i+1]
12.          if trace[i] == trace[i+1]: return True  # property proved

# simplified recursive description (see §III-D)
bool recBlock(s, i):
 1.  if i == 0: return False  # reached initial states
 2.  while trace[i-1] ∧ ¬s ∧ T ∧ s' ⊭ ⊥:
 3.      c = getPredecessor(i − 1, T_S, s')  # see §III-C
 4.      if not recBlock(c, i − 1): return False
 5.  g = generalize(¬s, i)  # see §III-B
 6.  add g to trace[i]
 7.  return True
```

Fig. 1. High-level description of IC3 (following [4]).

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. 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. Figure 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 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-1} 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 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 [9]. The algorithm is based on a smart exploration of the lattice of sub-clauses of the input clause. The original IC3 implementation [1] uses an approximated version of such procedure, trading effectiveness for computational efficiency. An even cheaper (and conceptually simpler) variant of the procedure was proposed in [4] and is used by the PDR implementation in ABC [6]. A third variant has been recently proposed in [8]. 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. In [8], 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 function T_S) 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 [1], only a simple syntactic generalization based on cone of influence is performed. A significant improvement was proposed in [4], 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 [10].

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 allows IC3 to generate counterexamples longer than the length of the trace when disproving properties, but it also generally improves performance for proving properties [1], [4].

E. Target-enlargement of P

The IC3 invariants and pseudo-code described above follow the PDR description of [4], which is slightly different from the original IC3 of [1]. In particular, in [1], *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 [4], it is shown that this behavior can be emulated by PDR by pre-processing 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.

IV. Setup of the comparison

A. 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 so 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 [3]. Our implementation is in C++, and its source code is available at https://nuxmv.fbk.eu/tests/difts2014. We have implemented all the most important 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.

B. Tested configurations

We distinguish 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-C,III-D, and III-E:

- for inductive generalization, we can select among the original IC3 procedure of [1] (*indgen-ic3*), the simpler one of [4] (*indgen-pdr*), and the CTG-based one of [8] (*indgen-ctg*); furthermore, we also consider a configuration in which relative induction is not used at all (*norelind*), and a simple implication check of the form $F_{i-1} \wedge T_S \wedge s'$ is used instead of (1) for blocking CTIs;
- for the computation of predecessors, we consider the SAT-based generalization procedure of [10] (*pre-gen*) and the cone of influence procedure of [1] (*pre-basic*);
- for the management of proof obligations, we can either use 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 [4], 1-step enlargement (*unroll-1*), and a more aggressive 4-step enlargement (*unroll-4*), inspired by the implementation of the TIP model checker [11].

Finally, as a further high-level parameter, we consider also the impact of pre-processing the transition system using sequential simplification techniques, commonly used by state-of-the-art model checkers, before invoking IC3. In particular, in our pre-processing configuration (*preproc*) we apply 2-step temporal decomposition [12] and detection of equivalent state variables using ternary simulation.

We also considered other, lower-level, parameters that may affect the performance of IC3:

- *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 [13] available from Github, both with (*minisat-simp*) and without (*minisat*) SAT pre-processing enabled, and the latest version of PICOSAT [14] (*picosat*).
- we consider two different algorithms for *CNF conversion*: the standard Tseitin encoding (*cnf-simple*) and the more sophisticated one presented in [15], 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 [4];
- *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 [1];
- finally, we also investigated the possibility to per-

form *approximated* calls to the SAT solver when possible. From our measurements (consistent with what reported in [4]), 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.[1] 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.

We considered as baseline for the experimental evaluation the configuration with the following settings: *indgen-pdr*, *pre-gen*, *queue*, *unroll-0*, *minisat*, *cnf-simple*, *onesolver*, *noactivity*. The rationale for using this configuration as baseline is that it corresponds to a basic implementation of the algorithm following the description of [4].

In the experimental analysis, we assumed all the above 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.

C. Benchmarks and execution environment

For the comparison we considered all the 765 single track benchmarks used in the hardware model checking competitions of the 2011, 2012, and 2013 editions [2].

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.

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. A deeper analysis, with possible different metrics, will require to define proper metrics that will also take into account correlations among the options, and this is left as future argument on investigation.

[1]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.

TABLE I
SUMMARY OF RESULTS FOR THE HIGH-LEVEL PARAMETERS

Configuration	# Solved	$\Delta_{baseline}$	Gained	Lost	Cumulative time (sec)
preproc	431	+10	22	12	25399
unroll-4	431	+10	29	19	31525
unroll-1	428	+7	23	16	34005
baseline	421	0	0	0	27242
indgen-ctg	412	-9	26	35	34632
stack	401	-20	14	34	34447
indgen-ic3	384	-37	7	44	30405
norelind	381	-40	5	45	32539
pre-basic	334	-87	9	96	31360

The source code, the data to reproduce the executed experiments, and all the log files can be downloaded from https://nuxmv.fbk.eu/tests/difts2014.

V. 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. High-level parameters

Fig. 2 shows the accumulated-time plots for the different high-level configurations, plotting the # of solved instances (y-axis) in the given total amount of time (x-axis) in logarithmic scale. To make the accumulated plots more readable, we cropped the curves corresponding to data below 1 second. Table I provides a summary of the information, where for each configuration we also show the difference in # of solved instances wrt. the baseline, as well as the number of instances gained (i.e. solved by the given configuration but not by the baseline) and lost. Finally, Fig. 3 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.)

These results lead to the following observations.

1) Inductive generalization: looking at the results of *indgen-ic3*, it seems that the cost of applying the original IC3 generalization procedure outweighs its potential benefits. An analysis of the log files revealed that, even on instances for which *indgen-ic3* leads to a substantial reduction in the number of generated cubes or in the length of the trace, the run-time often increases. Surprisingly however, in the majority of cases there is

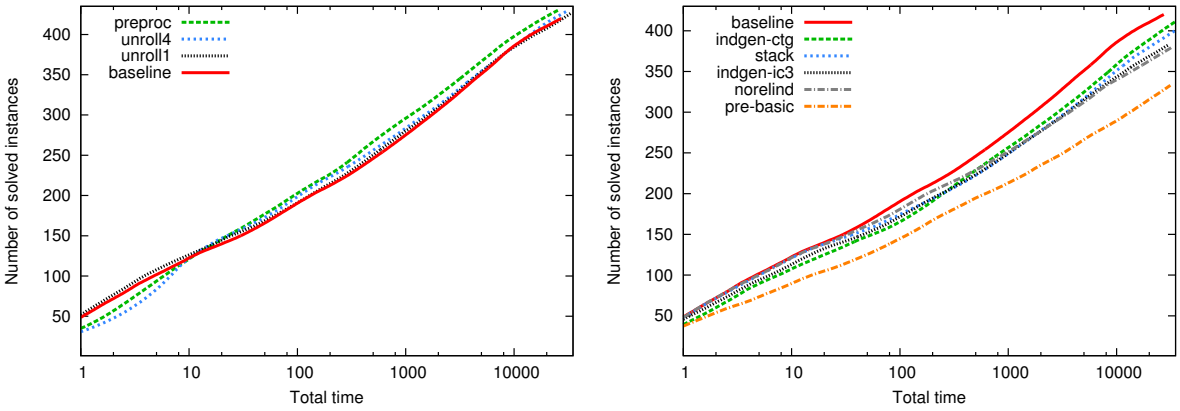

Fig. 2. Survival plots comparing the baseline configuration with configurations changing the "high-level" parameters. We provide two separate plots for better readability: the plot on the left shows configurations performing better than the baseline, the one on the right those performing worse.

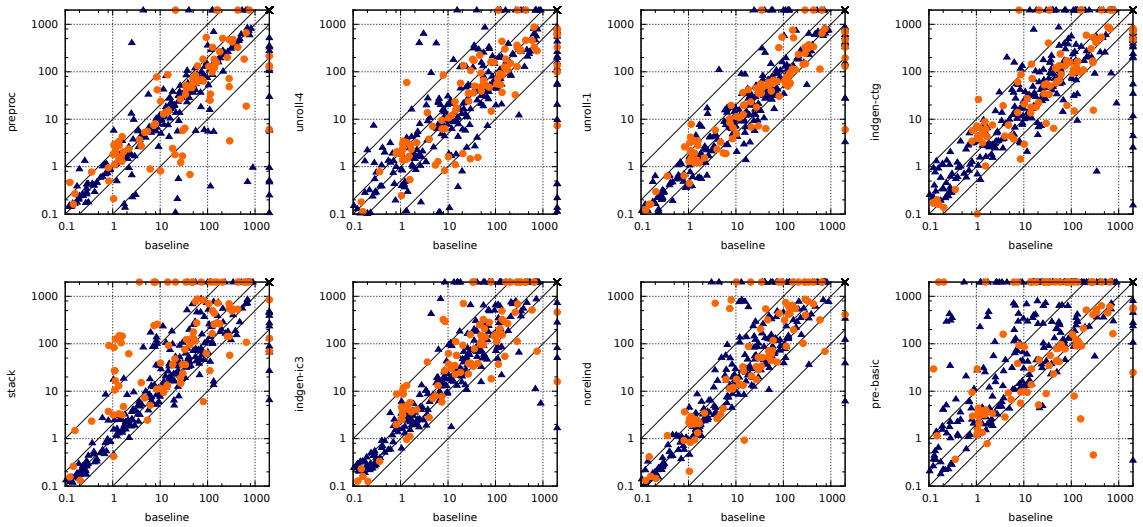

Fig. 3. Detailed comparison of high-level configurations vs the baseline. The baseline is always on the x-axis. Points above the diagonal indicate better performance of the baseline. Points on the borders indicate timeouts (900 s). Blue triangles denote safe instances, orange dots unsafe ones.

not even a reduction in size (# of clauses) of the inductive invariant produced for safe instances (that both *baseline* and *indgen-ic3* could solve), although the trace length generally decreases slightly.

Using the CTG-based generalization technique proposed in [8] (*indgen-ctg*) does not seem to pay off. This seems to contradict the results reported in [8], where CTG-based generalization is shown to lead to significant improvements in the # of solved instances, despite the overhead introduced for easier problems. However, our benchmark set is different than the one used in [8]. Indeed, if we exclude the backward-encoded BEEM models from our set, as was done in that work, *indgen-ctg* solves 5 more instances than *baseline*. In both cases, *indgen-ctg* and *indgen-pdr* (used in *baseline*) seem to have somewhat complementary strengths, as can be seen also by the scatter plots.

Finally, as expected disabling relative induction (*norelind*) significantly hurts performance. It is however surprising to see that *norelind* does not perform much worse than *indgen-ic3*.

2) Predecessor computation: our results confirm that generalizing predecessors of CTIs (either via ternary simulation [4] or via SAT [10]) is crucial for performance: this is the single most important parameter among those we considered.

3) Proof obligations management: our results confirm that using a simple stack instead of a priority queue for

TABLE II
SUMMARY OF RESULTS FOR THE LOW-LEVEL PARAMETERS

Configuration	# Solved	$\Delta_{baseline}$	Gained	Lost	Cumulative time (sec)
cnf-abc	448	+27	30	3	28431
sat-approx	438	+17	26	9	28833
activity	430	+9	17	8	31352
minisat-simp	429	+8	18	10	27413
baseline	421	0	0	0	27242
manysolvers	420	-1	14	15	32306
picosat	402	-19	11	30	32570

managing proof obligations leads to a visible degrade in performance. The performance degradation happens for both safe and unsafe instances alike (as can be seen in Fig. 3), and in fact out of the 34 instances lost by *stack*, 17 are safe and 17 are unsafe.

4) Target-enlargement: The results obtained by enabling *unroll-1* or *unroll-4* are somewhat inconclusive: they improve the # of solved instances, but they end-up in a noticeable increase of the run-time, and the numbers of lost instances are non-negligible. Moreover, the corresponding scatter plot show that there is a not clear trend.

5) Pre-processing: turning on *preproc* seems to generally help, 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. However, it might happen that our pre-processor actually increases the number of state variables in some cases (when applying temporal decomposition does not allow to discover further simplifications). In fact, in 9 out of the 12 lost instances the increase is very significant ($>$ 3x).

B. Low-level parameters

In this section we analyze the impact of the low-level parameters. Fig. 4 shows the "survival plots" for the different configurations. Table II provides a summary of the information and Fig. 5 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. 5), with (almost) 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 1.65, the mean 4.32 and the 9th percentile 3.42. However, also the number of generated cubes decreases slightly (median 1.0, mean 1.36). 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 1.35, the mean 3.42 and the 9th percentile 4.05; a similar trend is also shown for for the overall SAT solving time, with median 1.3, mean 3, and 9th percentile 2.85. 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 fact, in the 9 instances lost by *sat-approx*, on 7 cases the # of generated cubes is at least about 3 times more than for the *baseline* configuration. 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* does not seem to lead to any clear benefit. Indeed, some instances are gained, but the two accumulated plots turn out to be very similar and the corresponding scatter plot confirms that there is no clear trend.

4) SAT pre-processor: the use of SAT pre-processing (*minisat-simp*) leads to a small improvement in performance (see scatter), but several instances are lost. We notice that, surprisingly, using a specialized CNF conversion (namely *cnf-abc*) is 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 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.

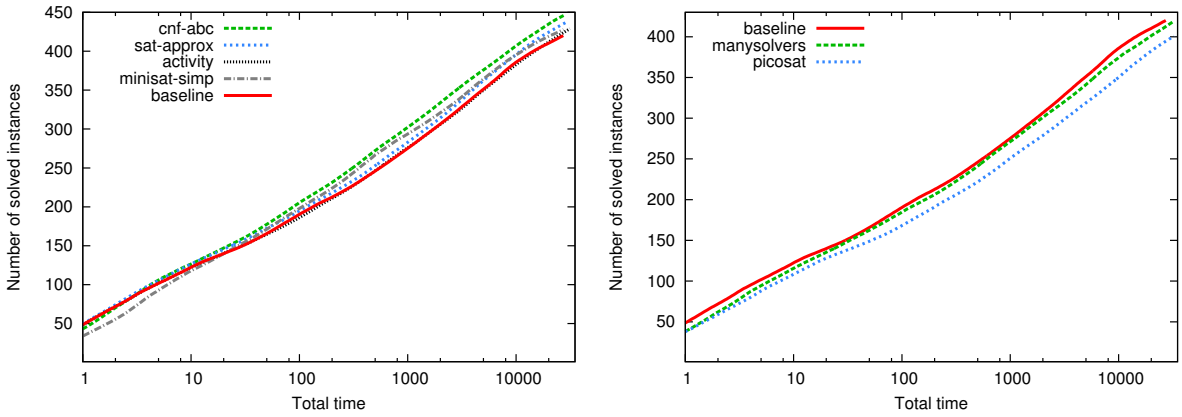

Fig. 4. Survival plots comparing the baseline configuration with configurations changing the "low-level" parameters. We provide two separate plots for better readability: the plot on the left shows configurations performing better than the baseline, the one on the right those performing worse.

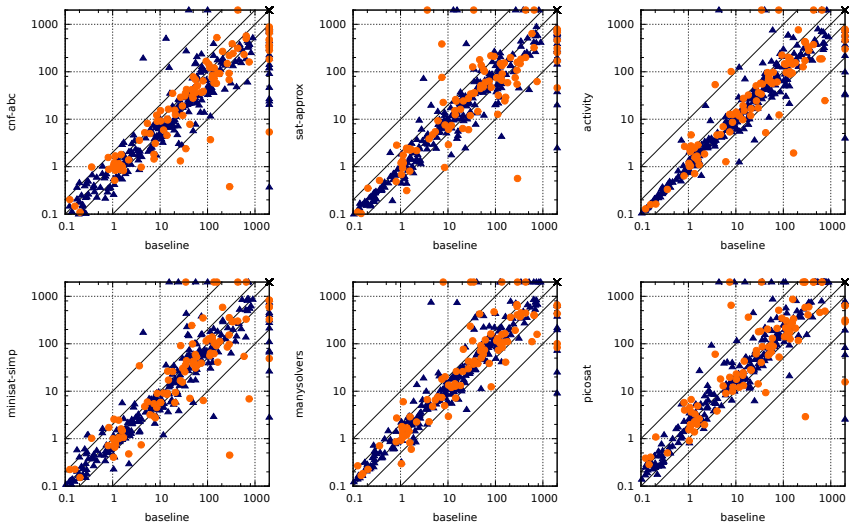

Fig. 5. Detailed comparison of low-level configurations vs the baseline. The baseline is always on the x-axis. Points above the diagonal indicate better performance of the baseline. Points on the borders indicate timeouts (900 s). Blue triangles denote safe instances, orange dots unsafe ones.

7) 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 accumulated 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).

C. Comparison with other implementations

We also compared our implementation against other state-of-the-art implementations, namely the "reference" IC3 implementation provided by Bradley [5] (*ic3ref*) and the version of PDR implemented within ABC [4] (*abcpdr*). We also considered a new configuration, called *bestcandidate*, obtained by enabling all the parameters that lead to an improvement in the # of solved problems wrt. *baseline*. Namely, *baseline* plus *preproc*, *unroll-4*, *cnf-abc*, *activity*, *sat-approx*, *minisat-simp*. We also compare against the virtual best, that takes the best configuration for each individual instance. The results are reported in Fig. 6 (for accumulated plots), in Fig. 7 (for the scatter plots), and summarized in Table III. The

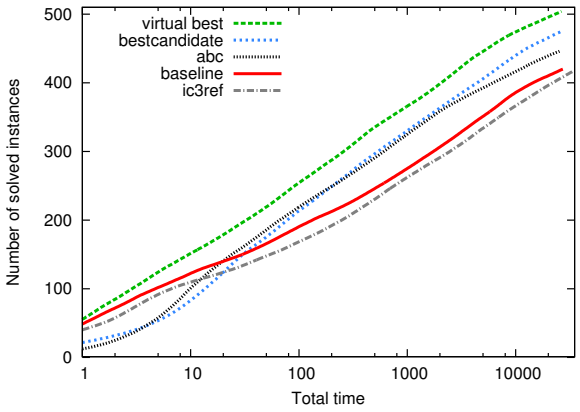

Fig. 6. Baseline vs best configuration, virtual best and other implementations.

TABLE III
SUMMARY OF RESULTS: BEST VS OTHER

Configuration	# Solved	$\Delta_{baseline}$	Gained	Lost	Cumulative time (sec)
virtual best	505	+84	84	0	26509
bestcandidate	477	+56	68	12	27341
abcpdr	447	+26	37	11	25360
baseline	421	0	0	0	27242
ic3ref	418	-3	32	35	33924

following observations arise from these results.

- It is clear that *ic3ref* is comparable to our *baseline* implementation as far as the # of solved problems is concerned. However, the scatter plot and the large numbers of gained (32) and lost (35) 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* (447 for *abcpdr* and 448 for *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. 7, 2nd from the left), we see that *abcpdr* is still significantly faster than *cnf-abc*, but it is difficult to identify the reason for this behavior (it might be due to a smarter implementation or to a better tuning for the instances) which is left to future investigations.
- Our *bestcandidate* configuration is a clear and significant improvement wrt. the *baseline* configuration both in terms of the # of solved problems and in solving time (as shown by the scatter plot).
- The *virtual best* configuration is way ahead of the others, suggesting that a portfolio approach (with better investigation of parameter configurations) might give significant advantages for this set of benchmarks.

D. 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. If we disregard the results for the *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 configuration confirm that a portfolio approach, with many configurations run in parallel, can give very significant performance advantages for this kind of problems.

VI. CONCLUSIONS

We have presented a systematical 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, more evident than other higher-level parameters (e.g. CTG-based generalization or model pre-processing). We also identified a set 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 somewhat complementary strengths, suggesting that a portfolio approach might give significant advantages.

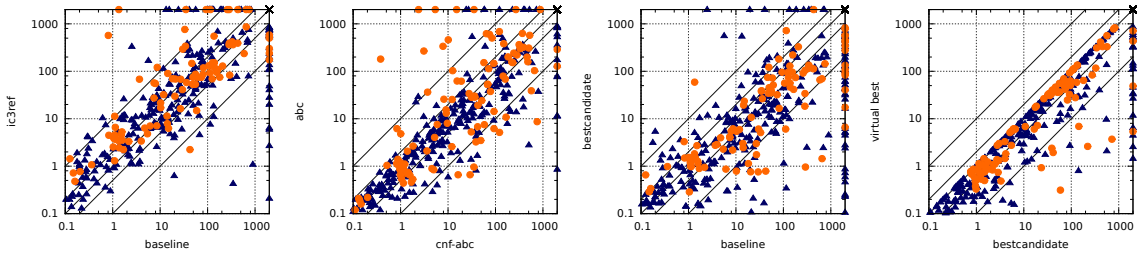

Fig. 7. Comparison of our baseline and best configurations vs ABC and IC3REF. Points on the borders indicate timeouts.

As future work, we aim at better investigating the relation existing among the different parameters, in order to better understand their respective impact, and identify an even better set of parameter configurations, possibly with the help of automatic parameter tuning procedures as in [16].

REFERENCES

[1] A. R. Bradley, "Sat-based model checking without unrolling," in *VMCAI*, ser. LNCS, R. Jhala and D. A. Schmidt, Eds., vol. 6538. Springer, 2011, pp. 70–87.

[2] "Hardware model checking competition," http://fmv.jku.at/hwmcc/.

[3] 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, pp. 334–342.

[4] 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, pp. 125–134.

[5] A. Bradley, "Ic3 reference implementation," https://github.com/arbrad/IC3ref.

[6] 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, pp. 24–40.

[7] D. Beyer and P. Wendler, "Algorithms for software model checking: Predicate abstraction vs. impact," in *FMCAD*, G. Cabodi and S. Singh, Eds. IEEE, 2012, pp. 106–113.

[8] Z. Hassan, A. R. Bradley, and F. Somenzi, "Better generalization in ic3," in *FMCAD*. IEEE, 2013, pp. 157–164.

[9] A. R. Bradley and Z. Manna, "Checking safety by inductive generalization of counterexamples to induction," in *FMCAD*, 2007, pp. 173–180.

[10] 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, pp. 135–143.

[11] K. Claessen and N. Sörensson, "A liveness checking algorithm that counts," in *FMCAD*. IEEE, 2012, pp. 52–59.

[12] M. L. Case, H. Mony, J. Baumgartner, and R. Kanzelman, "Enhanced verification by temporal decomposition," in *FMCAD*. IEEE, 2009, pp. 17–24.

[13] N. Eén and N. Sörensson, "An extensible sat-solver," in *SAT*, ser. LNCS, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Springer, 2003, pp. 502–518.

[14] A. Biere, "Picosat essentials," *JSAT*, vol. 4, no. 2-4, pp. 75–97, 2008.

[15] 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, pp. 272–286.

[16] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "Paramils: An automatic algorithm configuration framework," *J. Artif. Intell. Res. (JAIR)*, vol. 36, pp. 267–306, 2009.