Form Methods Syst Des (2011) 39:205–227
DOI 10.1007/s10703-011-0123-3

Benchmarking a model checker for algorithmic improvements and tuning for performance

Gianpiero Cabodi · Sergio Nocco · Stefano Quer

Published online: 31 May 2011
© Springer Science+Business Media, LLC 2011

Abstract This paper describes a portfolio-based approach for model checking, i.e., an approach in which several model checking engines are orchestrated to reach the best possible performance on a broad and real set of designs. Model checking algorithms are evaluated through experiments, and experimental data inspire package tuning, as well as new algorithmic features and methodologies. This approach, albeit similar to several industrial and academic experiences, and already applied in other domains, is somehow new to the model checking field. Its contributions lie in the description of how we: (1) characterize and classify benchmarks in a dynamic way, throughout experimental runs, (2) relate model checking problems to algorithms and engines, (3) introduce a dynamic tuning of sub-engines, exploiting an on-the-fly performance analysis, (4) record results of different approaches, and sort out heuristics to target different classes of problems. We provide a detailed description of the experiments performed in preparation of the Model Checking Competition 2010, where PdTRAV, our academic verification tool, won the UNSAT division, while ranking second in the OVERALL category.

Keywords Model checking · Binary decision diagrams · Satisfiability solving · Interpolant verification · Portfolio strategies

1 Introduction

After a couple of decades of active research and industrial efforts on Model Checking (MC) techniques, it is generally accepted that an industrial-scale model checker should include several engines and verification algorithms, to be exploited in a selective or coordinated way. Given the variety and difficult-to-predict shapes and characteristics of the problems under verification, the multi-engine approach is the only one able to provide scalability, robustness, and high coverage.

G. Cabodi · S. Nocco · S. Quer (✉)
Dip. di Automatica e Informatica, Politecnico di Torino, Turin, Italy
e-mail: stefano.quer@polito.it

The problem of building a general-purpose model checker is often considered an industrial task [1], involving professional software engineering and broad experimentation for parameter tuning. On the other hand, events such as the Hardware Model Checking Competition (HWMCC [2]) encouraged academic researchers to test the performance of their tools on a large benchmark set, thus improving the overall quality of academic packages.

To obtain the best possible behavior from a MC tool, a broadly adopted strategy is to measure the performance of every candidate algorithm on a representative set of problem instances (the so-called "training set"), and then to use only the methodology that offers the best overall performance. This approach is also called "winner-takes-all". However, choosing a single algorithm is not always a good approach, as often one approach is better than any other at solving some problem instances, but dramatically worse on other benchmarks. Indeed, the usage of this strategy results in the neglect of many algorithms that, though not competitive on average, offer very good performance on particular instances. Engineers thus face a potentially difficult "algorithm selection problem" [3–6].

The ideal solution of the algorithm selection problem would be to consult an "oracle" that knows the amount of time that each algorithm would take to solve a given problem instance, and then to select the algorithm with the best performance. Unfortunately, computationally cheap, perfect oracles of this nature are not available. Therefore, a reasonable compromise is to build an approximate run-time predictor, providing a heuristic approximation to a perfect oracle. Portfolio-based approaches [4] represent a widely accepted paradigm in this direction. In this paper, Huberman et al. propose a general method for combining existing algorithms, based on the notion of risk in economics. Tested by solving a canonical NP-complete problem, the methodology can be used for problems ranging from the combinatorics of DNA sequencing to the completion of tasks in environments with resource contention, such as the World Wide Web.

The SATzilla tool [5], presented at recent SAT competitions, witnesses the potential benefits of portfolio-based approaches. SATzilla adopts a portfolio approach which is selected partially off-line and partially on-line. Prediction is given by an empirical hardness model, based on 48 different design features, where ridge regression delivers the predicted run time for each native SAT solver within the portfolio. The tool thus identifies one or more presolvers and a main solver, to be run in sequence.

Pulina and Tacchella [6] propose a portfolio-based Quantified Boolean Formula (QBF) solver, called AQME. This tool, starting from an initial set of 141 different design features, firstly divides all instances into several clusters. Then, it tries to guess the best strategy to be adopted for each cluster. Although the authors consider 16 different solvers in the initial phase of the process, they show that the best results (in terms of instances solved and overall CPU time) are attained with only 8 solvers.

Compared to SAT and QBF, MC is an even more challenging problem, as the involved algorithms and data structures are inherently dissimilar, less regular, and performance less predictable. Anyway, also in this field, mechanizing the application of proof strategies is not a new concept, as it is an essential component of most general-purpose theorem provers, such as HOL [7], PVS [8] and ACL2 [9].

An interesting approach, in the direction of run-time prediction of MC performance, is proposed by Kamhi et al. [10]. Their work, developed within the framework of BDD-based symbolic MC, address techniques for performance visualization and monitoring, as well as semi-automated decisions taken by a "light expert system".

A more recent work is the one presented by Mony et al. [11]. In this case, the verification tool is transformation-based (the design is modified by a sequence of transformations such as redundancy removal, retiming, target-enlargement, localization, etc., till verification), and the expert system is based on a "tree of knowledge" and an "inference engine". The tree is a database, containing information learned during verification. Essentially, each node in the tree corresponds to a run of a particular engine instance (including all options) and the feedback received from that engine. The inference engine first associates to each node of the tree a defined priority depending on the outcome of the run. Then, for each verification step, it selects the node of the tree to which applying a new transformation and the kind of transformation to apply. The strategy was implemented in the SixthSense tool, used within IBM corporation, obtaining interesting results.

Following the above ideas, this paper describes the work done on our academic model checker PdTRAV (Politecnico di Torino Reachability Analysis and Verification) package, to participate to the Hardware Model Checking Competition 2010. The paper is also partly motivated by an evaluation of PdTRAV performed on a large set of industrial benchmarks under an industrial support.[1] Our main purpose is to use learning techniques to build an algorithmic portfolio, in order to achieve state-of-the-art performance on a broad, practical domain. Compared to [10] and [11], we focus more on the tasks performed by the tool developer than on the ones carried-on by the user, and we go deeper in characterizing and monitoring MC runs on academic and industrial designs. We thus aim at:

– Evaluating different MC engines, with a major effort on proof rather than falsification oriented engines.
– Classifying benchmarks by clearly separating easy from hard problems, being aware that the last category is clearly the most interesting one.
– Exploiting the benchmarking effort and the execution profiles obtained for tuning individual MC engines.
– Introducing a metric able to characterize the affinity (adequacy) between MC engines and benchmarks, and providing related techniques to estimate affinity.
– Developing an "expert system" able to control the activation of several MC engines, based on heuristics and proper (static and dynamic) data characterizing the problem.

This paper describes, with an historical perspective, the methodology adopted, the experimental set up, the data gathered during the entire process, and the final improvements obtained. Although we cannot claim that the overall framework is novel (see for example [4–6, 10, 11], we believe that the proposed scheme includes new ideas, such as dynamic tuning of MC engines, and the characterization of the engine-to-problem relationships.

Moreover, the experimental data, and the related classifications and evaluations introduced, represent a very interesting characterization of the HWMCC'10 benchmark set, and thus a valuable contribution for other research groups and tool developers in the field of MC.

1.1 Roadmap

The paper is organized as follows. Section 2 introduces our notation, a few introductory concepts on MC techniques, and some background on portfolio algorithms. Section 3 briefly describes the underlying academic MC tool (PdTRAV). Section 4 describes our overall ideas for benchmark classification, and benchmark-to-method affinity evaluation. Section 5 presents a simple module (the expert system) that exploits design and performance statistics to heuristically activate MC engines. Experimental results are reported in Sect. 6. Conclusions finally outline a few open problems and overall considerations for this work.

[1] SRC contract 2009-TJ-1968.

2 Background

2.1 Model and notation

The sequential systems we address are usually modeled either as Finite State Machines (FSMs), or as Kripke Structures. Although the latter ones are more general, and widely adopted in formal definitions, we select FSMs as our preferred model. The choice is mainly motivated by practical reasons, being FSMs a good compromise for models of sequential (hardware) circuits.

A Finite State Machine is a 6-tuple, $(S, I, \Sigma, \lambda, T, O)$ where: S is the set of states, $I \subseteq S$ is the set of initial states, Σ is the input alphabet, λ is the output alphabet, $T \subseteq S \times \Sigma \times S$ is the transition relation between the states, and $O \subseteq S \times \Sigma \times \lambda$ is the output relation.

The state space and the primary inputs are defined by indexed sets of Boolean variables $V = \{v_1, \ldots, v_n\}$ and $W = \{w_1, \ldots, w_m\}$, respectively. States correspond to valuations of variables in V, whereas transition labels correspond to valuations of variables in W. We indicate next states with the primed variable set $V' = \{v'_1, \ldots, v'_n\}$. Furthermore, in our notation superscripts identify a time frame. For example, we use $V^i = \{v^i_1, \ldots, v^i_n\}$ for representing the state variables at the i-th time step.

A set of states is expressed by a state predicate $S(V)$ (or $S(V')$ for the next state space). We assume that the transition relation $T(V, W, V')$ is given by a *circuit graph*, with state variables mapped to latches. Present and next state variables correspond to latch outputs and inputs, respectively.

A state path of length k is a sequence of states $\sigma_0, \ldots, \sigma_k$ such that, for all $0 \leq i < k$, $T(\sigma_i, \omega_i, \sigma_{i+1})$ is true given some input pattern ω_i.

2.2 Model checking

A state set S' is reachable from a state set S in k steps if, in the FSM state transition graph, there exists a path of length k, connecting some state in S to some state in S':

$$S(V^0) \wedge \left(\bigwedge_{i=0}^{k-1} T(V^i, W^i, V^{i+1}) \right) \wedge S'(V^k) \neq 0$$

Given an invariant property $P(V)$, the purpose of a MC algorithm is to verify whether its negation, $F(V) = \neg P(V)$, is reachable from $I(V)$ or not. In the first case $P(V)$ is true, otherwise it is false and a witness counter-example can be produced. In the sequel, we assume that the reader is familiar with the main hardware verification techniques, such as BDD-based reachability analysis [12], SAT-based Bounded Model Checking (BMC) [13], simple induction [14] and inductive invariants [15], interpolant-based MC [16], etc. Nevertheless, in the sequel, we will indicate with:

- IMG, the image operator, i.e., *To* = IMG (T, *From*) computes the set of states *To* reachable in one step from the states in *From*.
- ITP, a Craig interpolant, i.e., C = ITP (A, B) is an AND/OR circuit that can be directly derived from a refutation proof of $A \wedge B$.

2.3 Portfolio strategies

Portfolio techniques, commonly used in game theory and finance, provide an allocation of limited resources to give optimal trade-offs between expected performance and variability. Recently, portfolio-based approaches have been proposed for computation allocation, following *parallel* (several engines/algorithms are executed concurrently), as well as *sequential* (the execution of one algorithm only begins when the previous one has ended), and/or *partly sequential* (some combination of the two) distribution schemes. For example, Huberman et al. [4] proposed parallel portfolios for different search heuristics, whereas [5] and [6], developed within the frameworks of SAT and QBF decision engines, respectively, are intrinsically sequential.

Our approach is sequential as well, since we target a single-threaded application. By counting various setups and configurations as different engines, PdTRAV initially included 25 MC engines. This work led us to consider 11 new engines, and enrich our portfolio, to finally include the following 36 methodologies:

– 9 BDD-based methods [12], i.e., forward, backward and forward/backward, with 3 MC schemes available for each method, based on different clustering and early quantification, insertion of cut-point, etc.
– 24 ITP-based [16] techniques, with different partitioning and quantification schemes [17–21] (6 base engines multiplied by 4 different running configurations).
– 2 SAT-based strategies (simple induction [14] and inductive invariants [15, 22]).
– 1 BMC-based [13] method, targeting falsification.

3 The PdTRAV package

The Politecnico di Torino Reachability Analysis and Verification (PdTRAV) package can be considered as a set of MC engines oriented to evaluate and to benchmark new algorithmic ideas. It may represent a good starting point for experimental evaluation and comparison, as it won two of the sub-categories at the HWMCC'07 [23], ranked third in HWMCC'08, and first in the UNSAT category of the HWMCC'10. PdTRAV has a simple textual user interface, and different methodologies are typically selected and configured by command line switches. It puts no particular effort on input language (it just supports flat net-list input in BLIF [24] and AIGER [25] formats), compiler, GUI, etc. It includes falsification features, even if it is mainly oriented to the proof of (true) properties. No expert system (or portfolio manager) was present before this work, except an interpolant-based approach integrating a few other SAT- and BDD-based techniques [20, 21].

PdTRAV supports several symbolic reachability and MC methods, developed during the years:

– BDD-based representations and traversals, including forward, backward, combined (approximate)forward/(exact)backward algorithms [26, 27], partitioned BDDs and/or image computation procedures [28, 29].
– Interpolant-based verification, with ad-hoc abstraction and tightening techniques [18, 19], integrated SAT-based approaches [20, 21].
– Inductive reasoning (inductive invariants [22, 30]) and symbolic manipulation of And-Invert Graphs (AIGs [31]), with circuit-based quantification [32].
– BMC with AIG-based circuit compaction (before moving to CNF-based SAT calls) but without incremental SAT.

The tool also supports model transformations and reductions, typically activated as preprocessing steps of MC procedures. Although not comparable, in terms of features, power and versatility, to transformation-based verification, implemented in other MC tools, such as ABC [33] and SIXTHSENSE [34], PdTRAV includes, among others: Cone-Of-Influence

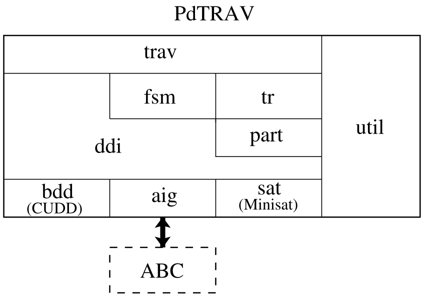

Fig. 1 The PdTRAV architecture: Packages and their relationship

(COI) reduction, retiming (forward and peripheral), re-synthesis based on (proved and trivially speculated) node equivalences, phase abstraction, handling of explicit and hidden constraints. Counterexample-based abstraction-refinement [35] is also available, but in a very basic and naive implementation.

The software architecture of PdTRAV is based on a modular and layered framework. In order to achieve its main purpose, fast prototyping capabilities for verification algorithms, we decided to clearly separate the functionality of the low-level evaluation engines, from the top level MC application layer. We thus created intermediate abstraction layers for bit-level Boolean function representation and manipulation, and for the main data structures involved in symbolic MC. The architecture basically shows:

– Low-level symbolic manipulation and decision engines: BDDs, SAT solver(s), AIGs.
– Mid-level modules managing various ADTs and objects required by verification algorithms: FSMs, transition relations, sets of reachable states.
– Top-level verification algorithms and related data structures.

A pictorial scheme of the architecture of PdTRAV is represented in Fig. 1. A brief description of native modules follows:

– `trav` (traversal). This module implements variants of various symbolic traversal and MC algorithms, both BDD- and SAT-based.
– `fsm` (finite state machine). As previously stated, FSMs are the basic sequential model representation within PdTRAV. The module provides functions to load, store and manipulate FSMs. Input, output and state variables; transition functions and (optionally) transition relations, constraints, and state sets are represented by BDDs and/or AIGs. Various transformation and optimization techniques are also provided by this module.
– `tr` (transition relation). Transition relations are the key structure for most supported symbolic traversals and MC algorithms. This module provides several algorithms to manipulate them, such as sorting, partitioning, clustering, constraining, etc. The module also includes various BDD-based symbolic image and pre-image computation routines.
– `part` (partitioning). This module provides routines to generate conjunctively or disjunctively partitioned Binary Decision Diagrams. Some of the provided functions are simply wrappers of lower levels procedures, heavily based on internal BDD data structures.
– `ddi` (decision diagram interface). This abstract interface manipulates Boolean functions and logic variables. The module basically provides ADTs for:
  • Boolean functions and arrays of Boolean functions, hiding the inner details of their implementation: BDDs (in monolithic and/or partitioned form), AIGs, CNF clauses.
  • Variables, sets of variables, arrays of variables.

  The module includes several core algorithms for general-purpose symbolic evaluations, to be exploited by other modules, such as linear existential quantification over conjunctions

of BDDs [36, 37], computation of Craig interpolants [16, 38, 39] exploiting SAT solver proofs, etc.
- `bdd` (Binary Decision Diagrams). The PdTRAV package relies on the CUDD [40] tool to perform low level operations on BDDs.
- `aig` (And-Invert Graph). This module is an extension of the basic `baig` package provided with VIS [41], which accounts for circuit-based representation and manipulation of Boolean functions.
- `sat` (SAT). The PdTRAV package mainly relies on the MINISAT [42] tool (version p1.14, with minor modifications) to solve satisfiability instances.
- `util` (utility). This module includes a set of functions for wrapping common low-level functionality of the operating system (i.e., process execution, assertion and exception handling, error dumping, etc.) and providing custom interfaces to generic data manipulation structures (arrays, lists, etc.).

Finally, PdTRAV also partially relies on the ABC [33] external library. ABC is a synthesis and verification software PdTRAV uses exclusively for combinational optimizations during internal verification steps, such as interpolant compaction. It exploits multilevel combinational synthesis features, such as balancing, rewriting, and refactoring algorithms. FRAIGing [43] is generally disabled, as too expensive for on-the-fly compactions. Sequential optimization and/or verification features provided by ABC are not used.

4 The tool benchmarking approach

In this section, we first describe how we classify verification problems (Sect. 4.1). Then, we exploit such a classification to measure the affinity (fitness) between MC engines and problems, after running each engine (Sect. 4.2). Finally, we propose dynamic (run-time) profiling and heuristics able to drive performance predictions (Sect. 4.3).

4.1 Classifying benchmarks

Verification problems are often classified adopting various criteria such as:

- Size-based grouping: Circuits are divided in sub-sets according to their gate size or, more often, their number of latches. The latter choice is a typical heuristic to select candidates for BDD-based approaches, as BDDs are usually not applied to circuits with more than 100–200 state elements.
- Property-based classification: Instances may be characterized as equivalence-checking problems, safety properties, liveness properties, etc.
- Circuit family- and/or architecture-based classification: Verification techniques often perform very differently on structurally different sets of benchmarks, such as pipelined structures, control logic, data-paths, memories, etc.
- Sequential depth, or diameter. This criterium is usually considered for BDD-based reachability (further distinguishing between forward and backward diameter, which can differ), though it can be identified even in SAT-based methods.
- Overall MC results: Benchmarks are classified through the time required by a model checker to complete the verification. Problems can be divided in several sub-groups, from easy-to-prove to hard-to-prove ones.

Obviously, size-, property- and family-based classifications are commonly used for preliminary decisions, as they rely on static data, typically available before any MC run. On the other hand, both the diameter of the circuits and the performance of a MC algorithm are more difficult to predict and estimate, as they are generally available only after completing the MC task. Furthermore, intermediate estimations of "coverage" (i.e., the advancements of partially completed tasks) are very difficult, due both to the non uniform behavior of verification strategies, and to the difficulty of defining coverage metrics.

Our proposal is to exploit the above mentioned classification criteria, and to enrich them by means of a set of run-time statistics. Such measures are observed at a finer level of granularity. They are computed either statically or dynamically, and updated throughout the MC runs. They include:

– BDD-based representation of the system transition relation (with cut-points). We classify the circuit taking into account the overall BDD-size (in terms of BDD nodes) and, possibly, the number of cut-point variables (pseudo-inputs) introduced. Cut-point insertion makes the approach scalable, but experience shows that too many cut points (hundreds to thousands) make sub-sequent MC tasks infeasible.
– BDD sizes within inner steps of symbolic traversals. For example, we consider peak BDD sizes within image and pre-image computations (which are related to the overall cost of symbolic image computations), BDD sizes of state sets, and the number of activations of dynamic reordering (sifting).
– AIG-size of combinational circuits (used for time-frame unrollings in BMC and BMC-like steps).
– SAT solver statistics for BMC or BMC-like SAT runs.
– Number of inductive invariants (equivalences and implications) proved by inductive methods.
– Size of generated interpolants.
– Number of reachability iterations, either in BDD- or SAT-based approaches.
– Analysis of the property under check. For instance, we analyze the circuit structure of the invariant property under check, in order to uncover equivalence checking problems, that typically require specific MC algorithms or tuning.
– Clock and constraint detection. Uncovering clock generators and hidden constraints is the core of several circuit transformations and reductions (e.g., phase abstraction) and may be used to initially classify circuits. For instance, we classified large circuits with clock as "industrial".

To give a first idea of our size-based classification strategy, we analyzed the complete HWMCC'08 [44] benchmark suite (645 designs with a single property). The size of such problems, in terms of primary inputs, state variables, and AIG nodes, is reported in Fig. 2. The figure clearly shows that most of the benchmarks can be classified as small or medium-size, whereas just a minority of them attain sizes comparable to industrial-scale problems.

4.2 Affinity measures: "a posteriori" static evaluation

To collect "a-posteriori" affinity measures, we verified all 645 properties in the HWMCC'08 suite with all 25 engines initially available in our portfolio. These runs deliver a matrix of statistics for each couple of available benchmark and MC engine/setting. Thus, it is possible to know the best choice for each problem, i.e., the engine and the setting that a perfect oracle would be able to select. After that analysis, to characterize the adequacy (fitness) of each single MC method for a given benchmark, we introduce the following affinity measures.

A first and rough possibility is to define affinity as a discrete value characterizing the success of MC each engines for each benchmarks. Whenever a technique M_i is able to

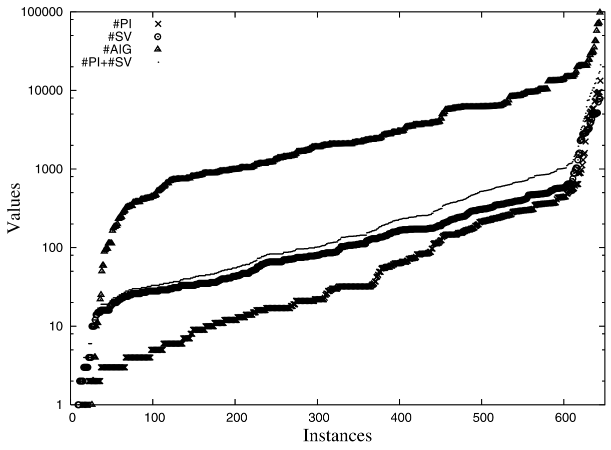

Fig. 2 Size statistics. The number of primary inputs (#PI), state variables (#SV), their sum (#PI+#SV), and number of AIG nodes (#AIG), is reported, on a logarithmic scale, as a function of the instance

complete the verification of a benchmark B_j, we include B_j in the set of completed experiments for M_i. However, we may also generate classifications based on dynamic data, at intermediate steps of the verification process, e.g., at given traversal depths (for BDD-based methods) or BMC bounds (for SAT-based approaches). For instance, it is possible to define $Partial_k(M_i)$ as the set of benchmarks where technique M_i was able to get to bound k. The above classification scheme is just binary, as a given benchmark is either in or out of a given set.

A more general formulation is based on scores representing the affinity A between techniques and problems. Thus,

$$A_{i,j} = A(M_i, B_j) = f_A(time, mem, stats)$$

is a measure of the affinity between method M_i and benchmark B_j, computed starting from execution time, used memory and other statistics. The measure can be either a continuous value, defined over a proper range, or it can be discretized. We associate the highest affinity to fast solutions, and the lowest one to failures, i.e., time-out or memory-out runs. For example, in case of bounded time (memory) resources, let $Time_{max}$ (Mem_{max}) be the limit for time (memory), and $Time_{i,j}$ ($Mem_{i,j}$) the time (memory) statistic for the (M_i, B_j) run. Then, a possible choice for continuous measures is given by

$$A^{Time}_{i,j} = Time_{max}/Time_{i,j}$$

$$A^{Mem}_{i,j} = Mem_{max}/Mem_{i,j}$$

where $A^{Time}_{i,j}$ ($A^{Mem}_{i,j}$) is equal to 1 for time-out (memory-out) runs, i.e., when $Time_{i,j} = Time_{max}$ ($Mem_{i,j} = Mem_{max}$).

As performance prediction (or problem coverage estimation) for MC is very difficult, we deliberately avoid exact estimations. We thus do not compute $f_A(time, mem, stats)$ in the continuous domain, but we adopt discretized values.

As a first attempt in this direction, we classified each benchmark in the HWMCC'08 suite as *easy*, *medium*, *hard*, and *unsolved*, considering the best results delivered by our portfolio. Those data are reported in Table 1.

The table shows that, given a time bound of 15 minutes, 519 problems out of 645 could be considered easy. Of the remaining ones, 66 are solved (49 medium and 17 hard), and 60

Table 1 A synthetic classification of the 645 HWMCC'08 designs, based on the verification results gathered with PdTRAV

Result	Easy	Medium	Hard	Unsolved
	[0, 10 s[	[10 s, 2 min[	[2 min, 15 min[	[15 min, ∞]
SAT	215	16	7	13
UNSAT	304	33	10	9
UNKNOWN				38
TOTAL	585 (238 + 347)			60

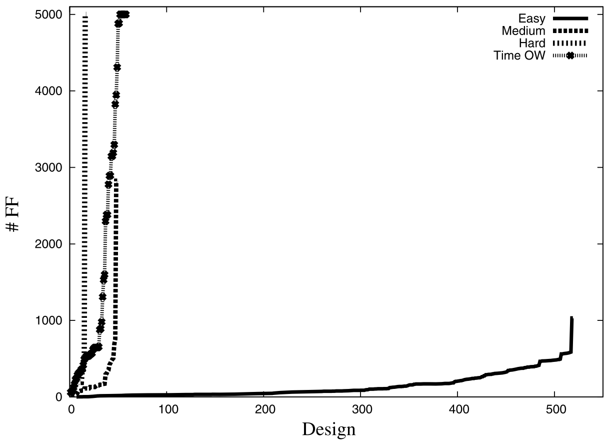

Fig. 3 Plotting the number of memory elements for easy, medium, hard and time overflow designs

remain unsolved. Overall, considering also known results from published references, 251 cases are classified as SAT, 366 are UNSAT, and 38 remain unknown (i.e., our tool was unable to complete 22 cases solved by other model checkers ranked at the HWMCC'08). Although very coarse, the above classification basically leads to the conclusion that just 77 benchmarks (12%) of the HWMCC'08 instances could be considered as challenging for our tools. Those are cases on which we had to focus.

As a further step to characterize affinity between benchmarks and engines, we measured the relation between circuit sizes and execution times. Figure 3 plots the number of latches for the four previously described execution classes. We limit the scale along the *y* coordinate to 5000 latches, even if a few benchmarks are beyond this value. It is easy to observe that easy cases are typically smaller (most of them are largely below 500 latches), whereas the number of latches increases for medium, difficult and unsolved cases. Notice that the plots also indicate that, whenever used as a prediction for overall run-times, circuit size (number of latches) is not a precise indication.

To be more specific, at a very coarse level, it was possible to identify 4 sub-groups of engines (BDD, ITP, IND and BMC). The problems covered by the four classes were: 446 for BDD, 545 for ITP, 478 for IND, and 287 for BMC. The best engine, from the ITP class, was able to solve 533 problems. The overlap among classes was also relevant, as just 25 problems were solved in one class only, respectively (9, 7, 1, 8). This value increased by filtering-out results of combined techniques, such as [20, 21], originally classified as ITP. In that case, the overall number of unsolved circuits was 42, distributed as (28, 4, 2, 8). The data clearly indicated that BDDs was definitely required within our portfolio, and the

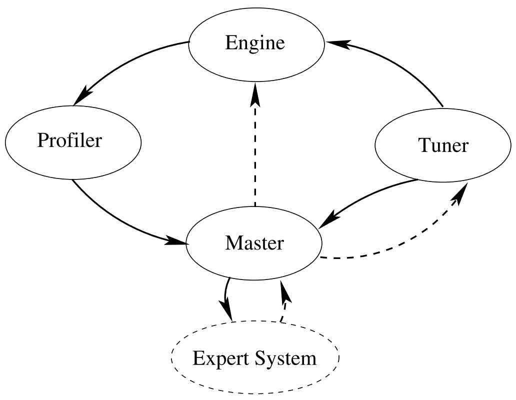

Fig. 4 The architecture of engine performance and tuning modules

optimization margin for the expert system was in the range between the best single engine (533) and the overall best (585).

Although the room for improvements was low, to provide dynamically updated affinity measures, we started a finer study of affinities, at the level of individual runs.

4.3 Affinity measures: dynamic evaluation

In this sections we concentrate on how to obtain affinity measures "a-priory", i.e., by exploiting data extracted from partial runs.

To reach the above goal, we adopt a modular, state-based approach, where each MC engine (the *Engine*) is controlled by three main modules (the *Profiler*, the *Tuner*, and the *Master*). The conceptual diagram is represented in Fig. 4. In the diagram, solid lines represent data flow, and dotted lines indicate control. Moreover, the *Engine* module roughly correspond to the groups identified in Sect. 4.2, i.e., our main MC approaches with their different options and tunings. Furthermore, the *Expert System* module (described in Sect. 5) includes the top level procedure, which is in charge of selecting and activating the verification routines, by interfacing with the *Master* module.

The characteristics of each module are described in the following paragraphs, where we will focus our discussion on our two main engines (BDD and ITP), whereas the other ones (IND and BMC) will not be described in details.

The Profiler Module The profiler module gathers statistics on all key procedures within the MC tool. More specifically, it collects information about the resource usage and the performance of each image computation, sifting procedure, interpolant evaluation, etc. All data are then summarized in a discrete form and collected in a *profiler table*. Tables 2 and 3 report a high level view of the profiler tables for the BDD and ITP engines, respectively.

Both tables keep track of data related to size and time performance. For the BDD engine, we first look at the data collected during transition relation (T) construction, then we dynamically keep track of image computation (IMG), symbolic traversal (Traversal), and dynamic reordering (Sifting). Based on these data, it is possible to classify several execution profiles for image computations by observing how much internal computational effort is required (peak BDD, sifting, execution time), compared to the size of the produced state sets. Similarly, it is possible to characterize the entire reachability analysis process, as traversals may vary from short (low diameter) with high intermediate BDD peaks, to long (counter-like) with almost no peak, and rather uniform distribution of BDD sizes and image computation

Table 2 A profiler for the BDD engine. The symbol #X indicates "the number of X", and $|X|$ "the size of X" (in terms of BDD or AIG nodes)

ENTITY	SIZE	TIME
T	# BDD nodes	Building
	# cut points (pseudo inputs)	
	# clusters	
IMG	$\lvert From \rvert$	Last VS Average computation
	$\lvert To \rvert$	
	Peak BDD nodes	
	# sifting	
Traversal	History of $\lvert Reached \rvert$ (e.g., flat, increase-light, peaks)	History of IMG (e.g., uniform, increase-light, peaks)
	Depth (e.g., short, medium, high)	
Sifting	Current threshold	Last VS Average Computation
	Average compaction ratio	

Table 3 A profiler for the interpolant engine. The symbol #X indicates "the number of X", and $|X|$ "the size of X" (in terms of BDD or AIG nodes)

ENTITY	SIZE	TIME
$C = \textsc{Itp}(A, B)$	$\lvert A \rvert$	Last VS Average Computation
	$\lvert B \rvert$	
	Size of the proof	
	$\lvert C \rvert$	
Traversal	History of $\lvert \textsc{Itp} \rvert$ (flat, increase-light, peaks)	History of ITP (uniform, increase-light, peaks)
	History of $\lvert R \rvert$	
	Depth (short, medium, high)	
	# false failures	
	# skipped steps	
$C = \textsc{Itp}(A, B)$ (with red. removal)	# support variables Average compaction ratio	Combinational optimization (Last VS Average)

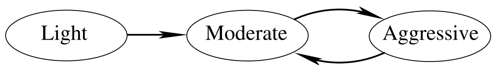

Fig. 5 The tuning policy graph

time. We proceed in a similar way for the interpolant-based approach, where we keep track of interpolant computation and optimization effort, and of the overall traversal, that is related to the sequential depth, and the number of generated false counter-examples.

The Tuner Module The *Tuner*, given a target policy, selects the proper parameter settings for the engine. It is driven using an FSM-based effort model, which can be described by the state graph of Fig. 5. In this case arcs represent state transitions. Roughly speaking, for each engine we have three main policies:

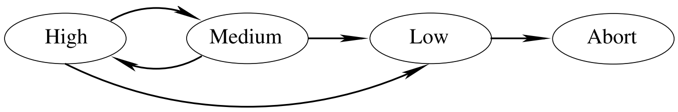

Fig. 6 The affinity state diagram

- *Light*, characterized by minimal overhead, for easy problems.
- *Moderate*, general-purpose, with some optimizations and related overhead, for medium-size problems.
- *Aggressive*, special-purpose, with high overhead, targeted for specific optimizations (e.g., image computation, reachable state set representation, deep or short traversals, ITP size or tightening, dynamic abstraction, etc.) and difficult properties verification.

The tuning policy is initially set to *Light*. Depending on performance profiling, the master can trigger a transition to *Moderate*. Whenever performance profiles give an indication of how to choose a more specific setting, e.g., optimize image computation (by re-clustering or by disjunctive partitioning), aggressive interpolant compaction, etc., the state control moves to the *Aggressive* state. As all MC algorithms potentially go through intermediate peak resource usage, we allow moving back from *Aggressive* to *Moderate*, whenever a significant decrease in resource usage is detected. This is greedily motivated by the purpose of reducing the overhead and speeding-up the tail execution phase, potentially demanding less resources. On the contrary, we do not allow a further backward transition to the *Light* state.

The Master Module The master module controls all activities of a given engine by observing profiled statistics (through the *Profiler* module), and activating dynamic tunings (through the *Tuner* module). Moreover, it acts as a slave module of the expert system (see Sect. 5) which activates the master of the selected engine, by specifying the proper resource limits.

Based on the tuning effort (*Light*, *Moderate* or *Aggressive*), the proposed time and memory resource bounds, and the profiler table, the master module computes an affinity value for the controlled engine. Similar to the tuning effort, affinity is managed through a FSM-like model, as shown in Fig. 6. Affinity values are in the range from *High* to *Low*, expanded to an *Abort* state, introduced to model no-affinity, and force the engine to stop its run. The initial affinity value is set by a preliminary static evaluation, called by the expert system. Usually, when an engine is selected, the affinity is set to *High*. After that, the affinity is adjusted depending on performance statistics. From the *High* state it can move to *Medium* or *Low* (e.g., very expensive unpredicted SAT call). *Low* is the state where tuning is typically set to aggressive and performance is observed to potentially stop the engine moving to the *Abort* state.

The *Master* module is also granted the option of internally adjusting its resource limits, depending on effort and affinity measures.

Case study 1: BDD traversal We now show how the BDD master module can heuristically implement a template-based policy for circuit pdtvisns3p00, one of the HWMCC'10 benchmarks solved by PdTRAV using BDDs. Figure 7 plots the BDD size of the *To* state set, the BDD size of the *Reached* state set, the number of live nodes, and the number of peak BDD nodes (as provided by the CUDD package) as a function of the forward traversal iteration. Figure 8 reports the CPU time for each traversal iteration (i.e., image computation) with and without the CPU time for the sifting procedure filtered-out.

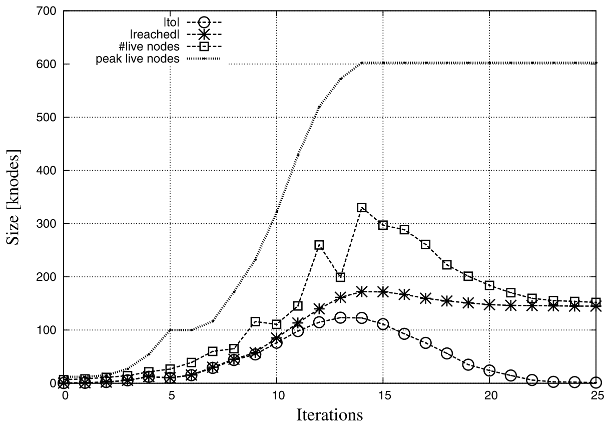

Fig. 7 BDD size of the state sets as a function of the traversal iteration for benchmark pdtvisns3p00

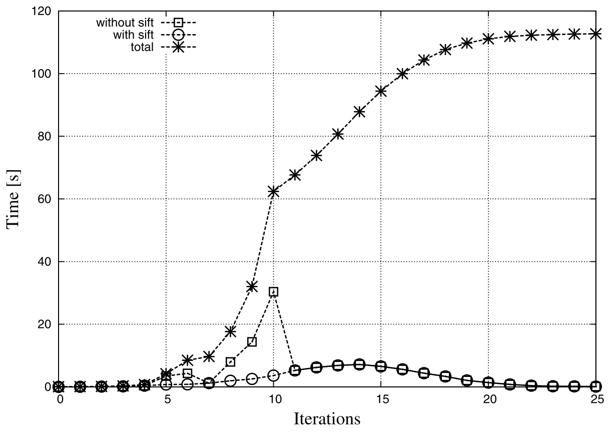

Fig. 8 Time for computing a BDD-based image as a function of the traversal iteration for benchmark pdtvisns3p00

The plots for *To* and *Reached* clearly show the intermediate traversal peak at iterations 13–14. After that, the size of the reached set becomes stable, whereas the size of the *To* set decreases. Image computation times have a similar profile, with sifting overhead clearly visible at iterations 5–10. It is also interesting to see how the ratio peak BDD size versus live BDD nodes (which filter out the nodes internally used and freed after each image computation) is in the range from 2 to 3 in the area before peak usage (iteration 14). This can be used heuristically to guess that image computation has an internal peak BDD usage from 2 to 3 times greater than the input/output state sets.

In our *template*-based policy symbolic traversals start with affinity and tuning set to *High* and *Light*, respectively. In our case, BDD sizes, and image computation times, increase rapidly. Moreover, sifting is triggered quite often. Depending on the chosen time and memory limits, we then change affinity to *Medium*. From this value affinity can move back to *High*, only if BDD sizes and image computation times stabilize or start decreasing. On the other hand, affinity goes to *Low* in correspondence of high BDD sizes, reached with steep increase rate, with sifting at high thresholds and/or high image execution times. Depending on a more detailed observation of the profiler table, tuning can go from *Moderate* to *Aggressive* for image and/or for reachable states. *Abort* state is triggered either at resource limits or (with flexible limits) based on a prediction of high resource usage, starting from profiled data. We generally consider this as the threshold between a *Moderate* and an *Aggressive* tuning, that we trigger with higher internal peak BDD nodes.

Case study 2: Interpolant-based MC In the case of interpolant-based MC (ITP) we basically concentrate on estimating the diameter of the circuit (and consequently the BMC bound at fixed-point or counterexample), and the effort required to represent and compute interpolants (for state sets). We initially guess a short diameter, unless otherwise suggested by a quick preliminary ternary simulation step. We consider a bound *Short*, *Medium* or *Long*, if the number of iterations is below 20, in the range 20–100 or above 100, respectively. In each range, we adopt a different tuning by observing SAT solver statistics and interpolant computation:

- SAT activity is characterized by SAT execution times and size of refutation proofs.
- Interpolants are characterized by the number of variables in their support, their AIG size (before and after combinational optimization), their growth rate across different traversal iterations.

ITP affinity starts *High*, and it is kept high as far as depth estimation is *Short* or *Medium*, and interpolant computation is relatively fast. Tuning is either *Light* or *Moderate*. Affinity is also kept *High* with *Long* runs, in case of quick interpolant computation. Tuning is possibly moved to *Aggressive*, either for more compaction effort, or to select different overall strategies (e.g., dynamic abstraction [18, 19]). Affinity changes to *Low* when both SAT calls and interpolants become difficult. In other cases, we go to *Medium* affinity, that is kept until SAT and interpolants become hard to compute. We put a limited effort on large interpolants, then we move to *Low* affinity and we possibly trigger the *Abort* choice.

5 Building a MC expert system

Our long-term goal is to go towards a multi-threaded application, with benchmarks characterized by multiple properties, and overall time limits ranging from minutes to hours. Under this perspective, the expert system we describe here is just an intermediate step, with a limited scope and a specific target framework, i.e., the HWMCC'10, with single threaded processes, single property, 15 minutes time limit, and mainly including easy-medium benchmarks.

Given this target, we basically take initial static (lightweight) decisions, followed by run of selected engines with increasing limits. Following other verification tools (see for instance [11, 33]), we run our engines from the cheapest to the most aggressive ones. More in detail:

- The sequence of engines to use is established by an initial classification of the benchmark, based on a static analysis of design statistics (see Sect. 4.1).
- Each one of the activated engines (see Sect. 4.3), estimates its affinity with the benchmark under check, and implements a dynamic self-tuning. Moreover, it can possibly abort the verification task by adjusting resource thresholds.

The resulting pseudo-code is given in Fig. 9. In line 2, function ANALYZE estimates an initial value for the affinity based on a static analysis of the design, i.e., it decides the verification algorithms to apply, based on the structural characteristics of the problem under check. In practice, the design is assigned to a class. After that, function REDUCE simplifies

```
 1  EXPERTVERIFY ($M$, limits)
 2      class = ANALYZE ($M$)
 3      ($M'$, class, result) = REDUCE ($M$, class)
 4      while (result == UNKNOWN)
 5          if (class == SMALL)
 6              result = MASTERBDD ($M'$, class, limits$_L$, flexible)
 7          else if (class == MEDIUM)
 8              result = MASTERBDD ($M'$, class, limits$_M$, flexible)
 9              if (result == UNKNOWN)
10                  result = MASTERITP ($M'$, class, limits$_L$, flexible)
11          else if (class == LARGE)
12              result = MASTERBMC ($M'$, class, limits$_M$, flexible)
13              if (result == UNKNOWN)
14                  result = MASTERIND ($M'$, class, limits$_L$, strict)
15          else if (class == SEC)
16              result = MASTERIND ($M'$, class, limits$_L$, flexible)
17          else if (class == INDUSTRIAL)
18              result = MASTERITP ($M'$, class, limits$_L$, flexible)
19          else // OTHER: no specific class identified
20              result = MASTERBMC ($M'$, class, limits$_S$, strict)
21              if (result == UNKNOWN)
22                  result = MASTERIND ($M'$, class, limits$_M$, strict)
23              if (result == UNKNOWN)
24                  result = MASTERITP ($M'$, class, limits$_L$, flexible)
25          class == UPDATE(class, result)
26      return result
```

Fig. 9 The top level "expert system" module

the model using both general purpose transformations (i.e., COI reduction) and some class-specific algorithms (e.g., two phase abstraction, retiming, hidden constraint detection, etc.). Function REDUCE sometimes is able to determine the overall result directly ending the verification procedure. Besides providing a simplified FSM model, function REDUCE also re-evaluates (and possibly re-assigns) the design class.

A loop is then entered, and a sequence of MC approaches is deployed, according to the initially chosen class, by activating the corresponding master module (lines 6, 8, 10, etc.). It is worth recalling that, based on the dynamic affinity computation discussed in Sect. 4.3, each master implements a finer engine tuning and possibly aborts the verification task. Every master essentially accepts four parameters, as some specific settings have been hidden in the pseudo-code:

1. The FSM under check M'.
2. The class, to be internally used for initial tuning and/or strategy selection.
3. A resource (time and memory) limit (in the pseudo-code, the limit has been discretized into three levels, i.e., short (S), medium (M) and high (H)).
4. A flag (*flexible*) or (*strict*) indicating whether the called procedure is allowed to automatically adjust the resource limit.

The above parameters are exploited by every master module as introduced in Sect. 4.3. Then, every master returns the verification result, which can be SAT, if the property has been disproved, UNSAT if it has been proved, or UNKNOWN for incomplete runs. In the last case, a different (master) verification strategy is applied, either within the same class, or even another one, modified by the UPDATE function at the end of each loop.

The description of the classes is the following one:

1. SMALL: This class is assigned to FSMs with a small number of latches (a few tens at most, 75 in our setting for HWMCC'10). BDD-based traversals are usually enough to solve these instances.
2. MEDIUM: This class still identifies designs within the range of applicability for BDDs, but larger than the SMALL ones (number of latches up to 150). In this case BDDs are attempted first, but we then resort to standard interpolation, if BDDs fail to complete the verification task.
3. LARGE: The LARGE class includes very large designs (e.g. $>$ 3000 latches, which are outside the scope (for the given limits) of all the "heavy" approaches. We thus only apply a few lightweight techniques, i.e., BMC and simple induction.
4. SEC: When the verification problem is recognized to be a Sequential Equivalence Checking instance (by static analysis of the property), the inductive engine is selected as the preferred verification routine. In our experience, this method (possibly coupled with retiming) is the most effective to solve this kind of problems.
5. INDUSTRIAL: Designs are tagged as INDUSTRIAL when some characteristics are fulfilled, e.g., one or more clock variables are defined or (possibly hidden) constraints are available. In this case we use interpolation.
6. OTHER: The OTHER class includes all the remaining designs, for which we apply a sequence of BMC, induction and interpolation, with increasing time limits.

Finally, it is important to remark that, according to the pseudo-code of Fig. 9, the same verification procedure may be called more than once as an effect of the class update. For instance, BDD-based traversals may be applied to a design initially tagged as SMALL, but then called again if this method leads to an abort and the class is modified to MEDIUM. Nonetheless, the actual settings used throughout these two calls are internally modified, so that there is no real work duplicated. This detail is omitted in the pseudo-code for the sake of simplicity.

6 Experimental results

PdTRAV is a state-of-the-art verification framework, oriented to evaluate promising verification techniques. The architectural description of the package is included in Sect. 3. The adopted algorithmic portfolio is described in Sect. 2.3. In this section, we present an *historical perspective* on how the tool had been tuned for the HWMCC'10, to get the best possible results in terms of solved designs, in the slotted time limit.

For our experiments, we adopted a benchmark "training set" represented by 645 HWMCC'08 [44] circuits, each one with a single property. Although this choice biased the tool on the adopted set, the proposed benchmarks represented the largest set of publicly available MC problems before the HWMCC'10. All our experiments ran on a Dual Core 3 GHz Workstation with 3 GBytes of main memory, hosting a Debian Linux distribution. Time limit was set to 900 seconds (as in the HWMCC competition), and the memory limits to 1 GByte. Since our main focus was on proved properties, in this section we will exclude the BMC engine, whenever providing detailed data and comparisons among the MC strategies.

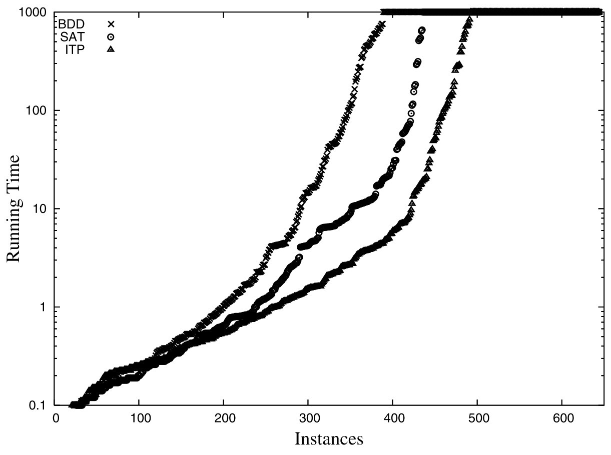

Fig. 10 Results for "standard" BDD, ITP and IND methods

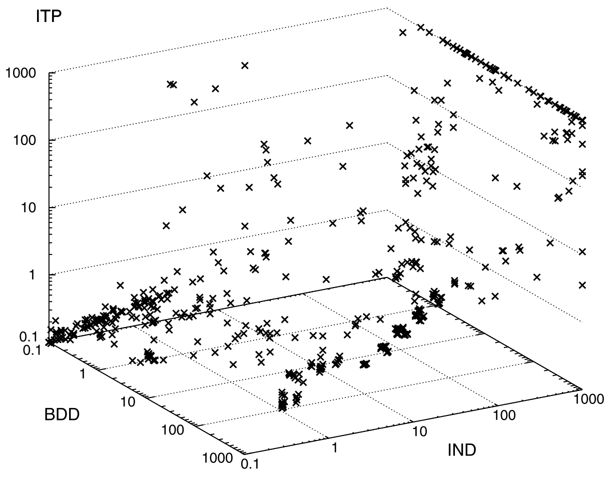

Fig. 11 Performance comparison for "standard" BDD, ITP and IND methods

Figure 10 shows an initial comparison among the 3 main MC approaches (able to both verify and falsify a property) in the PdTRAV portfolio. The graphs plot (on logarithmic scales) the results obtained with a "standard" implementation of the engines based on reachability analysis (BDD), interpolation (ITP), and induction (IND), on all the HWMCC'08 instances ordered by increasing CPU time.

All experiments with a CPU time equal to 1000 actually ran out of time. More interestingly, Fig. 11 represents the same set of data on a 3-D comparison among the techniques. The graph shows a relatively low correlation among the three main engines on the entire benchmark suite and strongly motivates (as discussed in Sect. 4.2) the need of a smarter approach than the pure "winner takes it all".

To get into the matter even more, we analyzed the 585 designs belonging to the HWMCC'08 suite solved by at least one of the 25 different strategies initially belonging to our portfolio. In more details, the less efficient technique was able to solve 370 problems, while the most efficient one 533. As a consequence, the difference between 533 and 585, i.e., 52 designs, was the initial optimization space left for any multi-engine or expert system strategy.

Before proceeding to more complex analysis and optimizations, we estimated the results we could attain with a few trivial partitioning and selection techniques:

- Adopting 7 different strategies (among the 25 originally available), each one of them running for 130 seconds, it was possible to solve 569 problems (out of 585).
- Splitting all designs in 4 different sub-sets, depending on their size, and adopting, on each set, a different group of strategies with different time slots it was possible to prove 576 properties. The best setting in this case was the following. The 4 classes include designs with: $[0, 100[$, $[100, 200[$, $[200, 500[$, $[500, \infty]$ memory elements. For the smallest set of circuits, the best resolution scheme was (again) to adopt 7 different strategies each one for 130 seconds. Those could solve 323 benchmark out of 329 belonging to the class. For the largest set, we had to use the interpolation-based engine (with a retiming pre-process) for about 800 seconds, and a forward BDD-based engine for the remaining 100 seconds. In this way, we could solve 20 out of 63 designs belonging to the class.

This experimental analysis showed that a very straightforward (and light weight) algorithms selection strategy could deliver interesting benefits, allowing PdTRAV to complete 576 designs instead of 533. Furthermore, as 576 is quite close to 585, i.e., the best possible result reachable with PdTRAV in its initial configuration, using one of the 25 different initial MC algorithms, we also worked in the direction of further improving the verification power of the PdTRAV engines.

After the previous experimental work we could make the following observations:

- Given the adopted CPU time limit, the HWMCC benchmark suite seems to include mainly easy-to-solve designs, few medium instances, and some hard or impossible to solve designs.
- The 900 seconds CPU time limit seems too small to appropriately face the hardest instances of the suite.
- The performance of PdTRAV can be improved by a very simple heuristic, based on the size of the designs and a CPU time preemption scheme. In this way, PdTRAV is able to verify a number of designs quite close to the optimum one which a perfect oracle would reach.
- The previous target, i.e., optimizing the number of completed instances on the entire suite, seems to be in contrast with other possible (somehow more interesting) targets, such as trying to solve the hardest (corner case) instances in the benchmarks set.

Convinced that more interesting results could be obtained with a finer analysis, we performed a second tuning phase driven by the following considerations:

- We focused on difficult and unsolved instances.
- We performed a deeper investigation of the 3 main MC engines, coming to a finer tuning of each of them.
- We insert in our portfolio 11 new engines obtained by mixing the original 3 standard MC strategies with "collateral" optimization techniques, such as:
  - Detecting equivalence checking cases.
  - Discovering hidden constraints (in the design or in the property, see [45]).
  - Optimizing BDD size through cut-point insertion.
  - Using ternary simulation.

  At the end of this phase we obtain a portfolio of 36 different engines/settings.
- We run all 36 engines on the 645 designs of the HWMCC'08 suite and we collect the data to put in place the expert system, as described in Sect. 5.

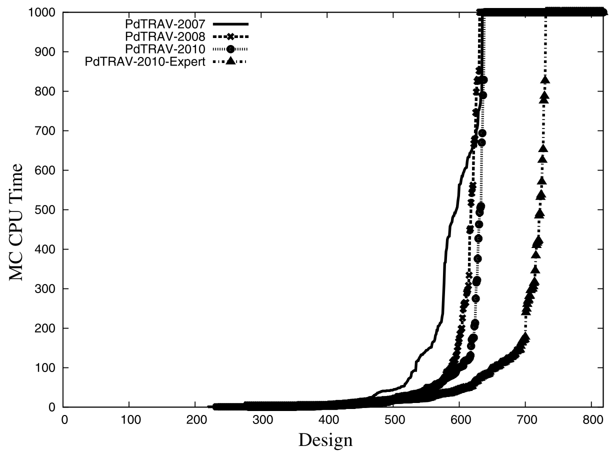

Fig. 12 Performance comparison for PdTRAV2007, 2008 and 2010

Once all those features had been added to our package, PdTRAV could verify 591 properties (out of 645). This means that it could solve 6 designs more than the previous maximum/optimum limit (585), and 15 more than the result obtained with a trivial heuristic based on the design size and time preemption.

To confirm these results, we finally presents a comparison of several versions of PdTRAV on the HWMCC'10 benchmark suite, including 818 problems (173 more than the HWMCC'08 suite). More specifically, we compare (on the same machine) the following PdTRAV versions: The one which participate to the 2007 competition (names PdTRAV-2007), the one of the 2008 competition (named PdTRAV-2008), the best engine available before the implementation of the expert system (PdTRAV-2010) and the one which participated at the competition, which includes the expert system described in this work (version PdTRAV-2010-Expert).

Notice that these data somehow duplicate the ones presented at the competition in the corresponding years, but also include some tool upgrades and updates. Moreover, all versions of the tool ran on the newest set of benchmark and on the same hardware and software platform, and are then directly comparable. Figure 12 plots the data obtained with these versions of the tool, ordered by increasing CPU time. The CPU time limit was again set to 900 seconds (as during the competition, even if, given the difference in platform, we can have some differences in the final data). Results somehow show continuous but modest improvements moving from PdTRAV-2007, to PdTRAV-2008 and to PdTRAV-2010, but much more evident with the final version of the tool.

To conclude, Fig. 13 reports a comparison among the expert system and the results obtained with the best engine available in PdTRAV just before the competition. In this case, we restrict our analysis on the HWMCC'08 suite, since this is the benchmark set for which the complete experimental evaluation of the original 25 techniques we took into account was available. The plot shows how close the expert system gets to the best obtainable result. More specifically, the expert system shows some overhead on some of the easy properties, but improvements on the medium-difficult ones.

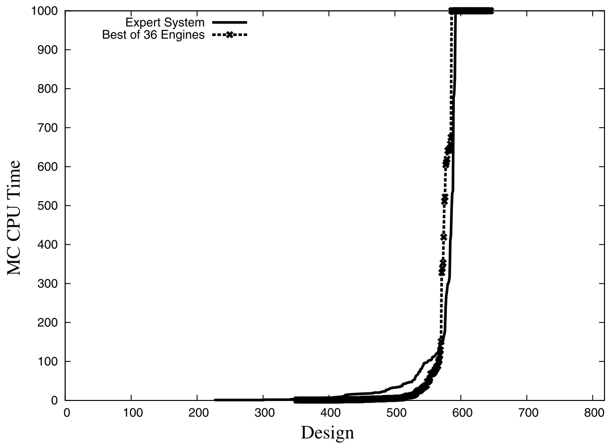

Fig. 13 Expert system compared with the best available result (obtained with 36 engines)

7 Conclusions and future work

This paper addresses the issue of dynamically generating and exploiting an experimental characterization/evaluation of benchmarks and MC approaches, within a multi-engine MC tool.

The overall framework is open to possible improvements, and the potentialities have shown to be promising. Moreover, the proposed scheme can inspire new improvements of existing algorithms and the forecast of new methodologies.

Although the report concentrates on the publicly available HWMCC suite, similar experiences have been collected on industrial benchmarks. To this respect, it should be noticed that neither the competition suite nor the given time limits are really appropriate to test portfolio-based approaches.

Among the future works, we would like to better investigate how to optimize the tool, in order to perform on multi-core architectures, by moving towards a multi-threaded application, with benchmarks characterized by multiple properties, and overall time limits ranging from minutes to hours.

Acknowledgements This work was supported in part by SRC contract 2009-TJ-1968.

References

1. Bjesse P, Leonard T, Mokkedem A (2001) Finding bugs in an alpha microprocessor using satisfiability solvers. In: Berry G, Comon H, Finkel A (eds) Proc computer aided verification, Paris, France, July 2001. LNCS, vol 2102. Springer, Berlin, pp 454–464
2. Biere A, Klaessen KL (2010) The hardware model checking competition web page. http://fmv.jku/hwmcc10
3. Rice JR (1976) The algorithm selection problem. Adv Comput 15:65–118
4. Huberman B, Lukose R, Hogg T (1997) An economics approach to hard computational problems. Science 275(5296):51–54
5. Xu L, Hutter F, Hoos HH, Leyton-Brown L (2008) Satzilla: Portfolio-based algorithm selection for sat. J Artif Intell Res 32(1):565–606
6. Pulina L, Tacchella A (2009) A self-adaptive multi-engine solver for quantified boolean formulas. Constraints 14(1):80–116
7. Gordon M (1989) Mechanizing programming logics in higher order logic. In: Current trends in hardware verification and automated theorem proving. Springer, Berlin

8. Srivas M, Rueß H, Cyrluk D (1997) Hardware verification using PVS. In: Formal hardware verification: methods and systems in comparison. Springer, Berlin
9. Kaufmann M, Manolios P, Moore JS (2000) Computer-aided reasoning: an approach. Kluwer Academic, Dordrecht
10. Kamhi G, Fix L, Binyamini Z (1998) Symbolic model checking visualization. In: Proceedings of the second international conference on formal methods in computer-aided design. FMCAD '98, London, UK. Springer, Berlin, pp 290–303
11. Mony H, Baumgartner J, Paruthi V, Kanzelman R, Kuehlmann A (2004) Scalable automated verification via expert-system guided transformations. In: Proc formal methods in computer-aided design. Springer, Berlin, pp 159–173
12. Cabodi G, Camurati P, Quer S (1994) Detecting hard faults with combined approximate forward/backward symbolic techniques. In: Proc IEEE ISCAS'94, London, UK, May 1994, pp 25–30
13. Biere A, Cimatti A, Clarke EM, Fujita M, Zhu Y (1999) Symbolic model checking using SAT procedures instead of BDDs. In: Proc 36th design automation conf, New Orleans, Louisiana, June 1999. IEEE Computer Society, Los Alamitos, pp 317–320
14. Sheeran M, Singh S, Stålmarck G (2000) Checking safety properties using induction and a SAT solver. In: Hunt WA, Johnson SD (eds) Proc formal methods in computer-aided design, Austin, TX, USA, November 2000. LNCS, vol 1954. Springer, Berlin, pp 108–125
15. Bjesse P, Claessen K (2000) SAT-based verification without state space traversal. In: Proc formal methods in computer-aided design, Austin, TX, USA, LNCS, vol 1954. Springer, Berlin
16. McMillan KL (2003) Interpolation and SAT-based model checking. In: Hunt WA Jr, Somenzi F (eds) Proc computer aided verification, Boulder, CO, USA, LNCS, vol 2725. Springer, Berlin, pp 1–13
17. Cabodi G, Murciano M, Nocco S, Quer S (2006) Stepping forward with interpolants in unbounded model checking. In: Proc int'l conf on computer-aided design, San Jose, California, November 2006. ACM Press, New York, pp 772–778
18. Cabodi G, Camurati P, Murciano M (2008) Automated abstraction by incremental refinement in interpolant-based model checking. In: Proc int'l conf on computer-aided design, San Jose, California, November 2008. ACM Press, New York, pp 129–136
19. Cabodi G, Murciano M, Nocco S, Quer S (2008) Boosting interpolation with dynamic localized abstraction and redundancy removal. ACM Trans Des Autom Electron Syst 13(1):309–340
20. Cabodi G, Camurati P, Garcia L, Murciano M, Nocco S, Quer S (2008) Trading-off SAT search and variable quantifications for effective unbounded model checking. In: Proc formal methods in computer-aided design, Portland, Oregon, November 2008, pp 205–212
21. Cabodi G, Garcia LA, Murciano M, Nocco S, Quer S (2010) Partitioning interpolant-based verification for effective unbounded model checking. IEEE Trans Comput-Aided Des Integr Circuits Syst 29(3):382–395
22. Cabodi G, Nocco S, Quer S (2009) Strengthening model checking techniques with inductive invariants. IEEE Trans Comput-Aided Des Integr Circuits Syst 28(1):154–158
23. Biere A, Jussila T (2007) The hardware model checking competition web page. http://fmv.jku.at/hwmcc07
24. Berkeley Logic Interchange Format Technical report, September 1996
25. The AIGER format. http://fmv.jku.at/aiger/
26. Cabodi G, Camurati P, Quer S (2000) Symbolic forward/backward traversals of large finite state machines. Euromicro J 46(12):1137–1158
27. Cabodi G, Nocco S, Quer S (2002) Mixing forward and backward traversals in guided-prioritized BDD-based verification. In: Brinksma E, Larsen KG (eds) Proc computer aided verification, Copenhagen, Denmark, July 2002. LNCS, vol 2102. Springer, Berlin, pp 471–484
28. Cabodi G, Camurati P, Quer S (1999) Improving the efficiency of BDD-based operators by means of partitioning. IEEE Trans Comput-Aided Des Integr Circuits Syst 18(5):545–556
29. Cabodi G (2001) Meta-BDDs: a decomposed representation for layered symbolic manipulation of boolean functions. In: Berry G, Comon H, Finkel A (eds) Proc computer aided verification, Paris, France, July 2001. LNCS, vol 2102. Springer, Berlin, pp 118–130
30. Cabodi G, Nocco S, Quer S (2007) Boosting the role of inductive invariants in model checking. In: Proc design automation & test in Europe conf, Nice, France, April 2007. IEEE Computer Society, Los Alamitos
31. Kuehlmann A, Ganai MK, Paruthi V (2001) Circuit-based Boolean reasoning. In: Proc design automation conference, Las Vegas, Nevada, June 2001. IEEE Computer Society, Los Alamitos
32. Cabodi G, Crivellari M, Nocco S, Quer S (2005) Circuit based quantification: back to state set manipulation within unbounded model checking. In: Proc design automation & test in Europe conf, Munich, Germany, March 2005. IEEE Computer Society, Alamitos
33. Brayton R, Mishchenko A (2010) Abc: an academic industrial-strength verification tool

34. Baumgartner J, Mony H, Paruthi V, Kanzelman R, Janssen G (2006) Scalable sequential equivalence checking across arbitrary design transformations. In: Proc formal methods in computer-aided design
35. Een N, Mishchenko A, Amla N (2010) A single-instance incremental SAT formulation of proof and counterexample abstraction. In: Proc int'l workshop on logic synthesis, May 2010
36. Coudert O, Berthet C, Madre JC (1989) Verification of sequential machines based on symbolic execution. In: LNCS, vol 407. Springer, Berlin, pp 365–373
37. Burch JR, Clarke EM, McMillan KL, Dill DL, Hwang LJ (1990) Symbolic model checking: 10^{20} states and beyond. In: Proc symposium on logic in computer science, Washington, DC, June 1990. Springer, Berlin, pp 428–439
38. Craig W (1957) Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. J Symb Log 22(3):269–285
39. Pudlák P (1997) Lower bounds for resolution and cutting plane proofs and monotone computations. J Symb Log 62(3):981–998
40. Somenzi F (2005) CUDD: CU decision diagram package—release 2.4.1. http://vlsi.colorado.edu/~fabio/CUDD/
41. Brayton RK et al (1996) VIS: a system for verification and synthesis. In: Alur R, Henzinger TA (eds) Proc computer aided verification. LNCS, vol 1102. Springer, Berlin, pp 428–432
42. Eén N, Sörensson N (2009) The Minisat SAT solver, April 2009. http://minisat.se
43. Mishchenko A, Chatterjee S, Brayton RK (2005) FRAIGs: a unifying representation for logic synthesis and verification. Technical report, EECS Dept, UC Berkeley, March 2005
44. Biere A, Jussila T (2008) The hardware model checking competition web page. http://fmv.jku.at/hwmcc08
45. Cabodi G, Camurati P, Garcia L, Murciano M, Nocco S, Quer S (2009) Speeding up model checking by exploiting explicit and hidden verification constraints. In: Proc design automation & test in Europe conf, April 2009, pp 1686–1691