Journal on Satisfiability, Boolean Modeling and Computation 9 (2015) 135-172

Hardware Model Checking Competition 2014: An Analysis and Comparison of Model Checkers and Benchmarks

Gianpiero Cabodi gianpiero.cabodi@polito.it
Carmelo Loiacono carmelo.loiacono@polito.it
Marco Palena marco.palena@polito.it
Paolo Pasini paolo.pasini@polito.it
Denis Patti denis.patti@polito.it
Stefano Quer stefano.quer@polito.it
Danilo Vendraminetto danilo.vendraminetto@polito.it
Dipartimento di Automatica e Informatica
Politecnico di Torino
Turin, Italy

Armin Biere biere@jku.at
Johannes Kepler University
Linz
Austria

Keijo Heljanko keijo.heljanko@aalto.fi
Aalto University
Aalto
Finland

Abstract

Model checkers and sequential equivalence checkers have become essential tools for the semiconductor industry in recent years. The Hardware Model Checking Competition (HWMCC) was founded in 2006 with the purpose of intensifying research interest in these technologies, and establishing more of a science behind them. For example, the competition provided a standardized benchmark format, a challenging and diverse set of industrially-relevant public benchmarks, and, as a consequence, a significant motivation for additional research to advance the state-of-the-art in model checkers for these verification problems. This paper provides a historical perspective, and an analysis of the tools and benchmarks submitted to the competition. It also presents a detailed analysis of the results collected in the 2014 edition of the contest, showing relations among tools, and among tools and benchmarks. It finally proposes a list of considerations, lessons learned, and hints for both future organizers and competitors.

KEYWORDS: *Model Checking, Equivalence Checking, Binary Decision Diagrams, Satisfiability Solvers, Interpolation, IC3, Property Directed Reachability*

Submitted December 2014; revised August 2015; published January 2016

1. Introduction

Semiconductor devices have proliferated into most aspects of our daily lives. Contemporary hardware devices often comprise a tremendous amount of complexity and diversity. Exposing all the bugs in such designs, which may number in the hundreds or even thousands, is an extremely challenging task. An unexposed hardware bug which escapes into silicon often risks severe penalties, ranging from economic loss and product market failure to safety concerns, including the risk of loss of human life. For this reason, the use of formal methods has become an attractive approach to overcome the coverage limitations inherent to simulation-based validation. Therefore, most semiconductor companies have investigated their applicability. Companies' individual choices of methods, tools, and application areas have varied, as has their level of success. With recent advances in the scalability of automated model checkers and sequential equivalence checkers, most companies have grown to rely upon these techniques to some extent. However, the inherent scalability limitations of these techniques remain a challenge to broader application, and hence they have not managed to completely displace simulation-based methodologies for the most part.

The Hardware Model Checking Competition (HWMCC) was founded in 2006 with several practical goals: Developing a standardized benchmark format reflective of industrially-relevant model checking problems; establishing a large and diverse set of challenging public benchmarks; providing a scientific framework to evaluate the effectiveness of research advances in the field; and overall, creating an increased motivation for continued research in model checking. First editions of the competition primarily focused on providing an infrastructure for comparing model checkers, creating a large set of public benchmarks, and involving young researchers in the field, riding along the success of previous SAT and SMT competitions. Since its foundation, the format of the competition has been in continuous evolution: Additional tracks were introduced, better I/O and OS conformance was imposed on the tools, resource limitations were increased to leverage the availability of more powerful hardware, etc. However, the main competition setup remains the same, with every competing tool for each track running on the same set of benchmarks, under the same time and memory limits.

Although the overall results of the competition have traditionally been presented either at the Formal Methods in Computer-Aided Design (FMCAD) or Computer-Aided Verification (CAV) conferences, an in-depth analysis of the details of the competitions has always been lacking. The purpose of this paper is to present an accurate analysis of the benchmarks, the tools, and their mutual relationships. We also present a detailed analysis of the results collected in the 2014 edition of the competition, providing some considerations on portfolios, model checking engines, and trends in formal verification. We finally propose a list of considerations, lessons learned, and hints allegedly useful for both future organizers and competitors.

1.1 Roadmap

The paper is organized as follows. Section 2 introduces the necessary background, and the adopted notation. Section 3 describes the evolution of the competition over the years. Section 4 illustrates all main organizational choices such as the contest system, and the final ranking criteria. Section 5 concentrates on contestants by briefly describing all tools

submitted to the 2014 competition (more details can be found in the reported references). Section 6 is devoted to the benchmark suite, its main characteristics and design subsets, as well as its evolution over the years. Section 7 presents an evaluation of the results, with a detailed analysis under various points of view. Section 8 outlines lessons learned from both the organizers' and the contestants' perspective. Finally, Section 9 concludes the paper giving some summarizing remarks.

2. Background

The hardware model checking competition has focused so far on hardware models described at the bit-level. A problem instance in the competition consists of a bit-level circuit (described in the simple and-inverter format AIGER [2]), combined with a set of properties and optional constraints. The AIGER format was designed to provide a structurally compact, and easy to parse, representation for bit-level model checking and sequential equivalence checking problems. It evolved over the years to match both the needs of the competition and requests from industrial users, in order to be used as an intermediate format in various verification and design flows. The AIGER format only considers bad state and justice properties. Constraints can be simple combinational invariant constraints as well as fairness constraints. The main goal in developing AIGER was to provide a common denominator of all bit-level formats, with clean and simple semantics, which nonetheless allows users to encode relevant industrial model checking problems.

On the application side, properties may have many different flavors. Classically there are safety and liveness properties, sequential equivalence as well as coverage goals, test generation, redundancy checks etc. In theory, checking these properties on bit-level models, that is to determine whether they hold, is PSPACE-complete. More practically, all of these properties can be mapped into the problem of symbolic reachability, e.g., whether a certain state in a sequential circuit can be reached starting from an initial state.

In AIGER this generic sequential reachability property is considered as a "bad state property" to highlight the connection to its dual problem, which is checking that a simple safety property holds. Such a reachability or bad state property is *satisfiable* (*SAT*) if a bad state is reachable. Otherwise it is considered to be *unsatisfiable* (*UNSAT*). In the *SAT* case, the model checker is also often expected to generate a witness trace, i.e., a sequence of states which shows how to reach the bad state from an initial state.

This semantic mirrors other related competitions, including the SAT [6], SMT [1], and CASC [55] competitions, where the problems considered have exactly the same "refutational" meaning. Also in these other competitions, satisfiability of a problem/property, which means existence of a model, requires the solver to produce a witness. This again matches the semantics of *SAT* in AIGER.

In the rest of the paper we will refer to *SAT* safety problems as those for which in principle a trace, showing that a bad state is reachable, exists. For liveness problems, *SAT* means that a trace exists which satisfies the justice properties and all the constraints. For an *UNSAT* problem, of either kinds, such a trace does not exist, e.g., the bad state is unreachable or there is no trace on which the justice properties hold (infinitely often under all specified constraints).

Even though in principle model checking can be reduced to reachability checking, encoding, and then using a safety/reachability checker might not be the fastest solution. In order to make use of algorithms specifically targeted towards solving liveness properties, the organizers introduced justice properties in AIGER. This is a compromise between fixing a specific high-level temporal logic, and what actual algorithms internally use. This restriction still permits to easily, and compactly, encode the temporal property part of many liveness property benchmarks, including benchmarks from industrial sources, and allowed the organizers to establish a *liveness* track (*LIVE*).

Many hardware models have multiple properties, thus it is natural to keep them together, and not to split them into several benchmarks. It is also believed that, in the context of multiple properties, model checkers can perform better if all properties are known at once. However, in a competition context, how to rank performance of model checkers on sets of benchmarks with multiple properties is still an open problem.

Initially the HWMCC considered only benchmarks with exactly one property, which was actually restricted to be a bad state property. This was the setting for the first competition, and it is still how the most important track of the competition, the *single* safety property track (*SINGLE*), is organized. The organizers continue to use this historical terminology, even though also for the liveness track a single (liveness) property per benchmark is considered. With the same abuse of notation the *multi* property track (*MULTI*) only considered bad state properties (safety properties), but more than just one per benchmark. In future competitions, the organizers might want to include an *open* track, where liveness and safety properties occur mixed in an arbitrary number per benchmark.

Finally, in industrial usage, model checking is often applied in the form of bounded model checking [10]. This is mostly due to capacity restrictions of complete, i.e., unbounded, model checkers. However, in certain cases, even verification plans have an explicit notion of how "deeply" formal verification must be performed, which in essence just specifies the bound up to which verification is deemed to be sufficient. In this context, on unsatisfiable benchmarks, the primary metric for assessing the performance of a model checker is most naturally given by the largest bound up to which the model checker can prove that a bad state can not be reached from the initial states, under a given time limit.

The *deep* bound track of the competition (*DEEP*) tries to determine the best model checker in this scenario. The benchmarks considered are taken from the single safety property track. Model checkers participating in this track are requested to print the bound up to which they claim a bad state can not be reached, while trying to solve a benchmark. The *DEEP* track is actually a virtual track, since the maximum bound for a model checker on a benchmark is determined during the *SINGLE* track. Only instances unsolved in the *SINGLE* track are considered for the ranking in the *DEEP* track. A model checker is considered buggy in the *DEEP* track if it prints a bound which is larger or equal to the length of a witness given by another model checker.

3. History

The first Hardware Model Checking Competition was held in Berlin, Germany, with CAV 2007. The *AIGER* format was introduced by the organizers as the standard benchmark format for the competition.

Originally, only the *SINGLE* track was carried out. Experiments were run on a cluster of 15 nodes. Each node was running a GNU/Debian Linux 2.6.8 operating system, featured a single Intel Pentium IV core running at 3 GHz, and included 2 GB of main memory. For each run, CPU time and memory were respectively limited to 900 seconds and 1.5 GB.

The second competition was held in Princeton, USA, with CAV 2008. The adopted hardware was the same of the previous year but the operating system was upgraded to Ubuntu Linux 7.10.

In 2010, the competition was held in Edinburgh, Scotland, with CAV. During this competition, the highly influential algorithm IC3 [12], sometimes referred to as Property Directed Reachability (PDR) [25], made its first appearance. The hardware setup was upgraded to a cluster of 32 Intel Quad Core nodes, running at 2.6 GHz, with 8 GB of main memory. Each model checker had full access to one node of the cluster, i.e., to 4 cores. Memory limitation on each run was increased to 7 GB.

In 2011 the competition was held in Austin, USA, with FMCAD. Two new tracks were introduced that year: The multiple properties track (*MULTI*) and the liveness track (*LIVE*). The original *AIGER* format specification was updated to version 1.9 in order to support invariant constraints (safety assumptions), liveness (justice) properties, and fairness constraints. The same hardware setup of the previous year was maintained.

In 2012 the competition was held in Cambridge, England, with FMCAD. The deep bound track (*DEEP*) was added during that edition, alongside previously introduced tracks. The same hardware configuration of 2011 was used, but the operating system on cluster nodes was upgraded to Ubuntu 12.04.

In 2013 the competition was held in Portland, USA, with FMCAD, keeping the same format, and hardware setup, of the previous year.

In 2014 the competition was held in Vienna, Austria. The results were presented at the FLoC Olympic Games. Only the *SINGLE*, *LIVE*, and *DEEP*, tracks were featured in that edition. For the first time, contestants were required to produce witness traces for the *SINGLE* track, in order to support *SAT* claims. The competition was run on a cluster at Aalto University composed of 32 nodes. Each node featured 2x Six-Core AMD Opteron 2435 running at 2.6 GHz with at least 16 GB of RAM. Such a configuration granted to each model checker exclusive access to 12 cores. Memory limitation on each run was also increased to 15 GB.

Table 1 shows the number of different research groups, and the number of distinct model checkers submitted to each competition. A dash associated to a given track for a year means that such a track wasn't there on that particular occasion.

Table 1. Number of different research groups, and number of submitted model checkers per year.

	SINGLE		*MULTI*		*LIVE*		*DEEP*		Total	
Year	#Groups	#Checkers	#Groups	#Checkers	#Groups	#Checkers	#Groups	#Checkers	#Groups	#Checkers
2007	8	19	–	–	–	–	–	–	8	19
2008	6	16	–	–	–	–	–	–	6	16
2010	8	21	–	–	–	–	–	–	8	21
2011	6	16	4	6	4	7	–	–	7	18
2012	7	18	6	9	4	8	5	13	7	25
2013	11	23	4	8	5	9	8	16	11	28
2014	9	16	–	–	7	8	7	10	9	19

4. Organizational Choices

Organizers usually perform a "preliminary phase" before the "official" experiments are run and the final ranking is computed. During this phase, competitors are allowed to tune and debug their tools on sample benchmarks from previous competitions. This is a necessary step in order to help submitters adapt to changes in the input format, and to address issues that could potentially lead to some contestants being disqualified.

Notice that, if a policy as strict as the one used in the SAT competition, in which resubmission is not allowed and incorrect solvers are immediately disqualified, would have been enforced, some tracks in past editions of HWMCC would barely have had a single competitor. An alternative approach, considered in the SMT competition, would be to penalize incorrect solutions with negative scores, but otherwise count correctly solved benchmarks towards the ranking. This way, however, the ranking would also include incorrect solvers, which the organizers regard as an even inferior solution.

For each track, the top three performing tools are awarded virtual gold, silver, and bronze medals respectively. As a research group is allowed to submit multiple tools or tool versions, virtual podium for medals is made by just considering the best result for each group. The ranking criterion is based on the number of solved instances in the allotted time limit, which was set to 900 seconds in all editions. Though this limit can be considered too low for many industrial scale problems, it was selected as a compromise based on the computing resources and time window (days) available to run the competition.

Using the number of solved instances within the timeout as the primary criterion for ranking contestants is arguably the most accepted approach. This scheme is also used in other competitions such as SAT, SMT, and CASC. The number of instances each contestant is able to solve is independent from one another, which allows to "replay" the competition independently and simplifies the use of competition results in papers. In the context of the SAT competition, various other ranking schemes were considered. After some deliberation, a democratic poll among SAT solver developers on which ranking scheme to use gave very strong support to only consider the number of solved instances per solver, with ties broken by CPU or wall-clock time. The organizers of the HWMCC followed the example set by the SAT competition.

Using wall-clock time[1.] versus using only process time, both as time limit and for tie-breaking, has the advantage to encourage parallelization of model checkers. However, a potential risk is to discourage development of new algorithms or improvements in single engines. In early incarnations of the SAT competition, process time was used for ranking solvers in a parallel track. This was the source of an engaging debate between organizers and solver submitters. In the end, it became clear that, at least for a parallel track, wall-clock time should be used. In SAT this has now been the standard for a couple of years. The SMT competition encourages parallel solvers, but, in our view, inconsistently uses process time to rank them.

Until 2011, time computation in the HWMCC has been based on (single threaded or process) CPU time. In other words, the score of each tool has been equal to the number of

1. The wall-clock time is the time necessary to a (mono-thread or multi-thread) process to complete the task, i.e., the difference between the time at which the task finishes and the time at which the task started. For this reason, the wall-clock time is also known as *elapsed time*.

problems solved in 900 seconds. Starting from 2012, a multi-threaded ranking, i.e., based on wall-clock time, has been used. This was done in order to emphasize multi-core CPU exploitation, and to encourage portfolio-based multi-threaded solutions. As a matter of fact, most of the latest participants follow this trend. The concurrency level of each tool can be guessed from experimental data, that report both CPU and wall-clock run times. In this sense, all HWMCC tracks are parallel tracks, which is also not ideal. It might be an interesting option for future incarnations of HWMCC to split the competition into two parts, a parallel and a sequential part, similar to the SAT competition, and use wall-clock time (limits) for the parallel tracks, and process-time (limits) for the sequential tracks.

For safety and liveness properties, an instance is considered to be solved by a tool if such a tool either proves the property to hold or to be violated. An instance is considered unsolved by a tool if the time or memory limits are exceeded or the tool crashes.

For the *MULTI* track, the scoring is based on a weighted formula, that takes into consideration the number of properties for each design, such as:

$$score \quad = \quad \frac{1}{N} \sum_{i=1}^{N} \left(\frac{\#solved_i}{\#properties_i} \right)$$

where N is the number of benchmarks, the index i is used to enumerate the benchmark in the set, and $\#solved$ and $\#properties$ denote the number of solved and overall properties for the benchmark, respectively.

For the *DEEP* track, the scoring is based on the following formula, which emphasizes robustness in reaching deep bounds:

$$score \quad = \quad \frac{1}{N} \sum_{i=1}^{N} \left(1 - \frac{1}{2 + maxbound_i} \right)$$

where, $maxbound_i = 0$ means that the model checker only proved that no initial state is bad, and accordingly $maxbound_i = -1$ is used to denote that the solver failed to even prove that. In the first case these benchmarks contribute a score of 0.5 to the average, and nothing in the latter case.

5. Model Checkers

In 2014 the competition attracted 9 research groups that submitted a total of 19 model checkers to the three available tracks. Some groups submitted multiple instances of the same tool, using different settings, while others presented multiple distinct tools, each targeting different types of problems or tracks. Some tools competed in more than one track simultaneously. A few of the submitted tools are highly specialized for solving *SAT* problems and/or for competing in the *DEEP* track.

The following subsections describe the model checkers participating at the 2014 competition, with a specific emphasis on the adopted algorithms, and settings. For each model checker, a brief historical description is provided, along with some notes on the authors, and the main objectives of the tool. A final comprehensive analysis concludes the section.

5.1 Contenders

5.1.1 *AIGBMC* and *BLIMC*

AIGBMC and *BLIMC* have both been developed by Armin Biere from Johannes Kepler University, Linz, Austria.

AIGBMC is part of the *AIGER* distribution. It was originally developed as reference implementation for the *AIGER* 1.9 format introduced at HWMCC 2011. The idea was to cross-validate competition results, and also to make it available before the competition to allow competitors to validate their adoption of the *AIGER* 1.9 format. *AIGBMC* implements an instance of the bounded model checking encoding presented in [31, 27], but only focuses on satisfiable instances. The first implementation used *PicoSat* [7] as SAT solver, while the new version (the one used in HWMCC 2014) relies upon Lingeling [8]. Lingeling uses sophisticated pre-processing techniques, though less powerful in incremental settings. For example, Lingeling "freezes" latch variables in the last unrolled time-frame for bounded model checking [24, 29], and it contains a function for "cloning" the SAT solver instance [42, 43, 44].

BLIMC is a bounded model checker for safety properties implemented on top of Lingeling. Beside serving as a test bed for the incremental usage of Lingeling, firstly it uses the same cloning idea as *AIGBMC* (but with a different conflict limit), and then it simplifies the transition relation using SAT level pre-processing as suggested in [29]. The simplified transition relation is then copied each time the problem is extended by a new time-frame.

It has to be noticed that *BLIMC* and *AIGBMC* partecipated at the competition but not at the final ranking.

5.1.2 *PdTRAV*

PdTRAV (*P*olitecnico *d*i *T*orino *R*eachability *A*nalysis and *V*erification) has been developed by the Formal Methods Group (Gianpiero Cabodi, Sergio Nocco, and Stefano Quer) at Politecnico di Torino, Italy [17].

It mainly targets safety property verification, and it is developed in C, with some C++ extensions, mainly needed for linking the core tool with external packages. The complete kit includes *AIGER* (for handling AIG file format), *CUDD* (as a low level BDD library), *MiniSat* (as a SAT solver), and *ABC* (exclusively used for combinational rewriting and sequential redundancy removal such as signal and latch correspondence). The portfolio includes random simulation (SIM), BMC, BDD-based (forward and backward) reachability, k-induction (IND), interpolation (ITP [16] and IGR [19]), and IC3.

The tool applies model transformations before running the final verification engines. Those simplifications are selected by a lightweight expert system (a simplified version of [18]). The tool is multi-threaded, running up to 8 different engines on a "winner-takes-it-all" basis. An expert system is in charge of selecting the proper engines for the current problem, according to a previous classification step and following further heuristics.

Alongside *PdTRAV*, a second version of the tool, called *PdTRAV*h, was submitted to HWMCC 2014. *PdTRAV*h is a slightly modified version of *PdTRAV*, with engines detached and controlled by a two-level thread/engine management scheme: Sub-managers are dedicated to ITP, IC3, BDD, and SAT-oriented (simulation and BMC) engines, grouped in sub-portfolios of 2 or 3 engines (or variants of the same engine with different settings).

5.1.3 *ABC*

ABC [14, 38] has been developed by the Berkeley Verification and Synthesis Research Center, namely by Baruch Sterin, Robert Brayton, Niklas Eén, and Alan Mishchenko from the University of California at Berkeley. It participated in the 2014 competition with four different versions, namely *SUPROVE*, *SIMPSAT*, *SUPDEEP*, and *SIMPLIVE*.

The development of *ABC* started from the re-engineering of MVSIS, a tool for the synthesis of multi-valued and Boolean circuits, with the objective to place a higher focus on AIG circuit representation. A fundamental premise of *ABC* is to leverage the synergy between synthesis and verification using efficient SAT-based Boolean reasoning on the AIG for combinational and sequential equivalence checking. The tool is written in C and offers a wide variety of algorithms for both synthesis and verification:

- Synthesis of combinational circuits: AIG balancing, rewriting and re-factoring, structural hashing, AIG sweeping, etc.
- Synthesis of sequential circuits: Retiming, redundant latch and signal elimination, sequential cleanup, etc.
- Equivalence checking of combinational circuits (SAT-sweeping).
- Equivalence checking of sequential circuits (both bounded and unbounded).

ABC includes the C version of *MiniSat* for handling Boolean reasoning problems and the package *CUDD* for BDD manipulation. In order to compete in the HWMCC, the Berkeley group developed a set of integrated model checkers as Python scripts on top of *ABC*. The main model checking engines integrated in those tools are: Property Directed Reachability (IC3), Bounded Model Checking (BMC), Rarity simulation (RareSim), Craig interpolation and BDD-based reachability. The IC3 algorithm is implemented as described in [25], with several different variants. *ABC* also includes several versions of BMC.

5.1.4 *TIP* and *TIPBMC*

TIP (Temporal Induction Prover) is a model checker targeting safety and liveness problems. It was originally developed by Niklas Eén and Niklas Sörensson during their PhD studies at Chalmers University, Sweden, with the purpose of showcasing how to efficiently implement BMC and induction using incremental SAT solving [41]. The version of *TIP* that has been competing in HWMCC is a complete rewrite of the original one, implemented by Niklas Sörensson and Koen Claessen. The source code for *TIP* is available on github [57].

TIP is implemented in C++ and relies heavily on *MiniSat* for all SAT-based algorithms. It contains only two engines for safety properties: BMC and IC3. For liveness, it uses k-liveness [23] (backed by IC3), and the more standard liveness-to-safety transformation [9] (backed by BMC). The only circuit transformations used are a simplified implementation of temporal decomposition [20], and redundant latch merging.

TIP is a single-threaded single-engine tool whose only concurrent feature is to interleave IC3 and BMC (in a rather ad hoc fashion). All engines are multi-property aware; even mixed liveness and safety can be handled within a single IC3 or BMC context. In the competition, different configurations were used for different tracks. For the *SINGLE* track, two versions

of *TIP* were submitted, one with only BMC and one with only IC3. For the *LIVE* track, a version with only k-liveness and a version with both k-liveness and BMC were submitted. For the *DEEP* track, a version with only BMC was used.

5.1.5 *V3* and *V3db*

V3 has been developed by Cheng-Yin Wu, under the consultation of Chi-An Wu and the supervision of Professor Chung-Yang (Ric) Huang, from the National University of Taiwan [66, 30, 15].

V3 has been written in C++, and it offers Boolean-level and word-level design manipulation and verification capabilities. The version submitted to the competition exploits *MiniSat* as the underlying SAT engine. It can manipulate both single and multiple safety properties as well as liveness properties.

Like other contestants, it exploits light-weight optimization techniques, but its main focus is to share information learned from different model checking algorithms rather than applying heavy synthesis optimization methods. In order to share information among verification engines, *V3* creates a virtual cloud (called "a *model checking cloud*") which serves as a communication channel for model checking algorithms. Information shared among threads include reached bounds and reachable states information.

V3 is portfolio based, and it exploits six model checking algorithms (sequential redundancy removal, simulation, BMC, induction, interpolation, and IC3) with a total of more than 30 different configurations. Regarding interpolation, the tool implements McMillan's original algorithm as well as a counterexample-guided version (named NewITP in [15]). There are two different configurations of IC3: One follows the original IC3 formulation that exploits multiple SAT solvers, while the other uses only one solver to maintain reachability over time-frames.

The tool implements a strict resource control mechanism for CPU time, memory usage and level of concurrency (number of cores that the tool can access). The strategy dynamically balances memory usage and the number of active threads over time.

Two versions of the tool, *V3* and *V3db*, were submitted to the HWMCC 2014.

5.1.6 *ShiftBMC* and *ShiftBMCpar*

The author of the *ShiftBMC* tool [34] is Norbert Manthey from the Technische Universität in Dresden, Germany.

The focus of *ShiftBMC* is to provide an incremental environment for the SAT solver Riss, developed by the same author, rather than to offer a complete multi-engine model checking tool. Its purpose is to showcase how recent developments in SAT technology can improve the performance of existing BMC tools.

Riss [36] includes several SAT related tools, such as the formula simplifier Coprocessor, and the parallel portfolio SAT solver Priss. Coprocessor [35] includes several recently developed CNF simplification techniques, and it can also be applied to QBF and MaxSAT formulas. Priss [33] is able to emit unsatisfiability proofs in the DRAT format [28].

ShiftBMC currently supports only single safety properties. It adopts *AIGBMC* to read and encode *AIGER* files, and it uses the *ABC* library to perform some initial simplifications. Once *ABC* has run, the circuit transition function is translated into CNF by the *AIGER*

tools, then Coprocessor performs some CNF simplifications, applying different techniques in a hand-crafted order. To avoid wasting too much time performing simplification on larger circuits, some heuristically selected limits are used to keep the CNF simplification time below a certain threshold. The final BMC problem is generated by writing the simplified transition formula into CNF by shifting the variables in a proper way.

Two versions of the tool, *ShiftBMC* and *ShiftBMCpar*, were submitted to the HWMCC 2014.

5.1.7 *IIMC*

IIMC (Incremental Inductive Model Checking) has been written by Zyad Hassan, Fabio Somenzi, Michael Dooley and Aaron Bradley from the University of Colorado, Boulder, USA [26].

IIMC is a parallel tool targeting safety property verification, language emptiness and checking of CTL properties. It is written in C++11 and uses *CUDD*, zChaff and *MiniSat* as external tools. *MiniSat* is used as a SAT solver for everything with the exception of IC3 which uses zChaff instead.

The pre-processing phase of the designs can involve any of the following techniques: Abstract interpretation of the model, BDD and SAT-based combinational redundancy removal, ternary simulation-based redundancy removal, phase abstraction, and redundancy removal via extraction of unit-literal invariants. To address a specific family of benchmarks, namely backward Beem (see Section 6.2.3), *IIMC* performs transition relation reversal as well.

IIMC is based on the concept of Incremental, Inductive Verification (IIV). IIV algorithms construct proofs by generating lemmas based on concrete hypothesis states, and subsequently derive more information from such lemmas through generalization. Incrementality refers to the fact that those lemmas hold relative to previously generated ones. The main IIV engines are IC3, Fair and IICTL [13]. The portfolio is enriched with the addition of a few BBD- and BMC-based strategies. The IC3 implementation benefits from some improvements such as lifting, counterexamples to generalization (CTG) [67] and localization reduction. The portfolio used during the competition exploits a thread-based approach and is composed of four engines running with default setup: IC3, reverse IC3, BMC, and either BDD-based forward reachability or IC3 with localization reduction. The usage of reverse IC3 is motivated by its good capabilities at finding counterexamples.

The portfolio setup for the *LIVE* track has been different in terms of engines, but maintains the same thread-based characteristics. In this context, the engines of choice are: Fair Cycle BMC (FCBMC), GHS (BDD-based language-emptiness check), and Fair.

5.1.8 *nuXmv*

nuXmv is developed by Cavada, Cimatti, Dorigatti, Griggio, Mariotti, Micheli, Mover, Roveri, and Tonetta from Fondazione Bruno Kessler (FBK), Trento, Italy [22, 21].

nuXmv (eXtended NuSMV) is the evolution of NuSMV, an open source symbolic model checker for finite-state systems that has been developed as a joint project between Carnegie Mellon University, University of Trento and Fondazione Bruno Kessler (FBK). At the HWMCC 2014 was submitted a portion of the entire tool, called *nuXmv*-bitcore, which

includes the subset of *nuXmv* functionalities capable of handling bit-level safety and liveness verification.

The tool is written in C/C++, and is designed to be SAT solver independent. It can be currently interfaced with both *PicoSat* and a version of *MiniSat* properly modified to produce resolution proofs.

nuXmv includes a vast portfolio of SAT-based safety property and liveness verification algorithms, such as McMillan's interpolation [37], ITPSEQ [59], DAR [61], k-induction [49], BMC [10], *AVY* [64], IC3 [12] alongside BDD-based reachability algorithms guided by regular expressions [56]. IC3 is implemented in three different variants: The "standard" IC3 algorithm (combining ideas from several implementations), IC3 with CTGs and IC3 with lazy abstraction-refinement [60]. The latter comes in two versions, depending on the approach used to handle the refinement step: The original one based on IC3, and a new variant based on BMC.

For liveness model checking, *nuXmv* implements BMC-based algorithms [11] as well as k-liveness [23] integrated within an IC3 framework. Some pre-processing techniques for the extraction of stabilizing constraints from liveness problems are also implemented.

5.1.9 *AVY*

AVY [62] was developed by Arie Gurfinkel, from the Software Engineering Institute at CMU, and Yakir Vizel from Princeton University.

The tool focuses on interpolation and PDR as presented in [64]. Extensions to the original tool are presented in [63], even though not all those strategies are part of the submitted version.

The main model checking algorithm it uses, called *AVY*, can be viewed as a synergy of both interpolation and IC3, even though it is more interpolation oriented. Like interpolation, it uses unrollings and interpolants to construct an initial candidate invariant and, like IC3, it uses local inductive generalization to keep the invariants in compact clausal form. *AVY* uses a single SAT solver instance globally: It can roam over the entire search space, and does not break it into local checks as part of a backward search. This addresses the main weakness of IC3 as it does not use any global knowledge during the search. The combination of interpolation and inductive reasoning allows *AVY* to benefit from the advantages of both methods as it uses the solver without guiding it during the search, but it does guide its proof construction.

The core algorithm is implemented in C++. *MiniSat* and Glucose are used as backend SAT solvers, while *ABC* is used for the AIG infrastructure and for other utilities. No sequential optimization is performed, but a Python interface orchestrates a portfolio of different verification engines. The portfolio organization is straightforward: A number of processes, each running different configurations of the main *AVY* algorithm, are run in parallel. Whenever a process finishes with a result the others are killed. Additionally, a process running the *ABC* version of PDR is executed. It is important to note that the tool does not include a native BMC engine.

5.1.10 *LS4*

LS4 has been developed by Martin Suda from Max-Planck-Institut für Informatik of Saarbrücken, Germany. The source code of the tool is publicly available [51].

LS4 is a tool for checking liveness properties of hardware designs, originally developed as a theorem prover for LTL. It internally relies on a single algorithm of the same name, first proposed in [53]. The algorithm is similar to IC3 on the reachability fragment, but employs a unique strategy for handling liveness. From a logical perspective, the algorithm is based on a calculus labeled superposition for PLTL described in [54]. Instead of saturating the given clause set, it uses partial models to guide the inference process. More details are available on Suda's PhD thesis [52].

LS4 is built on top of the SAT solver *MiniSat*, version 2.2. No circuit-specific transformations are used.

LS4 was not specifically tuned for performance in HWMCC and would most likely benefit from various circuit-specific pre-processing techniques, such as those described in [23].

5.2 Overall Analysis and Considerations

Table 2 shows which model checking engines and SAT solving tools are used by the contestants. The first column reports tools with their versions. For each of them, the table reports whether it uses Bounded Model Checking (BMC), interpolation (ITP), IC3, BDD, induction (IND), simulation (SIM), fairness (FAIR), IICTL [13], AVY, and LS4. The last column indicates the SAT solvers used. Notice that *AVY* and *LS4* appear as both tool names and engine names.

A detailed analysis on engine coverage, provided in Section 7, shows that IC3 and BMC are by far the engines yielding the highest number of solved instances. Some tools include other SAT-based engines such as k-induction and Craig Interpolation, implemented in different flavors. Few tools have BDD-based symbolic model checking. Random simulation is present in *SUPROVE*, *V3*, and *PdTRAV*. Abstraction-refinement schemes are present in various forms, following either a counterexample-based or a proof-based scheme, and including either localization type abstraction [65, 39] or speculative reduction [5]. Many tools incorporate some degree of simplifying transformations and property-preserving abstractions to reduce subsequent model checking resources. *ABC* is particularly strong in its reduction capability, and several other competitors use *ABC* as a simplifying pre-process.

As far as the SAT solvers are concerned, the majority of the tools adopt Minisat in different flavors. This seem to show that most tools did not use the latest SAT engines, often presenting better performance at the latest SAT competitions. A few tools do use more than one SAT solver, applying them for different purposes.

6. Benchmarks

6.1 Historical Perspective and Statistics

Tables 3, 4, and 5 report the number of proved (*SAT* or *UNSAT*) and unsolved instances over the years in the main categories (*SINGLE*, *LIVE* and *MULTI* tracks). These tables also illustrate the trend of benchmark characteristics during the various competitions. As a reference, during the HWMCC 2014, the benchmark set was composed of 230 different

Table 2. Comparison of model checking engines and algorithms among contestants.

		BMC	ITP	IC3	BDD	IND	SIM	FAIR	IICTL	AVY	LS4	k-live	L2S	SAT solver
AIGBMC		✓												Lingeling
BLIMC		✓												Lingeling
PdTRAV	*PdTRAV*	✓	✓	✓	✓	✓	✓							MiniSAT
	*PdTRAV*h	✓	✓	✓	✓	✓	✓							MiniSAT
ABC	*SIMPSAT*	✓	✓	✓	✓	✓	✓							MiniSAT
	SUPROVE	✓	✓	✓	✓	✓	✓							MiniSAT
	SUPDEEP	✓												MiniSAT
	SIMPLIVE	✓		✓								✓	✓	MiniSAT
TIP	*TIP*	✓		✓								✓		MiniSAT
	TIPBMC	✓												MiniSAT
	TIPRBMC	✓										✓		MiniSAT
V3	*V3*	✓	✓	✓		✓	✓							MiniSAT
	V3db	✓	✓	✓		✓	✓							MiniSAT
ShiftBMC	*ShiftBMC*	✓												Riss
	ShiftBMCpar	✓												Riss
IIMC		✓		✓	✓			✓	✓					MiniSAT zChaff
nuXmv		✓	✓	✓	✓	✓				✓		✓	✓	MiniSAT PicoSAT
AVY				✓						✓				MiniSAT Glucose
LS4											✓			MiniSAT

instances. Table 3 shows a rapid increase in the number of benchmarks in the early editions until 2010, then a decline in the following editions, where trivial and easy instances have been removed.

Table 3. Single Safety Track: Number of proved (*SAT* or *UNSAT*) and unsolved properties, and average benchmark size, over the years.

Year	Avg Latches	Avg ANDs	Solved			Unsolved	Total
			SAT	*UNSAT*	Total		
2007	582.18	9546.16	194	113	307	37	344
2008	450.09	7420.92	251	346	597	48	645
2010	562.02	8081.17	332	460	792	26	818
2011	940.94	11451.86	95	327	422	43	465
2012	2165.67	34470.90	76	140	216	94	310
2013	13298.50	141988.24	64	119	183	65	248
2014	15097.99	154319.16	54	100	154	76	230

Figure 1 shows a graph-based, more detailed, representation of those changes. Labels of vertices show the number of benchmarks used during the various competitions that resulted as solved (by at least one tool) or unsolved (by all tools). Those values coincide with the ones reported in Table 3. Labels of edges highlight the number of benchmarks introduced every year (floating incoming edges), discarded (floating outgoing edges), that maintain their status, and which ones were previously solved but result as unsolved or vice-versa. For instance, the set of solved instances for the year 2008 is composed of 597 benchmarks, 287 of which newly introduced, 298 solved and 12 unsolved in the previous year. From 2008 to 2010, none of these benchmarks were discarded, 595 were still solvable in the following competition whilst 2 became unsolved. Apart from the newly introduced or disregarded

benchmarks, there is a constant flow of circuits from the unsolved to the solved sets in both directions, with a predominance of the unsolved ones becoming solved.

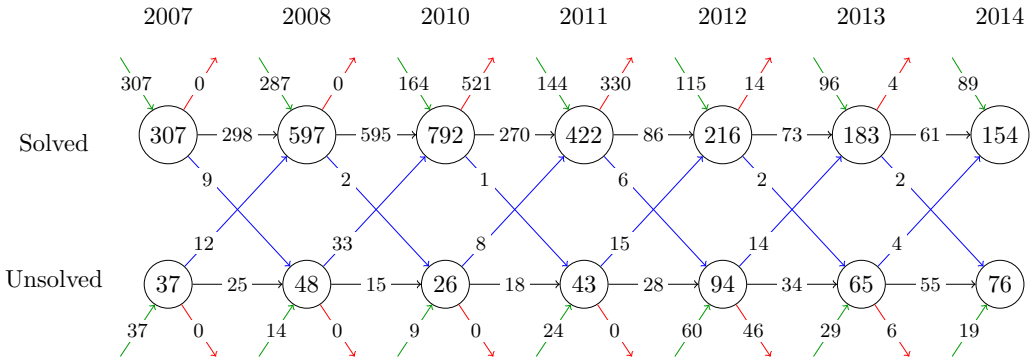


Figure 1. Evolution of *SINGLE* benchmark sets over the years.

Table 3 shows that the numbers of latches and gates have been steadily increasing, thereby making the verification instances allegedly more difficult, from year to year.

As far as the *LIVE* track is concerned, Table 4 shows that for the first two years the same set of liveness benchmarks was used. In the last two years, the data shows an impressive increase in number of instances.

Table 4. Liveness Track: Number of proved (*SAT* or *UNSAT*) and unsolved properties, and average benchmark size, over the years.

Year	Avg Latches	Avg ANDs	Solved			Unsolved	Total
			SAT	*UNSAT*	Total		
2011	101.58	1436.68	57	46	103	15	118
2012	101.58	1436.68	52	49	101	17	118
2013	11946.25	93144.05	80	73	153	32	185
2014	19474.58	150762.02	103	83	186	37	223

Figure 2 shows for those benchmarks the same information discussed for Figure 1. It is easy to notice that there are no outgoing edges as no circuit or property has ever been removed from the liveness benchmark sets.

The *MULTI* track was not carried out in 2014, hence Table 5 shows data only for the previous three years. Notice that in this track, tools showed a large number of discrepancies (i.e., properties reported as *SAT* by one or more tools, and as *UNSAT* by others). As a consequence, the table reports results (that is, the number of *SAT* and *UNSAT* properties) by explicitly taking into consideration the total number of discrepancies (column Discr.).

The number of benchmarks has increased over the years as well as their average size. The number of properties per benchmark has considerably increased in 2012 due to the growth in available industrially-generated benchmarks.

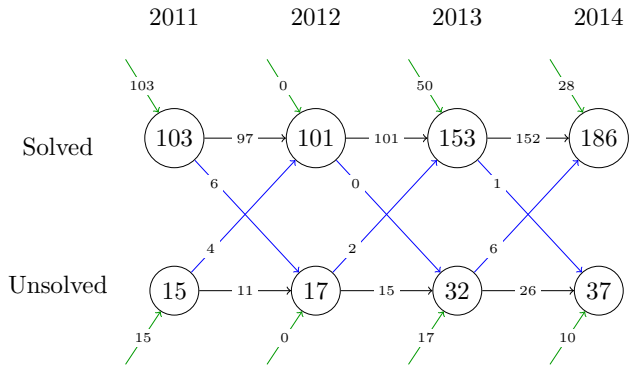


Figure 2. Evolution of *LIVE* benchmark sets over the years.

Table 5. Multiple Properties Track: Number of proved (*SAT* or *UNSAT*) and unsolved properties, and average benchmark size, over the years.

Year	Circuit	Avg Latches	Avg ANDs	Avg Prop.	Solved				Unsolved	Total
					SAT	*UNSAT*	Discr.	Total		
2011	24	819.29	6046.58	117.67	1500	1093	11	2604	220	2824
2012	76	9941.95	132736.96	1066.87	28765	20280	541	49586	31496	81082
2013	178	16887.25	202520.30	1208.03	32481	74782	5	107268	107762	215030

6.2 Main Benchmark Suites

During the HWMCC 2014, the benchmark set was composed of 230 different instances, coming from both academia and industrial settings. Among industrial entries, 145 instances belong to the SixthSense family (6s*, provided by IBM), 24 are Intel benchmarks (intel*), and 24 are Oski benchmarks. Among the academic related benchmarks, the set includes 13 instances provided by Robert (Bob) Brayton (bob*), 4 benchmarks coming from Politecnico di Torino (pdt*) and 15 Beem (beem*). Additionally, 5 more circuits, already present in previous competitions, complete the set.

6.2.1 6s Suite

The *6s* benchmark suite first appeared in the HWMCC 2011, and has grown in scope since. Only a very small fraction of the *6s* benchmarks are contrived. The majority are real industrial hardware verification problems, representing a large diversity of hardware components and verification obligations. Some of these represent verification problems directly from front-end language processing into the AIGER format. These types of hardware verification problems often significantly benefit from front-end transformations to simplify them before applying heavier-weight verification and falsification algorithms [40]. In some cases, transformations may outright solve them. Some of these benchmarks have already been pushed through an aggressive simplifying sequence of transformations, to enable a focus on

core verification algorithms. Some represent highly-transformed sub-problems, e.g., arising during sequential redundancy removal, which help enable the solution of the original verification objective.

These benchmarks are predominantly safety properties and sequential equivalence checking problems. Relatively few of them contain "constraints" (referred to as "invariant constraints" in AIGER terminology). This is largely due to a common industrial verification methodology of defining "drivers" for a test bench. In contrast to constraints, which specify illegal scenarios to be suppressed during verification, drivers are sequential state machines reactive to the design under verification which provide only legal stimuli by construction. Sequential constraints impose verification overhead not only to various formal verification algorithms, but also to semi-formal bug-hunting flows, logic simulation, and acceleration platforms.

Liveness benchmarks are also represented, though in a smaller proportion since only a relatively narrow subset of hardware verification tasks entail liveness checking. Additionally, "bounded liveness" is commonly used industrially to cope with the traditionally lower scalability of liveness checking, since bounded liveness is more natural to check in a simulation or acceleration platform, and since the resulting bounds offer useful insight into design performance.

A total of 421 *6s* benchmarks have been submitted to date. Figure 3 plots their main statistics. For the HWMCC 2014, a subset of the *6s* suite was chosen. Since all tracks supported only single properties, if a selected benchmark had multiple properties, one or a small number of single property benchmarks were derived from randomly-selected properties thereof. In some cases, this rendered a benchmark with challenging properties to be easily solvable. Figure 4 depicts the number of model checkers submitted to the *SINGLE* track (left-hand side) and to the *LIVE* track (right-hand side) which solved these. Note that there is not a significant trend of size versus difficulty to be observed. However, these figures also do not illustrate "reducibility" of these problems; a study of the size of a transformed benchmark, at the point that the solving engine solved it, would likely illustrate more of a trend.

6.2.2 The Oski Suite

The Oski benchmarks encode verification problems on a unit of the publicly available Oracle's OpenSPARC-T1 processor [32], implementing the 64-bit SPARC V9 architecture.

This processor, codenamed "Niagara", is representative of a multi-threaded multi-CPU architecture. The design features a CPU-cache crossbar (CCX) that manages data movement across eight SPARC cores, four L2 cache banks, the I/O bridge (IOB) and the floating-point unit (FPU) of the processor. The processor includes eleven concurrently working arbiters, each of which has eight 16-deep FIFOs for scheduling packet requests. The high level of concurrency makes it hard to exhaustively verify the design with simulation.

The Oski benchmarks are a complete implementation of end-to-end checkers [3] for the CCX portion that manages the data movement from any of the eight SPARC cores to any of the four L2 cache banks, IOB or FPU. The end-to-end checkers are written so that it is possible to get 100% coverage of the functionality of the design. All checkers are implemented as safety properties, originally written in SVA and later converted to AIGER.

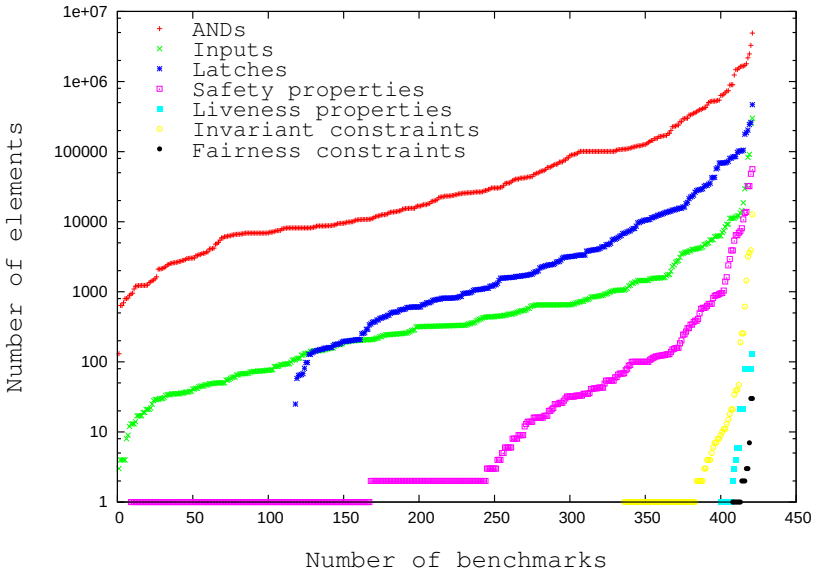


Figure 3. Design characteristics by benchmark.

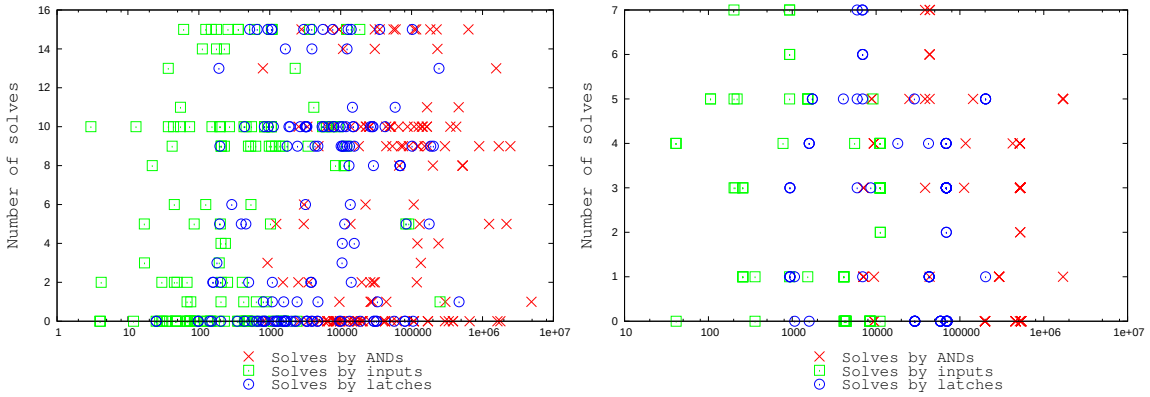


Figure 4. Number of solves with respect to benchmark characteristics for the *SINGLE* track (left-hand side) and *LIVE* track (right-hand side).

A manual analysis as well as a coverage analysis [50] has shown that a bounded proof depth of 24 cycles is enough to get the desired coverage on the design. Unfortunately, for the hardest checker (the one that verifies correctness of data outputs from data inputs), this bound is not easily reachable employing existing tools. Without using some manually crafted abstractions [3], it is not possible to reduce this required bound, so Oski is interested in how far the fully automated solutions can go.

6.2.3 The Beem Suite

The Beem suite [46], which stands for "BEnchmarks for Explicit Model checkers", originally included 300 instances deriving from more than 50 parametrized models. The suite was born in the domain of explicit model checking, where pure brute force is used by tools (such as Spin or Murphi) to exhaustively explore the search space. The models were originally specified in a low-level modeling language (called DVE) based on communicating extended finite state machines. This language is natively supported by the DiVinE [4] (Distributed Verification Environment) model checking tool, which acts as both a model checker and a library. As the language is easy to understand and parse, models have been translated in several other formats, such as AIGER, and a few of them have been selected for the HWMCC. Most of the circuits represent well-known examples and case-studies, such as mutual-exclusion algorithms, protocols, controllers, planning and scheduling algorithms. Models include safety properties, expressed as reachability of a predicate, and liveness properties, described in temporal logic.

6.2.4 The Other Suites

The remaining set of benchmarks includes instances selected among the hard-to-solve ones from previous competitions:

- bob* (13 instances) are benchmarks provided by Robert (Bob) Brayton [48]. Most of them are sequential equivalence checking problems.
- intel* (24 instances) were submitted by Zurab Khasidashvili for the first edition.
- pdt* (4 benchmarks) problems include two synchronous problems coming from modeling the k-king problem [47] (pdtsw*), and two sequential equivalence checking problems (pdtfifo1to0 and pdtpmsudc16).
- cmudme2 and nusmv* are benchmarks generated from existing SMV designs.
- All eijk* benchmarks are sequential equivalence checking problems generated by Van Eijk [58], based on original ISCAS'93 circuits.

7. Results

This section provides an analysis of results obtained by the competing tools at the HWMCC 2014, with the main purpose of characterizing strengths and weaknesses of various engines and techniques, evaluating affinities among the tools, and between tools and solved problems. We first discuss the results of *ABC*, selected as the winner of the *SINGLE* track according to the official 2014 rankings, then we present various statistics on all tools.

7.1 *ABC* results

ABC-based tools competed in the HWMCC 2014 in the following tracks:

- *SUPROVE* and *SIMPSAT* both competed in the *SINGLE* track, achieving respectively the first and second position in the overall ranking. More in detail, *SUPROVE* was the tool able to solve the largest number of problems: 124 in total (46 *SAT* and 78 *UNSAT*). *SIMPSAT* followed closely with 116 solved instances (50 *SAT* and 66 *UNSAT*).
- *SIMPLIVE* won the *LIVE* track with 177 solved problems in total (97 *SAT*and 80 *UNSAT*).
- *SUPDEEP* competed in the *DEEP* track, placing in eighth position with a score of 87.82. It also competed in the *SINGLE* track.

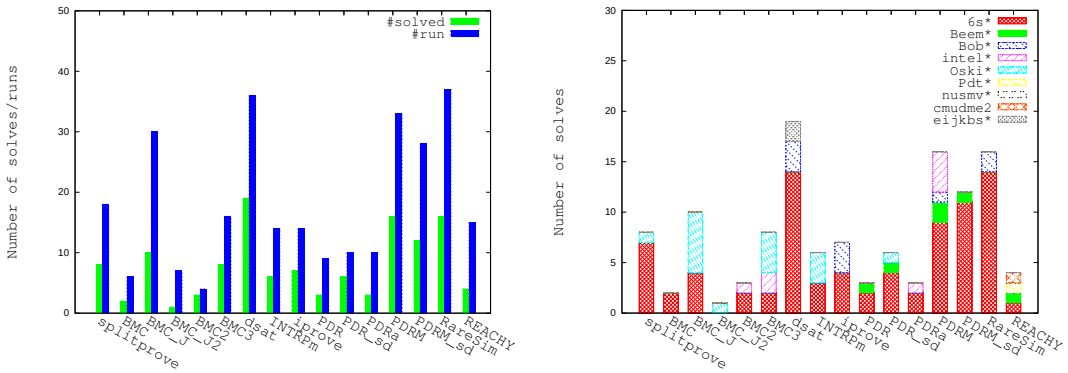
Looking at *SINGLE* track results, *SUPROVE* solved about 56% of *6s*, 71% of Oski, 25% of pdt, 29% of Intel, 70% of bob, and 40% of beem benchmarks. Furthermore it solved cmudme2, the two eijkbs, but no nusmv benchmarks. Both *SUPROVE* and *SIMPSAT* were the only tools able to solve the following benchmarks: oski3ub0i, 6s269r, bobsmvhd3, 6s349rb06, 6s7, oski4ui, 6s320rb1, and bobsmhdlc1. Both *SUPROVE* and *SIMPSAT* were among the few tools able to solve the following instances: bobsmmem, obsmmips, bobmitersynbm, bobsmfpu. Both *SIMPSAT* and *SUPROVE* resulted in a non-negligible number of memory overflows: 9 for *SIMPSAT* and 7 for *SUPROVE*. In particular, three instances (6s274r, 6s358r and 6s359) caused a memory overflow only for *SIMPSAT*, *SUPROVE* and *SUPDEEP*. This suggests that some of the engines or simplification techniques applied by these tools do not scale well on those instances, or that the degree of concurrency (and subsequent number of activated processes) was too high.

Figure 5(a) shows the numbers of solved benchmarks by individual *ABC* engines and the number of times a specific engine was activated. Data was obtained by additional experimentation done (offline) with *SUPROVE*. For each engine, the vertical bar on the left-hand side represents the number of benchmarks solved, whereas the vertical bar on the right-hand side indicates the number of times a specific engine was activated. Figure 5(b) represents, for each engine, the number of solved benchmarks classified by family. Overall data show that 34 problems were solved (after reductions) by combinational verification engines (dsat, iprove, splitprove), 24 problems by BMC (in different configurations), 16 by rarity simulation, 40 by PDR, 6 by interpolation, and 4 by BDD-based methods.

Figure 6 shows the simplification power of initial *ABC* optimization techniques in terms of number of latches and AIG gates, with axes labeled by thousands of latches and AND gates. Overall, while some simplification and abstraction can be hidden inside some engine, these plots show a significant impact of transformational techniques.

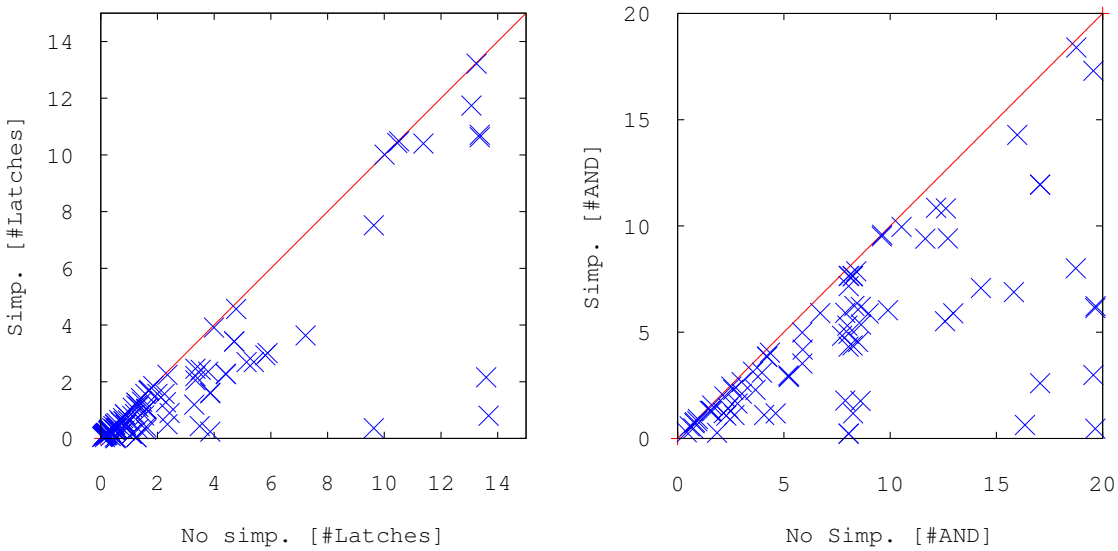
7.2 Single Safety Track

Figure 7 illustrates the distribution of time and memory usage for the *SINGLE* track. The former takes into account only solved instances, thus omitting time-outs and generically failed runs to improve readability. The latter includes the entire benchmark set.



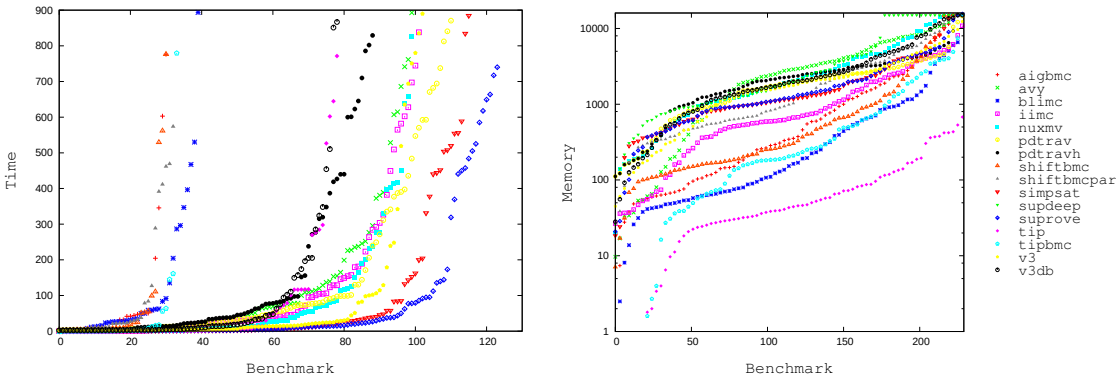
(a) (b)

Figure 5. ABC Engine Statistics: Number of benchmarks solved and number of runs per engine (a), and number of benchmarks solved per family (b).



(a) (b)

Figure 6. Design size reduction in terms of thousands of latches (a) and AIG gates (b).



(a) (b)

Figure 7. CPU times for all solved instances (a) and memory usage for the entire competition (b).

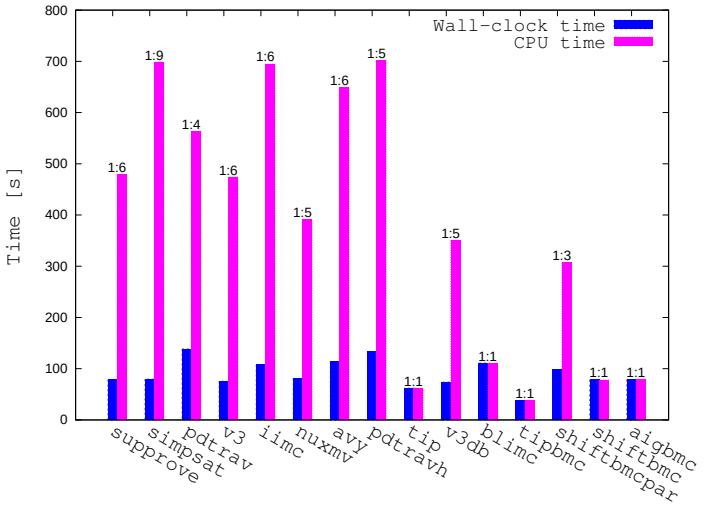
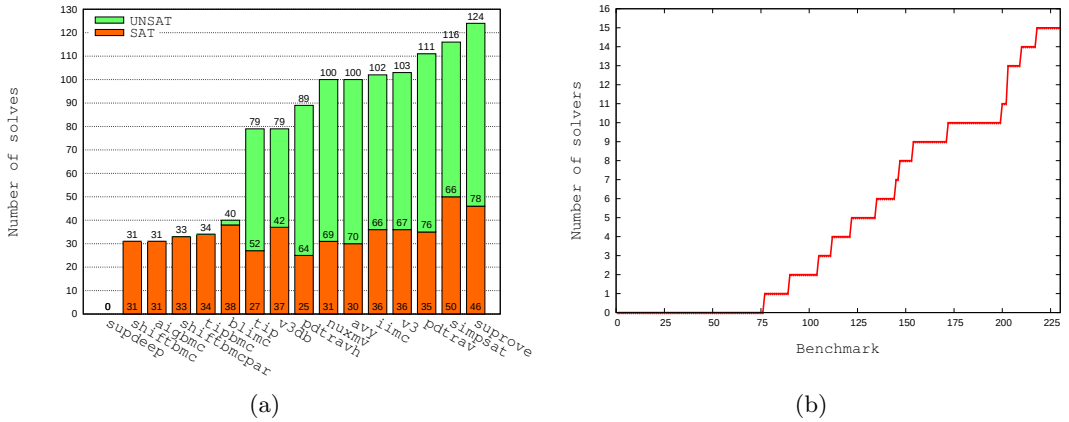


Figure 8. Total wall-clock time and total CPU time for all solved instances in the *SINGLE* track.

Portfolio tools, using multiple threads and processes, are by far the most widespread approach. Portfolio strategies range from simple schemes, basically launching a predefined and independent set of concurrent processes, to more complex strategies, where simplifying transformations are intertwined with model checking engines, under dynamic activation, control and tuning strategies. The degree of concurrency can be measured by the ratio of wall-clock time versus CPU time, ranging from 3 to about 8, for most tools, with the exception of *TIP* (and its variants), *BLIMC/AIGBMC*, *ShiftBMC*, and *LS4* for which wall-clock and CPU times do coincide. Figure 8 provides a more accurate view on this measure, and they clearly show the dominance of tools exploiting concurrency. This is obviously motivated by the rules of the competition, that encourage multi-core exploitation by establishing time limits in terms of wall-clock time.



(a) (b)

Figure 9. Solved instances represented as bars (a) and number of successful model checkers for each benchmark (b).

Obviously, memory limits, and thread (process) slow-down due to shared memory access, are key factors to be considered while developing a multi-engine tool. This is why none of the tools exploited the full availability of 12 cores.

Figure 9 illustrates the distribution of solved instances by model checker and the number of solves by benchmark. Figure 9 (a) shows that the top three model checkers, by number of solved instance, are *SUPROVE*, *SIMPSAT* and *PdTRAV*. In the final ranking of the *SINGLE* track, *SUPROVE* and *SIMPSAT*, tracks being part of the *ABC* suite, occupy conjointly the first position. Since *PdTRAV* presented a discrepancy on a newly introduced Oski benchmark, it was disqualified and therefore it does not appear in the final ranking. In its stead *V3* ranked second, followed by *IIMC*. A subsequent inquiry found that the problem affecting *PdTRAV* was due to a bug in the reduction routine invoked while applying hidden constraints. It can be noticed that *SUPDEEP* was not able to solve any instance, nevertheless we kept its entry in all the results provided in this paper for the sake of completeness.

It can be can clearly seen that some model checkers are just covering *SAT* instances, as they only use BMC engines, whereas other model checkers solve both *SAT* and *UNSAT* problems. Bounded model checkers cover a range of about 30 to 40 *SAT* problems, with *ABC* (*SUPROVE* and *SIMPSAT*) getting extra coverage by rarity simulation. Concerning *UNSAT* problems, a large majority of tools cover from 60 to 70 problems, with *ABC* and *PdTRAV* getting 78 and 76, respectively. *AVY*, being the only tool not to include a BMC engine, is clearly aimed at solving *UNSAT* problems. Nevertheless, it was still able to solve 30 *SAT* instances. This suggests that several *SAT* instances in the benchmarks set are not beyond the reach of UMC techniques. The plot in Figure 9 (b) evaluates the degree of hardness of problems, showing that 79 instances remain unsolved, whereas the degree of coverage of other problems follows an almost linear pattern, from hard to easy problems. In any case, a large number of designs were solved only by few model checkers.

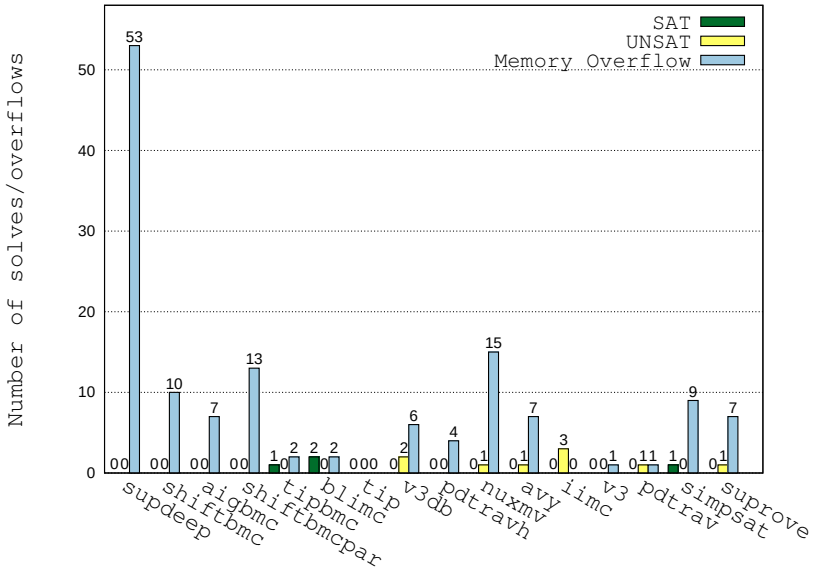


Figure 10. Number of unique solves (*SAT*, first bar, and *UNSAT*, second bar) and memory overflows (third bar) per contestant.

From a more in depth analysis of the full set of results, it is possible to identify the best performing tool for each benchmark family, considering the number of solved instances, and the average solving time as a tie-breaker. *SUPROVE* is the best tool with respect to the 6s, bob and Oski families while *IIMC* performs particularly well on the beem and pdt families. *nuXmv* and *PdTRAV* both solve the highest number of Intel benchmarks, but the former requires less time on average. In the case of the beem family, we can ascribe the good performance of *IIMC* to a particular pre-processing technique as described in 5.1.7. In the other cases we have no knowledge of ad hoc techniques targeting specific families.

In order to further characterize the competitors, Figure 10 shows the number of *unique solves*, i.e., instances on which only one model checker is successful, divided between *SAT* and *UNSAT* instances, and the number of memory overflows for each contestant. We analyzed the number on unique solves hoping to identify model checkers with orthogonal behavior, i.e., model checkers able to solve subsets of benchmarks otherwise unsolved. The results show that the degree of orthogonality among the tools is rather low, as the number of unique solves is small with respect to the whole benchmarks set. Most of the model checkers tend to cover about the same subsets of instances, with few exceptions. This is probably due to the high popularity of portfolio approaches. Unsolved instances may either be too complex to be tackled with state-of-the-art techniques and currently available hardware resources, or may require completely new approaches to be solved.

We also analyzed the tool results from the perspective of memory consumption, in order to assess scalability limits of each tool and the model checking techniques they employ. With the exception of *SUPDEEP*, most of the tools encounter on average ten memory overflows, with *IIMC* and *TIP* scoring as low as zero. This suggests that not a single instance is

completely intractable, memory-wise, but there are a few that require careful management of the available resources.

A special consideration in this regard can be made taking into account *nuXmv*. Such a model checker, on the high end spectrum in terms of overall results, surprisingly encountered the most memory overflows, disregarding *SUPDEEP*, over a set of benchmarks that did not cause problems to any other competitor. Upon further inquiry, its authors discovered an underlying bug with the way resolution proofs were stored and proof-logging was managed.

Figure 11 illustrates the correlation among tools using Pearson's index [45]. In statistics, the Pearson product-moment correlation coefficient is a measure of the linear correlation (dependence) between two variables X and Y. It gives a value between $+1$ and -1 inclusive, where 1 is total positive correlation, 0 is no correlation, and -1 is total negative correlation. It is widely used in the sciences as a measure of the degree of linear dependence between two variables. Pearson's correlation coefficient when applied to a sample is commonly computed as follows:

$$r = \frac{\sum_{i=1}^{n}(x_i - \overline{x}) \cdot (y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2} \cdot \sqrt{\sum_{i=1}^{n}(y_i - \overline{y})^2}}$$

In our framework, given a set of benchmarks B with cardinality n, the i index is used to enumerate benchmarks in B. Variables x and y represent model checkers, x_i and y_i take discrete value 0 (or 1), to denote failure (or success) of tools x and y with the i-th benchmark instance. $\overline{x}$ represents the ratio of solved instances for x ($\overline{x} = |solved_x|/n$), dually for $\overline{y}$.

In Figure 11, the correlation is computed taking into account all solved verification instances (a), only *SAT* designs (b), and only *UNSAT* benchmarks (c).

Theoretically, depending on the (negative or positive) index value, the correlation may be considered very strong when in the range $[0.70, 1.00]$, strong $[0.40, 0.69]$, moderate $[0.30, 0.39]$, weak $[0.20, 0.29]$, or negligible $[0.00, 0.19]$. Practically, in our cases, small coefficient variations may be considered as meaningless, and we use a coloring mapping strategy, to indicate weak/moderate/strong correlations. Darker gray is used to indicate higher affinity. Diagonals trivially represent self-affinity equal to 1.00. High values usually indicate that the tools share a large number of commonly solved benchmarks and that only a few designs were solved by just one of the tools. For example, the high correlation (0.80) between *TIPBMC* and *ShiftBMC* is motivated by the fact that 4 designs are solved only by *TIPBMC*, 1 only by *ShiftBMC*, and 30 by both tools. Conversely, small values often indicate a small number of commonly solved designs. For example, the low correlation (0.11) between *TIPBMC* and *AVY* is obtained because 7 designs are solved only by *TIPBMC*, 73 only by *AVY*, and 27 by both tools.

Figure 11(a) generally shows low affinity values, with the exception of pure BMC tools, that are mutually more related than other tools. *SUPDEEP* is completely unrelated with all other tools as it does not solve any design. Figure 11(b) (analysis of *SAT* problems) shows high affinity levels among all tools, which can be explained by the observation that *SAT* problems are mostly solved by BMC engines. *SUPROVE* and *SIMPSAT* show lower correlation with other tools, as they get a 10+ extra coverage by rarity simulation. Figure 11(c) (analysis of *UNSAT* problems) shows lower affinities on average, due to the fact that proof oriented portfolios are typically more diversified, and sensitive to model transformations/reductions/abstractions. Though in both Figure 11(b) and 11(c) tools from the

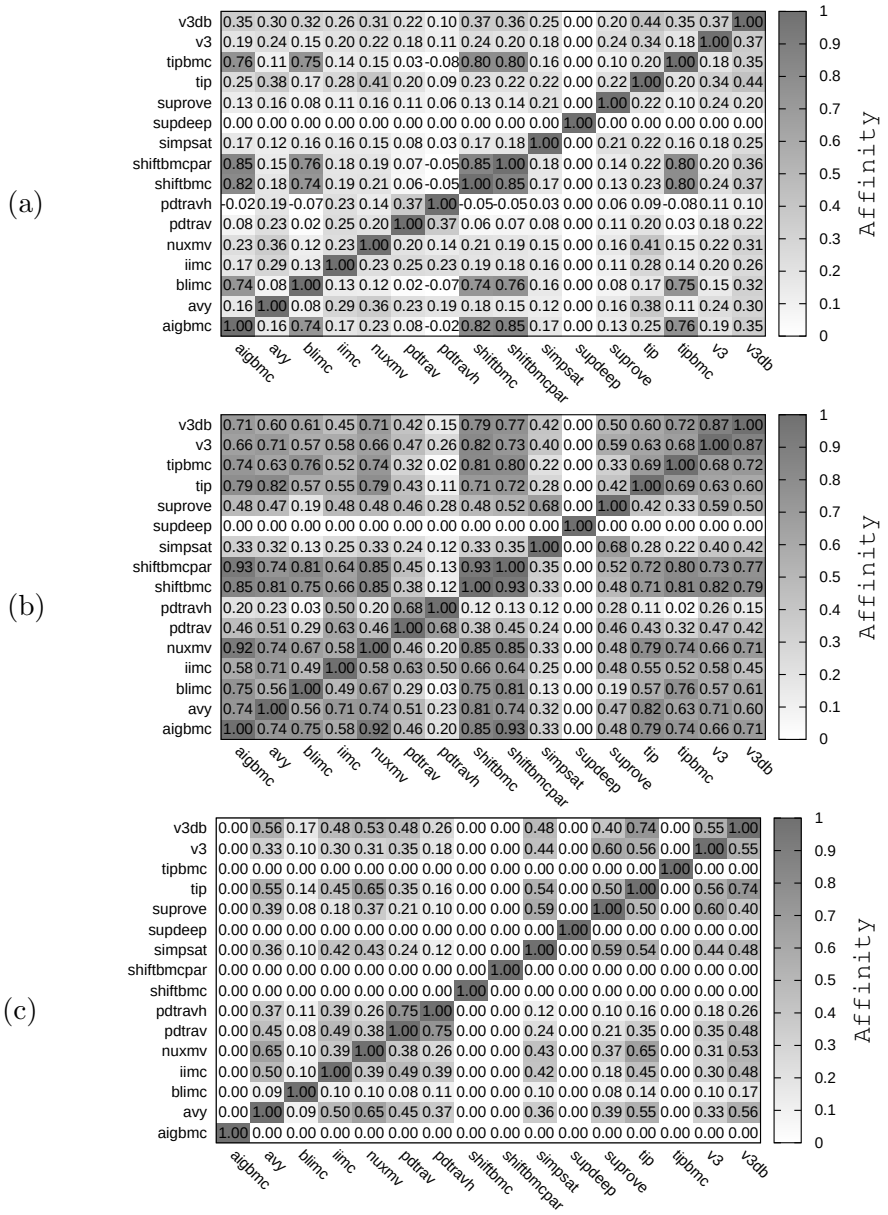
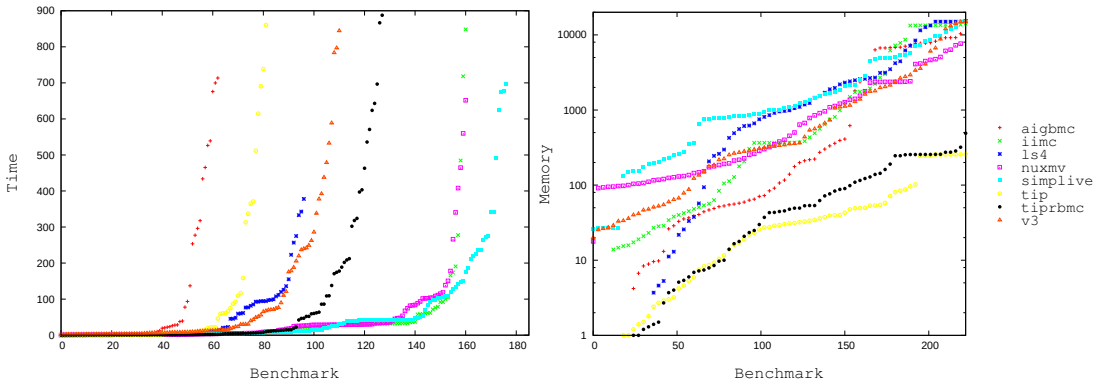


Figure 11. Correlation among tools: Pearson's Index over solved instances as heat-map for all *SAT* and *UNSAT* design (a), only *SAT* (b) and only *UNSAT* (c).

same group are characterized by a certain affinity, Figure 11(c) basically witnesses a certain diversification in the set of competing tools. A similar consideration could be derived by observing that the winner tool solved 124 problems, out of 151 globally solved. In Figure 11(c), most affinities are below the 0.60 threshold, with very few exceptions (excluding same group tools), such as *TIP-V3db* (0.74), *AVY-nuXmv* (0.65), *nuXmv-TIP* (0.65), and *SUPROVE-V3* (0.60).



(a) (b)

Figure 12. CPU times for all solved instances, and memory usage for all benchmarks.

As a final remark, notice that those results may be considered somehow disappointing. In fact, very high negative correlation values would have indicated complementary strength of the tools, i.e., complementary set of benchmark solved. That would have implied that a very simple portfolio scheme, somehow running both tools in a proper way, would have drastically increased the number of solved benchmarks, giving some new ground for improvements. Conversely, we have to argue that most of the tools, being per se already multi-engine, actually exploit most of the current (publicly available) technology.

7.3 Liveness Track

This experimental section closely follows Section 7.2 by presenting statistics on the *LIVE* track.

Figure 12 plots time and memory usage for all tools competing in the *LIVE* track.

Figure 13 provides a more accurate view on this measure, and it clearly shows the dominance of tools exploiting concurrency. This is obviously motivated by the rules of the competition, that encourage multi-core exploitation by establishing time limits on wall-clock time.

Figure 14 shows that the top three model checkers, by number of solved instance, are *SIMPLIVE*, *nuXmv* and *IIMC*, representing the final ranking for the *LIVE* track. As for the *SINGLE* track, it can be seen that some model checkers are just covering *SAT* instances, as they only use BMC engines, whereas other model checkers solve both *SAT* and *UNSAT* problems. The range of solved *SAT* instances is a bit more varied, with respect to the *SINGLE* track, ranging from 15 to 97, although most of the competitors score closer to 60 in the worst cases. Concerning *UNSAT* problems, a large majority of tools cover from 60 to 70 problems, with only *SIMPLIVE* reaching 80. Disregarding BMC-only model checkers, the results of *UNSAT* problems are rather uniform as well as the *SAT* ones. With the exception of *LS4*, whose author already acknowledged its shortcoming in the specific context of the competition, and *V3*, all the others contestants tend to cover the same subset on instances.

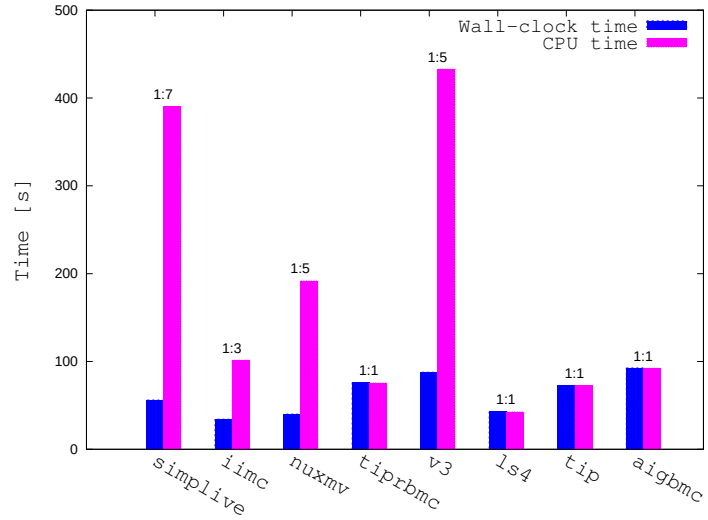


Figure 13. Total wall-clock time and total CPU time for all solved instances in the *LIVE* track.

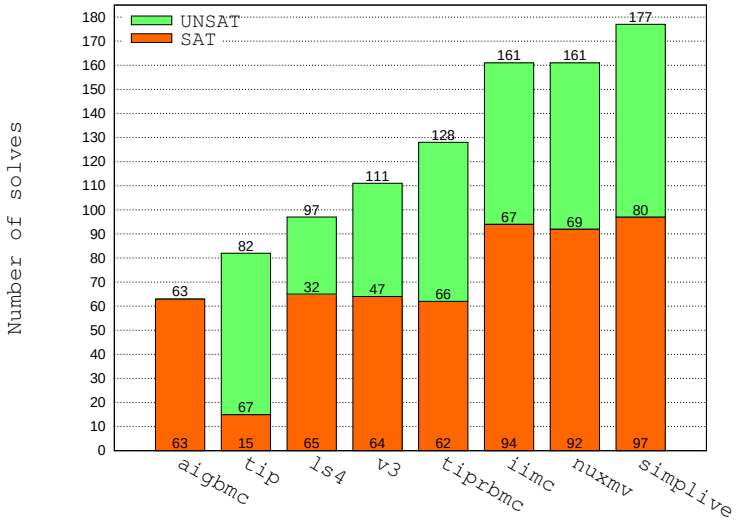


Figure 14. Solved instances per competitor.

Figure 15 shows the number of *unique solves*, divided between *SAT* and *UNSAT* instances, and the number of memory overflows for each contestant. As it can be seen from such a figure, data distribution follows a pattern similar to the one encountered whilst analyzing the *SINGLE* track, with a limited amount of unique *SAT* or *UNSAT* solves, thus it is possible to draw the same conclusions as before.

Figure 16 (a), (b), and (c) show tool correlation. As for the *SINGLE* track, *SAT* and *UNSAT* statistics (Figure 16 (a) and (b)) give better insights, confirming that SAT oriented techniques show higher correlation (e.g., 0.89 for *nuXmv-IIMC*, 0.82 for *TIPBMC-V3*). Much lower correlation ratios are found with *UNSAT* problems, where richer and more

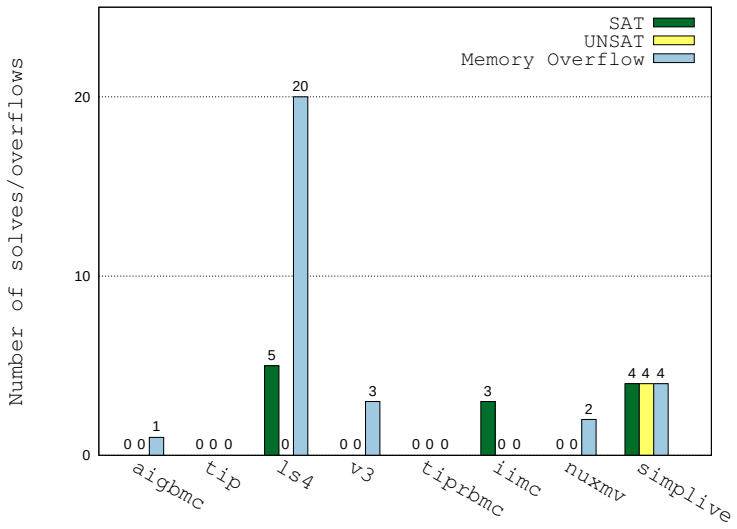


Figure 15. Number of unique solves (*SAT*, first bar, and *UNSAT*, second bar) and memory overflows (third bar) per contestant.

diversified portfolios cover non fully overlapping sets of benchmarks. It is also interesting to notice that Figure 16(b) shows some negative affinity scores.

7.4 Deep Bound Track

The *DEEP* track was introduced in HWMCC 2012, and has been present ever since, as deep bound capabilities are deemed rather important in several industrial settings. In a context in which unbounded model checking cannot provide a conclusive answer, being able to assess the progress achieved may be crucial. In order to do so, the depth at which a model checker could prove unsatisfiability is often the most relevant metric.

For the purposes of the track, the targeted benchmark set consists of all the unsolved instances for the *SINGLE* track, thus for the HWMCC 2014 edition resulting in 79 circuits.

Figure 17 illustrates the bound reached for each verification instance by each tool. The horizontal line set at bound 100 marks the cap point used for scoring purposes. The figure shows a certain uniformity of behavior for most of the tools, based on a BMC engine, with the exception of *IIMC* and *TIP*.

8. Lessons Learned

Some of the lessons learned have already been discussed in previous sections. We collect here some observations on the competition, pointing out pros and cons, and we propose some ideas for future editions. We first discuss some of the lessons learned from an organizational standpoint, then for the general user and/or competition participant.

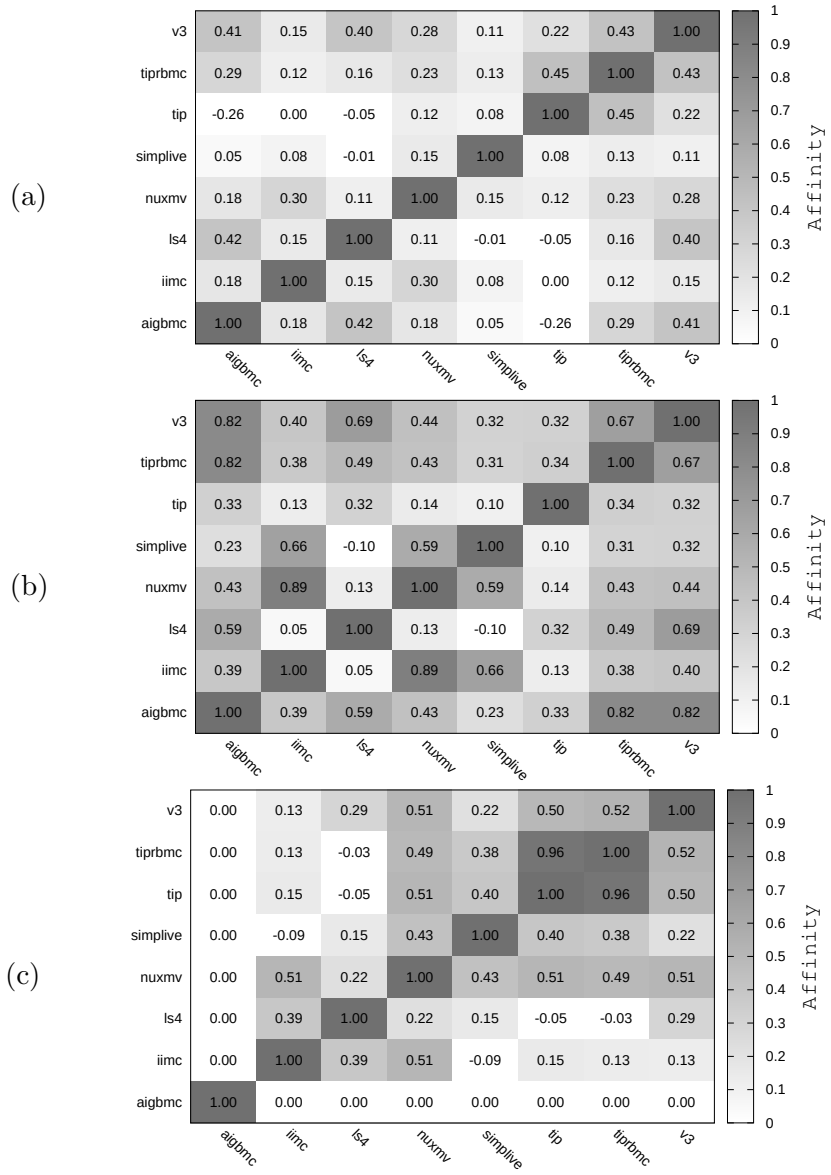


Figure 16. Correlation among tools: Pearson's Index over solved instances as heat-map for all *SAT* and *UNSAT* benchmarks (a), only *SAT* (b) and only *UNSAT* (c).

8.1 Lessons for Organizers

A major problem in organizing the competition is benchmark selection. This is an issue common to other competitions as well, as a certain renewal of an initial benchmark set is needed in order to avoid tool biasing and over-fitting. While in the beginning HWMCC editions always experienced a growth of the benchmark set, with various contributions from both participant groups and industrial partners, selection can now be done on a wide set

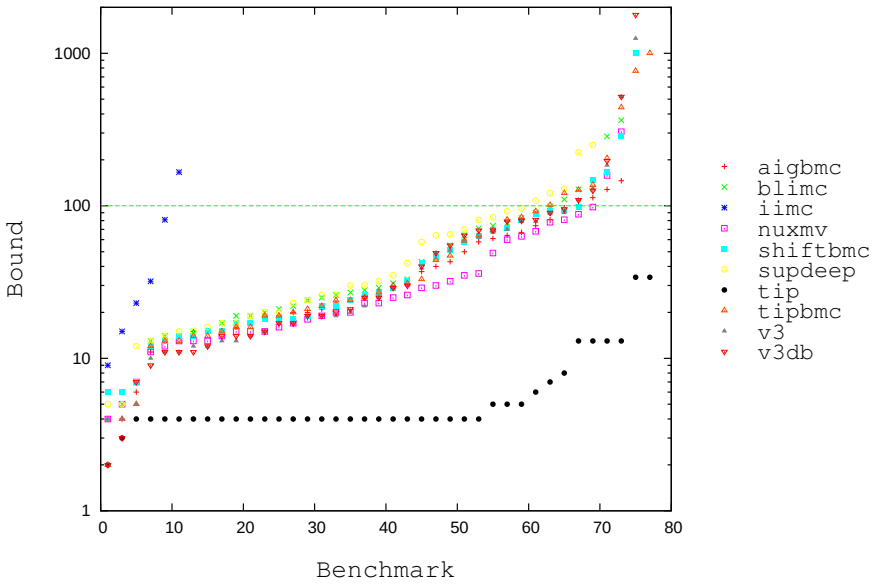


Figure 17. Reached bounds distribution by instance.

of available problems: Elimination of easy instances and randomization of choices, when possible, have been the driving criteria.

Overall, the set of benchmarks used for the 2014 edition includes many challenging problems, as witnessed by the large set of unsolved problems. Some unbalancing can be observed in the ratio of *UNSAT* versus *SAT* instances, with a certain dominance of the *UNSAT* ones. The set of industrial problems is undoubtedly relevant, mainly due to the recent contributions of IBM and Osky, added to the original Intel set. Nonetheless, the industrial set probably suffers from a certain polarization, and future contributions from other companies could greatly increase the value of the benchmark collection. As a last consideration, most of the industrial contributions are in the *SINGLE* track, whereas the number of industrial *LIVE* problems is still low.

The benchmark format, AIGER, was developed with a main goal of providing a common denominator of all bit-level formats, with clean and simple semantics, which still allows to encode relevant industrial model checking problems. We are aware of several aspects where AIGER fails short to reach this goal. First, word-level information available in some applications might help to speed-up model checking, but AIGER is a plain bit-level format, and thus such information is lost during the encoding process. Second, since functional models have much cleaner semantics than relational ones, particularly in a synthesis context, AIGER needs an additional encoding step to handle relational problems, such as latch based designs used in some companies. Third, application level properties might be mapped to multiple AIGER properties during the encoding process. Grouping or prioritizing AIGER properties to partially encode such relations is not supported. Beside these hard to overcome shortcomings there are several additional syntactic issues, including binary headers, arbitrary annotations and a user extensible section format, which will be addressed in future versions.

Concerning competition rules, the preliminary phase, oriented to tool congruency checks, could be significantly improved. Though not present in other competitions, it is an important practice in order to tackle problems related to formats (of AIGER files, counterexamples and log files) as well as tool configuration/installation on the target platforms. Discrepancy checking is another organizational issue, that is basically handled by detecting conflicting verdicts (by different tools) and by simulating counterexamples/witnesses. This potentially misses unsound/wrong *UNSAT* verdicts, as no proof of unsatisfiability (e.g., an invariant) is currently required to model checkers.

Time limits are another relevant issue, with impact on rankings and value of the overall competition itself. As already pointed out, the 900 second choice is a compromise, mainly motivated by computing resource limits. Though it is clear that higher limits would go in the direction of industry scale problems, its still difficult to envisage big changes in the near future.

Ranking criteria are a final topic deserving consideration. Although counting completed instances is a widely accepted measure for single (safety as well as liveness) tracks, ranking criteria for *MULTI* and *DEEP* tracks are more critical, and potentially subject to further discussion on how to weight the success (or partial success) on each problem in the given set.

8.2 Lessons for Developers and Users

Developers and tool users are mostly interested to competition results for their ability to show strengths and weaknesses of different tools/techniques, as well as the relevance of (subsets of) the used benchmarks. Though part of this paper has gone in this direction, it is extremely difficult (if not scientifically impossible) to drive conclusions from only the competition results.

Competition results (including log files) just provide little more than time and memory statistics. More detailed data would be needed for a thorough scientific analysis/evaluation. For end users, it would be convenient if all tools, alongside their specific settings, were made available after each year competition, for the sake of reproducibility. As a matter of fact, industries have never participated to the competition. Though there are obvious reasons for that, it is well known that many industrial groups test their tools on the competition benchmarks.

Industrial and academic toolmakers often exploit competition data as a starting point for more detailed comparisons, whenever competing tools (or their variants) are freely available.

Given the above limitations, the following concluding remarks could be drawn:

- Portfolio tools seem to be the current standard, and they seem to have an edge over single-engine tools.
- IC3 seems to be the most performing engine on all portfolio based model checkers.
- Minisat seems to be the preferred SAT solver by model checker developers. Authors probably chose to focus on other issues rather than integrating the newest, and possibly more powerful, SAT solvers in their tools. It could be interesting to assess the impact of such a choice on the competition results.

- BMC tools are the most natural competitors for the *DEEP* tack as they generally reach deeper bounds faster than other unbounded model checking engines. The top four model checkers in the *DEEP* track (including hors concours tools *BLIMC*/*AIGBMC*) are in fact BMC tools.
- All tools are now evaluated in terms of solved instances in the slotted time. Only wall-clock times, CPU times, memory usage and the final decision of tools over instances are known. Competitors should be encouraged to output more detailed information in order to allow the organizers, other competitors and other researchers in general, to better analyze the results and potentially learn more from competition data.

9. Conclusions

In this paper, after providing a historical overview of the Hardware Model Checking Competition, we have primarily focused on the 2014 edition. We have described the competing tools and the set of benchmarks. An experimental evaluation of the competition results is proposed, with the aim of characterizing the tools, their strengths and weaknesses as well as measuring their mutual affinities. We have also provided some data on benchmark families, and on the breakdown of individual engine contribution in a portfolio.

Tool descriptions have been directly obtained by their authors. Most of the experimental data have been derived from the HWMCC 2014 results. Data on *ABC* have been partially generated by new runs.

Though we are aware that more could be done in view of a more accurate observation and characterization of tool versus tool affinity and tool/engine versus problem coverage, this work sheds some light on HWMCC ranking and results. We deem the analysis we provide can be very interesting both for an industrial and an academic reader.

Acknowledgments

The authors would like to thank all HWMCC 2014 competitors for their support in describing the characteristics of their tools, as well as past contributors and organizers of the competition. Moreover, the authors express their appreciation to Vigyan Singhal, from Oski Technologies, for his analysis of the Oski benchmarks. Finally, a special acknowledgment goes to Jason Baumgartner, from IBM, for his contributions to the paper and his accurate analysis of 6s benchmarks.

References

[1] The SMT Competition Web Page. http://www.smtcomp.org/. Accessed: 2014-12-01.

[2] A. Biere. The AIGER And-Inverter Graph (AIG) Format. http://fmv.jku.at/aiger/. Accessed: 2007-05-01.

[3] P. Aggarwal, D. Chu, V. Kadamby, and V. Singhald. End-to-end formal using abstractions to maximize coverage: Invited tutorial. In P. Bjesse and A. Slobodova, editors, *Proc. of the Int. Conf. on Formal Methods in Computer-Aided Design*, pages 9–16, Austin, Texas, 2011.

[4] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček. DiVinE A Tool for Distributed Verification. In *Computer Aided Verification*, **4144** of *Lecture Notes in Computer Science*, pages 278–281. Springer Berlin Heidelberg, 2006.

[5] J. Baumgartner, H. Mony, A. Mishchenko, and R. K. Brayton. Speculative reduction-based scalable redundancy identification. In *Proc. Design Automation & Test in Europe Conf.*, pages 1674–1679. IEEE Computer Society, April 2009.

[6] D. Le Berre, O. Roussesl, and L. Simon. The International SAT Competitions Web Page. http://www.satcompetition.org/. Accessed: 2007-06-01.

[7] A. Biere. Picosat essentials. *JSAT*, **4**(2-4):75–97, 2008.

[8] A. Biere. Yet another local search solver and Lingeling and friends entering the SAT Competition 2014. In A. Belov, M. J. H. Heule, and M. Järvisalo, editors, *SAT Competition 2014*, **B-2014-2** of *Department of Computer Science Series of Publications B*, pages 39–40. University of Helsinki, 2014.

[9] A. Biere, C. Artho, and V. Schuppan. Liveness Checking as Safety Checking. In *FMICS*, 2002.

[10] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, London, UK, 1999. Springer.

[11] A. Biere, K. Heljanko, T. Junttila, T. Latvala, and V. Schuppan. Linear encodings of bounded LTL model checking. In *Logical Methods in Computer Science, 2(5)*, pages 1–64, 2006.

[12] A. R. Bradley. Sat-based model checking without unrolling. In J. Ranjit and D. A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI*, **6538** of *Lecture Notes in Computer Science*, pages 70–87. Springer, January 2011.

[13] A. R. Bradley, F. Somenzi, Z. Hassan, and Y. Zhang. An Incremental Approach to Model Checking Progress Properties. In *Proc. of the Int. Conf. on Formal Methods in Computer-Aided Design*, November 2011.

[14] R. Brayton and A. Mishchenko. ABC: An academic industrial-strength verification tool. In *Proc. Computer Aided Verification*, **6174** of *Lecture Notes in Computer Science*, pages 24–40. Springer-Verlag, 2010.

[15] C. Wu and C. Wu and C.Lai and C. Huang. A counterexample-guided interpolant generation algorithm for SAT-based model checking. In *Proc. Design Automation Conference*, pages 1–6, Austin, Texas, USA, June 2013. IEEE Computer Society.

[16] G. Cabodi, P. Camurati, and M. Murciano. Automated Abstraction by Incremental Refinement in Interpolant-based Model Checking. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 129–136, San Jose, California, November 2008. ACM Press.

[17] G. Cabodi, S. Nocco, and S. Quer. The PdTRAV tool. http://fmgroup.polito.it/index.php/download/viewcategory/3-pdtrav-package. Accessed: 2014-12-01.

[18] G. Cabodi, S. Nocco, and S. Quer. Thread-based multi-engine model-checking for multicore platforms. *ACM Transactions on Design Automation of Electronic Systems*, **18**(3):36:1–36:28, 2013.

[19] G. Cabodi, M. Palena, and P. Pasini. Interpolation with Guided Refinement: revisiting incremetality in SAT-based Unbounded Model Checking. In K. Claessen and V. Kuncak, editors, *Proc. of the Int. Conf. on Formal Methods in Computer-Aided Design*, pages 43–48, Lausanne, Switzerland, November 2014.

[20] M. L. Case, H. Mony, J. Baumgartner, and R. Kanzelman. Transformation-Based Verification Using Generalized Retiming. In *FMCAD*, November 2009.

[21] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The NuXmv Tool. https://nuXmv.fbk.eu/. Accessed: 2014-06-01.

[22] 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 A. Biere and R. Bloem, editors, *Proc. Computer Aided Verification*, **8559** of *Lecture Notes in Computer Science*, pages 334–342. Springer International Publishing, 2014.

[23] K. Claessen and N. Sorensson. A liveness checking algorithm that counts. In G. Cabodi and S. Singh, editors, *Proc. of the Int. Conf. on Formal Methods in Computer-Aided Design*, pages 52–59, Oct 2012.

[24] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In F. Bacchus and T. Walsh, editors, *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, **3569** of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.

[25] N. Een, A. Mishchenko, and R. K. Brayton. Efficient Implementation of Property Directed Reachability. In P. Bjesse and A. Slobodova, editors, *Proc. of the Int. Conf. on Formal Methods in Computer-Aided Design*, FMCAD '11, pages 125–134, Austin, Texas, 2011.

[26] Z. Hassan, F. Somenzi, M. Dooley, and A. Bradley. The IIMC tool. http://iimc.colorado.edu/. Accessed: 2014-12-01.

[27] K. Heljanko, T. A. Junttila, and T. Latvala. Incremental and complete bounded model checking for full PLTL. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, **3576** of *Lecture Notes in Computer Science*, pages 98–111. Springer, 2005.

[28] M. J. H. Heule, W. A. Hunt, and N. Wetzler. Verifying refutations with extended resolution. In M. P. Bonacina, editor, *Automated Deduction CADE-24*, **7898** of *Lecture Notes in Computer Science*, pages 345–359. Springer Berlin Heidelberg, 2013.

[29] S. Kupferschmid, M. D. T. Lewis, T. Schubert, and B. Becker. Incremental preprocessing methods for use in BMC. *Formal Methods in System Design*, **39**(2):185–204, 2011.

[30] C. Lai, C. Wu, and C. Huang. Adaptive Interpolation-Based Model Checking. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE Computer Society, 2014.

[31] T. Latvala, A. Biere, K. Heljanko, and T. A. Junttila. Simple bounded LTL model checking. In A. J. Hu and A. K. Martin, editors, *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings*, **3312** of *Lecture Notes in Computer Science*, pages 186–200. Springer, 2004.

[32] D. Lee. Opensparc - a scalable chip multi-threading design. In *VLSI Design, 2008. VLSID 2008. 21st International Conference on*, pages 16–16. IEEE, 2008.

[33] M. J. H. Heule and N. Manthey and T. Philipp. Validating Unsatisfiability Results of Clause Sharing Parallel SAT Solvers. In D. Le Berre, editor, *POS-14*, **27** of *EPiC Series*, pages 12–25. EasyChair, 2014.

[34] N. Manthey. The Shift BMC tool. http://tools.computationallogic.org/content/riss427.php. Accessed: 2014-12-01.

[35] N. Manthey. Coprocessor 2.0 – A flexible CNF simplifier. In A. Cimatti and R. Sebastiani, editors, *Theory and Applications of Satisfiability Testing SAT 2012*, **7317** of *Lecture Notes in Computer Science*, pages 436–441. Springer Berlin Heidelberg, 2012.

[36] N. Manthey. Riss 4.27. **B-2014-2** of *Department of Computer Science Series of Publications B*, pages 65–67. University of Helsinki, Helsinki, Finland, 2014.

[37] K. L. McMillan. Interpolation and SAT-based Model Checking. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Proc. Computer Aided Verification*, **2725** of *Lecture Notes in Computer Science*, pages 1–13, Boulder, CO, USA, 2003. Springer.

[38] A. Mishchenko. ABC: A System for Sequential Synthesis and Verification. http://www.eecs.berkeley.edu/~alanmi/abc/. Accessed: 2014-12-01.

[39] A. Mishchenko, N. Eén, R. Brayton, J. Baumgartner, H. Mony, and P. Nalla. GLA: Gate-Level Abstraction Revisited. In *DATE*, 2013.

[40] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann. Scalable Automated Verification via Expert-System Guided Transformations. In *Proc. of the Int. Conf. on Formal Methods in Computer-Aided Design*, November 2004.

[41] N. Eén and N. Sörensson. Temporal Induction by Incremental SAT Solving. *Electronic Notes in Theoretical Computer Science*, **89**(4):543–560, 2003.

[42] A. Nadel and V. Ryvchin. Efficient SAT solving under assumptions. In *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, pages 242–255, 2012.

[43] A. Nadel, V. Ryvchin, and O. Strichman. Preprocessing in incremental SAT. In *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, pages 256–269, 2012.

[44] A. Nadel, V. Ryvchin, and O. Strichman. Ultimately incremental SAT. In C. Sinz and U. Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, **8561** of *Lecture Notes in Computer Science*, pages 206–218. Springer, 2014.

[45] K. Pearson. Note on Regression and Inheritance in the Case of Two Parents. *Proceedings of the Royal Society of London*, **58**:240–242, 1895.

[46] R. Pelánek. BEEM: Benchmarks for Explicit Model Checkers. In *Model Checking Software*, **4595**, pages 263–267, 2007.

[47] S. Quer. Model Checking Evaluation of Airplane Landing Trajectories. *International Journal on Software Tools for Technology Transfer (STTT)*, **7**:1–2, April 2013.

[48] R. Brayton. Robert Brayton Home Page. http://www.eecs.berkeley.edu/~brayton/. Accessed: 2014-12-01.

[49] M. Sheeran, S. Singh, and G. Stålmarck. Checking Safety Properties Using Induction and a SAT Solver. In W. A. Hunt and S. D. Johnson, editors, *Proc. of the Int. Conf. on Formal Methods in Computer-Aided Design*, **1954** of *Lecture Notes in Computer Science*, pages 108–125, Austin, Texas, USA, November 2000. Springer.

[50] V. Singhal and P. Aggarwal. Using coverage to deploy formal verification in a simulation world. In *Proc. Computer Aided Verification*, pages 44–49. Springer, 2011.

[51] M. Suda. LS4: A PLTL-prover based on labelled superposition with partial model guidance. http://www.mpi-inf.mpg.de/~suda/ls4.html, 2012. Accessed: 2014-12-01.

[52] M. Suda. *Resolution-based methods for linear temporal reasoning*. PhD thesis, Universität des Saarlandes, December 2014.

[53] M. Suda and C. Weidenbach. A PLTL-Prover Based on Labelled Superposition with Partial Model Guidance. In *IJCAR*, 2012.

[54] M. Suda and C. Weidenbach. Labelled Superposition for PLTL. In *LPAR-18*, 2012.

[55] G. Sutcliffe. The CASC Web Page. http://www.cs.miami.edu/~tptp/CASC/. Accessed: 2014-12-01.

[56] T. Dina and C. Supratik and P. Paritosh. Efficient guided symbolic reachability using reachability expressions. In H. Hermanns and J. Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, **3920**, pages 120–134. Springer Berlin Heidelberg, 2006.

[57] The TIP (Temporal Induction Prover) solver. https://github.com/niklasso/tip. Accessed: 2014-12-01.

[58] C. A. J. van Eijk. Sequential Equivalence Checking Based on Structural Similarities. *IEEE Trans. on Computer-Aided Design*, **19**:814–819, July 2000.

[59] Y. Vizel and O. Grumberg. Interpolation-sequence based model checking. In A. Biere and C. Pixley, editors, *Proc. of the Int. Conf. on Formal Methods in Computer-Aided Design*, pages 1–8, Austin, Texas, 2009.

[60] Y. Vizel, O. Grumberg, and S. Shoham. Lazy abstraction and sat-based reachability in hardware model checking. In G. Cabodi and S. Singh, editors, *Proc. of the Int. Conf. on Formal Methods in Computer-Aided Design*, pages 173–181, 2012.

[61] Y. Vizel, O. Grumberg, and S. Shoham. Intertwined forward-backward reachability analysis using interpolants. In N. Piterman and S. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, **7795** of *Lecture Notes in Computer Science*, pages 308–323. Springer Berlin Heidelberg, 2013.

[62] Y. Vizel and A. Gurfinkel. The Avy tool. http://arieg.bitbucket.org/avy/. Accessed: 2015-12-12.

[63] Y. Vizel and A. Gurfinkel. DRUPing for Interpolants. In K. Claessen and V. Kuncak, editors, *Proc. of the Int. Conf. on Formal Methods in Computer-Aided Design*, pages 99–106, Lausanne, Switzerland, 2014.

[64] Y. Vizel and A. Gurfinkel. Interpolating Property Directed Reeachability. In *Proc. Computer Aided Verification*, **8559** of *Lecture Notes in Computer Science*, pages 260–276. Springer-Verlag, 2014.

[65] C. Wang, B. Li, H. Jin, G. D. Hachtel, and F. Somenzi. Improving Ariadnes bundle by following multiple threads in abstraction refinement. In *ICCAD*, November 2003.

[66] C. Wu, C. Wu, and C. Huang. The V3 tool. http://dvlab.ee.ntu.edu.tw/~publication/V3/. Accessed: 2014-07-01.

[67] Z. Hassan and A. R. Bradley. and F. Somenzi. Better generalization in ic3. In B. Jobstman and S. Ray, editors, *Proc. of the Int. Conf. on Formal Methods in Computer-Aided Design*, pages 157–164, 2013.