Benefits of Bounded Model Checking at an Industrial Setting

Fady Copty[1], Limor Fix[1], Ranan Fraer[1], Enrico Giunchiglia[2], Gila Kamhi[1], Armando Tacchella[2], and Moshe Y. Vardi[*] [3]

[1] Formal Property Verification, Intel Corporation, Haifa, Israel
[2] DIST, University of Genova, Genova, Italy
[3] Dept. of Computer Science, Rice University, Houston, USA

Abstract. The usefulness of Bounded Model Checking (BMC) based on propositional satisfiability (SAT) methods for bug hunting has already been proven in several recent work. In this paper, we present two industrial strength systems performing BMC for both verification and falsification. The first is *Thunder,* which performs BMC on top of a new satisfiability solver, *SIMO*. The second is *Forecast,* which performs BMC on top of a BDD package. SIMO is based on the Davis Logemann Loveland procedure (DLL) and features the most recent search methods. It enjoys *static* and *dynamic* branching heuristics, advanced *back-jumping* and *learning* techniques. SIMO also includes new heuristics that are specially tuned for the BMC problem domain. With Thunder we have achieved impressive capacity and productivity for BMC. Real designs, taken from Intel's Pentium© 4, with over 1000 model variables were validated using the default tool settings and without manual tuning. In Forecast, we present several alternatives for adapting BDD-based model checking for BMC. We have conducted comparison of Thunder and Forecast on a large set of real and complex designs and on almost all of them Thunder has demonstrated clear win over Forecast in two important aspects: capacity and productivity.

1 Introduction

The success of formal verification is no longer measured in its ability to verify interesting design behaviors; it is measured in its contribution to the correctness of the design in comparison to the contribution of other validation methods, i.e., simulation. Therefore, technologies and methodologies that enhance the productivity of formal verification are of special interest. Our research identifies Bounded Model Checking (BMC) based on propositional satisfiability (SAT) to be such a technology.

BMC based on SAT methods [bcrz99, bccz99, sht00] has recently been introduced as a complementary technique to BDD-based Symbolic Model Checking. The basic idea is to search for a counterexample in executions whose length is bounded by some integer k. Given this bound, the model checking problem can be efficiently reduced to a SAT problem, and can therefore be solved by SAT methods rather than BDDs.

---

[*]Work partially supported by NSF grants CCR-9700061 and CCR-9988322, BSF grant 9800096, and a grant from the Intel Corporation.

G. Berry, H. Comon, and A. Finkel (Eds.) : CAV 2001, LNCS 2102, pp. 436-453, 2001.

In this paper, we report our detailed evaluation of SAT-based BMC at an industrial setting. Our initial interest in BMC and SAT technology has been due to the several recent papers [bcrz99, bccz99, sht00] that have compared BDD-based model checking to SAT-based model checking and have concluded that many of the (BDD-based model checking) hard problems can easily be solved by SAT-based model checkers. The test cases used in the comparisons reported in [sht00] were drawn from the internal benchmark of a state-of-the-art BDD based symbolic model checker, RuleBase [bee96a, bee97a]. Therefore, in [sht00], no definite conclusions could be derived on the capacity benefit of the SAT technology, since all the verification cases were in the capacity ballpark of RuleBase. Although Biere et al. report in [bcrz99] that their SAT-based BMC consistently outperformed the BDD-based symbolic model checker, SMV, the results that they convey are on verification test cases made up of hundreds of sequential elements and inputs well in the capacity range of BDD-based symbolic model checkers.

Furthermore, prior comparisons [sht00] leave open the question whether the difference in performance and capacity is due to the underlying technology--BDD versus SAT, or is due to the difference between bounded and unbounded model checking. Moreover, both in [bcrz99, sht00] no extensive expert configuration and tuning have been done in the extraction of the performance numbers for BDD-based model checkers in their comparison with tuned SAT-based bounded model checkers.

In order to understand the clear benefit of bounded model checking and SAT technology at a formal-verification setting, we undertook the task of developing industrial strength BMC using both BDD and SAT algorithms and have thus provided the means for a fair comparison. On one hand, we have optimized Intel's unbounded BDD-based model checker, Forecast, for bounded model checking. On the other hand, we have developed a state-of-the-art SAT-based bounded model checker, Thunder.

Since our interest in SAT technology was in addressing the productivity problem of the current formal verification techniques, we have evaluated the benefits of BDD-based and SAT-based bounded model checking with respect to productivity. We have built a performance benchmark made up of a large number of hard real-life falsification test cases chosen from the unbounded Forecast's internal benchmark base. For each problem, we have built a falsification version that results in a counterexample of minimal length k, and a verification version of length k-1. In this manner, we have evaluated the power of SAT based bounded model checking for both verification and falsification.

In order to understand the benefits of SAT technology with respect to productivity, we tuned both Thunder and Forecast for the domain of bounded model checking and came up with a default best configuration for both engines. Since it is very hard to measure the tuning effort, we have compared tuned and default Forecast versus default Thunder. Surprisingly the default and best setting for Thunder was the same for all the test cases in the benchmark. Although Thunder significantly outperformed untuned Forecast; its performance was very similar to tuned Forecast for almost all the cases except for a few cases that could not be verified by any setting of Thunder. The performance benchmark therefore showed a clear productivity gain achieved by Thunder in the drastic reduction of user ingenuity and tuning effort in running the tools.

The capacity benchmark that we extracted by eliminating the pruning directives on all the test cases of the performance benchmark demonstrated that Thunder with no pruning effort could verify most of the test cases. These benchmarks, corresponding to circuits with thousands of sequential elements and inputs, are far beyond the capacity of Forecast and of any other BDD-based symbolic model checker. Therefore, the conclusion from the capacity benchmark was that Thunder has impressive capacity (can verify designs with over thousands of inputs and sequential elements) and potentially increases the productivity of the verification engineer by reducing the pruning effort significantly.

Thunder reads in RTL models, e.g., written in Verilog or VHDL, and in addition a set of assumptions and assertions expressed in our new temporal specification language, ForSpec [arm01]. Thunder is compatible with a wide-range of recently developed, state-of-the-art SAT solvers (e.g., GRASP, SATO, Prover). We report the benchmark results of Thunder based on a new SAT solver SIMO, developed at the University of Genova. SIMO is based on the Davis-Logemann-Loveland procedure (DLL) [dll62]. Similar to other state-of-the-art DLL-based algorithms, SIMO's strength is based on: (1) advanced procedures for *choosing* the next variable on which to split the search and (2) advanced *backtracking* mechanisms. SIMO features various forms of backtracking. In particular, besides the standard backtrack to the last choice point, SIMO implements a *Conflict-directed BackJumping schema, CBJ*, and *CBJ-with-Learning* [dec90a, pro93a, bs97a]. *CBJ-with-Learning* algorithm was chosen to be the best setting following intensive benchmarking with real-life test cases. In the context of heuristics to choose the splitting variable, we evaluated a wide range of known dynamic heuristics, both greedy (e.g., MOMS) and Boolean Constraint Propagation (BCP) [fre95a] based (e.g., Unit), and introduced a new dynamic heuristics, *UniRel2*, that proves to be the best for the Intel bounded-model checking benchmark. Unirel2 is a domain specific heuristics, since it gives preference to model variables, and also takes into account the simplification imposed on the auxiliary variables. Previous evaluation [sht00] of dynamic splitting heuristics reported static heuristics to be a clear winner over dynamic heuristics. Our results are not compatible with [sht00] in the sense that for our benchmark the dynamic splitting heuristics, Unirel2, worked much better than the available static heuristics in SIMO. Since we have not evaluated Unirel2 versus the original static heuristics introduced in [sht00], our conclusion is that dynamic splitting heuristics tuned for the domain of bounded model checking as is Unirel2 can be very robust for industrial size designs. Our intensive evaluation clearly pinpointed *Unirel2* and *CBJ-with-Learning* as the winning setting of Thunder for Intel's benchmark.

Our BDD-based model checker, Forecast, is built on top of a powerful BDD package, and contains most of the recently published state-of-the-art algorithms for symbolic model checking. In addition to the *unbounded* model checking algorithms in Forecast we developed *bounded* ones in order to give BDD based BMC a fair chance in the comparisons against Thunder. We tried to get an automatic (not requiring additional human tuning) default setting for Forecast as we have done for Thunder. We were not able to get a default setting that is good for all the test cases and an automatic static BDD variable ordering that beats the best humanly tuned variable order. Therefore, we compare both best default setting and tuned setting for Forecast with default setting of Thunder. The comparison reveals the productivity boost gained by Thunder, since the default setting of Thunder clearly outperforms the default setting of Forecast and is very competitive with the tuned Forecast setting.

As a summary, the unique contribution of this work is in the adaptation of unbounded BDD-based model checking to bounded model checking, optimizations of SAT based methods (mainly dynamic splitting heuristics) for bounded model checking and a thorough and fair evaluation of bounded model checking on SAT versus BDD based model checking making use of a rich set of real-life complex verification and falsification test cases.

The paper is organized as follows. In Section 2, we give an overview on Thunder and present experimental results that demonstrate the best SIMO and CNF generator configuration for Thunder. Section 3 describes our effort to achieve best results for BMC on BDD. In Section 4 we present experimental results comparing Thunder with Forecast. Section 5 describes our conclusion and future research directions.

2 Thunder: Bounded Model Checker on SAT

Thunder, our bounded checker on SAT technology, resembles the work of Bierre et al. [bccz99] in the reduction of the symbolic model checking problem to a bounded model checking problem and consequently to the problem of propositional satisfiability. Thunder, which makes use of a powerful DLL-based engine, SIMO, as its default SAT engine, is also compatible with other state-of-the-art SAT engines such as GRASP, SATO, Prover Plug-In™ [PPI, sta89]. We report in this paper our experience of Thunder with SIMO since our contribution is mainly in the tuning of DLL-based algorithms in the context of bounded-model checking.

2.1 Transforming the Bounded Model Checking Problem to Formulas

The basic idea in SAT based bounded model checking is to consider only paths of bounded length k and to construct a propositional formula that is satisfiable iff there is a counterexample of length k. BMC is concerned with finding counterexamples of limited length k, and thus it targets falsification and partial verification rather than full verification.

In order to fully verify a property one needs to look for longer and longer counterexamples by incrementing the bound k, until reaching the diameter of the finite state machine [bccz99]. However, the diameter might be very large in some examples, and there is no easy way to compute it in advance. This issue is addressed in [sss00] which incorporates induction in BMC that allows the algorithm to be used both for verification and falsification.

Assume that we have a finite state machine M with initial states I and transition relation TR, where both I and TR are encoded symbolically as Boolean formulas. Assume also, that we want to check if an invariance property P holds for all states reachable in a bounded number of steps. It is sufficient to focus only on invariance properties since the safety specifications expressed in our temporal language, ForSpec, are compiled into such invariance properties.

Our experience shows that the performance and capacity of Thunder is very dependent on the way we generate the propositional formulae describing the counterexample. Similarly to CMU's implementation of BMC, Thunder provides different settings that we describe below. We also provide experimental results that compare the various settings.

The propositional formula describing a path from s_0 to s_k requires s_0 to be an initial state and also that there is a transition from s_i to s_{i+1} for $0 \quad i < k$:

$$Path(s_0,\dots,s_k) = I(s_0) \quad TR(s_0, s_1) \quad \dots TR(s_{k-1}, s_k)$$

Thunder implements three different checks for a counterexample (similar to what is provided in CMU's BMC tool). The first one, referred to as *bound k*, looks for a violation of P in all the cycles from 0 to k:

$$Path(s_0,\dots,s_k) \quad (\quad P(s_0) \quad \dots \quad P(s_k))$$

The second check, referred to as *exact k*, looks for a violation of P exactly in the last cycle k:

$$Path(s_0,\dots,s_k) \qquad P(s_k)$$

Finally, the third check, referred to as *exact-assume k*, looks for a violation of P at cycle k and assumes P to be true in all the cycles from 0 to k-1:

$$Path(s_0,\dots,s_k) \quad P(s_0) \quad \dots \quad P(s_{k-1}) \qquad P(s_k)$$

As expected, using *exact* or *exact-assume* is significantly faster than *bound*, but then they solve an easier problem. For the sake of a fair comparison with BDD model checking, all the results in this section are obtained with *bound*. We will return in section 5 to the *exact* and *exact-assume* checks, since they are the only ones who can cope with the capacity challenging examples presented there.

We also implemented the Bounded Cone of Influence (BCOI) optimization proposed in [bcrz99]. This optimization rarely negatively affects so we use it as a default, such that all the results below are obtained in the presence of BCOI. Our experiments used a DLL-based SAT solver, SIMO [tac00], described in the next section.

2.2 DLL Based Satisfiability Engine - SIMO

As many other modern SAT solvers, SIMO [tac00] is based on the well-known Davis-Logemann-Loveland (DLL) algorithm [dll62]. DLL assumes the propositional formula to be in Conjunctive Normal Form (CNF) and it employs a backtracking search. At each node of the search tree, DLL assigns a Boolean value to one of the variables that are not resolved yet. The search continues in the corresponding sub-tree after propagating the effects of the newly assigned variable, using Boolean Constraint Propagation (BCP) [fre95a]. BCP is based on iterative application of the unit clause rule. The procedure backtracks once a clause is found to be unsatisfiable, until either a satisfying assignment is found or the search tree is fully explored. The last case implies that the formula is unsatisfiable.

SIMO's strength is based on: (1) advanced *backtracking* mechanisms (2) advanced procedures for *choosing* the next variable on which to split the search. Besides the standard backtracking to the last choice point, SIMO implements also *Conflict-directed Back-Jumping* (CBJ) and *CBJ-with-Learning* [dec90a, pro93a, bs97a]. In Section 3.2.1, we explain at a high-level the *CBJ-with-Learning* algorithm which was chosen to be the best setting following intensive benchmarking with real-life test cases.

In the context of heuristics to choose the splitting variable, we compare several dynamic heuristics and introduce a new dynamic heuristics, *UniRel2*, that proves to be the best for the Intel bounded-model checking benchmark. Section 3.2.2 explains at a high level the heuristics that have been compared and the experimental results that justify our decision.

2.2.1 CBJ-with-Learning

Since the basic DLL algorithm relies on simple chronological backtracking, and most heuristics are targeted to select the literal that satisfies the largest number of clauses, it is not infrequent for DLL implementations to get stuck in possibly large sub-trees whose leaves are all dead-ends. This phenomenon occurs when some selection performed way up in the search tree is responsible for the constraints to be violated. The solution, borrowed from constraint network solving [dec92], is to jump back over the selections that were not at the root of the conflict, whenever one is found. The corresponding technique is widely known as *Conflict-directed Back-Jumping* (CBJ) [pro93]. It has been reported from the authors of RELSAT [bs97], GRASP [ss96] and SATO [zha97] that CBJ proved a very effective technique to deal with real-world instances.

It turns out that in all these solvers, CBJ is tightly coupled with another technique, called *Learning*. CBJ can be very effective in "shaking" the solver from a sub-tree whose leafs are all dead ends, but since the cause of the conflict is discarded as soon as it gets mended, the solver may get repeatedly stuck in such local minima. To escape this pattern, some sort of global knowledge is needed: the causes of the conflicts may be stored to avoid repeating the same mistake over and over again. This process is usually called no-good or recursive learning. Our BMC experience with SIMO agrees with previous work [bs97] that reports that CBJ with relevance learning is essential for good performance in the domain of SAT.

2.2.2 Splitting Heuristics

The splitting heuristic needs to decide which variable to assign next from the set S of variables that were not assigned yet. Since the conversion to CNF [pg86] introduces many additional variables (one for each non-atomic sub-formula of the original formula) we restrict the set S to the variables of the original formula, also called *relevant* variables. As pointed out in [sht00], this optimization is very useful and our results confirm this conclusion.

SIMO features a static splitting heuristic that relies on a user-supplied order to choose each splitting variable among relevant variables. Additionally, SIMO has a wide range of dynamic splitting heuristics that showed to be very effective in our experience with bounded model checking.

SIMO's dynamic splitting heuristics fall broadly into two categories: BCP heuristics, and greedy heuristics. BCP heuristics choose the splitting variable by tentatively assigning truth-values to (some of) the unassigned variables and then performing BCP. In this way the exact amount of simplification produced by each possible assignment can be calculated. Moreover, BCP heuristics can detect *failed literals*, i.e., literals that once assigned produce a contradiction after a single sweep of BCP. *Greedy* heuristics choose the splitting variable by *estimating* the amount of simplification caused by an assignment. Relying on an estimate rather than an exact calculation makes greedy heuristics faster than BCP heuristics, but also less precise and incapable of detecting failed literals. In this regard, greedy heuristics can be seen as an approximation to the BCP ones. Both types of heuristics branch on the variable that produces- or is estimated to produce- the maximum simplification in the formula. We used heuristics from both categories in our experiments with SIMO.

Among the greedy heuristics, we have used *Moms* and *Morel* heuristics. For each open variable *p*, *Moms* computes the number of binary clauses in which *p* occurs, and uses this quantity as the expected amount of simplification when assigning *p*. *Morel* works in the same way as *Moms*, but its choice is restricted to relevant variables only.

From the class of BCP heuristics, we have used three BCP heuristics, called *Unit*, *Unirel*, and *Unirel2*. For each open variable *p*, *Unit* tentatively assigns both *p* and *p*: for both choices, BCP is performed and the number of unit-propagated variables is collected. If the heuristic yields a contradiction by assigning *p* (resp. *p*) then it immediately assigns *p* (resp. *p*): if also *p* (resp. *p*) fails, then *Unit* halts and backtracks, otherwise it goes on in trying to select a variable. If all variables are assigned during this process or all the clauses are satisfied, *Unit* reports that a satisfying assignment was found. *Unirel* works in the same way as *Unit*, except it considers only relevant variables when collecting the number of unit-propagated variables. *Unirel2*, on the other hand, tentatively assigns only relevant variables, but it collects the number of all the unit-propagated variables.

We compared the performance of *Moms, Morel, Unit, Unirel, Unirel2*, and *Static* heuristics in SIMO making use of a benchmark of 26 real-life test cases. The benchmark is evenly distributed between falsification and verification test cases. Unirel2 heuristics provides a clear performance and capacity boost over the other heuristics. We chose to report only timings of the dynamic heuristics, since SIMO does not include all the known static heuristics. The current static heuristics in SIMO performed much worse than the dynamic heuristics for our benchmark. However, in order to derive any accountable conclusions on the effectiveness of dynamic heuristics versus static heuristics, SIMO needs to be enriched with the latest static heuristics for bounded model checking [sht00].

For all the runs reported in Figure 1, we use 3-hour time-out limit. As can be seen, Moms heuristics is significantly inferior to Unit and Unirel2 (except for circuit12). On the other hand, Unirel2 provides a clear performance boost over Unit heuristics.

In the analysis of the results, let us concentrate on three representative heuristics from each category: Moms, Unit and Unirel2. Moms is the basic and most popular greedy heuristics. Unit is the simplest of the BCP-based heuristics, and Unirel2 is the overall fastest of the 6 (Static, Moms, Morel, Unit, Unirel, Unirel2) that we have tried. Our results indicate clearly that BCP heuristics perform better than greedy heuristics for this domain of problems. BCP heuristics take into account the structure of the CNF formula which closely reflects the structure of the original formula (before the CNF conversion). Indeed, in the CNF formula there are (possibly long) chains of implications. With BCP heuristics, a literal occurring at the top of a chain is preferred to a literal occurring in the middle of the same chain. This is not guaranteed to be the case with greedy heuristics, where only the number of occurrences counts. Moreover, both Unit and Unirel2 feature the failed literal detection mechanism that

Moms heuristics is missing. This mechanism allows Unit and Unirel2 to perform more simplifications at each node.

Unirel2 considers only the relevant variables (i.e., the model variables) whereas Unit heuristics considers all the variables as a candidate for splitting. Although the greedy nature of Unit heuristics makes it more accurate, in most cases the time spent to choose a variable will be much more in Unit than Unirel2 (since the number of all the variables can be significantly larger than the number of relevant variables). Therefore, to give up a bit on quality provides better overall performance for Unirel2.

3 Forecast – A BDD-Based Symbolic Model Checker

Several recent papers [bcrz99, bccz99, bccfz99, sht00] compare traditional BDD - based model checking with SAT-based model checking, showing that in many cases SAT technology dramatically outperforms BDD technology. Such comparisons (except [bccz99]), however, neglect one crucial aspect that distinguishes the two approaches. Traditional BDD-based model checking searches for counterexamples of *unbounded* length. In contrast, SAT-based model checking searches for counterexamples of a predetermined *bounded* length. Thus, prior comparison leaves open the question whether the difference in performance is due to the underlying technology--BDD vs. SAT, or is due to the difference between bounded and unbounded model checking. To answer this question, we undertook the task to first adapt a BDD-based model checker to bounded model checking and then compare its performance to a SAT-based model checker.

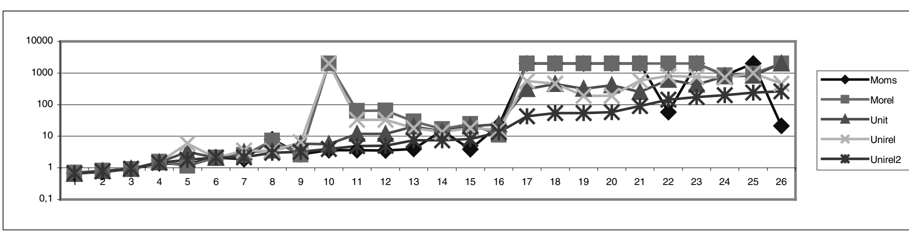


Fig. 1. Comparison of Thunder run-time with the dynamic heuristics Moms, Morel, Unit, Unirel and Unirel2 for a benchmark of 26 test cases on a logarithmic scale. In the reported runs, time-out has been set to 3 hours. The x axis indicates the test case where the y axis indicates the Thunder run-time. We can clearly see that Moms and Unit heuristics times out for 6 and 1 out of 26 test cases, respectively.

3.1 Adapting Forecast for Bounded Model Checking

Forecast is a BDD-based model checker developed and deployed in Intel, using an in-house BDD package. Forecast can run in two modes. In the standard mode, Forecast applies either forward or backward breadth-first-search *traversal* from a source set S to a target set T with respect to a transition relation TR, when *Image* refers to a pre-image or post-image operation:

```
Traversal(S,T,R)
Reach=Frontier=S
while (Frontier        ) {
   if (Frontier    T      ) terminate;
   Frontier := Image(Frontier,R) - Reach
Reach  := Reach     Frontier
}
```

A difficulty often faced by standard traversal is the excessive growth of the Frontier or the Reach set ("state explosion"). To address the former problem, Forecast can apply a *prioritized-traversal* algorithm, see [fkzvf00]. In prioritized traversal mode, we split the Frontier into two balanced parts when its BDD size reaches some predetermined threshold. Thus, instead of maintaining one frontier, the algorithm maintains several frontiers, organized in a priority queue. A given traversal step consists of choosing one frontier set and applying the image operator to that set. Thus, prioritized traversal can be viewed as a mixed bread-first/depth-first search.

How can we adapt standard traversal and prioritized traversal to bounded model checking? The first change is to bound the length of the traversal.

```
BoundedTraversal1(S,T,R,k)
I :=0
Reach=Frontier=S
 For (I = 0; I< k; I++) {
   If (Frontier    T      ) terminate;
   Frontier := Image(Frontier,R)-Reach
   Reach  := Reach     Frontier;
}
```

If the distance between S and T is less than or equal to k, then the running time of BoundedTraversal1 and Traversal would clearly coincide. Note, however, that termination is not an issue in bounded traversal. Thus, from a termination point of view, there is no need to maintain Reach.

```
BoundedTraversal(S,T,R,k)
 Frontier=S
  For (I = 0; I< k; I++) {
     If (Frontier    T      ) terminate;
      Frontier := Image(Frontier,R);
  }
```

However, besides guaranteeing termination, *Reach* was used in the classic algorithm to cut down on the size of *Frontier*. Thus, one would expect BoundedTraversal to run into huge Frontiers, resulting in weak performance. This is where prioritized traversal comes to the rescue. As before, we split the Frontier whenever its BDD gets larger than some threshold, maintaining a set of frontiers in a priority queue. With each frontier we maintain its distance from the source set S. We choose frontiers and apply the image operator, making sure that the bound is never exceeded. This results in a prioritized version of BoundedTraversal.

So far we have treated S and T in a symmetrical fashion. In practice, however, the initial states are defined in terms of many state variables, while the error state is defined in terms of a small number of state variables, called the *error variables*. Cone-of-influence (COI) reduction algorithms take advantage of that by eliminating state variables that cannot have any effect on the error variables. In the context of BMC, one can be more aggressive and eliminate variables that cannot have an effect on the error variables in a bounded number of clock cycles. This optimization called Bounded Cone-of-Influence (BCOI) was introduced in [bcrz99].

Forecast has a "lazy"mode [yt00] that effectively applies a BCOI reduction. This mode is effective only in backward traversal – for each pre-image, one identifies the relevant variables appearing in the frontier and builds a smaller TR based on those relevant variables. Naturally, this reduction is more effective when the Frontier has a small number of state variables, e.g., when the Frontier is close to the set of error states. We have adapted the "lazy model checking" mode of Forecast to BMC (i.e., we search for a counter-example for k pre-image steps).

3.2 Default Configuration for Forecast

Since we built the benchmark of bounded model checking from internal Intel's benchmark base[†] of Forecast, every test case had the best setting for unbounded Forecast meaning

- The right pruning directives to reduce the size of the model
- The best initial order that the FV expert user could get
- The best (CPU time-wise) configuration that the FV expert user could get

The time spent by the FV expert to get to the best initial order and tool configuration could not be derived from the benchmark. Furthermore, the configuration in the benchmark base was for unbounded Forecast. In order to make a fair comparison with Thunder, in search for a best default setting, we experimented with three recent state-of-the-art algorithms of Forecast described in Section 3.1 : bounded prioritized-traversal, unbounded prioritized traversal [fkzvf00], and bounded lazy model checking. For all the runs a partitioned transition relation was used.

We present results achieved by Forecast under two different configurations.

- *Automatic* : the initial variable order is automatically computed by a static variable ordering algorithm
- *Semi-automatic* : the initial order is taken from the order that was calculated by previous runs of Forecast with dynamic reordering[‡]

[†] All the properties verified were safety properties.

[‡] This evaluation is similar to RB2 configuration in [sht00a]; however, in our case the order gets refined by the dynamic reordering output of more than one run of the model checker.

Both of the configurations were run with dynamic reordering with the threshold of 500K BDD nodes (meaning dynamic reordering will be turned on when the total number of BDD nodes allocated exceeds 500K).

TestCase	Bound	Forecast **Lazy** (secs)	Forecast **Prioritized** (secs)	Forecast **Prioritized Unbounded** (secs)
Circuit 1	5	27.3	1340	114.1
Circuit 2	7	1.1	0.56	2.1
Circuit 3	7	2.1	15.00	106.1
Circuit 4	11	9.1	2233.00	6189.0
Circuit 5	11	TIMEOUT	107800.00	4196.2
Circuit 6	10	TIMEOUT	TIMEOUT	2354.1
Circuit 7	20	4187.2	TIMEOUT	2795.1
Circuit 8	28	TIMEOUT	TIMEOUT	TIMEOUT
Circuit 9	28	TIMEOUT	TIMEOUT	TIMEOUT
Circuit 10	8	TIMEOUT	TIMEOUT	2487.1
Circuit 11	8	TIMEOUT	TIMEOUT	2940.5
Circuit 12	10	TIMEOUT	TIMEOUT	5524.1
Circuit 13	37	TIMEOUT	TIMEOUT	TIMEOUT

Table 1. Automatic Setting Comparisons. Forecast performance comparisons for different configurations with automatically generated initial order. A time-out limit of 3 hours has been set.

Table 1 and Table 2 summarizes the time spent by Forecast in verifying these test cases when a time limit of 3 hours has been set. All experiments were run on HP J6000 work station with 2 Gigabyte memory. Table 1 reports Forecast runs when the initial order is automatically generated by the tool and Table 2 reports the results when Forecast is given a semi-manual order (i.e. the enhanced order is obtained by running Forecast with dynamic ordering several times).

The bottom line of Table 2 is the criticality of "a good initial order" for good performance of a BDD-based model checking. Without a good order, Forecast is far from being competitive. Although unbounded prioritized traversal does not outperform the other two algorithms for the test cases that all three complete, we selected it to be the winner configuration for the automatic default setting, since it times out much less than the other two (only three times). Although the success of unbounded prioritized traversal versus the bounded version is intriguing, we believe it to be due to the better suitability of the initial variable orders to the unbounded prioritized traversal.

Table 2 dilutes the effect of bad initial order; however still no winner configuration for all or most of the test cases can be chosen indicating the difficulty to set an always winning setting for BDD-based model checkers. Lazy model checking in Table 2 for the test cases that it can complete beats the other two. On the other hand, it cannot complete 6 verification cases in the time set. No clear winner could be found between the bounded and unbounded versions of Prioritized Traversal. Although performance of prioritized traversal is worse than lazy model checking for all the test cases where lazy model checking completes, it times out less (4 times).

As can be seen no good (overall winning) default setting could be selected for Forecast based on the results of Table 1 and Table 2. We have selected the unbounded prioritized traversal as the default setting, since it is a clear winner for Table 1 and not performing worse than the others for Table 2; moreover, the setting in Table 1 is more fair for comparison with default setting of Thunder, since the initial order selection time is included in the overall Thunder run time. Table 2 numbers do not include the time spent in the generation of the initial order time (i.e, the runs of symbolic model checking to generate good orders). However, note that although unbounded prioritized traversal is not guaranteed to find the counter-example of the minimal length, for all the falsification test cases that we have tried a counter-example of length k or less was reported.

TestCase	Bound	Forecast **Lazy** (secs)	Forecast **PrioritizedBounded** (secs)	Forecast **Prioritized UnBounded** (secs)
Circuit 1	5	7.4	21.0	21.8
Circuit 2	7	1.6	1.8	1.9
Circuit 3	7	2.3	5.2	5.6
Circuit 4	11	6.7	89.9	241
Circuit 5	11	6432.5	64.2	80.4
Circuit 6	10	TIMEOUT	44.6	36.8
Circuit 7	20	134.3	7250.2	TIMEOUT
Circuit 8	28	TIMEOUT	1421.1	1287.5
Circuit 9	28	TIMEOUT	TIMEOUT	1040.3
Circuit 10	8	147.4	693.1	694.6
Circuit 11	8	143.9	260.6	261.0
Circuit 12	10	2379.2	4657.0	1041.5
Circuit 13	37	TIMEOUT	TIMEOUT	4188.0
Circuit 14	41	TIMEOUT	1864.36	TIMEOUT
Circuit 15	12	423.1	TIMEOUT	TIMEOUT
Circuit 16	40	16.1	783.0	TIMEOUT
Circuit 17	40	TIMEOUT	TIMEOUT	33.1

Table 2. Semi-automatic Setting Comparions. Forecast performance comparisons for different configurations with semi-automatic generated good initial order.

4 Comparison of Thunder and Forecast

We evaluated bounded Thunder versus bounded Forecast with respect to performance and capacity. For each of these, we studied the aspect of productivity.

Our performance benchmark consists of 15 real-life falsification test cases. All the 15 test cases were from the unbounded Forecast benchmark base (meaning all the test cases could be falsified at special settings of Forecast). Since unbounded version of Forecast finds counterexamples of minimal length, we knew beforehand the minimal length k for the counterexamples that can be generated for each test case. Therefore, we could generate for each test case a bounded k-1 verification version. Furthermore, we added to our benchmark two hard verification cases where we requested both Forecast and Thunder to verify that no counter-example exists. In this manner, we evaluated the power of bounded Thunder versus the power of bounded Forecast for both verification and falsification test cases.

We built the capacity benchmark (made up of 11 test cases) by eliminating the pruning directives of some of the test cases in the performance benchmark and we added brand-new test cases clearly surpassing the capacity limits of Forecast and other state-of-the-art model checkers (i.e., verification test case with over 2000 sequential elements and inputs).

4.1 Analysis of Performance Benchmark Results

Table 3 compares the performance of default Thunder setting with default and tuned settings of Forecast. The default setting of Forecast (prioritized traversal + dynamic reordering + partitioned transition relation + automatic initial ordering) is far from being competitive. For tuned Forecast results, we report the configuration that has worked best. All the tuned configurations, except the ones explicitly reported do not activate dynamic reordering. As can be seen they include variations of transition relation (*tr part (partitioned)*, *tr mono (monolithic))*, variations of priorities (*min size, max states, BFS, DFS*) for prioritized search and variations of configurations for lazy model checking.

The comparison of default settings of Thunder and Forecast reveals that Forecast's default performance and capacity is far below Thunder's. On the other hand, the comparison results reveal that Thunder at default setting provides compatible performance to tuned Forecast results. For 6 benchmarks out of 17, Thunder default settings beat tuned Forecast setting's results by 2 to 3X (See in Table 3 the comparison on Circuit 3, 5, 8, 9, 10, and 11). For Circuit 13, Thunder default performance wins over Forecast tuned performance by 9X. Nevertheless, tuned Forecast results are 2 to 3 X better for Circuit 7 and Circuit 12. Thus, there is no clear winner with respect to performance when default Thunder and tuned Forecast's performance is compared. The only conclusion is that Thunder gives a significant productivity boost. In short, unlike Forecast Thunder does not require high tuning effort to perform well.

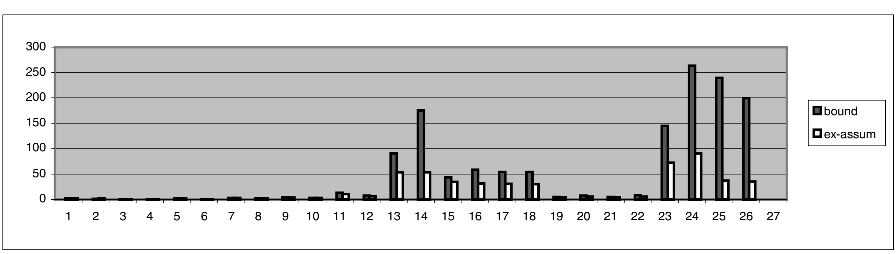
Through the performance benchmark, we also tested the capacity of Thunder versus Forecast. Three test cases that could be easily verified by tuned Forecast setting could not be verified by any heuristics of Thunder (Circuit 14 (bound 40, 41), Circuit 16 (bound 40), Circuit 17 (bound 60)). Although Thunder could not solve (except for Circuit 16) the bounded model checking problem for these test cases, it could solve a variation of the problem (exact, exact-assume described in Section 2.1). As seen in Figure 2, although *exact* and *exact-assume* modes are significantly faster than the *bound mode*, the problem solved is simpler. By exact-assume, we are verifying the existence of a counterexample of exactly length k. Clearly, the solution of k exact-assume verification cases where the existence of a counter-example of length 1 to length k are verified will be equivalent to verifying *bound k* problem. Although too time consuming, the fact that Thunder could solve the exact-assume problem for most of the hard test cases for the bound version, indicates that the solution of these problems is in the capacity range of Thunder.

4.2 Analysis of Capacity Benchmark Results

We generated the capacity benchmark by eliminating the pruning directives set to get the model checking cases through. The size of the test cases in the capacity benchmark containing thousands of sequential elements and inputs is clearly far beyond the capacity of Forecast and any other state-of-the art BDD-based symbolic model checker. Therefore, no results are reported for Forecast. Thunder has successfully verified a wide range of the test cases in the capacity benchmark indicating a clear win over Forecast for un-pruned test cases. The fact that Thunder could verify these test cases without the extensive pruning effort required for BDD-based model checker is also a clear indication of productivity gain achieved by Thunder.

In Table 4, we report the CPU time of the overall run of Thunder for 11 test cases. The test cases, circuit 1, 3 and 4, are the same test cases that have been used for the performance evaluation. For this benchmark, the pruning directives set by the user to get the verification fit the capacity of BDD-based model checking have been eliminated. We report the number of latches and inputs before and after the application of automatic pruning operation (cone-of-influence reduction with respect to property). As can be seen, using Thunder, test cases with over 9000 latches and inputs could be verified without requiring any additional manual pruning effort. In Table 4, the bounded model checking problem fed into Thunder SAT engine represents a verification case (i.e., Ncircuit8) of total 6832 inputs and sequential elements representing 121786 SAT variables and 358334 clauses. These results, although in the domain of bounded model checking, are a clear indication of the promise in this technology to establish model checking as a robust and popular technique at industrial validation environments.

Fig. 2. Performance comparison results of bound and exact-assume modes of Thunder for the same k**.** The x axis represents the test cases when the y axis represents Thunder run-time in seconds.



TestCase	Bound	Variables, Clauses in Thunder	Thunder Default (secs)	Bounded Forecast Default (secs)	Bounded Forecast Tuned(secs), Configuration
Circuit 1	5	5055, 14690	2.43	114	2.80, lazy
	4	3987, 11559	1.59		
Circuit 2	7	2000, 5727	0.81	2	0.56, lazy
	6	1688, 4820	0.64		
Circuit 3	7	3419, 8977	2.01	106	1.29, lazy
	6	2908, 7623	1.17		
Circuit 4	11	6740, 18884	1.91	6189	1.04, lazy
	10	6085, 17030	1.53		
Circuit 5	11	10258, 29515	10.12	4196	35.14, unbounded-prio, tr part
	10	9303, 26746	8.78		
Circuit 6	10	8829, 25587	5.51	2354	8.34, unbounded-prio, tr part
	9	7918, 22927	4.85		
Circuit 7	20	28769, 85033	236.29	2795	76.88, unbounded-prio: minimum size
	19	27316, 80732	140.65		
Circuit 8	28	38836, 116803	45.66	TIMEOUT	141.00, unbounded-prio: BFS, tr mono
	27	37427, 112558	52.85		
Circuit 9	28	37451, 112465	39.96	TIMEOUT	85.50, unbounded-prio : max states
	27	36092, 108377	50.65		
Circuit 10	8	8734, 25631	5.01	2487	13.90, lazy
	7	7517, 22031	5.79		
Circuit 11	8	8734, 25631	5.01	2940	13.89, lazy
	7	7517, 22031	5.76		
Circuit 12	10	8331, 24497	378.05	5524	159.20, unbounded-prio, tr part
	9	7429, 21826	139.47		
Circuit 13	37	60779, 169824	195.15	TIMEOUT	1586.00, unbounded-prio
	36	59118, 165175	217.75		
Circuit 14	41	51917, 154061	TIMEOUT 91.9 (exact-assume)	TIMEOUT	833.96, unbounded-prio maxstates
	40	50616, 150220	TIMEOUT 83.88(exact-assume)		
Circuit 15	12	9894, 29138	1070.65	TIMEOUT	17.31, unbounded-prio
	11		4209.1		
Circuit 16	40	40718,114344	TIMEOUT (exact-assume)	TIMEOUT	16.1, lazy + reorder
	20	20000, 56009	22.03		
Circuit 17	60	123323, 356126	TIMEOUT 4652.76 (exact-assume)	TIMEOUT	3657.3, tr part, forward reach
	20	41968,120996	247.27		

Table 3. Performance comparison results of default Thunder versus default and tuned Forecast. For Forecast**,** no timing for bound k-1 is reported (clearly it is less than the time reported for bound k).

Unpruned Test Cases	Bound	Num. Latches + Inputs before Automatic Pruning	Num. Latches + Inputs after Automatic Pruning	Variables, Clauses	Thunder time (secs)
Circuit 1	5	12011	152	6831, 19759	6.1
	4	12011	152	5403, 15591	5.1
Circuit 3	7	7054	0.81	24487, 65332	96.1
	6	7054	0.64	200552, 54774	16.37
Circuit 4	11	6586	2.01	119248, 353400	78.61
	10	6586	1.17	107838, 319404	68.2
Ncircuit 6	5	9704	1.91	21351, 61499	29.39
Ncircuit 7	5	17262	1.53	TIMEOUT	TIMEOUT
Ncircuit 8	6	6832	10.12	121786, 358334	576.24
Ncircuit 9	11	3321	8.78	35752, 105268	73.32
Ncircuit 10	6	1457	5.51	50578, 149668	267.91

Table 4. Results from the Capacity Benchmark.

5 Conclusions

In this paper, we have reported our effort to develop industrial strength BMC and the impressive productivity gain achieved by using SAT-based BMC (Thunder) versus BDD-based BMC (Forecast). This gain is achieved by drastic reduction in the required user ingenuity and tuning effort in running the tools. Our work agrees with previous work [bccz99, bcrz99, sht00] in the observation that SAT-based BMC can outperform BDD-based BMC. We show that this statement holds mainly in comparison of SAT-based BMC with untuned BDD-based BMC supporting our conclusion on productivity boost of SAT. Moreover, the evaluation of SAT-based BMC on verification test cases of over thousands of inputs and sequential elements reveals its outstanding capacity to verify designs far beyond the capacity ballpark of the state-of-the-art BDD-based model checkers.

The tuning effort that we have invested to get best default setting for SAT-based BMC introduces a new dynamic heuristics, *Unirel2*, which is a winner for Intel's bounded model checking benchmark supporting the statement made on the productivity gain achieved by Thunder over Forecast.

Acknowledgements

We would like to thank Roy Armoni for his contribution to the development of the infrastructure of Thunder.

References

[abe00] P. A. Abdulla, P. Bjesse, and N. E'en. Symbolic reachability analisys based on SAT solvers. In Proc. of the 6th International Conference of Tools and Algorithms for Construction and Analisys of Systems (TACAS 2000), volume 1785 of LNCS, pages 411-425, Berlin, 2000. Springer.

[arm01] R.Armoni, L.Fix, R.Gerth, B.Ginsburg, T.Kanza, S.Mador-Haim, E.Singerman, A.Tiemeyer, M.Y.Vardi. ForSpec: A Formal Temporal Specification Language, Submitted to ICCAD'01

[bee96a] I.Beer, S.Ben-David, C.Eisner, A.Landver. “RuleBase: An industry-oriented formal verification tool”. In Proc. Design Automation Conference 1996 (DAC'96).

[bee97a] I.Beer, C.Eisner, D. Geist, L.Gluhovsky, T.Heyman, A.Landver, P.Paanah, Y.Rodeh, G.Ronin, Y.Wolfsthal. “RuleBase: Model Checking at IBM”, Proceedings of CAV'97.

[bcm92] J.R. Burch, E.M. Clarke, and K.L. McMillan. Symbolic model checking: 1020 states and beyond.Information and Computation, 98:142-170, 1992.

[bcrz99] Armin Biere, Edmund Clarke, Richard Raimi, and Yunshan Zhu. “Verifying Safety Properties of a PowerPC Microprocessor Using Symbolic Model Checking without BDDs”. Proc. of Computer Aided Verification, 1999 (CAV'99).

[bccz99] A. Biere, A Cimatti, E. M. Clarke, and Y. Zhu. “Symbolic model checking without BDDs”. TACAS'99

[bccfz99] A. Biere, A. Cimatti, E. Clarke, M.Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In Proc. of the 36th Conference on Design Automation (DAC '99), pages 317-320. ACM Press, 1999.

[bry92] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys, 24(3):293-318, September 1992.

[bs97] R. J. Bayardo, Jr. and R. C. Schrag, "Using CSP Look-Back Techniques to Solve Real-World SAT Instances", pages 203-208, Proc. AAAI, 1997.

[dll62] M. Davis and G. Logemann and D. Loveland, “A machine program for theorem proving”, Journal of the ACM, vol. 5, 1962.

[dec90a] R. Dechter, "Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition", Artificial Intelligence, pages 273-312, vol. 41, n.3, 1990.

[fkzvf00a] R.Fraer, G.Kamhi, B.Ziv, M.Vardi, L.Fix, “Efficient Reachability Computation Both for Verification and Falsification”, Proceedings of International Conference on Computer-Aided Design, (CAV'00).

[fre95a] J.W. Freeman, "Improvements to propositional satisfiability search algorithms", PhD Thesis. University of Pennsylvania, 1995.

[mcm93] K.L. McMillan. Symbolic Model Checking: an Approach to the State Explosion Problem. Kluwer Academic Publishers, 1993.

[pg86] D.A. Plaisted and S. Greenbaum, "A Structure-preserving Clause Form Translation", Journal of Symbolic Computation, vol.2, pages=293-304, 1986.

[PPI] *Prover 4.0 Application Programming Reference Manual*, Prover Technology AB, 2000. PPI-01-ARM-1.

[pro93a] P. Prosser, “Hybrid algorithms for the constraint satisfaction problem", Computational Intelligence, vol. 9, n. 3, pages 268-299, 1993.

[sht00] O. Shtrichman, “Tuning SAT checkers for Bounded Model-Checking” Proc. of Computer Aided Verification, 2000 (CAV'00).

[sss00] M. Sheeran, S. Singh and G. Staalmarck, “Checking safety properties using induction and a SAT solver” Proceedings of Formal Methods in Computer Aided Design 2000 (FMCAD00)

[ss96] J.P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability.Technical report, University of Michigan, April 1996.

[ss98] M. Sheeran and G. Stalmarck. A tutorial on Stalmarck's proof procedure for propositional logic. In Proc. of the 2nd International Conference on Formal Methods in Computer Aided Design (FMCAD '98), volume 1522 of LNCS, pages 82-99, Berlin, 1998. Springer.

[sta89] G. Stalmarck. System for Determining Propositional Logic Theorems by Applying Values and Rules to Triplets that are Generated From Boolean Formula. Swedish Patent No. 467076 (approved 1992), US Patent No. 5276897 (1994), European Patent No. 0403454 (1995), 1989.

[tac00] A. Tacchella. "SAT Based decision procedures for knowledge representation and Formal Verification". PhD Thesis. University of Genova. 2000.

[wbcg00] P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining Decision Diagrams and SAT Procedures for Efficient Symbolic Model Checking. In Proc. of the 12th International Conference on Computer Aided Verification (CAV 2000), volume 1855 of LNCS, pages 124-138, Berlin, 2000. Springer.

[yt00] J.Yang, A.Tiemeyer." Lazy Symbolic Model Checking". DAC'00.

[zha97] H. Zhang. SATO: An efficient propositional prover. In William McCune, editor, Proceedings of the 14th International Conference on Automated deduction, volume 1249 of LNAI, pages 272-275, Berlin, July13-17 1997. Springer.