Tuning SAT checkers for Bounded Model Checking

Ofer Shtrichman

The Minerva Center for Verification of Reactive Systems, at the Dep. of Computer Science and Applied Mathematics, The Weizmann Institute of Science, Israel;
and IBM Haifa Research Lab
email: ofers@summer.weizmann.ac.il

Abstract. Bounded Model Checking based on SAT methods has recently been introduced as a complementary technique to BDD-based Symbolic Model Checking. The basic idea is to search for a counter example in executions whose length is bounded by some integer k. The BMC problem can be efficiently reduced to a propositional satisfiability problem, and can therefore be solved by SAT methods rather than BDDs. SAT procedures are based on general-purpose heuristics that are designed for any propositional formula. We show that the unique characteristics of BMC formulas can be exploited for a variety of optimizations in the SAT checking procedure. Experiments with these optimizations on real designs proved their efficiency in many of the hard test cases, comparing to both the standard SAT procedure and a BDD-based model checker.

1 Introduction

The use of SAT methods for Symbolic Model Checking has recently been introduced in the framework of Bounded Model Checking [4]. The basic idea is to search for a counter example in executions whose length is bounded by some integer k. The BMC problem can be efficiently reduced to a propositional satisfiability problem, and can therefore be solved by SAT methods rather than BDDs. SAT procedures do not suffer from the potential space explosion of BDDs and can handle propositional satisfiability problems with thousands of variables. The first experiments with this idea showed that if k is small enough, or if the model has certain characteristics, it outperforms BDD-based techniques [5].

SAT procedures are based on general-purpose heuristics that are designed for any propositional formula. In this paper we will show that the unique characteristics of BMC formulas can be exploited for a variety of optimizations in the SAT checking procedure. These optimizations were implemented on top of CMU's `BMC` [4][1] and the SAT checker `Grasp` [11, 12], without making use of features that are unique to either one of them.

[1] We distinguish between the tool `BMC` and the method BMC.

We benchmarked the various optimizations, and also compared them to results achieved by `RuleBase`, IBM's BDD-based Model Checker [1, 2]. `RuleBase` is considered one of the strongest verification tools on the market, and includes most of the reductions and BDD optimizations that have been published in recent years. The benchmark's database included 13 randomly selected 'real-life' designs from IBM's internal benchmark set. Instances trivially solved by `RuleBase` are typically not included in this set, a fact which clearly creates a statistical bias in the results. Thus, although we will show that in 10 out of the 13 cases the improved SAT procedure outperformed `RuleBase`, we can not conclude from this that in general it is a better method. However, we can conclude that many of the (BDD-based model checking) hard problems can easily be solved by the improved SAT procedure. A practical conclusion is therefore that the best strategy would be to run several engines in parallel, and then present the user with the fastest result.

Our results are compatible with [5] in the sense that their experiment also showed a clear advantage of SAT when k is small, and when the design has specific characteristics that make BDDs inefficient. We found it hard to predict which design can easily be solved by BMC, because the results are not strictly monotonic in k or the size of the design. We have one design that could not be solved with BMC although there was a known bug in cycle 14, and another design which was trivially solved, although it included a bug only in cycle 38. The SAT instance corresponding to the second design was 5 times larger than the first one, in terms of number of variables and clauses. We also found that increasing k in a *given design* can speed up the search. This can be explained, perhaps, by the fact that increasing k can cause an increase in the ratio of satisfying to unsatisfying assignments.

The rest of this paper is organized as follows: in the next two sections we describe in more detail the theory and practice of BMC and SAT. In Section 4 we describe various BMC-specific optimizations that we applied to the SAT procedure. In sections 5 and 6 we list our experimental results, and our conclusions from them.

2 BMC - the tool and the generated formulas

The general structure of an $\mathbf{AG}p$ formula, as generated in BMC, is the following:

$$\varphi: \quad I_0 \wedge \bigwedge_{i=0}^{k-1} \rho(i, i+1) \wedge (\bigvee_{i=0}^{k} \sim P_i) \tag{1}$$

where I_0 is the initial state, $\rho(i, i+1)$ is the transition between cycles i and $i+1$, and P_i is the property in cycle i. Thus, this formula can be satisfied iff for some i ($i \leq k$) there exists a reachable state in cycle i which contradicts the property P_i. Focusing on potential bugs in a specific cycle can be formulated by simply restricting the disjunction over P_i to the appropriate cycle. `BMC` takes an SMV - compatible model and generates a propositional SAT instance according

to Equation (1). The size of the generated formula is linear in k, and indeed empirical results show that k strongly affects the performance. As a second step, `BMC` transforms the formula to CNF. To avoid the potential exponential growth of the formula associated with this translation, it adds auxiliary variables, and performs various optimizations.

Every ACTL* formula (the subset of CTL* that contain only universal path quantifiers) can be reduced to a SAT instance, under bounded semantics [4]. While all safety properties can be expressed in the form of $\mathbf{AG}p$ [3], to handle temporal operators such as $\mathbf{AF}p$, `BMC` adds to φ the disjunction $\bigvee_{i=0..k-1} \rho(k,i)$, thus capturing the possibility of a loop in the state transition graph. Fairness is handled by changing the loop condition to include at least one state which preserves the fairness condition.

3 SAT checkers and `Grasp`

In this section we briefly outline the principles followed by modern propositional SAT-checkers, and in particular those that `Grasp` (Generic seaRch Algorithm for the Satisfiability Problem) is based on. Our description follows closely the one in [11].

Most of the modern SAT-checkers are variations of the well known Davis-Putnam procedure [7]. The procedure is based on a backtracking search algorithm that, at each node in the search tree, chooses an *assignment* (i.e. both a variable and a Boolean value, which determines the next subtree to be traversed) and prunes subsequent searches by iteratively applying the unit clause rule. Iterated application of the unit clause rule is commonly referred to as Boolean Constraint Propagation (BCP). 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 latter case implies that the formula is unsatisfiable.

A more generic description of a SAT algorithm was introduced in [11]. A simplified version of this algorithm is shown in Fig. 1.

At each *decision level* d in the search, a variable assignment $V_d = \{T, F\}$ is selected with the `Decide()` function. If all the variables are already decided (indicated by `ALL-DECIDED`), it implies that a satisfying assignment has been found, and `SAT` returns `SATISFIABLE`. Otherwise, the *implied assignments* are identified with the `Deduce()` function, which in most cases corresponds to a straightforward BCP. If this process terminates with no conflict, the procedure is called recursively with a higher decision level. Otherwise, `Diagnose()` analyzes the conflict and decides on the next step. First, it identifies those assignments that led to the conflict. Then it checks if the assignment to V_d is one of them. If the answer is yes, it implies that the value assigned to V_d should be swapped and the deduction process in line l_3 is repeated. If the swapped assignment also fails, it means that V_d is not responsible for the conflict. In this case `Diagnose()` will indicate that the procedure should `BACK-TRACK` to a lower decision level β (β is a global variable that can only be changed by `Diagnose()`). The procedure

```
// Input arg: Current decision level d
// Return value:
//   SAT():     {SATISFIABLE, UNSATISFIABLE}
//   Decide():  {DECISION, ALL-DECIDED}
//   Deduce():  {OK, CONFLICT}
//   Diagnose():{SWAP, BACK-TRACK}

SAT (d)
   {
l1:    if (Decide (d) == ALL-DECIDED) return SATISFIABLE;
l2:    while (TRUE) {
l3:       if (Deduce(d) != CONFLICT) {
l4:          if (SAT (d + 1) == SATISFIABLE) return SATISFIABLE;
l5:          else if (β < d || d == 0) // β is calculated in Diagnose()
l6:             { Erase (d); return UNSATISFIABLE; }
          }
l7:       if (Diagnose (d) == BACK-TRACK) return UNSATISFIABLE;
       }
   }
```

Fig. 1. Generic backtrack search SAT algorithm

will then backtrack $d-\beta$ times, each time `Erase()`-ing the current decision and its implied assignments, in line l_6.

Different SAT procedures can be modeled by this generic algorithm. For example, the Davis-Putnam procedure can be emulated with the above algorithm by implementing BCP and the pure literal rule in `deduce()`, and implementing chronological backtracking (i.e. $\beta = d-1$) in `diagnose()`. Modern SAT checkers include *Non-chronological Backtracking* search strategies (i.e. $\beta = d - j, j \geq 1$). Hence, irrelevant assignments can be skipped over during the search. The analysis of conflicts can also be used for adding new constraints (called *conflict clauses*) on the search. These constraints prevent the repetition of assignments that lead to conflicts. This way the search procedure backtracks immediately if a 'bad' assignment is repeated. For example, if `Diagnose()` concludes that the assignment $x = T, y = F, z = F$ inevitably leads to a conflict, it adds the conflict clause $\pi = (\sim x \vee y \vee z)$ to φ.

From the large number of `decide()` strategies suggested over the years, experiments with `Grasp` have demonstrated that the Dynamic Largest Individual Sum (DLIS) has the best average results [10]. DLIS is a rather straightforward strategy: it chooses an assignment that leads to the largest number of satisfied clauses. In this research we only experimented with DLIS, although different problem domains may be most efficiently solved with different strategies.

4 Satisfiability checking of BMC formulas

In this section we describe various BMC-specific optimizations that have been implemented on top of `Grasp`. Many of the optimizations deal with familiar issues that are typically associated with BDDs: variable ordering, direction of traversal (backward Vs. forward), first subtree to traverse, etc.

4.1 Constraints replication

The almost symmetric structure of Equation (1) can be used for pruning the search tree when verifying $\mathbf{AG}p$ formulas. In the following discussion let us first ignore I_0, and assume that φ is fully symmetric.

Conflict clauses, as explained in Section 3, are used for pruning the search tree by disallowing a conflicting sequence (i.e. an assignment that leads to an unsatisfied clause) to be assigned more than once. We will use the alleged symmetry in order to add *replicated clauses*, which are new clauses that are symmetric to the original conflict clause. Each of these clauses can be seen as a *constraint* on the state-space which, on the one hand preserves the satisfiability of the formula and, on the other hand, prunes the search tree.

Let us illustrate this concept by an example. Suppose that `deduce()` concluded that the assignment $x_4 = T, y_7 = F, z_5 = F$ always leads to a conflict (the subscript number in our notation is the cycle index that the variable refers to). In this case it will add the conflict clause $\pi = (\sim x_4 \vee y_7 \vee z_5)$ to φ. We claim that the symmetry of Equation (1) implies that, for example, the assignment $x_3 = T, y_6 = F, z_4 = F$ will also lead to a conflict, and we can therefore add the *replicated clause* $\pi = (\sim x_3 \vee y_6 \vee z_4)$ to φ. Let us now generalize this analysis. Let δ be the difference between the largest and lowest index of the variables in π (in our case $\delta = 7 - 4 = 3$). For all $0 \leq i \leq k - \delta$, the assignment $x_i = T, y_{i+3} = F, z_{i+1} = F$ will also result in a conflict and we can therefore add the replicated clause $m_i = (\sim x_i \vee y_{i+3} \vee z_{i+1})$.

Yet, φ is not fully symmetric. φ is not fully symmetric because of I_0 and because of the Bounded Cone of Influence reduction [5] [2]. The BCOI reduction eliminates variables that are not affecting the property up to cycle k. It can eliminate, for example, x_i for $k - 3 \leq i \leq k$ and y_j for $k - 5 \leq j \leq k$. Consequently cycle $k - 5$ will not be symmetric anymore to cycle $k - 3$ in φ. Typically variables are eliminated only from the right hand side, i.e., if a variable x_k is not eliminated, than for all $i < k$, x_i is also not eliminated. In the following discussion we concentrate on this typical case. Minor adjustments are needed for the general case.

There are two options to handle the asymmetry caused by the BCOI reduction. One option is to restrict the replicated clauses to $0 < i < k - \delta - \Delta$, where

[2] This is in addition to several other manipulations that `BMC` performs on φ which are easy to overcome, and will not be listed here.

Δ is the number of cycles affected by the BCOI reduction. Another option is to add replicated clauses as long as all their variables are contained in the BCOI.

The second option can be formalized as follows. Let C be the set of variables in the conflict clause. For a variable $\sigma \in C$, denote by $k_\sigma \leq k$ the highest index s.t. σ_{k_σ} is a variable in φ (without the BCOI reduction, $k_\sigma = k$ for all variables) and by i_σ the index of σ in C. Also, let $min_C = min\{i_\sigma\}$ and $\psi = min\{(k_\sigma - i_\sigma)\}$ for all $\sigma \in C$. Intuitively, ψ is the maximum number of clauses we can add to the 'right' (i.e. with a higher index) of the conflict clause. We now add replicated clauses s.t. the variable σ for which $i_\sigma = \min_c$ ranges from 0 to $min_C + \psi$.

Example 1. For the conflict clause $\pi = (\sim x_4 \vee y_7 \vee z_5)$, we have $C = \{x_4, y_7, z_5\}$ and $min_C = 4$. Suppose that $k_x = 5, k_y = 10$ and $k_z = 7$. Also, suppose that $k = 10$ and $\Delta = 5$ (since $k_x = 5$, Δ has to be greater or equal to $(10 - 5) = 5$). According to the first option, x will range from 0 to $(10 - 5 - (7 - 4)) = 2$. Thus, the replicated clauses will be $(\sim x_0 \vee y_3 \vee z_1)...(\sim x_2 \vee y_5 \vee z_3)$. According to the second option, we calculate $\psi = min((5 - 4), (10 - 7), (7 - 5)) = 1$, and therefore x will range from 0 to $(4 + 1) = 5$. Thus, this time the right most clause will be $(\sim x_5 \vee y_8 \vee z_6)$. □

Example 1 demonstrates that the second option allows for more replicated clauses to be added, and is therefore preferable.

The influence of I_0 is not bounded, and can propagate up to cycle k. Therefore a simple restriction on the replicated clauses is insufficient. A somewhat 'brute-force' solution is to simulate an assignment for every potential replicated clause, (i.e. assign values that satisfy the complement of m_i) and check if it leads to a conflict. The overhead of this option is rather small, since it only requires to assign $|m_i|$ variables and then `deduce()` once. If this results in a conflict, we can add m_i to the formula. However, the addition of wrong clauses can only lead to false positives, and therefore we can skip the simulation and refer to constraint replication as an under approximation method (this also implies that for the purpose of faster falsification, many other under approximation heuristics can be implemented by adding clauses to φ). Hence, we can first skip the simulation, and only if the formula is unsatisfiable, run it again with simulation.

The overhead of adding and simulating the replicated clauses is small in comparison to its benefit. In all the test cases we examined, as will be shown in Section 5, the replicated clauses accelerated the search, although not dramatically.

4.2 Static ordering

The variable ordering followed by dynamic `decide()` procedures (such as the previously mentioned DLIS strategy) is constructed according to various 'greedy' criteria, which do not utilize our knowledge of φ's structure. A typical scenario when using these procedures, in the context of BMC formulas, is that large sets of clauses associated with distant cycles are being satisfied independently, until they 'collide', i.e. it is discovered that the assignments that satisfied them contradict each other. Fig. 2 demonstrates this scenario, by showing two distant sets of

assigned variables (around the 5^{th} and 20^{th} cycles), that grow independently until at some point they collide. Similarly, they can collide with the constraints imposed by the initial state I_0 or the negation of the property in cycle k. To resolve this conflict, it may be necessary to go back hundreds of variables up the decision tree. We claim that this phenomena can potentially be avoided by guiding the search according to the (k-unfolding of the) Variable Dependency Graph (VDG). This way conflicts will be resolved on a more 'local' level, and consequently less time will be wasted in backtracking.

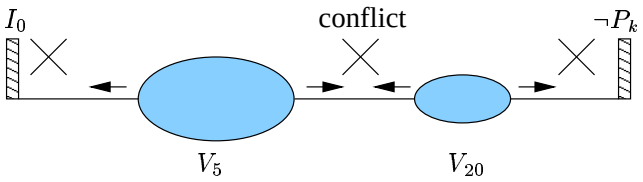

Fig. 2. With default dynamic ordering strategies, it is common that distant sets of variables are assigned values independently. We refer the reader to a technical report [9], where we show snapshots of the number of variables from each cycle that are assigned a value at a given moment. These charts prove that this phenomena indeed occur when using these strategies.

The most natural way to implement such a strategy is to predetermine a *static* order, following either a forward or a backward Breadth - First Search (BFS) on VDG. Indeed, our experiments have shown that in most cases this strategy speeds up the search.

Ordering strategies. We now investigate variations of the BFS strategy. Let us first assume that we are looking for a counter example in a particular cycle k. In this case a strict backward traversal may spend a significant amount of time in paths which include unreachable states. This fact will be revealed only when the search reaches $\overline{I_0}$ (we denote the set of variables in a sub-formula ψ by $\overline{\psi}$), which is placed last in the suggested order. Enforcing a static *forward* traversal, on the other hand, may result in a prolonged search through legal paths (i.e. paths that preserve the property), that will be revealed only when $\overline{P_k}$ is decided (these are the two 'walls' in Fig. 2). A similar dilemma is associated with BDD-based techniques (see for example [6] and [8]). It seems that the (unknown) ratio between the number of paths that go through unreachable states and the number of legal paths is crucial for determining the most efficient direction of traversal in both methodologies.

The strict backward or forward BFS causes the constraints, either on the first or the k-th cycle, to be considered only in a very 'deep' decision level, and the number of backtracks will consequently be very high, sometimes higher than the default dynamic strategies. Another problem with straight BFS results from

the very large number of variables in each cycle. Typically there are hundreds or even thousands of variables in each cycle. It creates a large gap between each variable and its immediate neighbors in VDG, and therefore conflicts are not resolved as locally as we would like to.

These two observations indicate that the straightforward BFS solution should be altered. On the one hand, we should keep a small distance between $\overline{P_0}$ and $\overline{P_k}$, and on the other hand we should follow VDG as close as possible. This strategy can be achieved, for example, by triggering the BFS with a set S of small number of variables from each cycle. As a minimum, it has to include $\overline{P_k}$ (otherwise not all the variables will be covered by the search). Different strategies can be applied for choosing the variables from the other cycles. For example, we can choose $\overline{P_i}$ for all i.[3]

When we generalize our analysis and assume that we are looking for a counter example in the range $0..k$, the set $S := \bigcup_{0 \leq i \leq k} \overline{P_i}$ is the smallest initial set which enables the BFS procedure to cover the full set of variables in a single path. Initial sets smaller than S will require more than one path. This will split the set of variables of each cycle into a larger number of small sets, and consequently create a big gap between them (i.e. between each node and its siblings on the graph). If two such distant siblings are assigned values which together contradict their parent node, then the backtrack 'jump' will be large. Increasing S, on the other hand, will create a large gap between neighboring variables on VDG (i.e. between a node and its sons on the graph). This tradeoff indicates that a single optimal heuristic for all designs probably does not exist, and that only experiments can help us to fine-tune S.

There are, of course, numerous other possible ordering strategies. Like BDDs, on the one hand it has a crucial influence on the procedure efficiency, and on the other hand, an ordering heuristic which is optimal for all designs is hard to find.

Unsatisfiable instances. A major consideration in designing SAT solvers, is their efficiency in solving unsatisfiable instances. Although the various optimizations (e.g. conflicting clauses, non-chronological backtracking) are helpful in these cases as much as they are with satisfiable instances, while satisfiable instances can be solved fast by a good 'guess' of assignments, an instance can be proven to be unsatisfiable only after an exhaustive exploration of the state-space.

We now show that the order imposed by the previously suggested backward BFS is particularly good for unsatisfiable BMC-formulas. In the following discussion we denote φ's sub-formulas $\bigvee_{i=0..k} \sim P_i$ and $\bigwedge_{i=0}^{k-1} \rho(i, i+1)$ by P and ρ respectively.

Let us assume that the property holds up to cycle k, and consequently φ is unsatisfiable. Since the transition relation ρ is consistent [4], a contradiction in φ will not be found before the first variables from $\overline{P}$ are decided. Yet, since

[3] This is not always possible because for $i < k$, $\overline{P_i}$ might be removed by the BCOI reduction.

[4] Inconsistent transition relations can occur, but typically can also be trivially detected.

typically $|\overline{P}| \ll |\overline{\rho}|$, it is possible that the search will backtrack on ρ's variables for a very long time before it reaches $\overline{P}$. Thus, by forcing the search to begin with $\overline{P}$, we may be able to avoid this scenario. However, starting from $\overline{P}$ is not necessarily enough, because this way we only shift the problem to the variables that define $\overline{P}$. It is clear that a BFS backwards on the dependency graph, from the property variables to the initial state is a generalization of this idea and should therefore speed up the proof of unsatisfiability.

4.3 Choosing the next branch in the search tree

The proposed static ordering does not specify the Boolean value given to each variable. This is in contrast to the dynamic approach where this decision is implicit. Here are four heuristics that we examined:

1. *Dynamic decision.* The value is chosen according to one of the dynamic `Decide()` strategies, which are originally meant for deciding both on the variable and its value. For example, the DLIS strategy chooses the value that satisfies the largest number of clauses.
2. *Constant, or random decision.* The most primitive decision strategy is to constantly assign either '0' or '1' to the chosen variable, or alternatively, to choose this value randomly. As several experiments have shown in the past [10], choosing a random or a constant value is not apriori inferior to dynamic decision strategies as one might expect. Any dynamic decision strategy can lead to the 'wrong side of the tree', i.e. can cause the search to focus on an unsatisfiable sub-tree. Apparently constant or random decisions in many cases avoid this path and consequently speed up the search.
3. *Searching for a flat counter example.* Analysis of bugs in real designs, leads to the observation that most of them can be reached by computations which are mostly 'flat', i.e. computations where the frequency in which the majority of the variables swap their values is low. This phenomenon can be exploited when 'guessing' the next subtree to be traversed. Suppose that the `Decide()` function chose to assign a variable x_i for some $0 \leq i \leq k$. Let x_l and x_r be the left and right closest neighboring variables of x_i that are already assigned a value at this point (if no such variable exists, we will say that x_l, or x_r, is equal to $\perp$). To construct a flat counter example, if $x_l = x_r$ we will assign x_i their common value. The following simple procedure generalizes this principle:

   $l =$ `largest number s.t.` $l < i$ `and` x_l `is assigned.`
   $r =$ `smallest number s.t.` $r > i$ `and` x_r `is assigned.`
   `if` $x_l \neq \perp$
   `if` $(x_l = x_r \ \| \ x_r = \perp)$ `return` x_l`; else return` $\{T, F\}$`;`
   `else`
   `if` $(x_r \neq \perp)$ `return` x_r`; else return` $\{T, F\}$`;`

   The non-deterministic choice can be replaced by one of the heuristics that were suggested above (e.g. dynamic, constant).

4. *Repeating previous assignments.* When the search engine backtracks from decision level d to β, all the assignments that are either `decide()`d or `deduce()`d between these two levels are undone by `erase()`. We claim that repeating previous assignments can reduce the number of backtracks. This is because we know that all assignments between levels $\beta + 1$ and d do not contradict one another nor do they contradict the assignments with decision level lower than β (otherwise the procedure would backtrack before level d). In order to decide on each variable's value for the first time, this strategy should be combined with one of the strategies that were described before.

4.4 A combined dynamic and static variable ordering

The static ordering can be combined in various ways with the more traditional dynamic procedures. We have implemented two such strategies:

1. *Two phase ordering.* The static traversal is used for the first max_s variables, and then the variables are `Decide()`-d dynamically.
2. *Sliding window.* Variables are chosen dynamically from a small set of variables, corresponding to a 'window' which progresses along the static order that we chose. Let $V : v_1..v_n$ be the static variable ordering, and let $V' : v'_1..v'_k$ be the (ordered) subset of V's variables that are currently not assigned a value. Let $1 \leq w \leq k$ be an arbitrary number denoting the *window* size. In each step, a variable is dynamically chosen from the set of variables that are within the borders of the window $[v'_1 - v'_w]$. Note that the two extreme ends of w, namely $w = 1$ and $w = k$, correspond to the pure static and dynamic orderings, respectively.

4.5 Restricting `Decide()` to dominating variables

While φ typically contains tens of thousands of variables, not more than 10%-20% of them are the actual model's variables. The other 80% are auxiliary variables that were added to φ in order to generate a compact CNF formula. It is clear that the model's variables are sufficient for deciding the satisfiability of the formula, and therefore it should be enough to `decide()` only them (however, if the formula has more than one satisfying assignment, some of the auxiliary variables should be assigned too). The same argument can be applied to a much smaller set of variables: the inputs. The input variables are typically less than 5% of the total number of variables, and can determine alone the satisfiability of the formula[5]. Thus, if we restrict `Decide()` to one of these small sets, we potentially reduce the depth of the decision tree, on the expense of more `deduce()` operations.

[5] Here we assume that all non-deterministic assignments are replaced by conditional assignments, where the 'guard' of the condition is a new input variable.

5 Experimental results

The Benchmark included 13 designs, on a 'one property per design' base. The properties were proven in the past to be false, and the cycle in which they fail was known as well. Thus, the Benchmark focuses on a narrow view of the problem: the time it takes to find a bug in cycle k, when k is pre-known. The iterative process of finding k is, of course, time consuming, which might be more significant than any small time gap between BMC and regular model checking.

The results presented in Fig. 3 summarize some of the more interesting configurations which we experimented with. In Fig. 4 we present more information regarding the SAT instance of each case study (the no. of variables and clauses) as well as some other `Grasp` configurations which were generally less successful. The right-most column in this figure includes the time it takes to prove that there is no bug up to cycle $k-1$, with the **SM** configuration. These figures are important for evaluating the potential performance differences between satisfiable and unsatisfiable instances.

We present results achieved by `RuleBase` under two different configurations. **RB1** is the default configuration, with dynamic reordering. **RB2** is the same configuration without reordering, but the initial order is taken from the order that was calculated with **RB1**. These two configurations represent a typical scenario of Model-Checking with `RuleBase`. Each time reordering is activated, the initial order is potentially improved and saved in a special order file for future runs. Thus, **RB2** results can be further improved.

`RuleBase` results are compared with various configurations of `Grasp`, where the first one is simply the default configuration without any of the suggested optimizations.

The following table summarizes the various configurations, where the left part refers to Fig. 3 and the right part to Fig. 4:

	`Grasp`	**+R**	**+SM**	**+SMF**	**+SMR**	**+SMP**	**+SMD**	**+W**$_i$
Ordering:	*Dyn*	*Dyn*	*Stat*	*Stat*	*Stat*	*Stat*	*Stat*	*Win i*
Value:	*Dyn*	*Dyn*	1	*Flat*	1	*Prev*	*Dyn*	*Dyn*
Variable set:	*All*	*All*	*Model*	*Model*	*Model*	*Model*	*Model*	*Model*
Replication:	*No*	*Yes*	*No*	*No*	*Yes*	*No*	*No*	*No*

The *Stat* ordering refers to the static order suggested in Section 4.2, whereas *Dyn* is the default dynamic decision strategy adopted by `Grasp` (DLIS). The *Win i* refers to a combined dynamic and static ordering, where variables within a window of size i are selected dynamically, as explained in Section 4.4. The '1', *Flat* and *Prev* default values refer to the constant, flat and previous values suggested in Section 4.3 (in **+SMP** we combined the *Prev* strategy with the default value '1'). The *Model* variable set refers to a restriction on `decide()` to model variables only, as described in Section 4.5. The Replication refers to constraint replication and simulation, as explained in Section 4.1. All configurations include the flag '+g60', which restricts the size of the conflict clauses (and consequently

also the size of the replicated clauses) to 60 literals. Other than very few cases, all the other possible configurations did not perform better than those that are presented.

The test cases in the figure below are separated into two sets: the 10 designs in the first set demonstrate better results for the optimized SAT procedure, and the 3 designs in the second set demonstrate an advantage to the BDD-based procedure. Both sets are sorted according to the **RB1** results.

Design #	K	**RB1**	**RB2**	`Grasp`	**+R**	**+SM**	**+SMF**	**+SMP**	**+SMR**
1	18	7	6	282	115	**3**	57	29	4.1
2	5	70	8	1.1	1.1	0.8	1.1	**0.7**	0.9
3	14	597	375	76	52	**3**	2069	**3**	**3**
4	24	690	261	510	225	**12**	27	**12**	**12**
5	12	803	184	24	24	**2**	**2**	**2**	3
6	22	*	356	*	*	18	**16**	38	18
7	9	*	2671	10	10	2	**1.8**	1.9	2
8	35	*	*	6317	2870	**20**	338	101	74
9	38	*	*	9035	*	**25**	277	126	96
10	31	*	*	*	9910	312	**22**	64	330
11	32	152	**60**	*	*	*	*	*	*
12	31	1419	**1126**	*	*	*	*	*	*
13	14	*	**3626**	*	*	*	*	*	*

Fig. 3. Results table (Sec.). Best results are bold-faced. Asterisks (*) represent run times exceeding 10,000 sec.

Remarks for Figures 3 and 4

1. The time required by `BMC` to generate the formula is not included in the results. `BMC` generates the formula typically in one or two minutes for the large models, and several seconds for the small ones. While generating the formula, the improved `BMC` generate several files which are needed for performing the various optimizations.
2. `RuleBase` supports multiple engines. The presented results were achieved by the 'classic' SMV-based engine. Yet, a new BDD-based engine that was recently added to `RuleBase` (January 2000), performs significantly better on some of these designs. This engine is based on sophisticated under and over approximation methods that were not yet published.
3. When comparing `RuleBase` results to BMC results, one should remember that the former has undergone years of development and optimizations, which the latter did not yet enjoy. The various optimizations that have been presented in this paper can be further improved and tuned. Various combinations of the dynamic and static orderings are possible, and it is expected that more industrial experience will help in fine tuning them. The

Design #	**vars**	**clauses**	**+SM**	**+SMD**	**+W$_{50}$**	**+W$_{100}$**	**+W$_{200}$**	**+SM** $(k-1)$
1	9685	55870	3	36	46	46	51	20
2	3628	14468	0.8	0.7	0.7	0.7	0.8	0.4
3	14930	72106	3	1216	8	3	17	934
4	28161	139716	12	26	31	42	61	26
5	9396	41207	2	3	2	3	3	1
6	51654	368367	18	243	111	418	950	28
7	8710	39774	2	1.8	2.5	1.9	2.8	1.3
8	58074	294821	20	123	163	86	105	30
9	63624	326999	25	136	164	153	181	230
10	61088	334861	312	125	70	107	223	1061
11	32109	150027	*	*	*	*	*	*
12	39598	19477	*	*	*	*	*	*
13	13215	6572	*	*	*	*	*	*

Fig. 4. Other, less successful configurations

implementation of the SAT checker `Grasp` can also be much improved even without changing the search strategy. It was observed by [10] that an efficient implementation can be more significant than the decision strategy.[6]

6 Conclusions

1. Neither BDD techniques nor SAT techniques are dominant. Yet, in most (10 out of 13) cases the optimized SAT procedure performs significantly better. As was stated before, only significant differences in performance are meaningful, because normally k is not pre-known. Such differences exist in 8 of the 10 cases.
2. The **SM**, **SMP** and **SMR** strategies are better in all cases compared to the default procedure adopted by `Grasp`. The **SM** strategy seems to be the best one.
3. The static ordering apparently has a stronger impact on the results than the strategy for choosing the next subtree. This can be explained by the fact that wrong choices of values are corrected 'locally' when the variable ordering follows the dependency graph, as was explained before. Surprisingly, the constant decision 'TRUE', which is the most primitive strategy, proved to be the most efficient (in another experiment we tried to solve design #10, which is the only one that is solved significantly better by other configurations, with a constant decision 'FALSE'. It was solved in about 3 seconds, faster than all other configurations). The 'flat' decision strategy performed better only in three cases. The 'Prev' decision was better than 'flat' in 6 designs, but only once better than the simple constant decision. Yet, it seems to be

[6] In [4], `SATO` [13] was used rather than `Grasp`. Although in some cases it is faster than `Grasp`, it is restricted in the number of variables it can handle, and seems to be less stable.

more stable in achieving fast results than both of them. As for the sliding window strategy, Fig. 4 shows that in most cases increasing the window size only slows down the search. The surprising success of the constant decision strategy can perhaps be attributed to its zero overhead. It can also indicate that most bugs in hardware designs can be revealed when the majority of the signals are 'on'. Only further experiments can clarify if this is a general pattern or an attribute of the specific designs that were examined in the benchmark.

4. Constraint replication (+simulation) requires a small overhead, which does not seem to be worthwhile when used in combination with static ordering. Yet, it speeds up the standard search based on dynamic ordering.
   This can be explained by the inherent difference between dynamic and static orderings: suppose that the assignment $x_1 = T$ and $y_{20} = F$ leads to a conflict, and suppose that their associated decision levels were 10 and 110 respectively when the conflict clause $(\neg x_1 \vee y_{20})$ was added to φ. In static ordering, the decision level for each variable remains constant. As a result, even if the search backtracks to a decision level lower than 10, the conflict clause will not be effective until the search once again arrives at decision level 110. In dynamic ordering, on the other hand, there is a chance that these two variables will be decided much closer to each other, and therefore the clause will prune the search tree earlier.
   Another reason for the difference is related to the typical sizes of backtracking in each of the methods. Since conflicts are resolved on a more 'local' level in the **SM** strategy, conflict clauses (either the original ones or the replicated clauses) are made of variables which are relatively close to each other in terms of their associated decision level. Therefore the non-chronological backtracking 'jump' caused by these clauses is relatively small.
5. Both SAT methods and BDD based methods do not have a single dominant configuration. BDDs can run with or without reordering, with or without conjunctive partitioning, etc. As for SAT methods, all the optimizations described in Section 4 can be activated separately, and indeed, as the results table demonstrate, different designs are solved better with different configurations. Given this state of affairs, the most efficient solution, as was mentioned in the introduction, would be to run several engines in parallel and present the user with the fastest solution. This architecture will not only enable the users to run SAT and BDD based tools in parallel, but also to run these tools under different configurations in the same time, which will obviously speed up the process of model checking.

Acknowledgments I would like to thank Armin Biere and Joao Marques-Silva for making `BMC` and `Grasp` publicly available, respectively, and for their most helpful assistance in figuring them out.

References

1. I. Beer, S. Ben-David, C. Eisner, D. Geist, L. Gluhovsky, T. Heyman, A. Landver,

P. Paanah, Y. Rodeh, G. Ronin, and Y. Wolfsthal. RuleBase: Model checking at IBM. In Orna Grumberg, editor, *Proc.* 9^{th} *Intl. Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lect. Notes in Comp. Sci.*, pages 480–483. Springer-Verlag, 1997.

2. I. Beer, S. Ben-David, C. Eisner, and A. Landver. RuleBase: An industry oriented formal verification tool. In *Proc. Design Automation Conference 96 (DAC96)*, 1996.
3. I. Beer, S. Ben-David, and A. Landver. On-the-fly model checking of RCTL formulas. In A.J. Hu and M.Y. Vardi, editors, *Proc.* 10^{th} *Intl. Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *Lect. Notes in Comp. Sci.*, pages 184–194. Springer-Verlag, 1998.
4. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS99)*, Lect. Notes in Comp. Sci. Springer-Verlag, 1999.
5. A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a power pc^{TM} microprocessor using symbolic model checking without bdds. In N. Halbwachs and D. Peled, editors, *Proc.* 11*st Intl. Conference on Computer Aided Verification (CAV'99)*, Lect. Notes in Comp. Sci. Springer-Verlag, 1999.
6. Wililiam Chan, Richard Anderson, Paul Beame, and David Notkin. Improving efficiency of symbolic model checking for state-based system requirements. In *International Symposium on Software Testing and Analysis (ISSTA98)*, Lect. Notes in Comp. Sci. Springer-Verlag, 1998.
7. M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7:201–215, 1960.
8. H. Iwashita, T. Nakata, and F. Hirose. Ctl model checking based on forward state traversal. In *IEEE/ACM International conference on Computer Aided Design*, pages 82–87, November 1996.
9. Ofer Shtrichman. Tuning sat checkers for bounded model checking – experiments with static and dynamic orderings. Technical report, Weizmann Institute, 2000. can be downloaded from www.weizmann.ac.il/∼ofers.
10. J.P.M Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *9th Portuguese Conference on Artificial Intelligence (EPIA)*, 1999.
11. J.P.M Silva and K. A. Sakallah. GRASP - a new search algorithm for satisfyability. Technical Report TR-CSE-292996, Univerisity of Michigen, 1996.
12. J.P.M Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48:506–516, 1999.
13. H. Zhang. SATO: An efficient propositional prover. In *International Conference on Automated Deduction (CADE-97)*, 1997.