Refining the SAT Decision Ordering for Bounded Model Checking [*] [†]

Chao Wang HoonSang Jin Gary D. Hachtel Fabio Somenzi

Department of Electrical and Computer Engineering
University of Colorado at Boulder, CO 80309-0425

Abstract

Bounded Model Checking (BMC) relies on solving a sequence of highly correlated Boolean satisfiability (SAT) problems, each of which checks for the existence of counter-examples of a bounded length. The performance of SAT search depends heavily on the variable decision ordering. We propose an algorithm to exploit the correlation among different SAT problems in BMC, by predicting and successively refining a partial variable ordering. This ordering is based on the analysis of all previous unsatisfiable instances, and is combined with the SAT solver's existing decision heuristic to determine the final variable decision ordering. Experiments on real designs from industry show that our new method improves the performance of SAT-based BMC significantly.

Categories and Subject Descriptors: B.6.3 [Logic design]: Design aids

General Terms: Verification

Keywords: Bounded Model checking, SAT, decision heuristic

1. Introduction

Bounded Model Checking (BMC [1]), based on Boolean satisfiability (SAT), has been widely accepted as a complement to model checking based on Binary Decision Diagrams (BDDs). In BMC, the existence of counter-examples to a linear time property of a bounded length is encoded in a Boolean formula—the formula is satisfiable if and only if a counter-example exists.

Many modern SAT solvers [15, 12, 7] are based on the Davis-Logemann-Loveland (DLL) search procedure [4]. At each node in the "search tree," a variable is selected and assigned either 0 or 1. The assignment may cause the values of other variables to be implied; these implications are applied iteratively to further prune the subsequent searches. Backtracking occurs whenever there is a conflict (i.e., the subformula becomes unsatisfiable). This procedure terminates whenever a satisfying assignment is found or the entire search tree has been explored.

Like many known search problems, the order in which Boolean variables are assigned (as well as the values assigned) affects the SAT solving performance significantly. In fact, different variable decision orderings imply different search trees, whose sizes and corresponding search times can be quite different. Because of the NP-completeness of the SAT problem, finding the optimal decision ordering is unlikely to be easier, and modern SAT solvers use heuristics to compute decision orderings that are "good enough." For example, Chaff uses the Variable State Independent Decaying Sum (VSIDS) heuristic [12]. (Pre-Chaff decision heuristics are surveyed by Silva in [14].)

Most aforementioned SAT solvers were designed to deal with general Boolean formulae. Useful information that is unique to the SAT problems encountered during BMC is lost during the translation. In particular, the sequence of SAT instances that BMC produces for increasing path length is made up of problems that are highly correlated; this means that information learned from previous SAT problems can help solving the current problem.

Based on this observation, we propose an algorithm to predict a good variable ordering for the current BMC instance. Such a linear ordering is computed by analyzing all previous unsatisfiable instances, and is successively refined as the BMC unrolling depth increases. We also propose two different approaches (static and dynamic) to apply this linear ordering; in both cases, the ordering is combined with the SAT solver's default decision heuristic to determine the final variable decision ordering.

We implemented our new method on top of the BMC package in VIS-2.0 [2] and the SAT solver Chaff [12]. Experimental studies on real-world circuits show that our new method can speed up BMC significantly: It reduced the run time on 32 out of the 37 circuits, and the average improvement in CPU time is 42%.

Related Work: The work most closely related is by Shtrichman [13], who regarded the SAT instance in BMC as a combinational circuit lying on a plane whose x-axis is the "time frames" and whose y-axis is the "registers." By breadth-first search on the Variable Dependency Graph (VDG), Shtrichman sorted the variables according to their positions on the "time axis." In contrast, our new method can be viewed as sorting the variables according to their positions on the other axis—the "register axis."

Ganai et al. proposed a hybrid representation [6] (both circuits and CNF formulae) in their SAT solver, to apply fast implication on the circuit structure and at the same time retain the merit of CNF formulae. Gupta el al. also applied implications learned from the circuit structure to help the SAT search, where the implications were extracted by BDD operations [9]. Lu et al. used circuit topological information and signal correlations to enforce a good decision ordering in their circuit SAT solver [11]. The correlated signals were identified by random simulation, and were applied to the SAT search so that they were most likely to cause conflicts.

The incremental nature of BMC was also exploited by several incremental SAT solvers [17, 5]. However, they primarily focused on incrementally creating a new SAT instance from the previous one, and on reusing previously learned conflict clauses. Refining

[*]This work was supported by SRC contract 2003-TJ-920.

[†]The authors thank Kenneth McMillan for helpful discussions.

DAC 2004, June 7–11, 2004, San Diego, California, USA.

```
while (1) {
    if ( make_decision( ) ) {
        while ( bcp( ) == CONFLICT ) {
            level = conflict_analysis( );
            if (level < 0)     return UNSAT;
            else               back_track(level);
    }}
    else
        return SAT;
}
```

Figure 1: DLL search procedure with backtracking.

the SAT decision ordering, on the other hand, has not been studied. We believe that our new method can be combined with these incremental techniques to further improve their performance.

2. Preliminaries

We represent the model as a 4-tuple $\langle V, W, I, T \rangle$, where V is the set of present-state variables, W is the set of inputs, $I(V)$ is the initial state predicate, and $T(V, W, V')$ is the transition relation. V' is the set of next-state variables.

Model checking a property with a finite-size witness or counter-example can be translated into a series of SAT problems. Take the invariant property $\mathsf{G}\,P$ (predicate P holds in all reachable states) as an example: $\mathsf{G}\,P$ is false on a model if and only if there exist finite-length paths from the initial states to states labeled $\neg P$. The existence of such paths can be formulated as

$$I(V^0) \wedge \bigwedge_{1 \le i \le k} T(V^{i-1}, W^i, V^i) \wedge \neg P(V^k) \ . \qquad (1)$$

Here, V^i is the set of state variables at the i-th time frame. In BMC, the finite length k keeps increasing until either a path is found (the property is proven false) or k exceeds a predetermined *completeness threshold* $\mathbb{N}$ [1] (the property is proven true).

Eq. 1 is usually encoded in Conjunctive Normal Form (CNF). A CNF formula is the conjunction of a set of *clauses*, each of which is a disjunction of *literals*. A *literal* is the positive (or negative) phase of a Boolean *variable*. Selecting a literal and making it *true* is called a *decision*. If a clause has only one free literal and all the other literals are *false*, it is called a *unit clause*. Every *unit clause* triggers an *implication*—its only *free* literal must be *true*. The process of applying the implication iteratively until no unit clause is left is called the *Boolean Constrain Propagation (BCP)*.

The basic DLL procedure for solving SAT problems is given in Fig. 1, which iteratively makes decisions and then applies BCP. A *conflict* may occur under a partial assignment—some clauses become *false* after BCP, indicating a previous decision is not appropriate. The level of that decision is identified by *conflict analysis*, following which, that decision is flipped by backtracking. A clause learned from the conflict is also added to the clause database to prevent the search from repeating the mistake; such a clause is called a *conflict clause*. The given formula is proven unsatisfiable if and only if there is a conflict without any decisions being made.

Whenever a formula is proven unsatisfiable, there exists a final conflict that can not be resolved by backtracking. Such a final conflict (represented by an empty clause) is the unique root node of a resolution subgraph (Fig. 2): the leaves are clauses of the original formula, and the internal nodes are the *conflict clauses* added during SAT search. By traversing this resolution graph backward, we can identify a subset of the original clauses that are responsible for this final conflict. This subset of original clauses, called the *unsatisfiable core* [18, 8], is sufficient to imply unsatisfiability. In Fig. 2, the black squares on the left side form an *unsatisfiable core*.

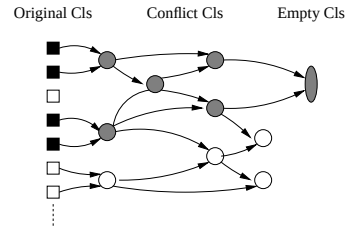

Figure 2: The resolution graph.

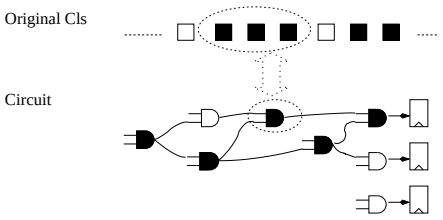

Figure 3: The unsatisfiable core and the abstract model.

3. Our Algorithm

A subset of clauses of the CNF formula identify a subset of registers and logic gates of the circuit, which implicitly define an abstraction of the model. A logic gate is considered to be in the abstract model if some clauses describing its gate relation appear in the unsatisfiable core. In Fig. 3, the black squares represent the unsatisfiable core, and the black gates represent logic gates in the abstract model.

These abstractions are over-approximations, because the element transition relations of the logic gates that are outside the current abstraction are assumed to be tautologies. However, such an abstract model is sufficient to prove that there is no counter-example of the current length k. Had we known the current abstract model by an *oracle*, we could have used it to help the decision-making during the SAT search. The idea is to make decisions only on the variables appearing in the current abstract model, since constraints among them are already sufficient to prove unsatisfiability. By doing so, only the logic relations among these variables will be explored; the other irrelevant variables and clauses will not be ignored.

Although there is no way of knowing the current abstract model before solving the SAT problem, abstract models extracted from previous unsatisfiable BMC instances can be good *estimates*. The idea is to guess a subset of important variables and apply them to solve the current instance (by assigning them first in the SAT search). When the estimation is perfect, no other variable in the formula needs to be assigned before we are able to prove unsatisfiability. Even when there are some discrepancies between the estimation and reality, the size of the search tree is expected to be significantly reduced. In Fig. 4, the shaded area represents the unsatisfiable core from the length-3 BMC instance; variables appearing in it are recorded and given higher priority during the decision-making on the length-4 BMC instance.

In practice, the SAT problems in BMC are usually highly correlated, e.g. their unsatisfiable cores share a large number of clauses. Furthermore, the vast majority of these SAT instances are unsatisfiable: For passing properties, they are always unsatisfiable; for failing properties, all but the last one are unsatisfiable. This means that there is a sufficient number of previous abstract models to compute and refine our estimation.

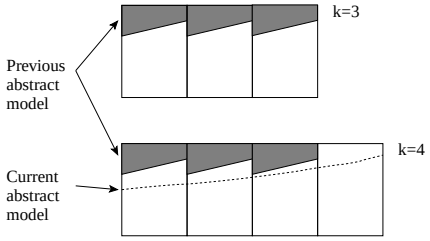

Figure 4: Previous abstractions to help the current BMC instance.

refine_order_bmc (M, P) {
  Initialize the list $varRank$;
  for each $k \in \mathbb{N}$ {
    F = **gen_cnf_formula** (M, P, k);
    $(isSat, unsatVars)$ = **sat_check** $(F, varRank)$;
    if $(isSat)$ return FALSE;
    else **update_ranking** $(unsatVars, varRank)$;
  }
  return TRUE;
}

Figure 5: BMC with the refined decision orderings.

3.1 Identifying Important Variables

To generate the unsatisfiable core, additional bookkeeping is required during the SAT solving process. In particular, for each conflict clause, its complete *conflict graph* must be recorded to memorize all the clauses that are responsible for it. Since some of these clauses may be conflict clauses themselves, at the end, we may have many conflict graphs forming a *Conflict Dependency Graph (CDG)* [3]. Variables appearing in the unsatisfiable core can be easily identified by traversing the CDG from the final conflict backward.

However, modern SAT solvers, like Chaff, periodically remove conflict clauses that are deemed irrelevant (or less relevant) to the current search. Disabling this feature may slow down the search significantly when solving difficult problems. On the other hand, if conflict clauses are allowed to be deleted, the dependency relation in the CDG might be broken, which makes the construction of a complete unsatisfiable core impossible.

In order to generate a complete unsatisfiable core without slowing down the search, we maintain separately a simplified version of the CDG. Our simplification is in the representation of conflict clauses—we only retain the dependency relations and replace each clause by a *pseudo ID*. Compared to the number of literals in the conflict clauses, which is often in the hundreds, the overhead of the pseudo ID (an integer) is small. The simplified CDG retains all the information required for identifying the unsatisfiable core, while the original clause database is left intact. Therefore, the periodic removal of irrelevant conflict clauses is not affected.

In practice, the additional overhead of maintaining and finally traversing our simplified CDG is low: The runtime increases by about 5%, and the memory overhead is usually negligible. Such a price is acceptable for our purpose.

3.2 Refining the Decision Ordering

For each previous unsatisfiable instance, we identify all the variables appearing in its unsatisfiable core. Variables from all the previous unsatisfiable cores are combined to determine a partial variable ordering for the current SAT search (represented by $varRank$). The overall algorithm, given in Fig. 5, accepts two parameters: the model M and the invariant predicate P. The list $varRank$ is used to store the scores. Procedure **gen_cnf_formula** is used to generate the CNF representation of the length-k BMC instance according to Eq. 1. The formula F is then given to **sat_check**, which also takes $varRank$ as a parameter. If F is unsatisfiable, **sat_check** returns all the variables appearing in the unsatisfiable core, which are used to update $varRank$. After we move to the next k, the updated ordering will be applied again. The entire procedure terminates as soon as F becomes satisfiable, or the unrolling depth k exceeds the *completeness threshold*.

Inside **update_ranking**, variables appearing in the unsatisfiable core are assigned non-zero scores; the rest are assigned zero. Let

Table 1: BMC vs. refine order BMC (both static and dynamic).

model	T/F (k)	bmc (s)	new bmc (s) sta.	new bmc (s) dyn.	model	T/F (k)	bmc (s)	new bmc (s) sta.	new bmc (s) dyn.
01_b	F	39	25	24	19_b	F	139	123	108
02_1_b1	(41)	6613	7200	5677	20_b	(28)	3748	5617	3992
02_1_b2	(28)	835	3648	894	21_b	F	93	80	76
02_3_b2	(65)	6944	494	476	22_b	(41)	6164	5134	3986
02_3_b4	(65)	6906	433	475	23_b	(25)	3968	3209	3644
02_3_b6	(59)	6861	352	368	24_1_b1	(22)	6045	748	1182
03_b	F	214	222	238	24_1_b2	(22)	4992	775	1053
04_b	F	85	70	67	24_1_b3	(22)	5075	782	1054
06_b	F	962	589	596	25_b	(90)	7107	3069	2922
11_b_2	(29)	3820	4533	2932	27_b	F	34	27	37
11_b_3	(28)	4160	3102	3515	28_b	F	782	855	683
14_b_1	(35)	201	2272	287	29_b	(22)	4917	5397	4270
14_b_2	F	35	30	35	31_1_b1	(21)	5728	3831	4491
15_b	F	12	13	12	31_1_b2	(21)	5838	2292	3552
16_1_b	(83)	6948	2256	4537	31_1_b3	(21)	4321	1904	3748
17_1_b1	(264)	7161	7114	6965	31_2_b1	(20)	5419	5215	2660
17_1_b2	(12)	29	816	44	31_2_b2	(19)	6924	3180	5475
17_2_b1	(167)	7160	4331	4629					
17_2_b2	(141)	7181	3475	3268	**TOTAL**		138k	86k	79k
18_b	(20)	1172	2999	1049	**RATIO**		100%	62%	57%

$bmc\_score(x)$ denote the score of variable x,

$$bmc\_score(x) = \sum_{1 \le j \le k} \textbf{in\_unsat}(x, j) \cdot j \ ,$$

where **in_unsat**(x, j) returns 1 if and only if x appears in the unsatisfiable core of the j-th BMC instance. Note that all previous unsatisfiable cores are used to determine the current ordering, because (1) we want to give preference to the variables appearing in most recent unsatisfiable cores, which usually have higher correlation to the current one, and (2) we want to avoid relying exclusively on any particular previous unsatisfiable core, which may not always be an accurate estimation of the current one.

3.3 Applying the Decision Ordering

We have applied our refined ordering to the SAT solver Chaff [12]; however, the proposed techniques can be easily adapted to other DLL-based SAT solvers. In Chaff, every literal l is associated with a $chaff\_score(l)$, whose initial value is its literal count in the CNF formula. During the SAT search, the score is updated periodically as follows,

$$chaff\_score(l) = chaff\_score(l)/2 + new\_lit\_counts(l) \ ,$$

where $new\_lit\_counts(l)$ is the number of *new* conflict clauses (since the last update) in which literal l appears. All the literals are sorted by the $chaff\_score()$, and one with largest score is selected and assigned first.

Our pre-computed $bmc\_score()$ can either replace or be combined with $chaff\_score()$ to determine the final ordering. We have experimented with two different ways of combining the two scores: in the *static* approach, the final decision ordering is determined primarily by $bmc\_score()$, with $chaff\_score()$ only as a tiebreaker. It is called static because this configuration is used throughout the

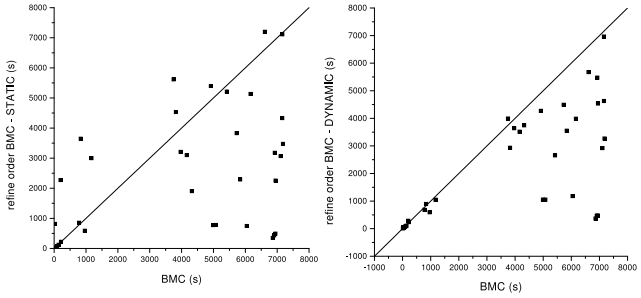

Figure 6: CPU time: BMC vs. refine order BMC.

SAT solving process. In the *dynamic* approach, the ordering is initially based primarily on $bmc\_score()$ with $chaff\_score()$ as a tie-breaker. However, if our estimation is found to be inaccurate, it switches back to the default VSIDS heuristic, where the sorting is based exclusively on $chaff\_score()$. The rationale behind this approach is that the VSIDS heuristic is designed to favor the most recently added conflict clauses, which may eventually dominate in terms of literal counts for difficult problems. Applying the VSIDS heuristic in those cases allows the search process to be driven primarily by these conflict clauses.

Whenever our estimation is not very accurate, or proving the unsatisfiability indeed needs almost all the variables, the SAT problem is considered *difficult*. If the number of decisions required to solve the problem is large, it is a good sign that the problem is difficult. Therefore, in our implementation of the dynamic approach, we switch back to the VSIDS heuristic as soon as the number of decisions is greater than 1/64 of the number of original literals.

4. Experiments

We implemented our algoirthm on top of the BMC package in VIS-2.0 [2, 16] and the SAT solver Chaff [12]. Our experimental studies were conducted on the set of IBM Formal Verification Benchmark circuits [10], with a 400MHz Pentium II with 1GB of RAM running Linux.

Table 1 compares the CPU time of our new method to the standard BMC. The first column is the name of the model. The second column indicates whether the given property is true or false. If the experiments cannot be finished within 2 hours, we compare the CPU times spent to reach the maximum unrolling depth that all methods can complete; in those cases, the maximum unrolling depth is given in parentheses. The following three columns give the CPU time of the standard BMC and our new method with both static and dynamic configurations. Trivial experiments that can be finished by all methods in less than 10 seconds have been excluded.

The average speedup of our algorithm is 38% (static) and 42% (dynamic). Overall, we have achieved performance improvement on 26 (for static) and 32 (for dynamic) out of the 37 circuits. In form of scatter plots, this is shown in Fig. 6: Dots that are under the diagonals represent the winning cases for our new method.

Some detailed information on a particular circuit, *02_3_b_2*, has been extracted. The two plots in Fig. 7 compare the standard BMC to our new method on the *number of decisions* and the *number of implications* at each unrolling depth. Note that smaller values for the number of decisions indicate smaller SAT search trees.

5. Conclusions

We have presented a new method to predict and then successively refine the variable decision ordering for the SAT problems encountered in BMC. Experiments on real designs have shown that our refined decision ordering can significantly speed up BMC. Further experimental analysis has indicated that the performance improvement is due to the reduction of the sizes of the search trees.

Our method exploits the unique characteristic of the SAT problems in BMC—the different SAT problems are highly correlated; therefore, it complements existing decision heuristics of the SAT solvers used for BMC. We believe that our method can be applied also to other SAT-based problems, as long as their subproblems have a similar incremental nature.

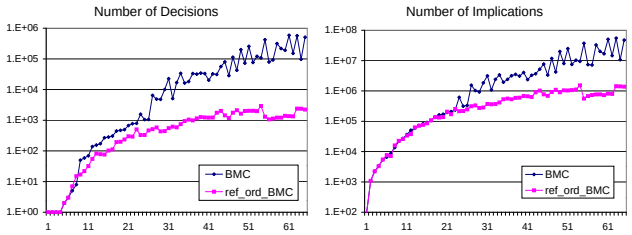

Figure 7: Statistics on *02_3_b_2* (x-axis is the unrolling depth).

References

[1] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Fifth International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, Mar. 1999. LNCS 1579.

[2] R. K. Brayton et al. VIS: A system for verification and synthesis. In *Eighth Conference on Computer Aided Verification*, pages 428–432. July 1996. LNCS 1102.

[3] P. Chauhan, E. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In *Formal Methods in Computer Aided Design*, pages 33–51. Nov. 2002. LNCS 2517.

[4] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[5] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4), 2003. First International Workshop on Bounded Model Checking.

[6] M. K. Ganai, P. Ashar, A. Gupta, L. Zhang, and S. Malik. Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver. In *Proceedings of the Design Automation Conference*, pages 747–750, June 2002.

[7] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Design, Automation and Test in Europe (DATE'02)*, pages 142–149, Mar. 2002.

[8] E. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Design, Automation and Test in Europe (DATE'03)*, pages 886–891, Munich, Germany, Mar. 2003.

[9] A. Gupta, M. Ganai, C. Wang, Z. Yang, and P. Ashar. Learning from BDDs in SAT-based bounded model checking. In *Proceedings of the Design Automation Conference* , pages 824–829, June 2003.

[10] IBM Formal Verification Benchmarks. URL: http:// www.haifa.il.ibm.com /projects/verification/RB_Homepage/ benchmarks.html.

[11] F. Lu, L. Wang, K. Cheng, J. Moondanos, and Z. Hanna. A signal correlation guided ATPG solver and its applications for solving difficult industrial cases. In *Proceedings of the Design Automation Conference*, pages 436–441, June 2003.

[12] M. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, pages 530–535, June 2001.

[13] O. Shtrichman. Tuning sat checkers for bounded model checking. In *Twelfth Conference on Computer Aided Verification* . July 2000. LNCS 1855.

[14] J. P. M. Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence*, Sept. 1999.

[15] J. P. M. Silva and K. A. Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer-Aided Design*, pages 220–227, Nov. 1996.

[16] URL: http://vlsi.colorado.edu/∼vis.

[17] J. Whittemore, J. Kim, and K. Sakallah. SATIRE: A new incremental satisfiability engine. In *Proceedings of the Design Automation Conference*, pages 542–545, June 2001.

[18] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design, Automation and Test in Europe (DATE'03)*, pages 880–885, Mar. 2003.