Refining the SAT Decision Ordering for Bounded Model Checking

Conference Paper · June 2004

DOI: 10.1109/DAC.2004.240441 · Source: DBLP

Chao Wang
University of Southern California

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 corresponds to the existence of counter-examples of a bounded length. These satisfiability problems are usually decided by SAT solvers, whose performance depends heavily on the variable decision ordering. The satisfiability problems in BMC have some unique characteristics that can be used to help the SAT decision making, but so far they have not been explored by the decision heuristics of most modern SAT solvers. We propose an algorithm to exploit the correlation among the sequence of SAT problems in BMC, by predicting and successively refining a partial variable ordering. This variable ordering is computed based on the analysis of all previous unsatisfiable instances, and is then combined with the SAT solver's existing decision heuristic to determine the final variable decision ordering. Experiments on real designs from industry showed that our new method improved the performance of SAT-based BMC significantly.

Categories and Subject Descriptors
B.6.3 [**Logic design**]: Design aids—*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 the Binary Decision Diagrams (BDDs) based model checking. Unlike BDDs, SAT algorithms built on top of circuits or Boolean formulae suffer less from the potential space explosion. Therefore, SAT-based BMC can handle many industrial-scale circuits that cannot be handled by BDD-based model checking.

In BMC, we are searching for counter-examples to a linear time property of a bounded length. The existence of these finite-length counter-examples can be encoded in a Boolean formula, such that the counter-examples exist if and only if the formula is satisfiable. A Boolean formula is satisfiable if and only if there exists a set of assignments that make the formula true.

Many modern SAT solvers [17, 20, 13, 8] are based on the Davis-Putnam-Logemann-Loveland (DPLL) procedure [5, 4]. Given a formula in the Conjunctive Normal Form (CNF), a recursive search is used to decide whether it is satisfiable or not. At each node in the "search tree," a variable is selected and assigned either 0 or 1. Applying this assignment to the original formula gives us a subformula that can be used for the subsequent searches. The assignment may also 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. some clauses become unsatisfiable under the current partial assignment). 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 these Boolean variables are assigned (as well as the values assigned to them) affects the SAT solving performance significantly. In fact, different variable decision orderings imply different search trees, whose sizes and corresponding search overheads 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 different heuristics to compute decision orderings that are "good enough." For example, the SAT solver Chaff uses the Variable State Independent Decaying Sum (VSIDS) heuristic. (Pre-Chaff decision heuristics can be found in the survey paper by Silva [16].)

Most aforementioned SAT solvers were designed to deal with general CNF formulae. Using them to decide the SAT problems encountered during BMC requires the translation of the resulting formulae into CNF. Useful information that is unique to BMC is lost during this 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 be used to 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 keeps increasing. 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.

1.1 Related Work

The work most closely related to ours is the one by Shtrichman [15], in which a predetermined variable ordering was extracted by Breadth-First Search (BFS) on the Variable Dependency Graph (VDG). After unrolling, the entire BMC instance was regarded as a large combinational circuit lying on a plane whose *x*-axis is time (i.e. time frames from 0 up to *k*), and whose *y*-axis is the 'registers.' By either forward or backward BFS on the VDG of this circuit, Shtrichman sorted the Boolean variables according to their positions on the 'time axis'. In contrast, our new method can be viewed as sorting the Boolean variables according to their posi-

[*] This work was supported by SRC contract 2002-TJ-920.

(The preliminary version.)
.

tions on the other axis – the 'register axis', so it is orthogonal to that of Shtrichman.

Recently, Ganai et al. proposed a hybrid representation [7] as the underlying data structure of their SAT solver. Both circuits and CNF formulae were included in order 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 (statically and dynamically) to help the SAT search, where these implications were extracted by BDD operations [10].

Lu et al. proposed to use circuit topological information and signal correlations to enforce a good decision ordering in their circuit SAT solver [12]. The correlated signals of the underlying circuit were identified by random simulation, and were then applied either explicitly or implicitly to the SAT search. In their explicit approach, the original SAT problem was decomposed into a sequence of subproblems for the SAT solver to solve one-by-one. In their implicit approach, correlated signals were dynamically grouped together in such a way that they were most likely to cause conflicts.

The incremental nature of the BMC instances was also exploited by several incremental SAT solvers [19, 6]. However, these works primarily focused on how to incrementally create a BMC instance with as little modification as possible to the previous one, and on how to re-utilize previously learned conflict clauses. Refining the SAT decision ordering, on the other hand, has not been studied. We believe that our new method can be combined with these existing incremental techniques to further improve their performance.

We implemented our new method on top of the bounded model checking command in VIS-2.0 [2, 18] and the SAT solver Chaff [13]. A set of real designs from industry were used for our experimental studies; these designs are from the public-domain IBM Formal Verification Benchmarks [11]. The experimental comparision show that our new method outperforms the standard BMC command in VIS significantly: It won on 32 out of the 37 circuits, and its average improvement in terms of the CPU time is 42%.

After establishing the background in the following section, we build a connection between unsatisfiable BMC instances and abstractions of the model in Section 3. Our new algorithm is presented in Section 4, followed by the analysis of the experimental results in Section 5. We then conclude in Section 6.

2. Preliminaries

We define the model under verification M as an *open system* represented by a 4-tuple $\langle V,W,I,T\rangle$, where $V=\{v_1,\ldots,v_n\}$ is the set of present-state variables, $W=\{w_1,\ldots,w_m\}$ is the set of combinational variables, $I(V)$ is the initial state predicate, and $T(V,W,V')$ is the transition relation. The variables in $V'=\{v'_1,\ldots,v'_n\}$ are the next-state variables.

In a sequential circuit, the variables in W are associated with either the primary inputs or the outputs of the combinational logic gates; the variables in V and V' are associated with the outputs and the data inputs of the registers. The transition relation is given as the synchronous composition of elementary relations,

$$T(V,W,V') = \bigwedge_{1\le i\le n} (v'_i \leftrightarrow w_i) \wedge \bigwedge_{1\le i\le m} T_i(W,V) \ . \qquad (1)$$

Each T_i is called a *gate relation* for it describes the behavior of a logic gate. For instance, if w_i is the output variable of a two-input AND gate with inputs w_j and v_k, then $T_i = w_i \leftrightarrow (w_j \wedge v_k)$. If, on the other hand, w_i is a primary input to the circuit, then $T_i = 1$. Each term of the form $(v'_i \leftrightarrow w_i)$ equates a next state variable to a combinational variable, describing that the output of a logic gate is fed to the data input of the i-th register.

```
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: DPLL search procedure with backtracking.

The model-checking problem for any property with a finite size witness or counter-example can be translated into a series of propositional satisfiability problems. When the model is finite, LTL, existential CTL, and ω-regular automata all satisfy this condition. For simplicity, the invariant property $\mathsf{G}P$ (predicate P holds in all reachable states) will be used in the following as an example. $\mathsf{G}P$ is false of 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

$$I(V^0) \wedge \bigwedge_{1\le i\le k} T(V^{i-1},W^i,V^i) \wedge \neg P(V^k) \ , \qquad (2)$$

which is satisfiable if and only if such paths exist. The unrolling depth k keeps increasing monotonically until either a counter-example is found or k exceeds a predetermined *completeness threshold* $\mathbb{N}$ [1]. In practice, Eq. 2 will be translated into a CNF formula and then given to the SAT solver, which decides whether it is satisfiable or not. The translation is linear with respect to both the size of the model and the number of time frames k.

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*. As an example, the following fragment of formula has three clauses, three variables (a, b and c) and six literals (a, $\neg a$, b, $\neg b$, c, and $\neg c$):

$$(a \vee \neg c) \wedge (b \vee \neg c) \wedge (\neg a \vee \neg b \vee c) \wedge \ldots \ .$$

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 has to be *true*. After each decision, there may exist some *unit clauses*. The process of applying the implication iteratively until no unit clause is left is called the *Boolean Constrain Propagation (BCP)*.

The basic DPLL search procedure for SAT is given in Fig. 1, which makes decisions and then applies BCP inside the *while* loop. If all the variables have been assigned and no conflict occurs, the formula is satisfiable and a complete set of assignments is returned. However, a *conflict* may occur after a partial assignment—some clauses become *false* after BCP, which indicates that a previous decision is not appropriate and needs to be modified. The level of that decision is identified by *conflict analysis*, following which, the "inappropriate" decision is recovered by backtracking. A clause learned from the conflict is also added to the clause database (conjoint with the original formula) 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 (in the pseudo code, this corresponds to

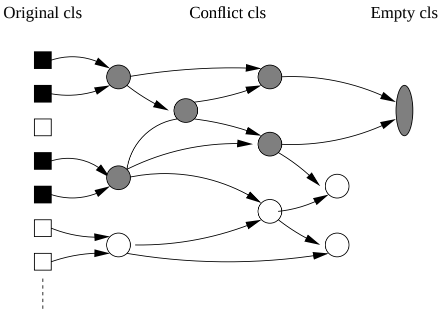

Figure 2: The resolution graph.

a backtrack level less than 0).

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, an example of which is shown in Fig. 2. The leaves of this subgraph are the clauses of the original formula (represented by squares on the left-hand side), and the internal nodes are the *conflict clauses* added during the SAT solving (represented by circles in the middle). By traversing this resolution graph from the unique root backward to the leaves, we can identify a subset of the original clauses that are responsible for this final conflict. In other words, this subset of original clauses, called the *unsatisfiable core* [21, 9], is sufficient to imply unsatisfiability. In Fig. 2, the conflict clauses that contribute to the final conflict are marked gray; the black squares on the left side form the subset of the original clauses in the *unsatisfiable core*.

3. Abstractions

Because of the connection between the CNF formulae and the model (circuit), the unsatisfiable core implies an abstraction of the model, which is represented as $\hat{M} = \langle \hat{V}, \hat{W}, \hat{I}, \hat{T} \rangle$. Here, $\hat{V}$, $\hat{V}'$ and $\hat{W}$ are subsets of V, V', and W respectively. $\hat{I}$ and $\hat{T}$ are all existential abstractions of their counterparts,

$$\hat{I}(\hat{V}) = \exists (V \setminus \hat{V})\,.\,I(V)$$
$$\hat{T}(\hat{V}, \hat{W}, \hat{V}') = \exists (V \setminus \hat{V}), (W \setminus \hat{W}), (V' \setminus \hat{V}')\,.\,T(V, W, V')\ \ .$$

In other words, $\hat{M}$ is constructed from M by removing some registers, inputs, and logic gates. Because all the clauses of the CNF formula are from the circuit, a subset of them identify a subset of registers and logic gates of the circuit, which implicitly define an abstraction. In Fig. 3, the top squares are the original clauses of the CNF formula, and the bottom is one copy of the circuit structure. (There are k copies of the circuit structure in the depth-k BMC instance, one for each time frame.) The black squares represent the unsatisfiable core, each clause of which corresponds to some registers and logic gates in the model. A logic gate is considered to be in the abstract model as long as some clauses describing its gate relation appear in the unsatisfiable core.

The abstraction based on the above definition is an over-approximation of the original model, because the element transition relations of the logic gates that are outside the current abstraction are assumed to be tautologies. Therefore, the abstract model simulates the concrete model in the sense that, if there is no counter-example to $\mathsf{G}P$ in the abstract model, there is no counter-example in the concrete model. Although the implication in the other direction is not always valid, the abstract model based on the unsatisfiable core is sufficient to prove that there is no counter-example of the current length k.

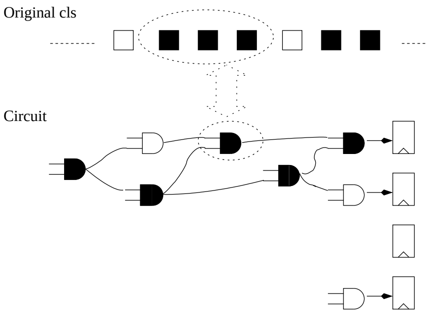

Figure 3: The unsatisfiable core and the abstract model.

Had we known the current abstract model (and the current unsatisfiable core) by an *oracle*, we could have applied 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. Whenever the size of the abstract model is small, the SAT search based on this approach is expected to be much faster.

Of course, there is no way to know the current abstract model unless we have solved the current SAT problem. However, abstract models extracted from previous unsatisfiable BMC instances can be good "estimates" of the current abstract model. In practice, the sequence of SAT problems in BMC are usually highly correlated—their abstract models share a large number of inputs, logic gates, and registers, which makes the estimation sufficiently accurate in most cases. Therefore, such an estimation can be applied to help solving the current SAT problem as described in the previous paragraph.

Our idea of identifying important variables from previous unsatisfiable BMC instances and applying them to solve the current instance is illustrated by Fig. 4. The shaded area represents the unsatisfiable core from the length-3 BMC instance. Variables appearing in this core are recorded and given higher priority during the decision-making on the length-4 BMC instance. In the best-case scenario where our estimation is perfect (i.e., it is sufficient to infer the unsatisfiability of the current instance), no other variable in the formula needs to be assigned before we are able to prove unsatisfiability. Even if there are some discrepancies between the estimation and reality, the size of the search tree is expected to be significantly reduced. This is because by making decisions on variables of the "abstract model" first, un-interesting parts of the search space will be quickly identified and pruned (by adding conflict clauses). In this sense, this approach will also help when the current BMC instance is indeed satisfiable.

The vast majority of the instances encountered in BMC are unsatisfiable. For passing properties, they are always unsatisfiable (i.e. no counter-example); for failing properties, all but the last instance is unsatisfiable. This means that there is a sufficient number of previous abstract models to compute and refine our estimation.

3.1 Identifying Important Variables

In order 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 usually have many such conflict graphs forming a *Conflict Depen-*

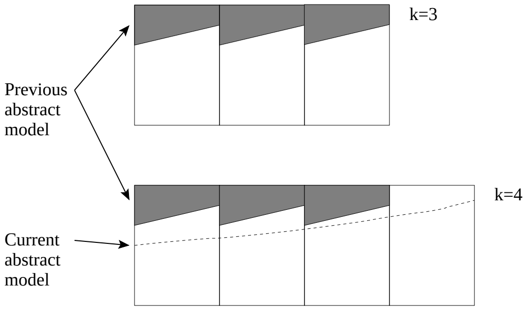

Figure 4: Previous abstractions to help the current BMC instance.

dency Graph (CDG) [3], in which one conflict graph depends on another. 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 some 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 at the same time, we maintain separately a simplified version of the CDG. Our simplification is mainly in the representation of the conflict clause—instead of recording both its content and the dependency relation, we only retain the dependency relation and replace its content by a *pseudo ID*. Our observation is that, for our purpose of identifying all the variables in the unsatisfiable core (a subset of the original clauses), only the dependency relation of these conflict clauses are required. Compared to the number of literals in the conflict clauses, which is often 100–200, the overhead of the pseudo ID (represented by an integer) is small. The simplified CDG will be used for identifying the variables in 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.

4. Our Algorithm

For each previous unsatisfiable instance, we identify all the variables appearing in its unsatisfiable core. Before solving the current SAT problem, variables from all the previous unsatisfiable cores are combined to determine a partial linear variable ordering. (It is called partial because only a subset of the variables may appear.) Each variable is given a score. In solving the current instance, variables with higher scores will be given higher priorities in the decision-making.

The overall algorithm is given as **refine_order_bmc** in Fig. 5, which accepts two parameters: the model M and the invariant predicate P. List *varRanking* is used to store the ranking and scores of the variables appearing in previous unsatisfiable cores. Integer k is the current unrolling depth. Procedure **gen_cnf_formula** is used to generate the CNF representation of the length-k BMC instance, according to Eq. 2. The satisfiability of the formula F is decided by the SAT procedure **sat_check**, which also takes the predetermined ordering *varRanking* as a parameter. Note that for the formula F, *varRanking* is usually a partial ordering.

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

Figure 5: BMC with the refined decision orderings.

If F is unsatisfiable, **sat_check** returns all the variables appearing in the unsatisfiable core, which effectively define an abstract model. Next, the set of new variables in *unsatVars* are used to update *varRanking*. After we move to the next k, the updated ordering will be applied again. The entire procedure terminates as soon as F becomes satisfiable (i.e., the property $\mathsf{G}P$ is proven to be FALSE), or the unrolling depth k exceeds the *completeness threshold* (when the property is declared TRUE).

Inside **update_ranking**, all the Boolean variables that have ever appeared in any previous unsatisfiable core are assigned non-zero scores; the rest are assigned zero. Let $bmc\_score(x)$ denote the score for the variable x, which is defined as follows,

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

where k is the current BMC unrolling depth, and $\mathbf{in\_unsat}(x, j)$ returns 1 if and only if variable x appears in the unsatisfiable core of the j-th BMC instance. The order of the variables in *varRanking* is based on the $bmc\_score()$: Those with higher scores get higher priority.

In the above scoring scheme, all previous unsatisfiable cores are used to determine the current ordering. This is designed for the following two observations: On the one hand, we want to give preference to the variables appearing in most recent unsatisfiable cores (i.e., those of the most recent BMC instances), which usually have higher correlation to the current one. On the other hand, we want to avoid relying exclusively on any particular previous unsatisfiable core, because it may not always be an accurate estimation of the current one.

4.1 Applying the Predetermined Decision Ordering

The following discussion is for the SAT solver Chaff [13]; however, our proposed techniques can be easily adapted to other DPLL-based SAT solvers. In Chaff's VSIDS decision heuristic, every literal l is associated with a $chaff\_score(l)$. The initial value of $chaff\_score(l)$ is the number of literal counts of l in the original formula (i.e., the number of clauses in which l appears). Then, it is updated periodically (e.g., every given number of decisions) as follows,

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

where $new\_literal\_counts(l)$ 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.

In principle, our pre-computed $bmc\_score()$ can either replace or be combined with $chaff\_score()$ to determine the final ordering.

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

Model	True/False or (*k*)	bmc (s)	refine_order_bmc	
			static (s)	dynamic (s)
01_batch	F	39	25	24
02_1_batch_1	(41)	6613	7200	5677
02_1_batch_2	(28)	835	3648	894
02_3_batch_2	(65)	6944	494	476
02_3_batch_4	(65)	6906	433	475
02_3_batch_6	(59)	6861	352	368
03_batch	F	214	222	238
04_batch	F	85	70	67
06_batch	F	962	589	596
11_batch_2	(29)	3820	4533	2932
11_batch_3	(28)	4160	3102	3515
14_batch_1	(35)	201	2272	287
14_batch_2	F	35	30	35
15_batch	F	12	13	12
16_1_batch	(83)	6948	2256	4537
17_1_batch_1	(264)	7161	7114	6965
17_1_batch_2	(12)	29	816	44
17_2_batch_1	(167)	7160	4331	4629
17_2_batch_2	(141)	7181	3475	3268
18_batch	(20)	1172	2999	1049
19_batch	F	139	123	108
20_batch	(28)	3748	5617	3992
21_batch	F	93	80	76
22_batch	(41)	6164	5134	3986
23_batch	(25)	3968	3209	3644
24_1_batch_1	(22)	6045	748	1182
24_1_batch_2	(22)	4992	775	1053
24_1_batch_3	(22)	5075	782	1054
25_batch	(90)	7107	3069	2922
27_batch	F	34	27	37
28_batch	F	782	855	683
29_batch	(22)	4917	5397	4270
31_1_batch_1	(21)	5728	3831	4491
31_1_batch_2	(21)	5838	2292	3552
31_1_batch_3	(21)	4321	1904	3748
31_2_batch_1	(20)	5419	5215	2660
31_2_batch_2	(19)	6924	3180	5475
TOTAL		138,632	86,212	79,021
PERCENTAGE		100%	62%	57%

However, relying exclusively on *bmc_score*() may not be practical, because it is only for a (usually small) subset of variables, and it is for variables instead of literals. (This means that both phases of each variable have the same score.) Therefore, we combine it with the *chaff_score*() in the decision-making, as opposed to using it as the only criterion.

We have proposed two different ways of combining the two scores: One is called static configuration, and the other is called dynamic. In both approaches, variables in the CNF formula are sorted periodically (e.g., every given number of decisions) and at the same time *chaff_score*() is updated as usual. However, the ordering in our static approach is primarily by *bmc_score*(), with *chaff_score*() only as a tiebreaker. It is called 'static' because this configuration is used throughout the SAT solving process.

In the 'dynamic' approach, the ordering is initially based primarily on *bmc_score*() with *chaff_score*() only 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.

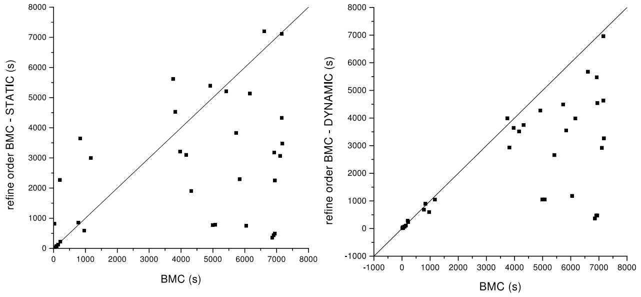

Figure 6: CPU time comparison: BMC vs. refine order BMC.

5. Experiments

We implemented our **refine_order_BMC** on top of the standard bounded model checking command in VIS-2.0 [2, 18] and the SAT solver Chaff [13]. The only difference between the standard BMC command and **refine_order_BMC** is in their SAT variable decision orderings. The BMC command in VIS is based on the basic encoding of BMC as in [1] and the basic induction proof as in [14]. Our experimental studies were conducted on the set of IBM Formal Verification Benchmark circuits [11], each of which has an invariant property $\mathsf{G}P$. The experiments were performed on a 400MHz Pentium II with 1GB of RAM running Linux, with the time-out limit set to 2 hours. Trivial experiments that can be finished by all methods in less than 10 seconds are excluded.

Table 1 compares the CPU time of our new method (with both static and dynamic configurations) to the standard BMC command in VIS. 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 the CPU times of our new method with both static and dynamic configurations. The last two rows of Table 1 give the total CPU times, and the average speed-up of our new methods over standard BMC.

The average speedup of **refine_order_BMC** (static) is 38%; the average speedup of **refine_order_BMC** (dynamic) is 42%. Overall, our new method has 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.

We also extracted some detailed information on solving the sequence of SAT problems of a particular circuit, *02_3_batch_2*. Four figures are given in Fig. 7, which compare the standard BMC to **refine_order_BMC** (static) on the *maximum decision level*, the *number of decisions*, the *number of conflict clauses*, and the *num of implications* at each unrolling depth. All of these numbers are much lower in our new method. In particular, smaller values for the maximum decision level and the number of decisions indicate smaller SAT search trees by our new method.

6. Conclusions

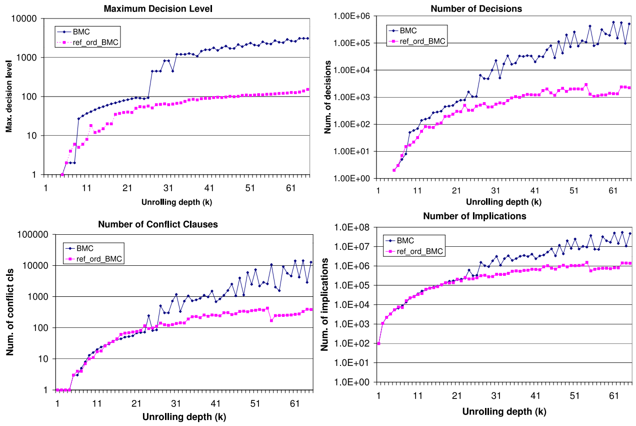

Figure 7: Statistics of BMC vs. refine order BMC (static) on Circuit *02_3_latch_2*.

We have presented a new method to predict and then successively refine a good variable decision ordering for the SAT problems encountered in BMC, which is based on the analysis of previous unsatisfiable instances and the estimation of the current *abstract model*. We have described both the static and the dynamic approaches in applying this partial ordering to the decision-making inside SAT solvers, by taking Chaff as an example. Experiments conducted on real designs have shown that our new method significantly outperforms the standard BMC. Further experimental analysis has indicated that the performance improvement is due to the reduction of the sizes of the search trees.

We believe that 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. Further experimental evaluation will help us fine-tune our procedures, especially for the dynamic approach. We believe that our method can be applied also to SAT-based problems other than BMC, as long as their subproblems have a similar incremental nature.

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 (TACAS'99)*, pages 193–207, Amsterdam, The Netherlands, Mar. 1999. LNCS 1579.

[2] R. K. Brayton et al. VIS: A system for verification and synthesis. In T. Henzinger and R. Alur, editors, *Eighth Conference on Computer Aided Verification (CAV'96)*, pages 428–432. Springer-Verlag, Rutgers University, 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 M. D. Aagaard and J. W. O'Leary, editors, *Formal Methods in Computer Aided Design*, pages 33–51. Springer-Verlag, 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] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7(3):201–215, July 1960.

[6] 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. http://www.elsevier.nl/locate/entcs/.

[7] 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, New Orleans, LA, June 2002.

[8] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 142–149, Paris, France, Mar. 2002.

[9] 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.

[10] 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 (DAC'03)*, pages 824–829, June 2003.

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

[12] 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 (DAC'03)*, pages 436–441, June 2003.

[13] 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, Las Vegas, NV, June 2001.

[14] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer Aided Design*, pages 108–125. Springer-Verlag, Nov. 2000. LNCS 1954.

[15] O. Shtrichman. Tuning sat checkers for bounded model checking. In E. A. Emerson and A. P. Sistla, editors, *Twelfth Conference on Computer Aided Verification (CAV'00)*. Springer-Verlag, Berlin, July 2000. LNCS 1855.

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

[17] 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, San Jose, CA, Nov. 1996.

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

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

[20] H. Zhang. SATO: An efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction*, pages 272–275, July 1997. LNAI 1249.

[21] 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, Munich, Germany, Mar. 2003.