Pruning Techniques for the SAT-Based Bounded Model Checking Problem

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
ofers@summer.weizmann.ac.il

Abstract. Bounded Model Checking (BMC) is the problem of checking if a model satisfies a temporal property in paths with bounded length k. Propositional SAT-based BMC is conducted in a gradual manner, by solving a series of SAT instances corresponding to formulations of the problem with increasing k. We show how the gradual nature can be exploited for shortening the overall verification time. The concept is to reuse constraints on the search space which are deduced while checking a k instance, for speeding up the SAT checking of the consecutive $k+1$ instance. This technique can be seen as a generalization of 'pervasive clauses', a technique introduced by Silva and Sakallah in the context of Automatic Test Pattern Generation (ATPG). We define the general conditions for reusability of constraints, and define a simple procedure for evaluating them. This technique can theoretically be used in any solution that is based on solving a series of closely related SAT instances (instances with non-empty intersection between their set of clauses). We then continue by showing how a similar procedure can be used for restricting the search space of individual SAT instances corresponding to BMC invariant formulas. Experiments demonstrated that both techniques have consistent and significant positive effect.

1 Introduction

SAT-based verification of invariants ($\mathbf{AG}p$) has been practiced for quite some time (see, for example [6]) under different names and for various verification tasks. Biere et. al. recently introduced the notion of Bounded Model Checking (BMC) [1], which extends these methods to LTL and reduces the verification problem to a pure propositional satisfiability problem. By doing so, it enables to exploit the power of advanced standard CNF-SAT solvers.

The basic concept of BMC is to search for a counter example in executions whose length is bounded by some integer k. For every model M, there exists a finite bound D, called the *Diameter* of M, such that M satisfies a property p iff no trace shorter or equal to D contradicts p. Thus, for a large enough k, this method is complete.

The BMC problem can be efficiently reduced to a propositional satisfiability problem whose size, in terms of number of variables, is linear in k. Since SAT is

T. Margaria and T. Melham (Eds.): CHARME 2001, LNCS 2144, pp. 58–70, 2001.

worst case exponential in the number of variables, k has a crucial effect on the ability to efficiently solve the BMC instance. Verification with BMC is normally based on a gradual process, where k is increased until one of the following occurs: a bug is found, the diameter D is reached, or the problem becomes intractable. In fact, experiments with real designs have shown that it is seldom the case that unsatisfiable instances (corresponding to bug free designs) can be efficiently solved for $k = D$[1]. Several methods were suggested recently to cope with this problem, including a procedure which can be seen as an extended version of the classic inductive proof: first, the property is proven correct up to cycle k. Then, the procedure checks whether this fact implies that the property is also true in cycle $k+1$. If not, k is increased, with the hope that the process will stop before reaching D[11]. In any case, BMC seems to be far more successful in falsification than in verification.

The tool `BMC` that was developed as part of [1], which reduces SMV-compatible models to a corresponding CNF-SAT problem, made it possible to evaluate these methods in comparison with standard BDD-based model checkers. Several such comparisons [2,12], caused BMC to gain recognition as a technique that can frequently outperform classic BDD-based model checking.

In a previous research we demonstrated how the unique structure of BMC invariant formulas can be exploited for various optimizations in the SAT solver, including pre-computation of variable ordering and addition of constraints on the search space [12]. In this paper we continue to explore ways in which generic CNF SAT solvers can be tuned for BMC or for other domains with similar characteristics. In particular, we investigate the possibility of exploiting BMC's gradual nature for speeding up the overall verification time. We will show how it is possible to exploit information gathered while solving a k-instance, for solving faster the consecutive $k+1$ instance. The basic idea is to reuse clauses that were deduced while solving previous instances. These clauses, called *conflict clauses* for reasons we will later explain, are naturally recorded in the standard SAT procedure with the aim of pruning parts of the search tree.

A similar idea was proposed by Silva and Sakallah [8] for the case of Automatic Test Pattern Generation (ATPG). They refer to the reused clauses as *pervasive clauses* and explain, in ATPG terms, under what conditions they are formed: if a circuit is tested with two fault models (i.e. the circuit formula is conjuncted with different formulas, each representing a different fault state. See the above reference for more details), the conflict clauses that were deduced from the circuit itself when checking the first instance are declared pervasive. These clauses can therefore be reused when checking the second instance. The authors define the more general question of 'when can clauses be declared pervasive' as an open problem. In this sense this paper addresses this challenge: we investigate the necessary conditions under which a conflict clause can be shared by two or more general SAT instances, and show a simple decision procedure for their evaluation. In Section 5 we will show how a similar procedure can be used as part

[1] Finding D in of itself is a hard problem, which we will not discuss in this paper.

of a different technique, called *constraints replication*, to add more constraints to a *single* SAT instance.

Experiments with both techniques proved their effectiveness. In a significant number of the test cases the overall verification time was reduced by 50 percent or more. More important, these improvements are rather consistent. As far as our experiments can show (15 different designs), the new techniques consistently reduces or leaves almost unaffected the solving time. Consistency has a strong practical advantage: rather than implementing it as a one more user activated flag, it encourages a change in the default configuration of the relevant tools.

The rest of the paper is structured as follows. We begin by giving necessary background on BMC and SAT in the next two sections. In Section 4 we describe the technical details of the suggested decision mechanism, and prove its soundness. In Section 5 we show how the same technique can be used for restricting the search space within a given SAT instance, as long as this instance stems from an invariant formula. In Section 6 we describe another related work called *incremental satisfiability*, which we believe will be better understood after reading the suggested method. Experimental results from our benchmark are given in Section 7, and some conclusions and ideas for future research are presented in Section 8.

2 Bounded Model Checking of Invariants

We focus on bounded model checking of invariants (**AG**p formulas). The general structure of the corresponding BMC instance is the following:

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

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 there exists a reachable state in cycle i ($j \leq i \leq k$) which contradicts the property P_i. The values of j and k can vary according to the range in which we are looking for the bug. $j = 0$ and $j = k$ are the two extremes corresponding to a full and exact search, respectively.

Our experiments were made on top of the enhanced versions of `BMC` [1] and `Grasp` [13,14], as were described in [12]. `BMC` takes an SMV – compatible model and generates a propositional SAT instance according to Equation (1). It also uses various heuristics to generate a variable ordering file, which is later used by `Grasp` to accelerate its search.

3 SAT Checking and Resolution

In this section we briefly outline the principles adopted by modern propositional SAT-checkers, and in particular those which `Grasp` is based on. Our description follows closely the one in [13].

Most of the modern SAT-checkers are variations of the well known Davis-Putnam procedure [4]. The procedure is based on a backtracking search algorithm that, at each node in the search tree, first decides on an *assignment* (i.e. both a variable and a Boolean value, which determines the next sub-tree to be traversed) and then iteratively applies the unit clause rule. The procedure backtracks once the current partial assignment contradicts one of the clauses. Each time that such a conflict occurs, `Grasp` analyzes the cause of the conflict. This analysis produces two distinct pieces of information:

1. The decision level which the procedure should backtrack to. Unlike the original Davis-Putnam procedure, `Grasp` supports non-chronological backtracks, thus if the current decision level is d, it can jump back to $d-i$ where $1 \leq i \leq d$. The mechanism for computing the backtrack level is elaborated in [13].
2. New clauses, called *conflict clauses*, are resolved and added to the clause database, thereby avoiding future occurrences of the same conflict. For example, if the assignment $x = T, y = F, z = F$ inevitably leads to a conflict, the addition of the conflict clause $\pi = (\neg x \vee y \vee z)$ will cause the search procedure to backtrack immediately if the above assignment is repeated. In Section 4.1 we will further elaborate on the resolution process which `Grasp` uses for computing these new clauses.

Conflict clauses are of special interest to us, because they possess valuable information for restricting the search. They are a result of time-consuming reasoning process, which can potentially be shared between SAT instances. It should be noted here that adding clauses that are consistent with the SAT instance (without adding new variables) typically makes the instance easier to solve, because it prunes parts of the search tree. This is only an empirical observation, not a theoretical result. Additional clauses can also slow down the process. First, there is an overhead associated with more clauses. This overhead is significant especially when deciding dynamically the next variable. Typically the next variable is chosen by a procedure which loops over all literals, looking for e.g. the assignment which leads to the maximum number of satisfied clauses. More clauses, therefore, slows down this process[2]. Secondly, the added clauses are not equally effective. The addition of one clause can prevent the formation of another, more effective clause, by pruning the sub-tree in which the other clause would have been created. These potential overheads caused most modern SAT solvers to permit a user restriction on the size and number of added clauses.

4 Constraints Sharing

Sharing conflict clauses between SAT instances can be applied whenever solving two or more SAT instances with non-empty intersection between their clauses sets. Constraints sharing is thereby expected to be far more effective in cases

[2] In our case we used predetermined static variable ordering, which eliminates this particular overhead.

where the solution is based on solving a series of SAT instances which share a large number of clauses. BMC and AI Planning problems [9] are two such cases. Pervasive clauses, the restricted version of constraints sharing, was also used in the past for several EDA problems [8,5], as was previously mentioned.

We begin the description of this technique with several simple definitions. In the following discussion, we use the same variables to denote CNF formulas and their associated sets of clauses. The difference will be clear from the context.

Let S_1 and S_2 be two sets of clauses associated with two CNF SAT instances, and φ_0 represent the set of clauses that are common to S_1 and S_2, i.e. $\varphi_0 = S_1 \cap S_2$. We will also need $\varphi_i = S_i \setminus \varphi_0$ $(i \in \{1, 2\})$, the non-overlapping subsets of S_1 and S_2. Finally, let ψ be a set of clauses that is deducible from φ_0, denoted by $\varphi_0 \vdash \psi$. Based on the following claim, we will be able to reuse ψ (which is computed while checking S_1) when checking S_2, by checking $S_2 \wedge \psi$:

Claim. if $\varphi_0 \vdash \psi$ then S_2 is satisfiable iff $S_2 \wedge \psi$ is satisfiable.

The claim is easy to justify: since $\varphi_0 \vdash \psi$ then $S_2 \vdash \psi$, which implies that $S_2 \leftrightarrow S_2 \wedge \psi$. Thus, S_2 is satisfiable iff $S_2 \wedge \psi$ is satisfiable.

In the general case, it is not common that two SAT instances share a large number of clauses. There is also a difficulty in mapping the variables between the two instances. However, according to Equation (1) it is apparent that with the exception of the clause $c_p : (\bigvee_{i=j}^{k} \neg P_i)$, φ_j^k is a subset of φ_{k+1}^t for all $t > k$. Thus, φ_1 is comprised of the single clause c_p[3].

In order to compute ψ, we need to isolate it from the set of conflict clauses that are deduced while checking S_1. Only then we can reuse it while checking S_2. One solution to the isolation problem is to check φ_0 rather than S_1. In the BMC case this can be done by omitting c_p from S_1. However, there are two drawbacks to this solution. First, φ_0 represents the transition relation, which is assumed to be consistent and therefore satisfiable. Experiments with this option demonstrated that typically φ_0 is trivially satisfied and, as a result, only a small number of conflict clauses are computed. Second, unlike solving S_1, this is an extra computation task which we prefer to avoid.

Thus, we are looking for a method to isolate ψ while checking S_1. Before we suggest an isolation mechanism, in the next sub-section we describe in more detail the mechanism which `Grasp` uses for computing conflict clauses.

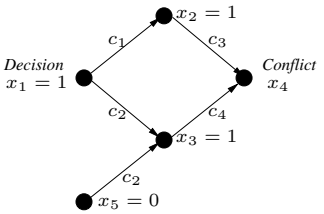
4.1 Derivation of Conflict Clauses

We explain the mechanism of deriving new conflict clauses by following a simplified version of an example first given by Silva and Sakallah in [13]. Assume the clause data base includes the clauses listed in Fig. 1(a), the current truth assignment is $\{x_5 = 0\}$, and the current decision assignment is $x_1 = 1$. Then the resulting *partial implication graph* depicted in Fig. 1 (b) describes the unit clause propagation process implied by this decision assignment.

[3] Note that this is true even if P_i is not an atomic proposition. In this case the equivalence $\bigvee_{i=0}^{n} P_i \equiv \bigwedge_{i=0}^{n} (p_i = P_i) \wedge \bigvee_{i=0}^{n} p_i$ is used, where p_i is a new propositional variable.

$c_1 = (\neg x_1 \vee x_2)$
$c_2 = (\neg x1 \vee x_3 \vee x_5)$
$c_3 = (\neg x_2 \vee x_4)$
$c_4 = (\neg x_3 \vee \neg x_4)$

(a)



(b)

Fig. 1. A clause data base (a) and a partial implication graph (b) of the assignment $x_1 = 1$ shows how this assignment, together with assignments that were made in earlier decision levels, leads to a conflict.

Each node in this graph corresponds to a variable assignment. The incoming directed edges $(x_1, x_j)...(x_i, x_j)$ labeled by clause c represent the fact that $x_1...x_i, x_j$ are c's literals and that the current value of $x_1, ..., x_i$ implies the value of x_j according to the unit clause rule. Thus, vertices that have no incoming edges correspond to decision assignments. The partial implication graph in this case ends with a conflict vertex. Indeed the assignment $x_1 = 1$ leads to a conflict in the value of x_4, which implies that either c_3 or c_4 cannot be satisfied. When such a conflict is identified, `Grasp` determines those variable assignments that are directly responsible for the conflict. In the above example these are $\{x_1 = 1, x_5 = 0\}$. The conjunction of these assignments therefore represents a sufficient condition for the conflict to arise. Consequently, the negation of this conjunction must be satisfied if the SAT instance is satisfiable. We can thereby add the new conflict clause $\pi : (\neg x_1 \vee x_5)$ to the clause database, with the hope that it will speed up the search.

4.2 Isolating ψ

In Section 4 we argued that it is necessary to identify those conflict clauses which are deduced solely from φ_0. These will be the reusable pervasive clauses. The description of the derivation process in the previous subsection sheds light on how this can be achieved. Under the assumption that it is possible to identify in advance the partition of S_1 into φ_1 and φ_0, we suggest the following isolation procedure:

1. Mark φ_0 clauses.
2. For every conflict clause π, if all clauses leading to the conflict are marked, then mark π and add it to ψ.

In the BMC case, marking φ_0 clauses is easy, because we know that all clauses except c_p belong to φ_0.

We demonstrate the isolation procedure by considering a case in which c_1 from the example in Section 4.1 is in φ_1, and $c_2...c_4 \in \varphi_0$. According to step 1, $c_2...c_4$ are marked. While resolving the conflict in e.g. c_4, we observe that the

unmarked clause c_1 is one of the clauses that lead to the conflict. We therefore do not mark the new conflicting clause π and do not add it to ψ.

Claim. By following the isolation procedure, we compute ψ such that $\varphi_0 \vdash \psi$.

Proof. The set of clauses in S_1 in any given time is comprised of three distinct subsets: φ_0, φ_1 and φ_π, where φ_π is the set of the dynamically added conflict clauses. To prove the claim we use induction on the size of φ_π. Initially φ_π and ψ are empty, thus obviously $\varphi_0 \vdash \psi$ is true in the base case. For the induction step, we assume $\varphi_0 \vdash \psi$ and add π to φ_π. We will focus on those cases where ψ is updated and denote the updated ψ by ψ', i.e. $\psi' = \psi \wedge \pi$. There are two such cases:

1. π is derived from φ_0 only. In this case $\varphi_0 \vdash \pi$. Together with the induction hypothesis we get $\varphi_0 \vdash \psi \wedge \pi$, which implies $\varphi_0 \vdash \psi'$.
2. Otherwise, π is derived from $\varphi_0 \cup \Pi$ (where Π is a subset of φ_π). We are interested in the case in which ψ is updated. According to step 2, this can only happen if all the clauses in Π are marked. In this case we have:

   (1) $\varphi_0 \vdash \psi$ (induction hypothesis)
   (2) $\varphi_0 \wedge \Pi \vdash \pi$ (assumption of case 2)
   (3) $\psi \vdash \Pi$ (Π's clauses are marked, therefore they were added to ψ.)
   (4) $\varphi_0 \vdash \Pi$ (from 1,3)
   (5) $\varphi_0 \vdash \pi$ (from 2,4)
   (6) $\varphi_0 \vdash \psi \wedge \pi$ (from 1, 5)

   and from (6), we have that $\varphi_0 \vdash \psi'$.

□

Once ψ is computed and saved to a file, we can simply merge it, after mapping the variable names, with S_2. $S_2 \wedge \psi$ should typically be solved faster than S_2 alone.

4.3 Implementation

While so far we referred to two SAT instances, the gradual nature of BMC allows to accumulate information from all previously checked instances with a lower k.

In the previous subsection we showed, given the list of φ_0 clauses, how to alter the SAT checker so it can generate ψ . In Fig. 2 we suggest a procedure which, based on this new feature, merges constraints sharing into the iterative BMC process. Constraints from previous runs are saved in a file called `<model>.psi` together with their corresponding k (line 2 in the procedure refers to this figure through the variable 'index'). These constraints are later merged into each new instance with a higher k. The procedure is bounded by the global variable D which holds the diameter of the design.

Bool **Solve** (model M, from j, to k, jump size jmp)
1. Generate the file `M.j-k.cnf` where φ_0 clauses are marked.
2. Add `M.psi` clauses with index $< k$ to `M.j-k.cnf`. Mark them as φ_0 clauses.
3. SAT-solve `M.j-k.cnf` while adding ψ clauses to `M.psi`.
4. If *result = unsatisfiable*
   if $(k < D)$ return **Solve** $(M, k+1, k+jmp, jmp)$ else return True.
   else print trace, and return False.

Fig. 2. An iterative Bounded Model Checking procedure with constraints sharing.

5 Internal Constraints Replication

In [12] we suggested a technique called *constraints replication*, which adds constraints to φ_0^k based on the almost symmetric structure of this formula (without the initial state I_0, φ_0^k has k equal parts up to variable names). In order to describe this technique we will use two new notations: x_i denotes a variable x in cycle i, and $\pi^{(i)}$ denotes the clause obtained by shifting i cycles each variable in the clause π. For example, if $\pi = (x_3, y_6)$ then $\pi^{(2)} = (x_5, y_8)$. We will use a similar notation for set of clauses.

Let π be a conflict clause which is deduced from a set of clauses $S \subset \varphi_0^k$, i.e. $S \vdash \pi$. We claim that if $S^{(i)} \subset \varphi_0^k$, then $S^{(i)} \vdash \pi^{(i)}$. Consequently, the *replicated clause* $\pi^{(i)}$ is also a conflict clause which can be added to φ_0^k. The problem is that since φ_0^k is not completely symmetric (due to I_0), it is not always the case that $S^{(i)} \subset \varphi_0^k$ (ignoring, temporarily, the question of i's range). In [12] we suggested a two-step 'trial and error' approach to solve this problem. Given a conflict clause π, first generate all replicated clauses by simultaneously increasing or decreasing the variables indices in π, as long as they stay in the range $0..k$. In the second step, which we refer to as the simulation phase, check if the complement of each replicated clause indeed leads to a conflict (recall that by definition every assignment that satisfies the negation of a conflict clause must lead to a conflict). If yes, add $\pi^{(i)}$ to φ_0^k. If not – discard it.

5.1 An Alternative Solution

The problem with the simulation phase is that checking whether a given partial assignment leads to a conflict may require a large computational effort. If we choose to minimize the overhead by limiting the search time, we take the risk that some 'good' replicated clauses are discarded.

Based on a procedure similar to the one described in Section 4.2, we would now like to offer an alternative to the simulation phase. This method will always identify the good replicated clauses and will hardly require any overhead. For the sake of simplicity we will handle here φ_0^k rather than φ_j^k. Only minor adjustments are needed for handling the more general case.

Our goal is to check efficiently whether a given set of clauses S has a shifted set $S^{(i)}$, and compute the range of i. The following procedure utilizes φ_0^k's structure to achieve this goal:

1. While generating φ_0^k, mark each clause c if $c^{(i)} \in \varphi_0^k$ for $i = 0..k$ (all clauses except I_0 and c_p.
2. For every conflict clause π, if all clauses leading to the conflict (the S clauses) are marked, then mark π as 'replicable'. In addition, record l_S and h_S, the lowest and highest cycle index in S, respectively.
3. For each replicable clause π, add a replicated clause $\pi^{(i)}$ for i in the range $-l_S...(k - h_S)$.

Example 1. Consider the conflict clause $\pi : (\neg x_1 \vee z_2)$ which is deducible from the set of clauses S:

$$c'_1 = (\neg x_1 \vee y_2) \qquad c'_2 = (\neg x_1 \vee y_1 \vee z_2)$$
$$c'_3 = (\neg y_2 \vee z_3) \qquad c'_4 = (\neg y_1 \vee \neg z_3)$$

($c'_1..c'_4$ are structurally equivalent to $c_1..c_4$ of Fig. 1. Here we use the notation in which subscripts represent cycle numbers). If $c'_1..c'_4$ are marked in step 1, then π is replicable. We note that $l_S = 1$ and $h_S = 3$. Thus, if e.g. $k = 5$, we can add (in addition to π itself) the replicated clauses:

$$\pi^{(-1)} : (\neg x_0 \vee z_1) \quad \pi^{(1)} : (\neg x_2 \vee z_3) \quad \pi^{(2)} : (\neg x_3 \vee z_4).$$

5.2 When Do Replicated Clauses Become Pervasive?

After defining the conditions for adding clauses both outside (pervasive clauses) and inside (replicated clauses) the SAT instance, we now investigate the circumstances in which these two techniques can be combined, i.e. when do replicated clauses become pervasive. We once again use S to denote the set of clauses that imply the conflict clause π.

Claim. A replicated clause $\pi^{(i)}$ is pervasive if π is pervasive (where i is in the range as defined in step 2 of the procedure listed in Section 5.1).

Proof. Given that π is pervasive, it implies that $S \subset \varphi_0^{k+1}$. Since π is also replicable, S only includes clauses that were marked in step 1 of the procedure. Together this implies that $S^{(i)} \subset \varphi_0^{k+1}$ and therefore $\varphi_0^{k+1} \vdash \pi^{(i)}$. Thus, $\pi^{(i)}$ can be reused with φ_0^{k+1}. □

6 More Related Work: Incremental Satisfiability

The idea of solving SAT instances incrementally can be attributed to Hooker [7]. He proposed an algorithm that given a satisfiable instance and an additional clause, it checks whether the satisfiability is preserved when the new clause

is added to the formula. His experiments showed that solving large instances incrementally can be faster than solving them as one monolithic formula. It was later extended by Kim et al. [10] and used for path delay fault testing, a process in which the effect of faults on delays in certain paths is checked. The large number of paths typically requires the partition of the problem into a series of instances, each representing a subset of the tested paths. All the paths share the same prefix P, which empirically is far larger than the suffixes $s_1...s_i$. Incremental satisfiability is then used in the following way. A satisfying assignment for P is sought, and conflict clauses are added to P (those clauses that are deducible directly from P). If P is unsatisfiable, the process halts because the conjunction of P with S_i for all i is obviously unsatisfiable. Otherwise, the trace is used as an initial assignment when checking each of the instances $P \wedge S_i$ for all i. In case the initial trace does not lead to a satisfying assignment, the standard backtrack process is invoked.

The resemblance between their and our work is rather clear: the prefix P in their work is $\varphi_0 = S_1 \cap S_2$ in ours. The addition of conflict clauses that were computed while looking for a satisfying assignment to P is equivalent to the option of checking φ_0 directly. In Section 4 we argued that this is ineffective in the BMC case, because φ_0 is satisfied *too fast* for creating a substantial number of conflict clauses, which are essential for speeding up the search later.

7 Experimental Results

To experiment with the two suggested techniques, we randomly chose 15 different hardware designs from IBM's internal benchmark set. The results of this experiment show that constraints sharing has a consistent positive effect, or only marginal negative effect due to its overhead. However, as was explained in Section 3, this consistency can not be guaranteed for all future cases. Replicating clauses and sharing them, as described in Section 5, also had a very positive effect, although somewhat less consistent.

The results of the 15 cases can be divided into 3 groups: the first group includes 6 designs, which were solved at least 50 percent faster due to the suggested techniques. The second group includes 7 designs, which are solved very fast with or without the new techniques. The satisfying assignment is found in these cases before a significant number of conflict clauses are created, and therefore sharing them or replicating them has little effect, if any. In some of these cases the overhead is larger than the benefit, which results in a small negative effect. The last group includes 2 designs that timed-out with all methods.

In Fig. 3 we present results of five representative cases from the first two groups.

The last instances of designs 3, 8, 9 and 10 are satisfiable, while design 14 is unsatisfiable in all 5 instances. The *C-Sharing* strategy refers to constraints sharing, where the 'added clauses' line indicates the number of clauses that were added to each instance. The *Flip* strategy is a variation of the C-Sharing strategy: rather than using the same configuration for all instances (by 'configurations' we

Strategy		Design # 10					Design # 14				
	$k \rightarrow$	*27*	*28*	*29*	*30*	*31*	*14*	*15*	*16*	*17*	*18*
Normal	time(sec)	61	102	174	144	**14**	**10**	91	192	*	*
C-Sharing	time(sec)	63	77	80	47	16	**10**	58	**155**	1.6E4	*
	added clauses	0	973	1092	1208	1253	0	925	2117	3474	6116
Flip	time(sec)	-	50	-	62	-	-	**31**	-	**4219**	-
	added clauses	0	1112	1206	1361	1408	0	972	1827	3152	6057
C+rep	time	**48**	**21**	**19**	**44**	30	13	48	214	6211	*
	replicated	2094	1704	1216	1075	450	5932	5656	7778	1.7E4	*
	added clauses	0	482	1113	1536	2014	0	3374	5773	9806	1.6E4

Strategy		Design # 3					Design # 9				
	$k \rightarrow$	*10*	*11*	*12*	*13*	*14*	*34*	*35*	*36*	*37*	*38*
Normal	time (sec)	**3**	10	13	238	**1**	34	39	43	49	61
C-Sharing	time (sec)	**3**	12	**7**	75	**1**	35	38	44	47	58
	added clauses	0	207	571	955	1553	0	4	8	9	10
Flip	time (sec)	-	**9**	-	**8**	-	-	42	-	53	-
	added clauses	0	255	656	1126	1709	0	12	13	17	18
C + rep	time	4	14	11	23	2	**25**	**28**	**31**	**33**	**45**
	replicated	1229	1508	1954	2277	0	32	33	34	35	1.5E4
	added clauses	0	726	1877	3024	4380	0	33	67	102	138

strategy		Design # 8				
	$k \rightarrow$	*31*	*32*	*33*	*34*	*35*
Normal	time (sec)	**13**	**14**	**14**	**18**	38
C-Sharing	time (sec)	**13**	**14**	15	**18**	**29**
	added clauses	0	4	9	11	14
Flip	time (sec)	-	15	-	22	-
	added clauses	0	11	14	18	20
C + rep	time (sec)	15	16	18	19	**29**
	replicated	58	30	62	32	4638
	added clauses	0	60	91	155	188

Fig. 3. Representative results of four strategies show the advantage of constraints sharing and replicated clauses in reducing the overall verification time. *C-sharing* refers to the standard constraints sharing procedure, and the *Flip* strategy refers to a procedure where the search strategy alternates in each instance. *C + rep* is the same as C-Sharing, with the addition of internally replicated clauses. Best results are bold-faced, and asterisks (*) represent run times exceeding 20,000 sec.

refer to different ordering strategies, as were listed in [12]), we switched it every run. The instances in the odd columns were solved with an alternative configuration, and are therefore left empty to avoid confusion. The generally – better results are related to the different set of clauses that were added to each of these instances. This was a repeating phenomenon in the experiments we conducted, which indicates that adding clauses that were deduced by a different configuration can cause larger portions of the search space to be pruned. Obviously this can only be a good strategy if the alternative strategy similarly performs,

on average, as the default one. The *C + rep* strategy is the same as C-sharing, with the addition of replicated clauses. The 'replicated' line refers to the number of replicated clauses that were added. These clauses can become pervasive (see Section 5.2), which explains the increase in the number of 'added clauses' in the last line.

8 Summary

We introduced constraints sharing, a technique for sharing information between SAT instances whose clauses sets have a non empty intersection. This technique can be seen as a generalization of an older method called pervasive clauses, which was first introduced in the context of ATPG. We showed how this technique exploits the gradual nature of bounded model checking for shortening the overall verification time. We also showed how the same principle can be used, in the case of invariants checking, for adding constraints within a single SAT instance. Experimental results demonstrate the rather consistent positive effect that both of these methods have. Based on this observation, we implemented the two improvements as part of the default configuration of our versions of `BMC` and `Grasp`.

There are two experimental research directions that can be based on these techniques. First, using constraints sharing when checking two different properties of the same design. Although the percentage of shared clauses is expected to be smaller in this case (the property's clauses are different, and they impose a different *cone of influence*), it should nevertheless accelerate the overall verification time. Secondly, using the same techniques in other domains, such as AI Planning problems. SAT-based planning has been used in the past in a very similar way to BMC: a solution is found by solving a series of SAT instances, where each instance corresponds to a different number of allowed steps in the plan. See e.g. [3] and [9] for more details on this subject.

References

1. 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.
2. 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^{th} *Intl. Conference on Computer Aided Verification (CAV'99)*, Lect. Notes in Comp. Sci. Springer-Verlag, 1999.
3. R. I. Brafman and H. H. Hoos. To encode or not to encode: linear planning. *IJCAI*, pages 988–993, 1999.
4. M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7:201–215, 1960.
5. L.G.e. Silva, L.M. Silveira, and J.P.M. Silva. Algorithms for solving boolean satisfiability in combinational circuits. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*, March 1999.

6. J.F. Groote, J.W.C. Koorn, and S.F.M. van Vlijmen. The safety guaranteeing system at station Hoorn-Kersenboogerd. Logic Group Preprint Series 121, Utrecht University, 1994.
7. J. N. Hooker. Solving the incremental satisfiability problem. *Journal of Logic Programming*, 15:177–186, 1993.
8. J.P.M. Silva and K. A. Sakallah. Robust search algorithms for test pattern generation. In *Proceedings of the IEEE Fault-Tolerant Computing Symposium*, June 1997.
9. H. Kautz and B. Selman. Planning as satisfiability. In *Proc. of the* 10^{th} *European Conf. on AI*, pages 359–363, 1992.
10. J. Kim, J. Whittemore, J.P.M. Silva, and K. A. Sakallah. Incremental boolean satisfiability and its application to delay fault testing. In *IEEE/ACM International Workshop on Logic Synthesis (IWLS)*, June 1999.
11. M. Sheeran, S. Singh, and G. Stalmarck. Checking safety properties using induction and a sat-solver. In Hunt and Johnson, editors, *Proc. Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD 2000)*, 2000.
12. O. Shtrichman. Tuning SAT checkers for bounded model checking. In E.A. Emerson and A.P. Sistla, editors, *Proc.* 12^{th} *Intl. Conference on Computer Aided Verification (CAV'00)*, Lect. Notes in Comp. Sci. Springer-Verlag, 2000.
13. J.P.M. Silva and K. A. Sakallah. GRASP – a new search algorithm for satisfiability. Technical Report TR-CSE-292996, Univerisity of Michigen, 1996.
14. J.P.M. Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48:506–516, 1999.