Applying SAT methods in Unbounded Symbolic Model Checking

K. L. McMillan

Cadence Berkeley Labs

Abstract. A method of symbolic model checking is introduced that uses conjunctive normal form (CNF) rather than binary decision diagrams (BDD's) and uses a SAT-based approach to quantifier elimination. This method is compared to a traditional BDD-based model checking approach using a set of benchmark problems derived from the compositional verification of a commercial microprocessor design.

1 Introduction

Symbolic model checking [6, 7] is a method of verifying temporal logic properties of transition systems that relies on a symbolic representation of sets (*i.e.*, as formulas rather than as an explicit lists). In the finite state case, this method has become identified with Binary Decision Diagrams [4] (BDD's), a canonical form for Boolean formulas that has proved to be quite efficient for this purpose in practice. Because of the success of BDD's, and related structures, other forms of expressing Boolean functions in symbolic model checking have remained largely unexplored. In this work, we will consider the use of conjunctive normal form (CNF) as a representation in symbolic model checking. The use of this form makes it possible to adapt efficient algorithms used in solving the satisfiability problem (SAT) to the most important operation in symbolic model checking, *quantifier elimination*.

In particular, we will show that, with a slight modification, modern SAT algorithms based on the Davis-Logemann-Loveland (DLL) approach can be used to eliminate universal quantifiers from an arbitrary Boolean formula, producing a result in CNF. This makes it possible, using standard methods, to compute a CNF formula equivalent to the CTL formula AXp, where p is an arbitrary Boolean formula. This in turn makes it possible to evaluate any CTL formula using fixed point characterizations of the CTL operators (and, in fact, the formulas of the more general μ-calculus).

We will observe that this procedure, using CNF and SAT-based quantifier elimination, can be exponentially more efficient than model checking using BDD's, in cases where the resulting fixed points have compact representations in CNF, but not as BDD's. We will also compare the SAT-based approach with the BDD-based approach on a set of benchmark model checking problems derived from microprocessor verification.

Related work In the past, SAT methods have been applied in model checking in a variety of ways. In *bounded model checking* [2], the transition relation of a system is unfolded k times, allowing any counterexamples of up to k states to be found using a SAT solver. Unless a bound on the length of counterexamples is known, however, this method cannot actually verify the given property, it can only produce counterexamples. The method presented here is not bounded, and produces a guarantee of correctness when the property is true.

SAT solvers have also been used in a hybrid method to detect when a fixed point has been reached, while quantifier elimination is performed by other means (generally by the expansion of the quantifier as $\exists v.f = f\langle 0/v\rangle \vee f\langle 1/v\rangle$, followed by some simplification method). Examples of this approach include [3, 1, 17]. Because of the expense of quantifier elimination in this method, it is usually limited to sequential machines with a very small number of inputs (typically zero or one). By contrast, the approach presented here uses SAT methods in the actual quantifier elimination step, and is not limited in terms of the number of inputs (examples with hundreds of inputs have been verified). SAT algorithms have also been used to, in effect, generate a disjunctive decomposition for BDD-based image computations [9]. Here, BDD's are not used – the image computation is entirely based on SAT methods and produces a result in CNF.

Finally, another approach to using SAT in model checking is based on unfolding the transition relation to the length of the longest "shortest path" between two states [15]. The fact that this length has been reached can be verified using a SAT solver. Thus, unlike bounded model checking, the method can provide a guarantee of correctness for any property. Unfortunately, the longest "shortest path" can be exponentially longer than the diameter of the state space (for example, the longest shortest path for an n-bit register is 2^n, while the diameter is 1). The method presented here does not involve unfolding the transition relation, and requires a number of iterations bounded by the diameter, as does traditional symbolic model checking.

Outline of this paper In section 2 we outline the standard DLL approach to SAT using conflict-based learning. We first consider the satisfiability problem for CNF formulas, then use this result verify the validity of arbitrary Boolean formulas. In section 3, this basic algorithm is extended to convert an arbitrary Boolean formula into CNF, rather than simply checking its validity. In section 4 we extend this algorithm to eliminate universal quantifiers in the result. We also consider the problem of quantifier elimination under a restriction (*i.e*, a "don't care" condition). In section 5, we then apply this quantifier elimination procedure in a symbolic CTL model checking algorithm, and show how to detect convergence of the fixed point series. Finally, in section 6, we compare this approach to a standard method using BDD's.

2 The basic SAT algorithm

The satisfiability problem (SAT) is to determine whether a Boolean formula in conjunctive normal form (CNF) has a satisfying assignment. We sketch here a

generalized SAT algorithm using conflict-based learning, introducing only sufficient detail to allow understanding of the algorithms that follow. Details of the implementation are crucial for performance, but are not covered here. The reader may refer, for example, to [16, 13] for a detailed treatment.

Preliminaries Let $\mathcal{V}$ be a finite set of *variables* and let $\mathcal{B}$ stand for the set $\{0, 1\}$. A *literal* is a variable $v \in \mathcal{V}$, or its negation $\neg v$. A *clause* is a disjunction of a set of zero or more literals $l_1 \vee \cdots \vee l_n$ (where the disjunction of zero literals is taken to mean the constant $\mathbf{0}$). A CNF formula is a conjunction of a set of zero or more clauses $c_1 \wedge \cdots \wedge c_n$ (where a conjunction of zero clauses taken to mean the constant $\mathbf{1}$). In the sequel, we will speak of a clause as a *set* of literals, and a CNF formula as a *set* of clauses, the disjunction or conjunction, respectively, of these sets being implicit. A clause will be said to be *trivial* if it contains both a variable v and its negation $\neg v$. A trivial clause is equivalent to the constant $\mathbf{1}$. In the sequel, will will take "clause" to mean "non-trivial clause".

An *assignment* is a partial function from $\mathcal{V}$ to $\mathcal{B}$. An assignment is said to be *total* when its domain is $\mathcal{V}$. A total assignment A is said to be *satisfying* for formula f when $f(A)$, the value of f given A under the usual interpretation of the Boolean connectives, is 1. We will equate an assignment A with a conjunction of a set of literals, specifically the set containing $\neg v$ for all $v \in \mathrm{dom}(A)$ such that $A(v) = 0$ and v for all $v \in \mathrm{dom}(A)$ such that $A(v) = 1$. Thus, for example, we will take $a \wedge \neg b$ to stand for the assignment $\{(a, 1), (b, 0)\}$.

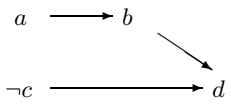
Boolean constraint propagation The basic SAT algorithm builds up an assignment by making a sequence of arbitrary decisions. During this process, additional implied assignments are generated by a process called *Boolean constraint propagation* (BCP). That is, given an assignment A and a clause c in the CNF formula, if all the literals in the clause but one are false in A, then the remaining literal must be true in any satisfying assignment extending A. Thus, this *implied* literal can be added to A without loss of generality. BCP builds an *implication graph*, a DAG in which the vertices are literals, and each vertex is implied by its predecessors.

More formally, for a given CNF formula f and assignment A, let the *implication graph* $\mathrm{IG}(A, f)$ be a directed acyclic graph (V, E), where V is a set of literals. For any vertex $l \in V$, let $\mathrm{preds}(v)$ denote $\{l' \in V \mid (l', l) \in V\}$. The implication graph has the following properties:

- Every literal in A is a root.
- For every vertex l not in A, the CNF formula f contains the clause $l \vee \bigvee_{m \in \mathrm{preds}(l)} \neg m$. We will denote this clause $\mathrm{cl}(l, A, f)$.
- For all $v \in \mathcal{V}$, V does not contain both v and $\neg v$.

We also assume that the graph is maximal, in the sense that no extension of the graph satisfies the above conditions. Note, however, that the above conditions do not uniquely define the implication graph. We will refer to literals in A as the "roots" of the graph, although technically one-literal clauses in f (unit clauses) can also induce vertices with no predecessors.

As an example, suppose that $f = (\neg a \vee b) \wedge (\neg b \wedge c \wedge d)$ and $A = a \wedge \neg c$. A possible implication graph $\mathrm{IG}(A, f)$ is shown below:



We will denote by A_f the assignment induced by the implication graph $\mathrm{IG}(A, f)$. That is, $A_f = \bigwedge V$, where $(V, E) = \mathrm{IG}(A, f)$. In our example, $A_f = a \wedge \neg c \wedge b \wedge d$. It is straightforward to show by induction on the edge relation E that $f \wedge A$ implies A_f.

Conflict-based learning Given an assignment A, a clause is said to be in *conflict* when all of its literals are false in A_f. If any clause in the CNF formula f is in conflict, then our assignment A cannot be extended to a satisfying assignment. In this case, a technique called *conflict-based learning* is used to deduce a new clause that will, in effect, prevent us from becoming blocked in the same way in the future. This new clause, called the *conflict clause*, is deduced by resolving existing clauses in f using the implication graph as a guide.

First, we must define *resolution*. Given two clauses of the form $c_1 = v \vee A$ and $c_2 = \neg v \vee B$, we say that the *resolvent* of c_1 and c_2 is $A \vee B$, provided $A \vee B$ is non-trivial (*i.e*, contains no contradictory literals). For example, the resolvent of $a \vee b$ and $\neg a \vee \neg c$ is $b \vee \neg c$, while $a \vee b$ and $\neg a \vee \neg b$ have no resolvent, since $b \vee \neg b$ is trivial. It is easy to see that any two clauses have at most one resolvent. The resolvent of c_1 and c_2 (if it exists) is a clause that is implied by $c_1 \wedge c_2$ (in fact, it is exactly $(\exists a)(c_1 \wedge c_2)$).

Now, suppose that $c = l_1 \wedge \cdots \wedge l_n$ is a clause in f that is in conflict. We know that the implication graph contains the literals $\neg l_1 \ldots \neg l_n$. Further suppose that some literal $l_i \in c$ is not in A (*i.e.*, is not a root of the implication graph). Then by definition f contains the clause $\mathrm{cl}(\neg l_i, A, f) = \neg l_i \vee \neg m_1 \vee \cdots \vee \neg m_k$ where $m_1 \ldots m_k$ are the predecessors of $\neg l_i$ in the implication graph. The resolvent of c and $\mathrm{cl}(\neg l_i, A, f)$ is a clause that is itself in conflict.

As an example, suppose that that we add the clause $(\neg b \vee \neg d)$ to the example above. This clause is in conflict, since the implication graph contains both b and d. Taking the resolvent of $\mathrm{cl}(d, A, f) = (\neg b \wedge c \wedge d)$ with the conflicting clause $(\neg b \vee \neg d)$, we obtain an implied clause $(\neg b \vee c)$, which is also in conflict. Resolving this clause with $\mathrm{cl}(b, A, f) = (\neg a \vee b)$, we obtain another implied clause $(\neg a \vee c)$, also in conflict.

The following is a generic conflict-based learning procedure that takes a clause in conflict and produces an implied clause (also in conflict) by repeatedly applying resolution steps until some termination condition T is satisfied, or no further steps are possible:

```
0    procedure deduce(c, A, f)
1    while ¬T and exists l ∈ c such that ¬l ∉ A
2            let c = resolvent of cl(¬l, A, f) and c
3    return c
```

Since the resulting clause is implied by f, it can be added to f without changing its satisfiability.

Basic SAT procedure We now put together the methods of Boolean constraint propagation and conflict-based learning to obtain a generic procedure for determining the satisfiability of a CNF formula f:

```
0     procedure SAT(f)
1     let A = ∅
2     repeat
3           if f contains 0 return "unsatisfiable"
4           else if some clause c in conflict
5                 add clause deduce(c, A, f) to f
6                 remove some literals from A
7           else if A_f is total, return "satisfiable"
8           else
9                 choose a literal l such that l ∉ A and ¬l ∉ A
10                add l to A
```

The procedure heuristically guesses new literals (*i.e.*, decisions) to add to the assignment A. If at any point a conflict occurs in the implication graph $\mathrm{IG}(A, f)$, we call the procedure `deduce` to generate an implied clause, which is added to f, and then we "backtrack", removing some literals from A. Otherwise, the procedure terminates when either the implied assignment A_f is total (in which case we have a satisfying assignment) or the empty clause $\mathbf{0}$ is deduced (in which case the f is unsatisfiable). We assume throughout that the implication graph $\mathrm{IG}(A, f)$ is updated incrementally to reflect any changes in A or f.

Note that there are many heuristic choices to be made in this procedure. Notable among these are the choice of which literals to add to A (the decision heuristic), the choice of literals to eliminate by resolution in the conflict-based learning procedure, and the order of building the implication graph. These heuristic choices vary between solvers and strongly effect the efficiency of the procedure, but are essentially orthogonal to the methods introduced here.

Proving validity of arbitrary Boolean formulas Given an arbitrary Boolean formula (not in CNF), there is a standard procedure for constructing a CNF formula that is unsatisfiable exactly when p is valid. We assume a set of *input variables* $V_I \subset V$, and a Boolean formula p over V_I. For simplicity, we also assume p uses only disjunction and negation, and contains no double negation (*i.e.* subformulas of the form $\neg\neg q$). For every subformula q of the form $r \vee s$, we introduce a distinct variable v_q. To each subformula of p we can now associate a literal l_q, which is q if q is an input, v_q if q is a disjunction, and $\neg v_r$ if q is of the form $\neg r$. Now, we construct a CNF formula $\mathrm{CNF}(p)$, which contains, for each subformula q of the form $r \vee s$, the clauses

$$\{(v_q \vee \neg l_r), (v_q \vee \neg l_s), (\neg v_q \vee l_r \vee l_s)\}$$

It is not difficult to show that for any assignment A to V_I, there is a unique satisfying assignment A' of $\mathrm{CNF}(p)$ consistent with A, and such that $A'(l_p) = p(A)$. A a result, the CNF formula $\mathrm{CNF}(p) \wedge \neg l_p$ is unsatisfiable exactly when p is valid.

3 Characterizing Boolean functions in CNF

Now, given an arbitrary formula p, rather than checking the validity of p, we wish to construct an equivalent formula to p in conjunctive normal form. This can be done by a slight modification of the basic SAT algorithm. Briefly, we construct a SAT problem for the validity of p, and run the SAT algorithm as described above. However, if a satisfying assignment is found, instead of terminating, we construct a new clause that is in conflict (*i.e.*, rules out the satisfying assignment) and continue the procedure. This new clause, which we will call a *blocking clause*, must have the following properties:

- It must contain only variables in V_I,
- It must be false in the current assignment A_f, and
- It must be implied by $l_p \wedge \mathrm{CNF}(p)$.

The following procedure uses blocking clauses to compute a CNF formula χ equivalent to p:

```
0     procedure toCNF(p)
1     let f = CNF(p) ∧ ¬l_p, χ = 1, and A = ∅
2     repeat
3           if f contains 0, return χ
4           else if some clause c in conflict
5                 add clause deduce(c, A, f) to f
6                 remove some literals from A
7           else if A_f is total
8                 choose a blocking clause c'
9                 add c' to f and χ
10          else
11                choose a literal l such that l ∉ A and ¬l ∉ A
12                add l to A
```

Each time a satisfying assignment is obtained, the procedure generates a new clause whose complement characterizes a set of satisfying assignments (*i.e.*, it rules out a set of cases where p is false). When the CNF formula becomes unsatisfiable, these clauses precisely characterize p. We can argue partial correctness of this procedure as follows. The procedure maintains the invariant that p implies χ (since only clauses implied by p are added to χ). Further, at all times f is equivalent to $\mathrm{CNF}(p) \wedge \neg l_p \wedge \chi$. Thus, on termination, when $f = \mathbf{0}$, there is no assignment that makes p false and χ true, in other words, χ implies p. If the procedure terminates, therefore, χ is a CNF formula equivalent to p.

Of course, the important question is how to choose a blocking clause when a satisfying assignment is reached. Such a clause can be generated using the conflict-based learning procedure. However, to ensure that the conflict clause we generate involves only input variables, we must use an alternate implication graph to generate it, in which all the roots are assignments to input variables.

Such a graph can be generated in the following way. Suppose we have a satisfying assignment A_f for f. Let $A' = A_f \downarrow V_I$ (the projection of A_f onto the input variables) and let $f' = \text{CNF}(p) \wedge \chi$. We can show that $A'_{f'} = A_f$, that is, the implication graph $\text{IG}(A', f')$ induces the same assignment as $\text{IG}(A, f)$. This can be argued as follows: first, we know that A_f is a satisfying assignment for f', since f' contains a subset of the clauses in f. Further, since $\text{CNF}(p)$ determines the non-input variables as a function of the input variables, it follows that A_f is the *unique* satisfying assignment consistent with A'. Finally, since the truth value of any subformula of p can be inferred from the truth values of its own immediate subformulas, it follows that the assignment $A'_{f'}$ is total. Since A' implies $A'_{f'}$ and A_f is the *only* satisfying assignment consistent with A', it follows that $A'_{f'} = A_f$.

Now, in particular, since f contains the clause $(\neg l_p)$, it follows that the clause (l_p) is in conflict in $\text{IG}(A', f')$. As a result, by computing $c' = \texttt{deduce}(l_p, A', f')$, we obtain a clause that is implied by $\text{CNF}(p) \wedge l_p$ (since this implies f'), and is in conflict (*i.e.*, false in the current assignment). Further, we can ensure that c' involves only input variables by modifying the termination condition T in `deduce` so that resolution may terminate only when c' contains only input literals. This must eventually occur since all the roots of the implication graph $\text{IG}(A', f')$ are input literals. Thus c' is a blocking clause.

Clearly, it would be wise from a performance point of view to maintain the alternate implication graph $\text{IG}(A', f')$ incrementally, updating it as changes occur in A_f. In this way we avoid reconstructing the entire graph each time a satisfying assignment is found. Further, there is no need to add a literal to A' if that literal is already implied by the existing assignment. Thus, A' will typically be smaller than $A_f \downarrow V_I$, which will generally result in shorter blocking clauses.

As an example, suppose that $p = (a \wedge b) \vee c$, and suppose that we have guessed the assignment $A = a$. A possible implication graph is shown in (a) below. Note that the literal $\neg l_p$ always occurs in the implication graph because f includes the unit clause $(\neg l_p)$. At this point, we have a satisfying assignment. Projecting onto the input variables, we have $A' = a \wedge \neg b \wedge \neg c$. The resulting alternate implication graph $\text{IG}(A', f')$ is shown in (b) below. Note that the implied assignment is the same as in the original graph, and further, clause (l_p) is in conflict. Thus, we begin the conflict-based learning process with clause (l_p). We first eliminate l_p, to obtain $(l_{a \wedge b} \vee c)$. Since this contains a non-input variable $v_{a \wedge b}$, we must continue. Thus, we eliminate $l_{a \wedge b}$, obtaining the blocking clause $(b \vee c)$. Adding this clause, we undo the assignment to a and continue. Suppose we once again guess $A = a$. This time, instead of reaching a satisfying assignment, we find the new clause $(b \vee c)$ in conflict. Thus, we perform the normal conflict analysis, obtaining the conflict clause $(\neg a)$. Propagating implications, this yields a new satisfying assignment where $A' = \neg a \wedge b \wedge \neg c$. This in turn yields the blocking clause $(a \vee c)$. This clause is in conflict in the empty assignment, thus, the standard conflict analysis infers the empty clause, and the procedure terminates, returning the CNF formula $(b \vee c) \wedge (a \vee c)$.

a) implication graph b) alternate graph rooted at inputs

4 Quantification and Image Computations

As noted in the introduction, in order to compute AXp, we need to be able to eliminate universal quantifiers. That is, given a Boolean formula p, and a set of variables $W = w_1, \ldots, w_n$, we would like to construct a Boolean formula equivalent to $\forall W.p$. This is quite easily done if p is a CNF formula, as we simply delete all the literals of the form w_i or $\neg w_i$. That is, the universal quantifier distributes over the conjunction, and also over the disjunction within a clause, since the literals in the clause have independent support. It follows that to compute a CNF formula for $\forall W.p$, we have only to convert p to CNF form using procedure `toCNF` and then delete the literals w_i and $\neg w_i$ from the result. This procedure may be highly inefficient however. In the extreme case, we may generate an exponentially large CNF formula for p, which then becomes equivalent to the single clause $\mathbf{0}$ when quantification is applied.

To alleviate his problem, we can introduce the quantification step into the SAT algorithm itself. This yields the following procedure:

```
0     procedure forall(W, p)
1     let f = CNF(p) ∧ ¬l_p, χ = 1, and A = ∅
2     repeat
3           if f contains 0, return χ
4           else if some clause c in conflict
5                 add clause deduce(c, A, f) to f
6                 remove some literals from A
7           else if A_f is total
8                 choose a blocking clause c'
8a                remove literals of form w_i or ¬w_i from c'
9                 add c' to f and χ
10          else
11                choose a literal l such that l ∉ A and ¬l ∉ A
12                add l to A
```

This differs from the previous algorithm only in the addition of line 8a, which universally quantifies the variables $w_1, \ldots, w_n$ in the blocking clause c'. This procedure maintains the invariant that $\forall W.p$ implies χ (since c' is always implied by p, it follows that $\forall W.c'$ is always implied by $\forall W.p$). Further, at all times f is equivalent to $\mathrm{CNF}(p) \wedge \neg l_p \wedge \chi$. Thus, on termination, when $f = \mathbf{0}$, there is no assignment that makes p false and χ true, in other words, χ implies

p, hence $\forall W.\chi$ implies $\forall W.p$, hence $\chi \Rightarrow \forall W.p$. If the procedure terminates, therefore, χ is a CNF formula equivalent to $\forall W.p$.

Quantifier elimination under a restriction Typically in a symbolic model checking application, we are given a restriction r on the result of an operation. That is, we only care about the value of the resulting formula when r is true. In this case, we would like to evaluate $(\forall W.p) \downarrow r$, which is defined to mean some formula g such that $r \wedge g = r \wedge \forall W.p$. This can be accomplished using the quantifier elimination procedure `forall` by simply replacing CNF(p) with CNF(p) $\wedge$ CNF(r) $\wedge\, l_r$. This in effect restricts the satisfying assignments to those that satisfy r. When the algorithm terminates, χ is a CNF formula for $(\forall W.p) \downarrow r$. Further, in the case when $r \Rightarrow \forall W.p$, there is no satisfying assignment, hence the algorithm returns $\chi = \mathbf{1}$. This is a useful property, as we will shortly observe.

5 Symbolic Model Checking Using SAT methods

We now consider the use of the above quantifier elimination algorithm in symbolic CTL model checking. We assume the reader is familiar with the temporal logic CTL, as defined by Clarke and Emerson [8], and with symbolic model checking methods [12]. To give a symbolic CTL model checking algorithm using a given representation for Boolean functions, it suffices to give an algorithm for reducing the formula AXp, where p is a Boolean formula, into the given form. The remaining operators of the logic can then be derived using standard equivalences and fixed point characterizations.

Briefly, we assume a set of state variables $\mathcal{S} = s_1, \ldots, s_n$ and a set of combinational variables $W = w_1, \ldots, w_k$. The transition relation of the model is given by a set of equations of the form $s_i' = \delta_i(\mathcal{S}, W)$. For a given propositional formula p, we can characterize AXp as:

$$AXp = \forall W.\ p\langle \delta_i / s_i \rangle$$

That is, we can compute a CNF formula equivalent to AXp by syntactically substituting each s_i by δ_i, and then applying our universal quantifier elimination algorithm to the combinational variables $w_i, \ldots, w_k$. Now, for example, we can compute AGp, using the "frontier set simplification" method, as the conjunction of the following sequence:

$$Z_1 = p$$

$$Z_{i+1} = (AX\ Z_i) \downarrow \bigwedge_{j=1}^{i} Z_j$$

To evaluate the "frontier set" Z_{i+1}, we can use our algorithm for quantifier elimination under a restriction. This sequence converges when $\bigwedge_{j=1}^{i} Z_j \Rightarrow (AX\ Z_i)$, in which case Z_{i+1} is the constant $\mathbf{1}$. This gives us a way to detect convergence without solving an additional SAT problem, and yields the following procedure for computing a Boolean formula equivalent to AGp:

```
0       procedure AG(p)
1       let Z = Q = p
2       while Z ≠ 1
3               let Z = (∀W. p⟨δ_i/s_i⟩) ↓ Q
4               let Q = Q ∧ Z
5       return Q
```

It is straightforward to derive algorithms for the other CTL modalities using their fixed point characterizations. Note that the existential modalities can be obtained using the duality $EXp = \neg AX \neg p$.

6 Comparison with BDD-based symbolic model checking

We now compare this approach to symbolic model checking using SAT techniques to a standard approach using binary decision diagrams (BDD's). In our experiments, both techniques use the same fixed point iteration for AGp (though we will also compare to a BDD method using a forward traversal method). The BDD-based technique is implemented in the Cadence SMV system.[1] It uses a "conjunctive partitioning" approach [5] to the reverse image computation. It also uses "dynamic variable ordering" [14] to optimize the BDD variable order during the computation. The quantifier elimination algorithm `forall` was implemented by modifying the ZCHAFF SAT solver [13] from Princeton University. The decision variable heuristics and implication graph mechanism in that solver are unchanged. The implementation is quite inefficient, in that it reconstructs the entire implication graph $\mathrm{IG}(A', f')$ when a satisfying assignment is found, rather than maintaining this structure incrementally. Thus, there is considerable room for improvement in performance results presented. All computations are performed on workstation using a 900MHz Pentium III processor, and the Linux operating system.

A simple example To begin with, it is generally held that BDD's provide a more compact representation for Boolean functions than, for example CNF. However, there are some interesting cases where the CNF form can be exponentially more compact than the BDD form. Consider, for example, the following simple sequential machine that we will call `swap`. The state of `swap` consists of n k-bit binary numbers, $x_0, \ldots, x_{n-1}$. We assume that $n \leq 2^k$. The input to the machine is a number i, in the range $0, \ldots, n-1$. At each step, the machine swaps the values of x_i and $x_{i-1 \bmod n}$. In the initial state of the machine, we have, for all j, $x_j = j$. We would like to check the property $AG(x_0 \neq x_1)$, that is, the first two numbers are always different. What is interesting about this problem is that $AG(x_0 \neq x_1)$ is precisely the set of states such that all the x_i's are distinct. This is similar to a situation that occurs in real systems that rely on the fact that no resource is allocated to two different users.

We note that the BDD representing the fact that "all the x_i's are distinct" is exponential for any variable order. On the other hand, there is a cubic CNF

[1] `http:/www-cad.eecs.berkeley.edu/~kenmcmil`

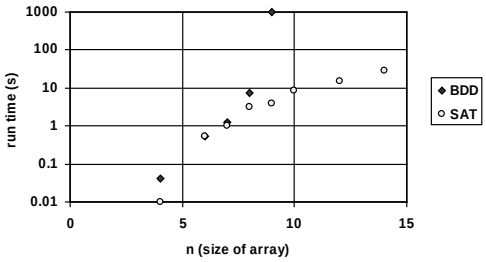


Fig. 1. Run time for SAT- and BDD-based model checkers on `swap`

representation for this proposition, of the form:

$$\bigwedge_i \bigwedge_{j>i} \bigwedge_v (x_i \neq v \vee x_j \neq v)$$

Of course, since our SAT-based algorithm does not compute a *minimal* CNF formula for AGp, we are not guaranteed to obtain a result of this size. However, it is plausible that the CNF-based technique could perform exponentially better than a BDD-based technique on this example.

In fact, this turns out to be the case. Figure 1 plots the run time performance of a BDD-based model checker and our SAT-based model checker in computing $AG(x_0 \neq x_1)$, as we increase n, letting $k = \lceil \log_2 n \rceil$. The BDD-based model checker was stopped at $n = 10$, having exhausted 250MB of memory. We note that in fact the BDD run time is increasing exponentially, while the time for the SAT-based model checker increases approximately as $n^{4.5}$. Similar results are also obtained using BDD's with a forward traversal approach. It appears therefore, that the CNF approach can be more efficient than binary decision diagrams in some cases.

Microprocessor verification benchmarks We now compare these two techniques on a more substantial set of benchmarks. These are publicly available model checking problems derived from the compositional verification of one unit of a commercial microprocessor design, using techniques describe in [10].[2] All of the formulas to be checked in this benchmark suite are of the form AGp, where p is Boolean, and all are true in the initial states (*i.e.*, no counterexamples are generated).

These benchmark problems, as generated by the Cadence SMV system, contain a large number of functionally equivalent nodes. This is due partly to the fact that the logic in the RTL design description is unoptimized, and also to the fact that the abstraction steps performed by Cadence SMV can generate many redundant expressions. The benchmarks were preprocessed using a technique

[2] The design is the PicoJava II (TM) design from Sun Microsystems, Inc. The RTL-level source code for this design is available from Sun Microsystems. The benchmark suit is described at `http:/www-cad.eecs.berkeley.edu/~kenmcmil`, and can be generated from the source code using the Cadence SMV tool.

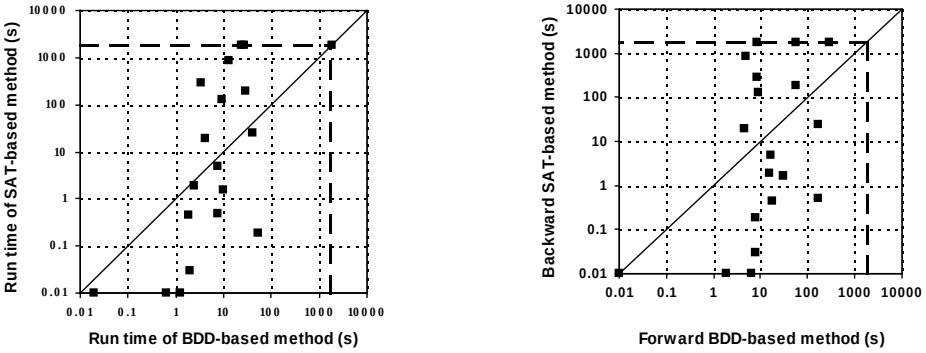


Fig. 2. Microprocessor verification benchmark

called "BDD sweeping" [11] to combine functionally equivalent nets, since this technique was found by Kuehlmann and Baumgartner to substantially improve the performance of SAT-based bounded model checking at little expense.[3]

In the algorithm for AGp, we also perform a lightweight optimization of the CNF formula that results from the quantifier elimination procedure. This procedure is based on a ZBDD representation of the clause set, and results in the elimination of some subsumed clauses, and resolution of some clause pairs (*i.e.*, it can resolve $a \vee b$ and $a \vee \neg b$ to a). This is done because the quantifier elimination procedure is observed to produce many redundant clauses. The time required for this optimization is generally small compared to the total run time, but it is not included in run time figures presented below, which are therefore slightly optimistic.

The left graph in figure 2 plots the performance of the SAT-based method against the performance of the BDD-based method. Points above the diagonal line indicate a faster run time for the BDD-based technique, while points below the line indicate a faster time for the SAT-based approach. A time of 1800 seconds indicates that a computation was stopped at 1800 seconds without completing. This time is indicated by the heavy dashed lines in the figure. In no case was a computation stopped because of memory exhaustion.

For reference, right graph compares the run times of the "backward" SAT-based approach against a forward traversal method using BDD's. This comparison is less direct, but it shows at least that the previous comparison is not a "straw man".

What we observe in this experiment is that, while the total run time is much smaller for the BDD-based technique, for most individual problems, the SAT-based method is faster (in some cases by two orders of magnitude). In the direct comparison, the SAT-based method performed better in 11 cases, while the BDD-based method performed better in 7 cases. However, the graph clearly shows that the variance in run times is greater for the SAT-based method than

[3] Personal communication.

for the BDD-based method. Overall, this is a promising result, since there is likely to be much room for improvement in the SAT-based methods, especially given the highly inefficient implementation of the procedure that was used in the comparison. It suggests at the very least that it would be a good policy to devote a short time to model checking using the SAT-based method before trying the BDD-based approach.

7 Conclusion

We have observed that a traditional DLL-style SAT solver can be modified to perform quantifier elimination on Boolean formulas, producing a result in CNF. This in turn provides a basis for symbolic model checking that is not based on BDD's. This may provide an advantage over BDD-based model checking in the case when the CNF form is more compact than the BDD form (which may happen, for example, when resource allocation is involved), or if the SAT-based image computation step proves to be faster than the BDD-based approach. In a preliminary benchmark comparison against a standard BDD-based approach, the new approach appears promising, especially given the relative maturity of BDD-based model checkers.

In particular, there are several ways in which the current implementation might be improved. In the first place, as mentioned above, our implementation is highly inefficient, since it does not maintain the implication graph incrementally (no self-respecting SAT solver would do this!). Secondly, the BDD-based approach is using the "conjunctive partitioning" method, based on early quantification to simplify the representation of the transition relation. Without this technique, it is unlikely that any of the benchmarks could be completed. On the other hand, the SAT-based method is computing the reverse image in a single step. It is possible that computing the image incrementally, using the early quantification technique, would also improve the performance of the SAT-based method correspondingly. Finally, the method suffers from the fact that "blocking clauses" may only contain input variables. It is straightforward to construct examples that produce an exponentially large number of blocking clauses simply because quantified clauses cannot be learned that involve larger subformulas. In fact, an ordinary SAT solver that could only learn clauses over input variables would be quite inefficient. If a solution can be found for this problem, a dramatic improvement in performance might result.

References

1. P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In *TACAS 2000*, volume 1785 of *LNCS*. Springer-Verlag, 2000.
2. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS 1999*, pages 193–207, 1999.
3. P. Bjesse. Symbolic model checking with sets of states represented as formulas. Technical Report CS-1999-100, Department of Computer Science, Chalmers technical university, March 1999.

4. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.
5. J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. B. Denyer, editors, *VLSI '91*, Edinburgh, Scotland, August 1991.
6. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, June 1990.
7. O. C., C. Berthet, and J.-C. Madre. Verification of synchronous sequential machines based on symbolic execution. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France*, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1989.
8. E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In Dexter Kozen, editor, *Logic of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*, Yorktown Heights, New York, May 1981. Springer-Verlag.
9. A. Gupta, Z. Yang, P. Ashar, and A. Gupta. SAT-based image computation with application in reachability analysis. In *FMCAD 2000*, pages 354–371, 2000.
10. R. Jhala and K. L. McMillan. Microarchitecture verification by compositional model checking. 2001.
11. A. Kuehlmann and F. Krohm. Equivalence checking using cuts and heaps. In *Design Automation Conf.*, pages 263–268, 1997.
12. K. L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.
13. M. W. Moskewicz, C. F. Madigan, Y. Z., L. Z., and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535, 2001.
14. R. Rudell. Dynamic variable ordering for binary decision diagrams. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 42–47, 1993.
15. M. Sheeran, S. Singh, and G. Stalmarck. Checking safety properties using induction and a sat-solver. In *Formal Methods in Computer Aided Design*, 2000.
16. 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, November 1996*, 1996.
17. P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Computer Aided Verification*, pages 124–138, 2000.