SAT-Based Unbounded Symbolic Model Checking

Hyeong-Ju Kang, *Student Member, IEEE,* and In-Cheol Park, *Senior Member, IEEE*

***Abstract*—This paper describes a Boolean satisfiability checking (SAT)-based unbounded symbolic model-checking algorithm. The conjunctive normal form is used to represent sets of states and transition relation. A logical operation on state sets is implemented as an operation on conjunctive normal form formulas. A satisfy-all procedure is proposed to compute the existential quantification required in obtaining the preimage and fix point. The proposed satisfy-all procedure is implemented by modifying a SAT procedure to generate all the satisfying assignments of the input formula, which is based on new efficient techniques such as line justification to make an assignment covering more search space, excluding clause management, and two-level logic minimization to compress the set of found assignments. In addition, a cache table is introduced into the satisfy-all procedure. It is a difficult problem for a satisfy-all procedure to detect the case that a previous result can be reused. This paper shows that the case can be detected by comparing sets of undetermined variables and clauses. Experimental results show that the proposed algorithm can check more circuits than binary decision diagram-based and previous SAT-based model-checking algorithms.**

***Index Terms*—Boolean satisfiability checking (SAT), formal verification, symbol manipulation, symbolic model checking.**

I. Introduction

As THE complexity of system-on-chip design is increasing rapidly, design verification has been attracting more attention in recent years. A promising verification method called formal verification has emerged during the last decade to detect corner-case errors that are hardly detected by simulation [1]–[3]. Formal verification has a variety of methods, and among them, the most widely used is model checking. Given a model and a property, model checking verifies whether or not the model satisfies the property. A major difficulty in applying model checking to practical designs is the state explosion problem that the problem size grows very rapidly as the size of a target design increases.

In the earlier model-checking algorithms, a target design is expressed explicitly [1], [2], [4], [5]. As the algorithms require much memory, they can be applied to only small circuits. To cope with the memory problem, symbolic model checking was proposed as an alternative [6], [7], where the sets of states and the state-transition relation are expressed as formulas. Since most of the recent hardware model-checking algorithms are symbolic, this paper focuses only on symbolic model checking.

In 1986, R. E. Bryant proposed binary decision diagrams (BDDs) to represent Boolean formulas [8], and in 1990, K. L. McMillan *et al.* proposed a BDD-based model checking algorithm [6], [7]. The algorithm can handle circuits with about 10^{20} states, and many methods have been proposed to improve the algorithm. Although the BDD-based model checking is efficient and being widely used, it is still restricted to a limited range of circuits because of the memory explosion. Many works have been proposed to overcome the explosion, but there is no one that resolves the problem completely.

Bounded model checking (BMC) [9], [10] uses Boolean satisfiability checking (SAT) procedures instead of BDDs. The SAT problem is to determine whether a given Boolean formula has a satisfying assignment or not [11]–[13]. In BMC, given a bound k, k transitions are unfolded into an equation. The property to check is transformed into an equation such that the equation is true if and only if the property is false within k transitions. The conjunction of the two equations is checked by the SAT procedure. Although BMC algorithms can check large circuits, they guarantee the property's truth only for k transitions. Recently, a BDD-based BMC algorithm has been proposed in [14]. In opposition to BMC, the model checking that is not limited by a bound is called unbounded model checking (UMC) [15].

In this paper, we propose a SAT-based UMC algorithm in which the sets of states and the transition relation are expressed in conjunctive normal form (CNF). The proposed algorithm performs quantification required in image or preimage computation by using a satisfy-all procedure that generates all satisfying assignments. A two-level logic minimizer compresses the set of the assignments generated by the satisfy-all procedure to achieve better performance by reducing the number of minterms in a state-set.

The proposed UMC algorithm exploits an enhanced satisfy-all procedure that is the most significant part of the algorithm. The satisfy-all procedure is implemented by modifying a conventional SAT procedure. The satisfy-all procedure in [15] generates a better satisfying assignment from a found assignment, but requires the redrawing of an implication graph. The proposed procedure can do the similar process without the redrawing. The proposed procedure exploits line justification used in automatic test pattern generation (ATPG) [16] to find necessary variable assignments.

In addition, the proposed satisfy-all procedure exploits a cache table, as is used in BDDs [17]. In traversing a search space, the procedure saves intermediate results in a cache table. When meeting a situation equivalent to a previous one, the proposed procedure accesses the corresponding result stored in the table. Each table entry has a field to indicate the possibility of

Manuscript received May 23, 2003; revised September 15, 2003, February 9, 2004, and April 14, 2004. This work was supported in part by the Korea Science and Engineering Foundation through the MICROS Center, in part by the Ministry of Science and Technology and the Ministry of Commerce, Industry and Energy through the System IC 2010 project, and in part by the Ministry of Information and Communication through the IT-SoC project. This paper was recommended by Associate Editor J. H. Kukula.

The authors are with the Division of Electrical Engineering, Department of Electrical Engineering and Computer Science, Korea Advanced Institute of Science and Technology, 373-1 Daejeon, Korea (e-mail: dk@ics.kaist.ac.kr; icpark@ics.kaist.ac.kr).

Digital Object Identifier 10.1109/TCAD.2004.841068

reuse in the current situation. The procedure can detect equivalent situations efficiently by managing the field and considering the number of undetermined variables and clauses. Although a similar method is proposed for an ATPG-based UMC algorithm in [18], it cannot be applied to general satisfy-all algorithms.

The rest of this paper is organized as follows. Section II discusses the differences between the related works and the proposed algorithm. Section III explains the proposed satisfy-all procedure, and Section IV describes the cache table. Section V presents the proposed UMC algorithm, and Section VI shows experimental results. Concluding remarks are made in Section VII.

II. Related Works

The main reason why BDDs cannot deal with large circuits is that BDDs are based on canonical representation requiring much memory. Therefore, some works have been made to reduce the memory requirement by using noncanonical representation.

A. Gupta and *et al.* proposed an algorithm that represents the sets of states in BDDs and the transition relation in CNF [19], [20]. The algorithm performs quantification with a procedure similar to ours. However, when a value is assigned to a variable, the algorithm checks whether the state-set BDD becomes a zero BDD. The proposed algorithm does not require the separate checking because the sets of states are also represented in CNF and the checking is implicitly included in the proposed satisfy-all procedure.

Formula-based algorithms that the sets of states and the transition relation are represented in formulas were proposed in [21] and [22]. They usually use their own representations, such as binary expression diagrams (BEDs) and reduced Boolean circuits (RBCs), and do quantification $\exists x.f(X)$ by generating two equations $f(X)|_{x=0}$ and $f(X)|_{x=1}$ and taking the disjunction of the two equations. Though the algorithms include some formula reduction techniques, they can be applied to a limited range of circuits that have a small number of inputs.

K. L. McMillan proposed a SAT-based UMC algorithm [15] that performs quantification $\exists x.f(X)$ by invoking a satisfy-all procedure with a CNF formula equivalent to $\sim f(X)$. When a satisfying assignment is found, the procedure redraws an implication graph. The redrawing is effective in cutting much search space from a satisfying assignment, but it consumes time and memory. The proposed procedure does not need the redrawing and exploits several enhancing techniques.

A cache table was used in an ATPG-based UMC algorithm called *success-driven learning* in [18]. In the algorithm, only the current state variables and the input variables can be decided, while the values of the other variables are determined by implication. Equivalent states are detected by observing the cut between the gates whose output values are determined and those not. This detection method can be applied only to the decision strategy. A SAT algorithm, however, can exploit various selection methods of decision variables in which any decision candidates can be selected to improve the performance. A new method that can be applied to the general cases is described in this paper.

```
1: SAT(f_CNF,V) {
2:   while(1) {
3:     if all variables in V are decided {
4:       return SAT;
5:     } else {
6:       choose one undetermined variable v from V;
7:       assign a value to v;
8:       if(deduce()=conflict) {
9:         analyze the conflict;
10:        add conflict clauses to f_CNF;
11:        if(backtrack()=fail) return UNSAT;
12:      }
13:    }
14: }
15:}
```

Fig. 1. Conventional SAT procedure.

III. Quantification by a Satisfy-All Procedure

A. Problem Definition

Given a formula represented in CNF, a SAT procedure finds an assignment that satisfies the formula or proves that no such assignment exists. The procedure is a very useful tool for managing CNF. A SAT-based UMC algorithm usually has a SAT procedure modified to perform operations like quantification.

The existential quantification is denoted as follows:

$$g(X) = \exists Y.\big(f(X,Y)\big) \tag{1}$$

where X and Y are sets of variables. The equation means an assignment of X satisfies $g(X)$ if and only if there is an assignment of Y such that the assignments of X and Y satisfy $f(X,Y)$. The existential quantification is one of major computations required in model checking. Computing the existential quantification is somewhat different from the conventional SAT problem, as the former requires all the satisfying assignments.

The problem to find all the assignments satisfying a formula is named satisfy-all in [8]. In this paper, we will extend the satisfy-all problem to include the quantification. The problem is now to obtain all the assignments of X that satisfy $g(X) = \exists Y.(f(X,Y))$ when a CNF formula $f_{\mathrm{CNF}}(X,Y)$ and variable sets X and Y are given. If Y is empty, this problem becomes equal to that of [8].

B. Conventional SAT Procedure

The SAT procedure determines if there is an assignment that satisfies a given formula represented in CNF. A CNF formula is the conjunction of clauses, each of which is the disjunction of literals, where a literal is a variable or its complement. An arbitrary Boolean formula f can be transformed into a CNF formula f_{CNF} with a representative variable v_f by using the procedure in [10] and [23], where $(f_{\mathrm{CNF}} \wedge v_f)$ is satisfiable if and only if f is satisfiable and $(f_{\mathrm{CNF}} \wedge \sim v_f)$ is satisfiable if and only if $\sim f$ is satisfiable. In this CNF form, an operation on CNF formulas can be implemented easily. Given a binary operation $\circ$, the CNF formula $(f \circ g)_{\mathrm{CNF}}$ is the conjunction of $f_{\mathrm{CNF}} \wedge g_{\mathrm{CNF}}$ and a CNF formula equivalent to $(v_{f\circ g} \leftrightarrow v_f \circ v_g)$.

Many SAT solvers have been proposed based on the Davis and Putnam procedure [11]. Fig. 1 is a pseudocode of the general SAT procedure. After a variable is selected and its value is

```
1: SatisfyAll(f_CNF,X,Y) {
2:   make G empty;
3:   while(1) {
4:     if all clauses are satisfied {
5:       if the improved procedure is being used {
6:         s' = Justify();
7:         insert the assignment s' of X into G;
8:       } else {
9:         insert the assignment of X into G;
10:      }
11:      insert an excluding clause to f_CNF;
12:      if(backtrack()=fail) return G;
13:    } else {
14:      choose an undetermined variable v from X ∪ Y;
15:      assign a value to v;
16:      while(deduce()=conflict) {
17:        analyze the conflict;
18:        insert conflict clauses to f_CNF;
19:        if(backtrack()=fail) return G;
20:      }
21:    }
22:  }
23:}
```

Fig. 2. Proposed satisfy-all procedure.

decided, $\mathrm{deduce}()$ reveals implications resulted from the decision. If a conflict occurs during $\mathrm{deduce}()$, the procedure inserts some conflict clauses after analyzing the conflict, and then backtracks to a decision point.

C. Proposed Satisfy-All Procedure

The satisfy-all procedure takes a CNF formula $f_{\mathrm{CNF}}(X,Y)$ and variable sets X and Y, and returns all the assignments of X that satisfy $g(X) = \exists Y.(f(X,Y))$. The major difference from the conventional SAT procedure is that the satisfy-all procedure does not stop when a satisfying assignment is found, because all the satisfying assignments are needed to construct $g(X)$. For a satisfying assignment found, the procedure stores the assignment and backtracks to continue as in the case that a conflict occurs.

Fig. 2 is a pseudocode of the proposed satisfy-all procedure, where G is the set of assignments of X that have an assignment of Y satisfying $f(X,Y)$, in other words, the assignments of X satisfying $g(X)$. When the procedure is completed, G has all the assignments satisfying $g(X)$. An assignment of X satisfying $g(X)$ can have several assignments of Y satisfying $f(X,Y)$. To search for such an assignment of X only once and reduce the searching time, an excluding clause is inserted as shown at the eleventh line in Fig. 2, which is equivalent to the complement of the found assignment of X. The statements in lines 5–7 will be described in Section III-E.

A satisfying assignment is detected when all the clauses are satisfied. A clause is called satisfied if one of the literal values in the clause is 1 and unsatisfied if all of them are 0. A clause which is neither satisfied nor unsatisfied is called undetermined. A clause has a field *num_satisfied* to indicate whether it is satisfied or not. The field denotes the number of literals of value 1 in the clause. A variable has two lists of clauses that include its positive and negative literals. When a variable is set or unset, the *num_satisfied* fields of the clauses belonging to the variable's list are incremented or decremented, respectively.

D. Converting a Set of Assignments Into a CNF Formula

The satisfy-all procedure in the previous subsection provides a set of all the assignments satisfying $g(X)$. However, the set should be converted into a CNF formula to use it in the proposed model checking algorithm. Since the set of assignments G can be regarded as a two-level logic circuit expressed in sum-of-products form, it can be transformed into a CNF formula by using the procedure in [10] and [23]. As a result, we obtain a CNF formula g_{CNF} with a representative variable v_g.

Before converting the assignment set into a CNF formula, a two-level logic minimizer is invoked to reduce the size of the assignment set and the resulting CNF formula. Without the minimizer, the number of assignments in G can be very large, degrading the performance. Any two-level logic minimizer can be used, but we have used *Espresso* [24] to obtain experimental results. To reduce the overhead of *Espresso*, the proposed procedure controls the number of cubes to be processed by *Espresso* at a time.

E. Justification

The satisfy-all procedure proposed in Section III-C is a general satisfy-all procedure. X and Y can be any subsets of the variable set and f_{CNF} can be any CNF formula. However, if X and Y are restricted to specific subsets and f_{CNF} is a CNF formula transformed from a circuit, the procedure can have better performance. In the preimage computation, X is a set of the current state variables, and Y is a set including the input variables I, the next state variables, and the intermediate or representative variables. In this section, we will show that a better satisfying assignment can be generated from a found satisfying assignment.

Let us assume that in Fig. 2, f_{CNF} is a CNF formula transformed from a circuit with a representative variable v_f or a combination of those formulas. The values of the variables in X and I determine the values of the variables in the difference set, Y–I. To satisfy $f(X,Y)$, the values of the variables in X and I should meet a condition: they should make $v_f = 1$. For a general subset of variables, one more condition should be met: the values of the variables should cause no conflict. Since the variables in X and I are independent variables, they cannot cause any conflict if they make $v_f = 1$. Therefore, it is sufficient to check the first condition.

When an assignment of X and I leads to the values of the variables in Y-I satisfying $f(X,Y)$, not all of the variables in X and I are necessary to meet the condition. The variables but those needed to make $v_f = 1$ are not necessary to satisfy $f(X,Y)$. The unnecessary variables can have unknown values instead of 0 or 1. An unknown variable in X has twice as much search space as a 0- or 1-valued variable can have.

After finding a satisfying assignment, the proposed satisfy-all procedure classifies the unnecessary variables by using line justification of ATPG [16]. Originally, the $\mathrm{Justify}()$ routine [16] in line justification determines the values of input variables for an output value of a gate. In the proposed procedure, the $\mathrm{Justify}()$ routine only selects the input variables that are necessary to make the output variable have the desired value. For example, if the output of an AND gate is 0, the justification procedure selects an input variable whose value is 0. Because all the clauses are

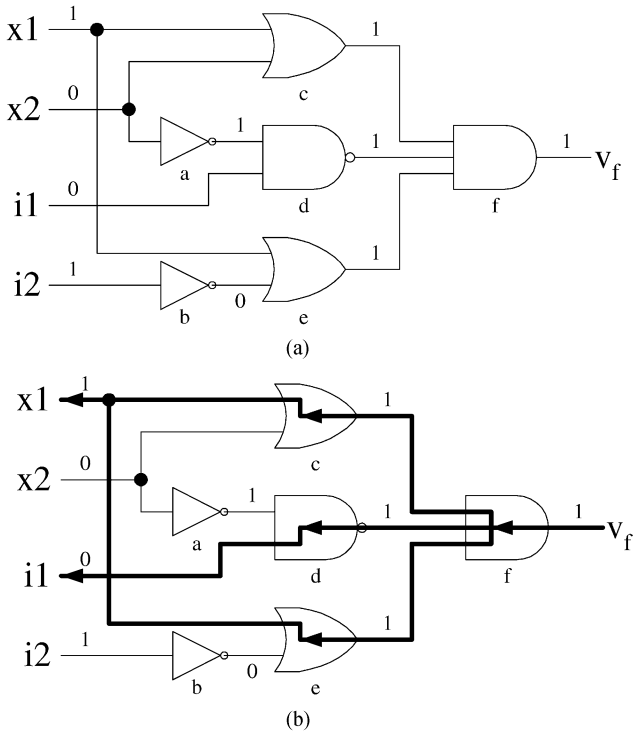


Fig. 3. (a) Satisfying assignment is found. (b) A better assignment is generated by the Justify() routine.

satisfied without conflict, it is guaranteed that there is at least one input variable whose value is 0. If the output of an AND gate is 1, Justify() selects all the input variables of the gate. As all the clauses are satisfied without conflict, all the input variables have value 1. The Justify() routine selects variables by traversing from v_f to the variables in X and I. The X and I variables selected by the Justify() routine are the variables necessary to satisfy $f(X, Y)$.

Fig. 3 is an example, where $X = \{x1, x2\}$ and $I = \{i1, i2\}$. The assignment $(x1, x2, i1, i2) = (1, 0, 0, 1)$ sets v_f to 1 in Fig. 3(a), in other words, it satisfies f. However, in Fig. 3(b), the Justify() routine reveals that $(x1, x2, i1, i2) = (1, \text{-}, 0, \text{-})$ is enough to set v_f to 1, where "-" means an unknown value. Therefore, $(x1, x2) = (1, \text{-})$ is also a satisfying assignment. The assignment $(x1, x2) = (1, \text{-})$ is better than the original assignment $(x1, x2) = (1, 0)$ because the former covers more solutions.

The proposed satisfy-all procedure exploits the Justify() routine. After finding a satisfying assignment, the proposed procedure calls the Justify() routine to generate a better assignment as shown in the sixth line of Fig. 2. The new assignment is added to G and an excluding clause is induced from it. The information on the circuit structure is required to use the Justify() routine, which in the proposed procedure is extracted and kept when f_{CNF} is generated.

In some cases, the output of a gate can be justified more than once, as the Justify() routine can select one of many possible inputs. For example, if the output of an AND gate is 0, any input variable whose value is 0 can be selected. Selecting a different variable may induce a different assignment. Exploring all the selections gives all the assignments that can be generated from an original assignment, but may take too much time. Therefore, only one variable is selected and explored for each gate in the proposed algorithm. In the proposed procedure, each variable v is associated with a set of transitive fanin X variables, X_v. Gates are justified in depth-first search order. If the Justify() routine is to select a variable to justify a gate, it computes a set $D_v = X_v - X_n$ for each candidate variable, where X_n is the set of necessary X variables found already. The routine chooses a variable that has the smallest D_v.

McMillan's algorithm uses a similar method [15], where an implication graph is redrawn for backtracking. Instead of redrawing the graph, the proposed procedure uses the given circuit structure. As the proposed method cannot be used for the case that only the transformed CNF formula f_{CNF} is given without the original formula or circuit, McMillan's method can be used to substitute the proposed justification technique if the variable representing the original formula such as v_f in Fig. 3 is specified.

F. Managing Excluding and Conflict Clauses

The performance of a satisfy-all procedure becomes lower as the procedure finds more satisfying assignments. The reason is related to excluding clauses inserted. The more clauses, the more time the procedure consumes to manage them. To overcome this problem, the proposed procedure compresses the sets of the satisfying assignments and the excluding clauses periodically. *Espresso* is also used for this compression. A found satisfying assignment is saved in G_{pre}. When G_{pre} has a specified number of satisfying assignments, the proposed procedure removes the excluding clauses corresponding to the assignments in G_{pre}, and processes G_{pre} by invoking *Espresso*. Then, the assignments in the compressed G_{pre} are transformed into new excluding clauses and inserted to f_{CNF}. The assignments in G_{pre} are moved to G. This method reduces the number of assignments in G and the number of excluding clauses, leading to reduced memory usage and processing time.

As excluding clauses are similar to conflict clauses, the two clauses are processed similarly. When a SAT procedure meets a conflict, it generates a conflict clause to avoid meeting similar conflicts. Excluding clauses prevent the proposed procedure from finding the same satisfying assignment again. In modern SAT procedures, conflict clauses are deleted if they are decided to be irrelevant. In the proposed procedure, irrelevant excluding clauses are also deleted. How to decide an excluding or conflict clause to be deleted is dependent on the specific details of the base SAT procedure.

When examining whether all the clauses are satisfied at the forth line of Fig. 2, the proposed procedure does not consider the satisfying states of excluding clauses. Excluding clauses represent that a part of solution space is not required to be searched, not that the part must not be searched. Sometimes, a better assignment is achieved by disregarding the states of excluding clauses. For example, let us consider a case that (1, 0, 1) is a satisfying assignment for state variables $(x1, x2, x3)$ and an excluding clause $\text{cl} = (\sim x1 \vee x2 \vee \sim x3)$ is added. When all the clauses except the excluding clause cl are satisfied by an assignment of (1, 0, x), the assignment is not considered as a satisfying assignment if the procedure cares for the satisfying state of cl.

In the forth line of Fig. 2, the proposed procedure does not care for the satisfying states of conflict clauses either. Conflict clauses can be classified into two categories. In category A, a conflict clause is induced only by the clauses in the original equation and other clauses in category A. If all the clauses in the original equation are satisfied, all the conflict clauses in A are also satisfied. Thus, it is not needed to check them. Category B includes conflict clauses whose induction depends on excluding clauses and other clauses in B. Since it is not necessary to consider the satisfying states of excluding clauses, those of the conflict clauses in this category are not necessary either.

IV. Using a Cache Table in the Satisfy-All Procedure

A. Motivation

The BDD procedure and the satisfy-all procedure are similar to each other, as both of the procedures traverse down after deciding the value of a variable. If special cases such as conflicts in the satisfy-all procedure and leaf nodes in the BDD procedure are encountered, they traverse up. One of the most efficient techniques used in the BDD procedure is to use a cache table [17]. After a vertex v is traversed, the result for the subgraph with root v is saved in the cache table. When the procedure traverses v again, it refers to the table and uses the saved result instead of evaluating it again. However, the satisfy-all procedure has a difficulty in employing this technique. The difficulty is mainly caused by the fact that table-referring time is not clearly defined in the satisfy-all procedure. BDDs have sufficient information on whether the previous results can be reused at a decision point, while the CNF does not. Therefore, the satisfy-all procedure should detect the case while traversing the search space. Actually, the table in [17] is a hash-based cache table. The proposed procedure exploits the cache characteristic of the table, i.e., storing intermediate results and reusing them.

B. Basic Principles

To construct a table and reuse its contents, we have to examine whether a table entry can be used in the current time. Before explaining a method to do it, some terminologies are defined for clarity. All the following theorem and definitions are related to a satisfy-all problem with a variable set V and a formula f expressed in CNF.

Definition 1: A variable is called undetermined if it has an unknown value.

Definition 2: A situation A is a pair of a set $V_A \subset V$ of variables that are not determined and an assignment $g_A : (V - V_A) \rightarrow \{0,1\}$ on $V - V_A$.

Definition 3: Given an assignment $g : V \rightarrow \{0,1\}$, its restricted assignment on $V_A \subset V, g_{\mathrm{VA}} : V_A \rightarrow \{0,1\}$ is

$$g_{\mathrm{VA}}(v) = g(v), \qquad \text{for } v \in V_A. \tag{2}$$

Definition 4: A situation A induces a satisfying assignment $s : V \rightarrow \{0,1\}$ if s satisfies f and $s(v) = g_A(v)$ for $v \in V - V_A$.

Definition 5: Two situations A and B are called equivalent when the following conditions are satisfied:

1) $V_A = V_B$.
2) For each satisfying assignment s_A induced by A, there is a satisfying assignment s_B induced by B that has the same restricted assignment on V_A as s_A.
3) For each satisfying assignment s_B induced by B, there is a satisfying assignment s_A induced by A that has the same restricted assignment on V_B as s_B.

Definition 5 means equivalent situations have the same satisfying assignments for undetermined variables. If the satisfying assignments for a situation are saved in a table and another situation equivalent to it is met later, the satisfying assignments to be induced by the later situation can be obtained from the table entry without traversing the search space deeper. The following theorem gives a sufficient condition for two situations to become equivalent.

Theorem 1: If two situations A and B have the same sets of undetermined variables (UVs) and the same sets of undetermined clauses (UCs), the two situations are equivalent.

The proof is straightforward. The assignments of UVs depend only on UCs. Since the two situations have the same set of UCs, a satisfying assignment of UVs in one situation is also a satisfying assignment in the other situation. Theorem 1 implies that a table entry can be selected by comparing the set of UVs and the set of UCs.

The definition of equivalent situations does not cover all of the equivalent search states. And Theorem 1 cannot reveal all of the equivalent situations, because the CNF formula has less information than BDDs. It would be another difficult problem to detect all of the equivalent search-states with CNF. The proposed cache scheme, therefore, uses a subset of the equivalent states.

C. Table Management

It is time consuming to compare the current sets of UVs and UCs with those in the table entries. This section describes how to detect equivalent situations with ease. In conclusion, equivalent situations can be detected by comparing only the numbers of UVs and UCs.

A table entry consists of a UV set, a UC set, and intermediate results, and has its unique identifier for read. Entries whose UV sets are of the same size are connected by a linked list. Each variable has a list of table entries where it is undetermined. Each clause has a similar entry list, too. When an entry is inserted into the table, it is connected to a proper linked-list and added to the lists of UVs and UCs.

A table entry has a field, *num_mark*, to represent the number of its UVs and UCs that are determined and satisfied in the current situation. This field is incremented or decremented when the value of a variable is set or unset and a clause becomes satisfied or undetermined, respectively. In a situation, a table entry whose *num_mark* is 0 has a special meaning. The sets of UVs and UCs of the entry are subsets of the current situation. In other words, it is possible to use the entry in the situation. A table entry whose *num_mark* is not 0 cannot be used in the situation. As $\mathrm{num\_mark} = 0$ guarantees that the sets in the table entry are subsets of the situation, it is not necessary to compare all the elements in the sets. By checking the size of the sets, we can know whether the sets are equivalent to each other.

```
1: SatisfyAllCache(f_CNF,X,Y) {
2:   make G empty;
3:   I = input variables in Y;
4:   while(1) {
5:     if all clauses are satisfied {
6:       insert the assignment of X into G;
7:       if(backtrack()=fail) return G;
9:     } else {
10:      choose an undecided variable v from X ∪ Y;
11:      assign a value to v;
12:      while(1) {
13:        while(deduce()=conflict) {
14:          analyze the conflict;
15:          insert conflict clauses to f_CNF;
16:          if(backtrack()=fail) return G;
17:        }
18:        if(access_table()=true) {
19:          add_assignments_from_table();
20:          if(backtrack()=fail) return G;
21:        } else {
22:          add_table();
23:          break;
24:        }
25:      }
26:    }
27:  }
28:}
```

Fig. 4. Proposed satisfy-all procedure with a cache table.

Hence, it is not difficult to examine whether a table entry can be reused. Every time a variable is set or unset, the *num_satisfied* field of some clauses and the *num_mark* field of some table entries are updated. When the procedure meets a situation, it selects a linked-list of the table by using the number of the current UVs. Then the procedure searches for an entry whose *num_mark* is 0 and whose UC set has the same size as the current UC set.

D. Implementation

Fig. 4 describes the satisfy-all procedure equipped with a cache table. The cache table is accessed after $\mathrm{deduce}()$ is completed because the situations found in doing $\mathrm{deduce}()$ are redundant. In $\mathrm{access\_table}()$, a linked-list is selected by the number of UVs. In traversing the list, the function searches for an entry that has *num_mark* of 0 and the same number of UCs as that of the current situation. If it finds one, it returns true. Otherwise, it returns false. When $\mathrm{access\_table}()$ does not find a proper entry, $\mathrm{add\_table}()$ generates an entry to be added into the table and connected to the linked-list. It adds the entry id to the entry lists in the UVs and UCs.

Each table entry keeps the range of the satisfying assignments of X variables in G that the situation of the entry induces. When an entry is generated, the next insertion point in G is saved in the entry. When the procedure backtracks at a decision level, the last insertion point in G is saved in the table entries generated at the decision level.

If a proper entry is found in $\mathrm{access\_table}()$, the satisfying assignments induced by the entry are used to generate new assignments. If the current situation consists of $V_A \subset V$ and an assignment $g_A : (V - V_A) \rightarrow \{0, 1\}$ and s is an assignment in the table entry, a new assignment s_{NEW} is generated as follows:

$$s_{\mathrm{NEW}}(v) = \begin{cases} s(v), & v \in V_A \\ g_A(v), & \text{otherwise.} \end{cases} \tag{3}$$

If the new assignment does not have a variable in the UV set, the procedure backtracks to the maximum decision level of the variables in the assignment. Otherwise, the procedure backtracks by one level.

Fig. 5 illustrates the data structure of the proposed procedure. Each variable has its identification number and value. The value can be 0, 1, or unknown (x in the figure). Each variable has a list of clauses to be included. The term *entries* in a variable means a list of cache table entries whose UV sets have the variable. A clause has its identification number and a list of cache table entries. A clause keeps the number of its literals of value 1 in the *num_satisfied* field. G has the assignments of X variables. A cache-table entry is accessed first by the number of UVs. Entries with the same number of UVs are connected in a form of a linked list. An entry has the locations of its first and last satisfying assignments in G. Two lists of undetermined variables and undetermined clauses are also included. The *num_mark* field indicates whether the entry can be used.

E. Enhancing Techniques

This section deals with several techniques that can be used with a cache.

1) Cache Access Region: Every cache entry does not have the same hit ratio, as situations equivalent to a certain cache entry may occur more frequently. The number of satisfying assignments that a cache can accommodate is also important. It would be beneficial to maintain only cache entries associated with a high hit ratio and many satisfying assignments. The benefit of a cache entry, however, is not known *a priori* when the proposed procedure decides whether to store the cache entry. It is difficult to predict how many hits will occur at the cache entry. The proposed procedure, therefore, exploits the predetermined cache access region and accesses the cache table only when the search state belongs to the region.

The proposed procedure specifies the cache access region in the number of UVs that affects the two factors significantly. The two factors, however, have conflicting effects to the number of UVs. A cache entry with a less number of UVs usually has higher hit ratio, and one with a more number of UVs is associated with more satisfying assignments. To solve this conflict, a cost function is defined as the product of the hit ratio and the number of satisfying assignments. The cache access region is determined by experiments.

2) Justification: If the satisfy-all procedure uses the $\mathrm{Justify}()$ routine described in Section III-E, a table entry keeps the range of the assignments in G_{SEED} instead of G. A satisfying assignment saved in G is different from a found assignment on X variables due to the $\mathrm{Justify}()$ routine. The original X assignment and the input-variable assignment is saved in G_{SEED}. One more step is needed to reuse the results saved in a table entry. After generating a new assignment s_{NEW} from an assignment s in G_{SEED}, the proposed procedure deduces the values of other variables from s_{NEW}. As an assignment in G_{SEED} includes an input-variable assignment, it is guaranteed that all the clauses are satisfied by the implication. Then the procedure uses the $\mathrm{Justify}()$ routine to make a new satisfying assignment s'_{NEW}.

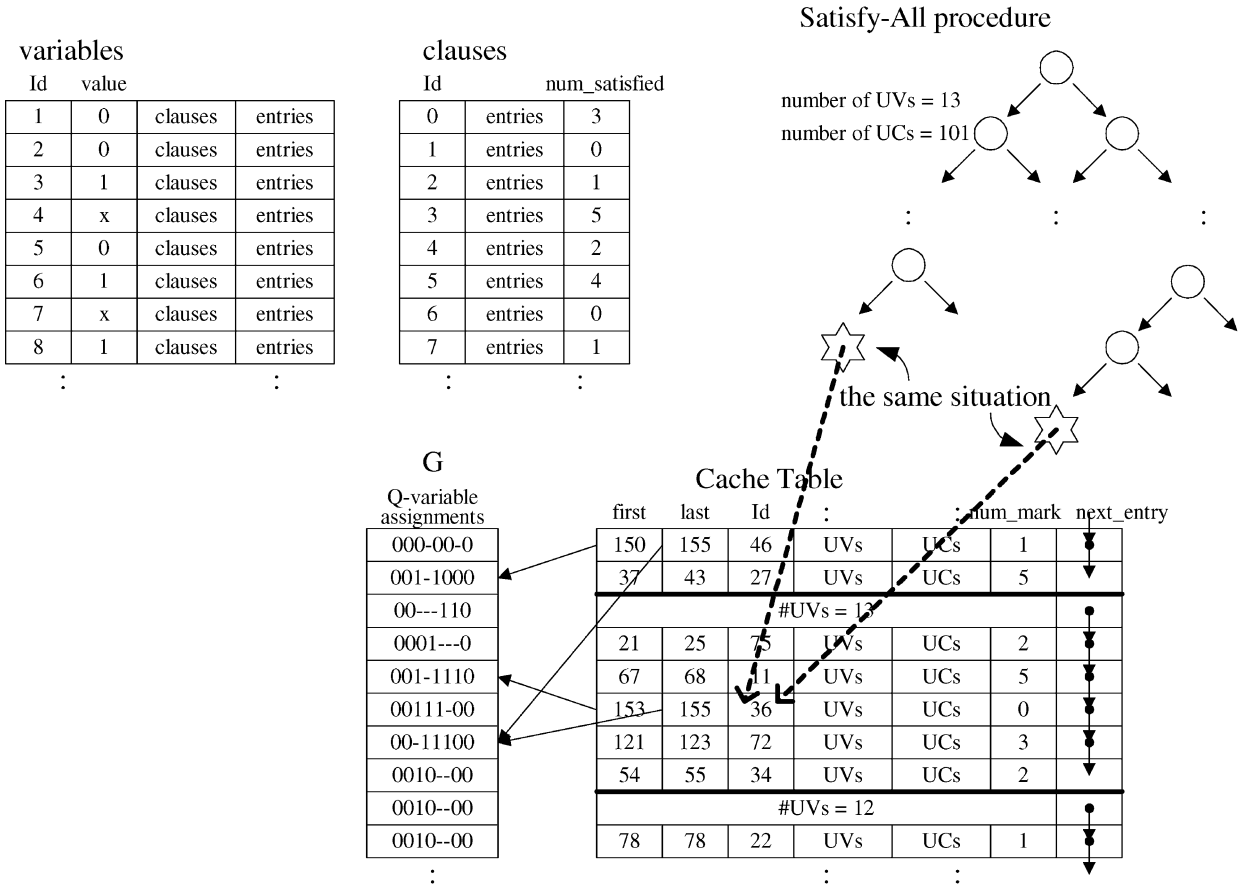


Fig. 5. Data structure of the proposed satisfy-all procedure.

Introducing G_{SEED} may lead to memory explosion. The memory requirement, however, is not much in real problems because the justification scheme prunes much search space. Moreover, some cache entries are invalidated in order to accept excluding clauses, which will be described next in detail. An element in G_{SEED} can be removed if it is not included by any valid cache entry.

3) Excluding Clauses: As adding excluding clauses can change the satisfying assignment set, excluding clauses should be treated differently from conflict clauses. To reuse the satisfying assignments in an equivalent situation, this change should be considered. The proposed algorithm overcomes this problem by counting the undetermined excluding clauses when obtaining the number of the UCs. However, this counting leads to another problem. When adding excluding clauses, satisfying states of the excluding clauses are not known for every cache entry generated previously. We can know only that the excluding clauses are undetermined in the cache entries related to the decision path that produces those excluding clauses. The other cache entries should be invalidated.

4) Compressing Solutions: Satisfying assignments that belong to different cache entries should be compressed separately. When all the satisfying assignments are found under a cache entry and no cache entry is generated at the lower decision levels, the satisfying assignments under the cache entry are compressed and excluding clauses are changed. To reduce the time to invoke *Espresso*, satisfying assignments are compressed only when their number is greater than a specified limit. When the justification is used, however, reused is not satisfying assignments in G but seeds in G_{SEED}. Therefore, satisfying assignments in G under different cache entries can be compressed together. Seeds are not allowed to be compressed, but a seed can be deleted only when it does not belong to valid cache entries any longer.

5) Hit Between Nonequivalent Situations: In [18], equivalent search states are defined generally to reuse a cache entry in more cases. As the definition is not adequate for general SAT procedures, a restricted definition that can be applied to SAT procedures well is used in this paper. In addition, a technique complementing the restricted definition is introduced to reuse the satisfying assignment in a cache entry in more cases.

In the previous descriptions, satisfying assignments in a previous situation can be reused in the current situation only if the two situations are equivalent. In that case, the satisfying assignments under the previous situation produce all the satisfying assignments under the current one. However, there are cases that a previous situation is not equivalent to the current situation, but that the satisfying assignments under the previous situation are associated with a subset of the satisfying assignments under the current situation. In the cases, the satisfying assignments in the previous situation can be reused in the current situation, too. As the satisfying assignments generated from the previous situation are only a subset of the total satisfying assignments that can be found under the

current situation, the procedure continues to explore deeper search space instead of backtracking.

There are two cases in which this subset relation occurs. The first case is that the UV sets of the previous and current situation are the same, but the UC set of the current situation is a subset of that of the previous situation. The other case is that the UC sets are the same, but the UV set of the current situation is a superset of that of the previous situation. The former case can be detected by comparing the difference of the sizes of the UC sets and the *num_mark* field. Let us assume that the sizes of the UV sets of the current situation and a table entry are the same and the sizes of their UC sets are $\#\mathrm{UC}_{\mathrm{current}}$ and $\#\mathrm{UC}_{\mathrm{entry}}$. The subset relation occurs when the *num_mark* field of the table entry equals to $\#\mathrm{UC}_{\mathrm{entry}} - \#\mathrm{UC}_{\mathrm{current}}$. The latter case can be detected by comparing the sizes of the UV sets. A table entry with the same UC set size, $num\_mark = 0$, and less UV set size can produce a subset of the satisfying assignments under the current situation.

V. SAT-Based UMC Algorithm

The conventional UMC algorithm is based on BDDs. In this section, we will present a SAT-based UMC algorithm in which BDD operations are substituted with CNF operations. Logical operations, such as $\sim$ and $\vee$, can be implemented easily with the representative variables. After constructing a CNF formula q_{CNF} that relates the representative variable of the resulting CNF formula and the representative variables of the operand CNF formulas, the procedure returns a conjunction of q_{CNF} and the CNF formulas of the operands.

A. Preimage and Fix-Point Calculation

Some CTL operators require preimage computations. A preimage is computed as follows:

$$S(V) = \exists V'.\big(T(V,V') \cdot S'(V')\big) \tag{4}$$

where $T(V,V')$ is the transition relation, V and V' are the sets of the current and next state variables, $S(V)$ and $S'(V')$ are the sets of the current and next states, and $S(V)$ is the preimage of $S'(V')$. In our algorithm, $S'(V')$ and $T(V,V')$ are represented in CNF, and quantification is done by the satisfy-all procedure described in the previous section. In fact, $T(V,V')$ and $S'(V')$ do not depend only on the current and next state variables, but have the input and output variables and some intermediate variables inserted when their CNF formulas are constructed. Those variables must be quantified, too.

Fig. 6 is a pseudocode that computes a preimage, where $D(f)$ is a function that returns a set of variables on which the formula f depends. At the first time, S'_{CNF} depends on the current state variables, so the variables should be substituted by the corresponding next state variables. The satisfy-all procedure in Sections III and IV returns a set of the assignments of the current state variables G, which is equivalent to the preimage of S'_{CNF}. The set is processed by *Espresso* to reduce the number of assignments. At last, the reduced set is converted into a CNF formula to be returned.

```
1: preImage({S'CNF, vS'},{TCNF, vT }) {
2:   X = V;
3:   Y = D(TCNF) – V;
4:   substitute variables of V in S'CNF with the
       corresponding variables of V';
5:   Y = Y ∪ D(S'CNF);
6:   obtain a CNF formula qCNF equivalent
       to (vf↔vS' ∧vT);
7:   fCNF = S'CNF∧TCNF∧vf∧qCNF;
8:   G = SatisfyAll(fCNF,X,Y);
9:   G' = espresso(G);
10:  return convertCNF(G');
11:}
```

Fig. 6. Preimage calculation.

```
1: fixPointEF({pCNF,vp},TCNF) {
2:   FCNF = pCNF; vF = vp;
3:   RCNF = pCNF; vR = vp;
4:   while(!empty({FCNF, vF})) {
5:     obtain a CNF formula qCNF equivalent
         to (vTtemp ↔ ~vR);
6:     change the intermediate variables in {FCNF,vF};
7:     TtempCNF = TCNF∧RCNF∧qCNF;
8:     {FpreCNF,vFpre} =
         preImage({FCNF,vF},{TtempCNF,vTtemp});
9:     obtain a CNF formula qCNF equivalent
         to (vRnew↔vR∨vFpre);
10:    RnewCNF = RCNF∧FpreCNF∧qCNF;
11:    RCNF = RnewCNF; vR = vRnew;
12:    FCNF = FpreCNF; vF = vFpre;
13:  }
14:  return {RCNF,vR};
15:}
```

Fig. 7. Fix-point calculation in the case of EF p.

A fix point is obtained by successively applying the preimage calculation as described in Fig. 7 that shows a pseudocode calculating a fix point in the case of EF p. Although the pseudocode presents only the case of EF operation, it can be easily transformed to other operators like EG and EU. The transition relation T_{CNF} and a set of states p_{CNF} are assumed to be given in CNF. In the code, F_{CNF} is the set of frontier states, and R_{CNF} is the set of reached states. In order not to visit the same states more than once, the proposed procedure uses the temporary transition relation T_{tempCNF}, which is a conjunction of T_{CNF} and the complement of R_{CNF}. As shown in the sixth statement of the code, the variables in F_{CNF}, except the state variables, should be changed into new variables to prevent the overlapping with those variables in R_{CNF}. After a preimage CNF formula is constructed, a new reached state set R_{newCNF} is constructed by the disjunction of R_{CNF} and the preimage. When F_{CNF} is empty, the procedure ends to return R_{CNF}.

B. Inclusion and Empty Checking

The inclusion checking procedure is straightforward, so its pseudocode is not given in this paper. Checking whether a set of states represented by a formula f_0 is included in a set of states represented by a formula f_1 is equivalent to determining whether the formula $(f_0 \wedge \sim f_1)$ is satisfiable. Therefore, the inclusion-checking procedure is to construct a CNF formula of $(f_0 \wedge \sim f_1)$ and to determine its satisfiability by a conventional

TABLE I
CHARACTERISTICS OF ISCAS'89 CIRCUITS

Circuits	#In	#Out	#Gate	#DFF
s713	35	23	393	19
s1269	18	10	569	37
s1423	17	5	657	74
s1512	27	21	780	57
s4863	49	16	2342	104
s13207	31	121	7951	669
s38584	12	278	19253	1452

SAT solver. The empty-checking procedure is similar to the inclusion checking except that the former determines whether the input formula f, instead of $(f_0 \wedge \sim f_1)$, is satisfiable.

C. Enhancing Safety-Property Checking

Safety properties ensure that some bad states are never reached [2]. They are usually written as AG $\sim p$ and encountered many times in practical model-checking problems. One of the well-known techniques that improve the safety-property checking is to check whether the reached set has some states included in the initial states at each iteration. By this technique, we can know whether the safety property is false earlier than a fix point is found.

VI. EXPERIMENTAL RESULTS

The performance of the proposed algorithm is compared with those of a BDD-based UMC tool and another SAT-based UMC algorithm in [15]. The proposed algorithm is written in C and C++ languages and compiled by g++ with option $-$O3. The proposed satisfy-all procedure is implemented by modifying Chaff [13]. The experiment is done on a SUN Blade 2000 workstation that has a 900 MHz processor and 1 GB of memory. The BDD-based model checking tool compared is SMV [25] because it is one of the most efficient model checking tools ever known. All the results are obtained from backward fix-point computation.

A. Test Sets

Some of ISCAS'89 benchmark circuits are used for target designs. Table I shows the characteristics of the benchmark circuits. The first column lists the circuits. The second–fifth columns show the number of inputs, outputs, combinational gates, and D flip/flops, respectively. All the checked properties are safety properties with a form AG $\sim p$, where p is a state. We assume that all the state variables are set to 0 initially.

B. Statistics About a Cache Table

Fig. 8 shows statistical data related to a cache table when a preimage of a state in circuit s1423 is calculated by the proposed satisfy-all procedure. Fig. 8(a) includes the number of misses and hits and the average number of satisfying assignments per hit. Fig. 8(b) shows the hit ratio and the expected number of reused satisfying assignments that is the product of the hit ratio and the average number of satisfying assignments per hit. The horizontal axis represents the ratio of the number of UVs to the number of the total variables. The figures indicate that cache

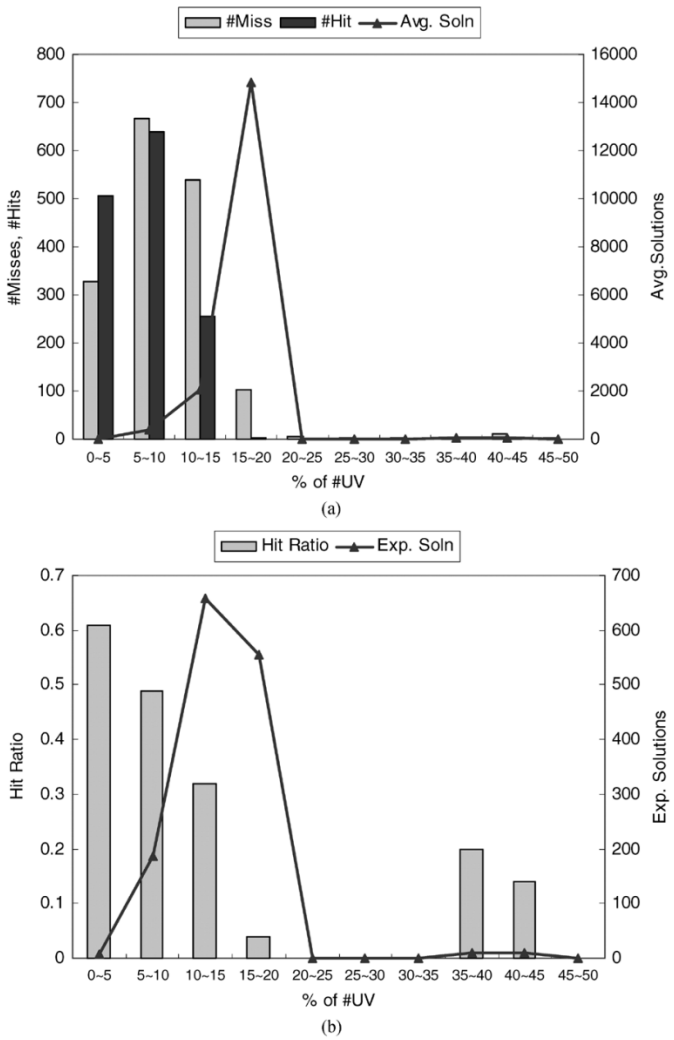


Fig. 8. Statistic of a cache table. (a) Number of hits and misses and the average number of solutions per hit. (b) Hit ratio and the expected number of reused solutions.

entries in which 5%–25% of the total variables are undetermined are more worth maintaining than other entries.

The regions that have more than 50% undetermined variables are not shown because of a very low access and a low hit ratio. In a SAT procedure, a few first decisions imply many variables. Therefore, the procedure spends most of the time when there are small undetermined variables and, thus, the number of accesses to the regions with many undetermined variables is usually low. Since sets of undetermined variables in the regions are much different, their hit ratios are very low, too.

C. Results With Options

Table II shows the processing time and memory usage that the proposed procedure takes with several options. The first row shows four options that can be adopted by the proposed procedure: employing a cache table, justification, using excluding clause, and excluding clause management. The column below each option indicates whether the option is used or not: X means it is not used and O means otherwise. The procedures are applied to three circuits: s713, s1423, and s38584. A false safety property is generated arbitrarily for each circuit. In the table, the term

TABLE II
PROCESSING TIME AND MEMORY USAGE WITH OPTIONS

	Cache	Justification	Excluding	Excluding Management	s713		s1423		s38584	
					Time	Mem	Time	Mem	Time	Mem
Proposed Algorithm	X	X	X	X	TO	-	TO	-	TO	-
	X	X	O	X	93.12	38	TO	-	TO	-
	X	O	O	X	< 1	< 1	5.34	9	188.88	55
	X	O	O	O	< 1	< 1	8.27	9	241.38	53
	O	X	X	X	672.50	497	TO	-	TO	-
	O	O	X	X	205.92	129	TO	-	TO	-
	O	O	O	X	< 1	< 1	14.21	10	172.98	76
	O	O	O	O	< 1	< 1	15.21	9	195.61	72

(Memory : MB, Time : seconds)

TABLE III
EXPERIMENTAL RESULTS FOR THE PROPOSED ALGORITHM WITH VARIOUS WEIGHTING SCHEMES

(circuit) _(property)	Proposed + weight X		Proposed + weight Y		Proposed + no-weight	
	Time	Mem	Time	Mem	Time	Mem
s1269_0	20.75	24	11.30	24	12.30	24
s1269_1	4.17	13	3.96	13	5.15	9
s1269_2	18.25	24	11.82	24	12.20	24
s1269_3	29.97	27	19.26	27	17.16	27
s1269_4	30.51	27	18.85	27	18.42	27
s1423_0	<1	<1	<1	<1	<1	<1
s1423_1	11.80	8	407.81	27	15.21	9
s1423_2	<1	<1	<1	<1	<1	<1
s1423_3	46.92	27	39.69	27	37.60	34
s1423_4	52.37	17	235.90	22	69.77	16

(Memory : MB, Time : seconds)

TO and <1 mean that the algorithm takes more than 1 h and less than 1 s or one MB memory, respectively. The table shows each option affects the overall performance significantly. Excluding clauses and justification play an important role in proving properties, which means that the two methods reduce the number of cubes and literals significantly. The excluding clause management reduces memory usage. The employment of a cache table reduces the processing time for s38584, whereas it increases the processing time for s1423. The cost to manage the cache table is more than the benefit in small circuits. The results suggest that it would be a good strategy not to use the excluding clause management and the cache table at first. If the first try fails due to memory explosion, a second try is performed with adopting the management. If the first try consumes too much time, the cache technique is utilized. In the experiment, *Espresso* takes less than 5% of the total time.

Table III compares various weighting schemes that the proposed algorithm can exploit. In the table, the first column indicates the circuit and the property to be proved. Five false safety properties are generated arbitrarily for s1269 and s1423. The columns with "*Proposed+weight X*" ("*Proposed+weight Y*") show the results obtained with giving higher weights to $X(Y)$ variables in (1). The last two columns are generated with no variable weighting and all of the options in Table II. The table shows that the variable weighting affects the performance much, but it is not obvious which weighting is better. No one always beats the others. The no-weighting scheme, however, gives robust results.

Sheng and Hsiao's algorithm selects a decision variable among X variables or input variables [18]. The selection strategy can result in a simple cache scheme that stores less information than the proposed cache scheme. Their algorithm, however, cannot exploit many brilliant variable selection strategies adopted in modern SAT solvers. As selecting a decision variable from X variables is similar to giving higher weight to X variables, we can use the results of $+$ *weight X* and $+$ *no-weight* schemes to compare Sheng and Hsiao's algorithm to the proposed one. This indirect comparison is performed as Sheng and Hsiao's algorithm is based on ATPG and BDDs and no code implementing their algorithm is available in the public domain. It is not clear which one is better in Table III. The results, however, show that a cache scheme equipped with an unbiased-variable-selection strategy can give better results for some circuits such as s1423_3.

D. Comparison of SMV and McMillan's Algorithm

To compare the performance of the proposed algorithm, McMillan's algorithm and SMV, six benchmark circuits are used: s1269, s1423, s1512, s4863, s13270, and s38584. Five false safety properties are generated arbitrarily for each circuit.

Table IV shows the processing time and the memory usage. In the table, the term MO means that the algorithm consumes more than 0.8 GB of memory, and TO means that it takes more than 2 h. McMillan's algorithm is associated with *Espresso* reduction after a preimage computation because otherwise memory explosion occurs for many cases. The proposed algorithm gives no weight to variables and employs all the options in Table II.

Compared to the proposed algorithm, McMillan's algorithm takes more time because the algorithm wastes time in redrawing implication graphs and searching equivalent spaces. For large

TABLE IV
EXPERIMENTAL RESULTS FOR SMV, MCMILLAN'S ALGORITHM, AND THE PROPOSED ALGORITHM

(circuit) _(property)	SMV		McMillan's [15] + Espresso		Proposed	
	Time	Mem	Time	Mem	Time	Mem
s1269_0	-	MO	14.62	169	12.30	24
s1269_1	-	MO	5.49	81	5.15	9
s1269_2	-	MO	13.57	161	12.20	24
s1269_3	-	MO	20.68	187	17.16	27
s1269_4	-	MO	19.97	171	18.42	27
s1423_0	3.84	9	<1	<1	<1	<1
s1423_1	9.42	22	TO	-	15.21	9
s1423_2	4.01	9	1.83	29	<1	<1
s1423_3	4.73	13	TO	-	37.60	34
s1423_4	5.25	14	TO	-	69.77	16
s1512_0	4.63	13	5.97	116	< 1	< 1
s1512_1	4.54	12	5.42	106	< 1	< 1
s1512_2	4.54	14	5.45	101	< 1	< 1
s1512_3	4.59	11	4.92	95	< 1	< 1
s1512_4	4.56	14	5.43	97	< 1	< 1
s4863_0	-	MO	-	MO	TO	-
s4863_1	-	MO	-	MO	158.67	71
s4863_2	-	MO	-	MO	230.70	89
s4863_3	-	MO	-	MO	1361.49	129
s4863_4	-	MO	-	MO	1548.68	227
s13207_0	-	MO	25.42	163	11.92	18
s13207_1	-	MO	81.07	618	38.46	22
s13207_2	-	MO	98.94	747	48.13	25
s13207_3	-	MO	103.28	758	48.68	25
s13207_4	-	MO	-	MO	TO	-
s38584_0	-	MO	-	MO	195.61	72
s38584_1	-	MO	-	MO	93.29	57
s38584_2	-	MO	-	MO	104.07	65
s38584_3	-	MO	-	MO	105.10	65
s38584_4	-	MO	-	MO	TO	-

(Memory : MB, Time : seconds)

circuits, McMillan's algorithm takes more memory because it does not have an excluding clause management scheme. The proposed algorithm can prove more properties and circuits than SMV. The main reason for SMV's failure is due to memory explosion. As the proposed algorithm usually takes more time than SMV, there is a tradeoff between memory usage and processing time.

VII. CONCLUSION

In this paper, we have presented a SAT-based UMC algorithm. The sets of states and the transition relation of a target design are represented in CNF, which is noncanonical and memory efficient. The quantification, one of the most frequent operations encountered in model checking, is processed by a satisfy-all procedure that generates all satisfying assignments. This paper has proposed an efficient satisfy-all procedure. The found satisfying assignments are processed by line justification used in ATPG. Two-level logic minimization is exploited to reduce the size of the state set, and excluding clauses are managed to reduce memory usage. In addition, we have presented a cache table that can be employed in the proposed satisfy-all procedure. A cache table has been widely used in BDD procedures, but not in satisfy-all procedures because a method of referring to the table was not clearly defined. We have shown that the results in an entry can be reused when the entry has the same sets of undetermined variables and clauses as the current situation. The set comparison is simplified to a size comparison by managing a field in each entry.

The proposed UMC algorithm obtains a preimage and a fix point by applying the proposed satisfy-all procedure. A logical operation is performed by adding a few clauses that are equivalent to the operation on representative variables.

Experimental results show that the proposed algorithm can verify more benchmark circuits than the pervious BDD-based and SAT-based UMC algorithms. The proposed algorithm can prove more false safety properties than McMillan's SAT-based algorithm and the BDD-based algorithms due to the efficient techniques proposed in this paper such as caching, two-level logic minimization, and so on.

REFERENCES

[1] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA: MIT Press, 1999.

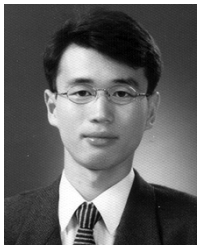
[2] C. Kern and M. R. Greenstreet, "Formal verification in hardware design: A survey," *ACM Trans. Design Automation Electron. Syst.*, vol. 4, pp. 123–193, 1999.

[3] H. Choi, B. Yun, Y. Lee, and H. Roh, "Model checking of S3C2400X industrial embedded SOC product," in *Proc. ACM/IEEE Design Automation Conf.*, 2001, pp. 611–616.

[4] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.*, vol. 8, pp. 244–263, 1986.

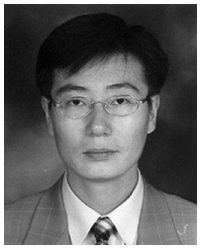
[5] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra, "Automatic verification of sequential circuits using temporal logic," *IEEE Trans. Comput.*, vol. C-35, no. 8, pp. 1035–1044, Aug. 1986.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 1020 states and beyond," in *Proc. Symp. Logic Comput. Sci.*, 1990, pp. 428–439.
[7] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill, "Sequential circuit verification using symbolic model checking," in *Proc. ACM/IEEE Design Automation Conf.*, 1990, pp. 46–51.
[8] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, pp. 677–691, Aug. 1986.
[9] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, vol. 1579, 1999, pp. 193–207.
[10] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using SAT procedures instead of BDDs," in *Proc. ACM/IEEE Design Automation Conf.*, 1999, pp. 317–320.
[11] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, pp. 201–205, 1960.
[12] J. P. Marques-Silva and K. A. Sakallah, "Boolean satisifiability in electronic design automation," in *Proc. ACM/IEEE Design Automation Conf.*, 2000, pp. 40–45.
[13] M. W. Moskevicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering and efficient SAT solver," in *Proc. ACM/IEEE Design Automation Conf.*, 2001, pp. 530–535.
[14] G. Cabodi, P. Camurati, and S. Quer, "Can BDDs compete with SAT solvers on bounded model checking?," in *Proc. ACM/IEEE Design Automation Conf.*, 2002, pp. 117–122.
[15] K. L. McMillan, "Applying SAT methods in unbounded symbolic model checking," in *Proc. Int. Conf. Computer-Aided Verification*, vol. 2404, 2002, pp. 250–264.
[16] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*. Piscataway, NJ: IEEE Press, 1999.
[17] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package," in *Proc. ACM/IEEE Design Automation Conf.*, 1990, pp. 40–45.
[18] S. Sheng and M. Hsiao, "Efficient preimage computation using a novel success-driven ATPG," in *Proc. Design, Automation, Test Eur.*, 2003, pp. 822–827.
[19] A. Gupta, Z. Yang, P. Ashar, and A. Gupta, "SAT-based image computation with application in reachability analysis," in *Proc. Int. Conf. Formal Methods Computer-Aided Design Electron. Circuits*, vol. 1954, 2000, pp. 354–371.
[20] A. Gupta, Z. Yang, P. Ashar, L. Zhang, and S. Malik, "Partition-based decision heuristics for image computation using SAT and BDDs," in *Proc. Int. Conf. Computer-Aided Design*, 2001, pp. 286–292.
[21] P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta, "Combining decision diagrams and SAT procedures for efficient symbolic model checking," in *Proc. Int. Conf. Computer-Aided Verification*, vol. 1855, 2000, pp. 124–138.
[22] P. A. Abdulla, P. Bjesse, and N. Eén, "Symbolic reachability analysis based on SAT-solvers," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, vol. 1785, 2000, pp. 411–425.
[23] G. Tseitin, "On the complexity of derivations in propositional calculus," in *Studies in Constructive Mathematics and Mathematical Logics*, A. O. Slisenko, Ed. New York: Plenum, 1968, pt. II, pp. 115–125.
[24] Espresso [Online]. Available: http://www-cad.eecs.Berkeley.edu/ Software/software.html
[25] SMV, K. L. McMillan. [Online]. Available: http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/



Hyeong-Ju Kang (S'00) received the B.S. and M.S. degrees in electrical engineering in 1998 and 2000, respectively, from the Korea Advanced Institute of Science and Technology (KAIST), Daejon, where he is currently pursuing the Ph.D. degree in electrical engineering and computer science.

His research interests include formal verification of a very large scale integrated (VLSI) design, multimedia/communication VLSI systems, and embedded processor design.



In-Cheol Park (M'93–SM'02) received the B.S. degree in electronics engineering from Seoul National University, Seoul, Korea, in 1986, and the M.S. and Ph.D. degrees in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Daejon, in 1988 and 1992, respectively.

Since June 1996, he has been an Assistant Professor and is now an Associate Professor in the Department of Electrical Engineering and Computer Science, KAIST. Prior to joining KAIST, he was with IBM T. J. Watson Research Center, Yorktown, NY, from May 1995 to May 1996, where he performed research on high-speed circuit design. His current research interests include computer-aided design algorithms for high-level synthesis and very large scale integrated architectures for general-purpose microprocessors.

Prof. Park received the Best Paper Award at the International Conference on Computer Design in 1999, and the Best Design Award at the Asia–South Pacific Design Automation Conference in 1997.