Using SAT based Image Computation for Reachability Analysis

Pankaj Chauhan Edmund M. Clarke Daniel Kroening

September 2003

CMU-CS-03-151

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research is sponsored by the Semiconductor Research Corporation (SRC) under task ID 1027.001 and the National Science Foundation (NSF) under Grant Nos. CCR-9900309 and CCR-0098072. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of SRC, NSF, or the United States Government.

Keywords: Hardware Verification, SAT

Abstract

Satisfiability procedures have shown significant promise for symbolic simulation of large circuits, hence they have been used in many formal verification techniques, including automated abstraction refinement, ATPG etc. We show how to use modern SAT solvers like Chaff and GRASP to compute images of sets of states and how to efficiently detect fixed point of the sets of states during reachability analysis. Our method is completely SAT based, and does not use BDDs at all. The sets of states and transition relation are represented in clausal form, which can be processed by SAT checkers. The SAT checker subsequently generates the set of newly reached states in clausal form as well. At the heart of our engine lie two efficient algorithms. The first algorithm shortens the cubes that the SAT checker generates by a static-analysis algorithm, which significantly reduces the number of cubes the SAT checker needs to enumerate. The second algorithm reduces the space required to store sets of states as a set of cubes by a recursive cube-merging procedure. We demonstrate the effectiveness of our procedure on ISCAS sequential benchmarks for reachability. In particular, our algorithm does not have BDD size explosion surprises and deteriorates in a predictable manner.

1 Introduction

Image Computation and Reachability Analysis Computing the set of states reachable in one step from a given set of states under a transition relation forms the heart of many symbolic state exploration algorithms, including reachability analysis, model checking [8, 6, 7], etc. This operation is called *image computation*. Let us consider a state transition relation T over the set of states S. The set of states is defined by the set of valuations over a vector of state variables $\mathbf{x}$. We denote a set or a vector of variables in a boldface. The transition relation $T(\mathbf{x}, \mathbf{i}, \mathbf{x}')$ relates states defined by valuations of present state variables $\mathbf{x}$ and inputs $\mathbf{i}$ to states defined by valuations of next state variables $\mathbf{x}'$. Note that we are using propositional formulas $S(\mathbf{x})$ and $T(\mathbf{x}, \mathbf{i}, \mathbf{x}')$ to represent set of states and set of transitions. The image of $S(\mathbf{x})$ under $T(\mathbf{x}, \mathbf{i}, \mathbf{x}')$ is given by the following equation.

$$Img(S(\mathbf{x}')) = \exists \mathbf{x}.T(\mathbf{x}, \mathbf{i}, \mathbf{x}') \wedge S(\mathbf{x}) \tag{1}$$

In reachability analysis, beginning with the set of initial states S_0, images are repeatedly computed until the set of states does not grow any more, in other words, the least fixed point of the following equation is computed.

$$X = \mu X.(S_0 \cup Img(X)) \tag{2}$$

Following simple algorithm computes this fixed point.

Reachability(S_0)
$S_{reach} \leftarrow \phi$
$i \leftarrow 0$
while $(S_i \neq \phi)$ {
    $S_{reach} \leftarrow S_{reach} \cup S_i$
    $S_{i+1} \leftarrow Img(S_i) \backslash S_{reach}$
    $i \leftarrow i + 1$
}
return S_{reach}

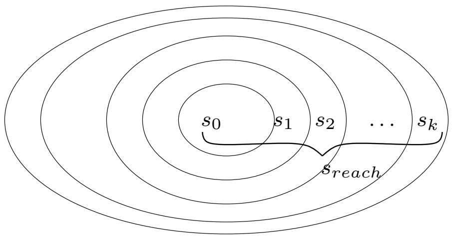

Figure 1: Reachability algorithm

In this algorithm, S_i denotes the set of newly discovered states in each iteration. Once there are no more states to be discovered, we have reached a fixed point.

As noted earlier, the sets of states and sets of transitions are traditionally represented by BDDs ([3, 8, 5, 4]). It is well known that while BDDs are compact representations of many functions, they unfortunately suffer from size explosion for many circuits. A BDD based model checker is like a black box.

A slight change in circuit or variable order can make model checking infeasible. Moreover, there are some functions like multipliers, where the BDDs are always exponentially large in the number of variables. BDD based model checkers do not have a gradual degradation in performance, and the performance is often not predictable. We offer a SAT procedure based image computation and fixed point detection mechanism that is robust and degrades gradually. The runtime of our algorithm depends on the size of the input circuit and the diameter of the circuit only.

SAT Procedures Recently, propositional SAT checkers have demonstrated tremendous success on various classes of SAT formulas. The key to the effectiveness of SAT checkers like Chaff [16], GRASP [19], and SATO [20] is non-chronological backtracking, efficient conflict driven learning of conflict clauses, and improved decision heuristics.

SAT checkers have been successfully used for Bounded Model Checking (BMC) [2], where the design under consideration is unrolled and the property is symbolically verified using SAT procedures. BMC is effective for showing the presence of errors. However, BMC is not at all effective for showing that a specification is true unless the diameter of the state space is known. Moreover, BMC performance degrades when searching for deep counterexamples. The basic problem with BMC is that there is no mechanism to detect whether a fixed point has been reached while exploring state space. A more serious problem is that the transition relation is unrolled for progressively increasing number of steps; hence, searching for deeper counterexamples becomes impractical. Our algorithm is used to detect fixed points in image computations and the SAT checker never has to deal with multiple unrollings of the transition relation. In each SAT checker run, only one step of the image computation is done at a time.

The efficiency of SAT procedures has made it possible to handle circuits with a few thousand of variables, much larger than any BDD based model checker is able to do at present.

The basic framework for these SAT procedures, shown in Figure 2, is based on Davis-Putnam-Longeman-Loveland (DPLL) backtracking search, . The function `decide_next_branch()` chooses the branching variable at current *decision level*. The function `deduce()` does *Boolean constraint propagation* to deduce further assignments. In the process, it might infer that the present set of assignments to variables do not lead to any satisfying solution. This is termed as a conflict, as at least one CNF clause remains unsatisfied. In case of a conflict, new clauses are learned by `analyze_conflict()` that hopefully prevent the same unsuccessful search in the future. The conflict analysis also returns a variable for which another value should be tried. This variable may not be the most recent variable decided, leading to a *non-chronological* backtrack. If all variables have been decided, then we have found a satisfying assignment and the procedure returns. The strength of various SAT checkers lies in their implementation of constraint propagation, non-chronological backtracking, decision heuristics, and learning.

```
while(1) {
   if (decide_next_branch()) {        // Branching
      while (deduce() == conflict) {  // Propagate implications
         blevel = analyse_conflict(); // Learning
         if (blevel == 0)
            return UNSAT;
         else
            backtrack(blevel);        // Non-chronological
                                      // backtrack
      }
   }
   else                               // no branch means all vars
                                      // have been assigned
      return SAT;
}
```

Figure 2: Basic DPLL backtracking search (used from [16] for illustration purpose)

Modern SAT checkers work by introducing conflict clauses in the learning phase and by non-chronological backtracking. Implication graphs are used for Boolean constraint propagation. The vertexes of this graph are literals, and each edge is labeled with the clause that forces the assignment. When a clause becomes unsatisfiable as a result of the current set of assignments (decision assignments or implied assignments), a conflict clause is introduced to record the cause of the conflict, so that the same futile search is never repeated. The conflict clause is learned from the structure of the implication graph. When the search backtracks, it backtracks to the most recent variable in the conflict clause just added, not to the variable that was assigned last.

In our algorithm, we use the Chaff SAT checker [16], as it has been demonstrated to be the most powerful SAT checker on a wide class of problems.

The rest of the paper is organized as follows. In Section 2, we describe our basic SAT based reachability algorithm. This algorithm is inefficient as given, therefore, in Section 3, we describe efficient data structure for storing sets of states as DNF cubes and a cube enlargement procedure. In Section 4, related work is described. Finally, we conclude in Section 6 with directions for future research.

2 SAT Based Fixed Point Computation

SAT checkers like Chaff read propositional formulas represented in conjunctive normal forms (CNFs). We present an algorithm that does not use any BDDs. We assume that the transition relation $T(\mathbf{x}, \mathbf{i}, \mathbf{x}')$ is already represented in as a set of CNF clauses. It is customary to convert any transition relation represented as a set of propositional formula to CNF form by introducing intermediate variables. This translation is polynomial in the size of the original circuit. We represent the set of newly reached states after each iteration of the reachability loop (S_i in Figure 1) as a set of disjunctive normal form (DNF) cubes. The set of all reachable states after each step (S_{reach}) is also represented in DNF. Since S_{reach} is in DNF, $\neg S_{reach}$ will be automatically in CNF. As Chaff needs CNF representation, we convert S_i from DNF to CNF by introducing intermediate variables. In each iteration i, we ask the SAT checker to find satisfying assignments to the formula below.

$$S_{i-1}(\mathbf{x}) \wedge T(\mathbf{x}, \mathbf{i}, \mathbf{x}') \wedge \neg S_{reach}(\mathbf{x}') \tag{3}$$

This formula corresponds to step 5 of the basic reachability algorithm (Fig. 1). This formula asks the SAT checker to compute a satisfying assignment such that the present state variables $\mathbf{x}$ and input variables $\mathbf{i}$ satisfy the predicate $S_{i-1}(\mathbf{x}) \wedge T(\mathbf{x}, \mathbf{i}, \mathbf{x}')$, i. e., the set of states reachable from the newly discovered states in the previous iteration. We conjoin with this the negation of the set of all accumulated states so far ($\neg S_{reach}$), thus we ask SAT checker to compute only the states that have not been seen so far. If the SAT checker concludes that the formula is unsatisfiable, then it means that the set of newly reached states S_i is empty, and we have reached fixed point. On the other hand, if the SAT checker finds a satisfying assignment to this formula in present state $\mathbf{x}$, input $\mathbf{i}$, intermediate and $\mathbf{x}'$ variables, the projection of this assignment on $\mathbf{x}'$ variables gives a subset of newly reached states. Note that this partial assignment to $\mathbf{x}'$ is consistent with the full assignment that the SAT checker finds to the formula 3. The formula 3 describes all constraints on the next set of states. Therefore, the projection is a valid state reachable from S_{i-1} following the transition relation T. Therefore, the following lemma easily follows.

Lemma 2.1 *The projection of any partial satisfying assignment to Equation 3 in* $\mathbf{x}$*,* $\mathbf{i}$*,* $\mathbf{x}'$ *and intermediate variables to just* $\mathbf{x}'$ *is a valid partial assignment in* $\mathbf{x}'$ *describing a newly discovered state reachable from* S_{i-1} *following* T*.*

We add this state to $S_i(\mathbf{x})$ as a DNF cube $\mathbf{d}$, after translating the next state variables in the cube to present state variables. The negation of $\mathbf{d}$ is a CNF clause, which is added as a conflict clause in the SAT engine. This clause $\neg\mathbf{d}$ is called a *blocking clause*. Thus after finding each satisfying assignment, the set $S_{reach}(\mathbf{x}')$ grows.

We present the high level algorithm in Figure 3. The algorithm has two loops. The outer loop carries out different steps of image computation, while the inner loop is implicit in the SAT checker, and enumerates sets of the newly reached states in a particular step.

Each satisfying cube $\mathbf{d}$ is added to S_i and S_{reach} after enlarging it to $\mathbf{d}'$ in step 6. The addition of $\mathbf{d}'$ to S_{reach} is done in the SAT checker when the blocking clause $\neg\mathbf{d}'$ is added. As noted earlier, negation of S_{reach} is automatically in CNF.

/* It is assumed that S_0 is given in DNF form */
SatReachability(S_0)
1 $i \leftarrow 1$
2 $S_{reach} \leftarrow S_0$
3 **while** $(S_{i-1} \neq \phi)$ {
4     $S_i \leftarrow \phi$
    /* $DNFtoCNF$ converts a formula to CNF by introducing intermediate variables */
5   **for** (each satisfying partial assignment $\mathbf{d}$ in $\mathbf{x}'$ to $DNFtoCNF(S_{i-1}(\mathbf{x})) \wedge T(\mathbf{x}, \mathbf{i}, \mathbf{x}') \wedge \neg S_{reach}(\mathbf{x}'))$
        /* $\mathbf{d}$ contains only next state variables */
6         $\mathbf{d}' \leftarrow EnlargeCube(\mathbf{d})$
7   Add $\neg \mathbf{d}'$ as a blocking clause
8         $S_i \leftarrow AddCube(S_i, next2current(\mathbf{d}'))$
9         $S_{reach} \leftarrow AddCube(S_{reach}, \mathbf{d}')$
10   **endfor**
11 }
12 **return** $next2current(S_{reach})$

Figure 3: Outline of SAT based reachability algorithm

3 Efficient Implementation of SAT Based Reachability

This algorithm, as it is, is very inefficient and hence impractical. The first problem comes from the way the SAT checker computes satisfying assignments or cubes. Chaff SAT checker gives values to all variables in any assignment. We then project this assignment to $\mathbf{d}$, which assigns values to all next state variables $\mathbf{x}'$. Therefore $\mathbf{d}$ describes only one newly reached states. Enumerating states one at a time is clearly very inefficient. However, most times, only a partial assignment to all variables satisfies the clause database given to SAT. A partial assignment to $\mathbf{x}'$ describes more than one state at a time, the larger the set the few the number of assignments. This is advantageous in two ways, first the blocking clause for $\mathbf{d}$ prunes the SAT search space drastically, second, the number of state enumerations required go down considerably. Therefore, it is desired that the partial assignment be as small as possible. It is clearly to our advantage to get as small cubes as possible, since smaller cubes cover a larger number of assignments. Given a cube computed by Chaff, it may be possible to throw away certain assignments from the cube, and still get a satisfying cube. By a static analysis of the transition relation, we infer the unnecessary assignments in $\mathbf{d}$. This procedure *EnlargeCube* is called in line 6 on $\mathbf{d}$ to get a smaller cube $\mathbf{d}'$.

The second problem is that the representation of the sets S_{reach} and even S_i can grow too large. For example, if we consider a counter that counts up to 2^{30}, each iteration of the while loop will add only one state to S_{reach}. Thus we will have to represent 2^{30} clauses for S_{reach}. However, the DNF clause 1 represents all possible values of the counter. In other words, after a satisfying assignment to S_{reach} is found, we can combine multiple clauses to get a smaller partial assignment. For example, the DNF clauses $x_1 \wedge x_5 \wedge \neg x_6$ and $x_1 \wedge x_5 \wedge x_6$ can be combined to $x_1 \wedge x_5$. An efficient data structure is needed to support this *AddClause* operation, since many clauses may be added, and each clause can potentially be combined with more than one existing clause. We use a hash table, eacy entry of which contains a trie.

We give a cube enlargement heuristic procedure next, which is followed by a description of an efficient data structure that stores S_i and S_{reach}. The enlargement procedure reduces the number of set enumerations, hence the amount of time, while the seconf procedure reduces the space requirement.

3.1 Cube Enlargement

There are five types of variables that appear in the SAT formula 3: present state variables $\mathbf{x}$, circuit inputs i, next state variables $\mathbf{x}'$, intermediate variables $\mathbf{i_s}$ introduced while converting S_{i-1} to CNF, and the intermediate variables $\mathbf{i_t}$ introduced while converting the transition relation $T(\mathbf{x}, \mathbf{i}, \mathbf{x}')$ to CNF. The SAT checker finds a satisfying assignment $\mathbf{c}$, possibly to all these variables. However, the cube $\mathbf{d}$ of line 7 in the algorithm (Fig. 3) is just in terms of $\mathbf{x}'$ variables. In order to reduce the number of assignments in $\mathbf{d}$, we present the following procedure. This procedure assumes that the transition relation is given in functional form, i.e., there is a transition function $f_i(\mathbf{x})$ for each next state variables x_i. Let $Supp(f_i)$ denote the support set of f_i, i.e., the variables that f_i depends on. This assumption is true for circuit descriptions. When an assignment to a next state variable x_i' can be ignored, we say that x_i' is *.

Free Variables

First, we describe the concept of free variables, i.e., the variables that are free to choose any value, despite SAT checker assigning them specific values. In order to detect whether the variable v is free or not, the following conservative tests are used. If v is an input variable or an intermediate variable, then it is definitely free. Moreover, for functional transition relations, we don't even need to check if an intermediate variable appears in other transition functions or not, since intermediate variables are generated from local translation for f_i. The only real problem is if v is a present state variable. The only constraints that are placed on the present state variables are from S_{i-1}. To see if the present satisfying assignment $\mathbf{c}$ restricts v or not, we can just check that assignment to v does not affect the present satisfying assignment. This can be efficiently detected as follows. While translating the DNF corresponding to S_{i-1} to CNF, we introduce one intermediate variable for each DNF cube. In essence, the truth value of each DNF cube is captured in the corresponding variable. Suppose $i_1, i_2, \ldots, i_k$ are the intermediate variables corresponding to DNF cubes $D_1, D_2, \ldots, D_k$ in S_{i-1}. The translation of the S_{i-1} constraint in Eqn. 3 looks like:

$$(i_1 \vee i_2 \vee \ldots \vee i_k) \wedge (i_1 \Leftrightarrow D_1) \wedge (i_2 \Leftrightarrow D_2) \wedge \ldots \wedge (i_k \Leftrightarrow D_k) \tag{4}$$

Each equality $i_j \Leftrightarrow D_j$ gives rise to a set of CNF clauses, which we haven't expanded for the sake of brevity.

If the satisfying assignment c makes any of intermediate variables true, the corresponding DNF cube is true, and we don't need to see if any other DNF cube is true or not, since the truth of only one DNF cube satisfies the S_{i-1} clauses. So we find the first intermediate variable i_l that is set to true. All present state variables *not mentioned in the DNF cube D_l are irrelevant* for satisfying S_{i-1} constraint, hence, they can be assumed to be free.

Free Transition Functions

Let us denote the set of free variables in the support of a transition function f_i as $FreeSupp(f_i)$.

The main idea is that if a transition function f_i (for x_i') depends on a variable v (which is either a present state variable, input or an intermediate variable from $\mathbf{i}_t$), and the following conditions are satisfied, we can guarantee that x_i' can be set to either 1 or 0. Thus, the value of x_i' can be safely ignored from the present assignment.

1. Variable v is free.

2. Values of non-free variables in $Supp(f_i)$ are propagated and do not force f_i to a particular value. For example, when $f_i = x_1 \wedge x_2$, x_1 is free, f_i satisfies condition 3, but $x_2 = 0$ in S_{i-1}. This forces f_i to 0. So constant values of non-free variables are propagated first.

3. The function f_i does not share free support with any other transition function, i.e., $FreeSupp(f_i) \cap FreeSupp(f_j) = \phi, j \neq i$. Moreover, the variable v appears in the formula for f_i in exactly one place. In other words, the variable v does not appear in any other propositional function.

Note that there may be other conditions under which x_i' can still choose both values. However, this conditions allow us to do a static analysis of the circuit.

The third condition is too restrictive in practice. Usually, transition functions do share common variables. In order to infer that a next state variable x_i' can assume both values, we can simplify the transition functions by further constant-propagating values of some free variables as well (remember that we already constant-propagate the values of non-free variables). For example, suppose that $f_1 = x_1|i_2$ and $f_2 = x_1|i_3$, and x_1, i_1 and i_3 are free variables. Suppose the SAT assignment is $a = 0, i_1 = 0, i_2 = 1$. Since both f_1 and f_2 share the variable x_1, we can not safely say that both x_1' and x_2' are *. However, we can replace x_1 by 0, and propagate the effects, giving us $f_1 = i_2$ and $f_2 = i_3$. Now, both f_1 and f_2 become independent, as can be set to *. Note that since x_1 is a free variable, we could have chosen $x_1 = 1$, different from the SAT assignment. In order to determine which variables to assign values to, till the functions become independent, we use a greedy strategy. We order the variables by the number of times they appear in all transition functions. Beginning with the variable that occurs the most number of times, we keep replacing the variables by constants (from SAT assignment) and propagating the effect, until transition functions become independent and next state variables can be inferred to be *s. In the worst case, all transition functions become constants, whose values agree with the satisfying assignment.

Most analysis of this procedure can be done statically just once. The only changing part is detection of free present state variables. Note that this is just one alternative. There can be other options. For example, we considered using efficient approximate set cover algorithms to find out the literals that cover all clauses of formula 3. Another option is to use BDD based symbolic simulation to infer multiple cubes. The given cube enlargement procedure produces one smaller cube. However, using BDDs for simulation of the circuit for one step and then applying constants to some of BDD variables to contain the BDD sizes can yield a set of many states at once.

3.2 Efficient Set Representation

The set of states are represented as a set of DNF cubes. However, it is easy to see that each new cube that is added to S_{reach} has a potential to be merged with other cubes to form shorter cubes. For example, the boolean function 1 is an exponentially compact representation than four DNF cubes $a \wedge b, \neg a \wedge b, a \wedge \neg b, \neg a \wedge \neg b$. We describe the following procedure to add a cube to the existing set of cubes. We assume that the variables in the cubes are ordered. The set is represented by a hash table, where each hash table entry stores a subset of cubes in a trie form. Each trie stores cubes that are made up of the same CNF variables. The hash table is indexed by the hash computed from a signature of a cube. In the following algorithm (Figure 4), assume that the DNF cube **d** is represented as a vector of integers, each integer identifying a particular propositional variable, negative if the literal is negative, positive otherwise. For example, if $a, b, c, d, e, f...$ are variables, then they are identified by the integers $1, 2, 3, 4, 5, 6, ...$ So a cube $(a \wedge \neg c \wedge \neg f)$ is represented by the vector $\mathbf{d} = [1, -3, -6]$. The function *ComputeSignature* computes a bit string that is used to compute the hash value for a cube. The bit string is ordered, just as variables in the cubes are, and contains 1 for each variable (in any phase) present and 0 for the variables not present in the cube. The trailing $0s$ are removed to get a shorter bit string. So the signature for the cube **d** is 101001. Not that even though there may be variables numbered $7, 8, 9, ..,$ the $0s$ corresponding to them do not appear in the signature.

Since each trie stores cubes made up of the same variables, the cubes are represented by bit strings of the same length as the number of variables in the cubes of the trie. Essentially, if a literal is positive, the bit corresponding to it is 1, and 0 otherwise. So the cube **d** is stored as 100.

The crux of the *AddCube* procedure is between lines 5–19. Given an incoming cube **d**, it tries to find all other cubes from the trie that differ from **d** in just one bit. For each such cube $\mathbf{d}'$ found (lines 10–13), the cube computed by merging **d** and $\mathbf{d}'$ is added to S by calling *AddCube* recursively. The merged cube is essentially cube **d** with the matched bit (i^{th} bit) removed. If **d** doesn't match at the i^{th} bit, then the next bit is checked. Once the traversal over trie is done, we check if **d** was merged with anything. If it was, then we no longer keep **d**. Line 18 just updates the hash table with potentially modified trie.

Note that the algorithm doesn't guarantee absolute minimum cubes. In fact, to do so, we may need to keep all cubes, even after they are merged, in the hopes of merging them with other future cubes. But the main focus of the algorithm is to reduce space, and not get the absolute smallest cubes. Another point to note is that since the SAT checker always finds new states that haven't been discovered so far, we assume that the trie T_d does not already contain **d**.

The complexity of this algorithm in the worst case can be $O(n^2)$, not counting the recursive calls. Here n is the number of state variables. Each of line 1, 3–4, 6 and 17 cost $O(n)$, while hash lookups and updates on lines 2 and 18 are essentially constant time operations. Lines 10 and 11 cost $O(m) + O(m-1) + \ldots + O(1) = O(m^2) = O(n^2)$.

3.3 Complexity of the Set Representation

The set of DNF cubes representing S_i or S_{reach} possess a useful property. By the negation of S_{reach} in the SAT formula (Eqn. 3), we guarantee that no newly generated DNF cube shares a satisfying assignment with any existing cube in sets S_i or S_{reach}. Thus the set of DNF cubes representing these sets are disjoint, in that they do not have any common assignment. For example, the DNF cube $b \vee c \vee d$ cannot occur if the cube $a \vee b \vee c$ is already present, as they share a common assignment $a = b = c = d = 1$. However, cube $\neg a \vee b \vee c \vee d$ can occur. If we can detect that the set of cubes is a tautology, we can terminate the reachability, as we have reached all the states. Our cube addition algorithm is *online* in nature. We now actually prove that if the set of DNF cubes are given *a priori* that do not share a common a satisfying assignment, then detecting if it is a tautology is polynomial. The general problem of detecting DNF tautology is NP-complete, so is its dual CNF satisfiability.

We call the problem of tautology detection of a set of DNF cubes that do not share any common assignment pairwise an *R-TAUT* problem. The procedure for tautology detection works simply by counting the number of satisfying assignments. Suppose there are c DNF cubes made up from a total of m variables. The cubes do not share common assignments pair wise. Suppose cube i has literals $l_1, l_2, \ldots, l_{c_i}$. There are a total of 2^m possible assignments to variables, and if each assignment is a satisfying assignment, then the DNF cubes are a tautology. Cube i describes a total of 2^{m-c_i} assignments agreeing with the literals in cube i. As we know that no two cubes i and j share any common

```
ComputeSignature(d, m)
/* m is the size of the cube d, assume d is sorted */
j ← 1
for (i from 1 to d[m])
    if (i = |d[j]|) /* variable is present in d */
        s_d[i] ← 1
        j ← j + 1
    else
        s_d[i] ← 0
    endif
endfor
return s_d
```

```
AddCube(S, d, n, m)
/* n is the total number of variables, m is the number of variables in d */
s_d ← ComputeSignature(d, m)
/* T_d is the trie in which d will be stored */
T_d ← HashLookup(S, s_d)
/* compute the representation of d to store in T_d */
for (i from 1 to m)
    b[i] ← (d[i] > 0)?1 : 0
endfor
match ← false
T_d ← TrieInsert(T_d, b)
curr_node ← T_d
for (i from 1 to m)
    b[i] ← 1 − b[i] /* flip the i^th bit */
    if (TrieLookup(curr_node, b[i : m]) = true)
        /* match at the ith bit */
        T_d ← TrieDelete(T_d, b)
        /* insert the merged cube */
        S ← AddCube(S, d[1 : i − 1] :: d[i + 1 : m], n, m)
        match ← true
    endif
    b[i] ← 1 − b[i] /* flip it back to what it was */
    curr_node ← (b[i] = 1)?curr_node.right : curr_node.left
endfor
if (match = true)
    T_d ← TrieDelete(T_d, b)
/* update the trie for this cube */
S ← HashUpdate(S, s_d, T_d)
return S
```

Figure 4: Procedures *AddCube* and *ComputeSignature*.

assignment, the total number of assignments that satisfy both cube i and j are precisely $2^{m-c_i} + 2^{m-c_j}$. The total number of satisfying assignments for the set of DNF cubes is just

$$2^{m-c_1} + 2^{m-c_2} + \ldots + 2^{m-c_c}.$$

Clearly, the additions can be carried out in time polynomial in m and c, the size of problem input. Hence the theorem

Theorem 3.1 *R-TAUT is in P.*

Note that in our case, the set of DNF cubes are not given a priori. However, this procedure can be run periodically on S_{reach} to find out if all states have been reached, in which case we stop the search. We also use this counting mechanism to report the number of reached states.

4 Related Work

The first completely SAT based reachability algorithm was reported by McMillan in [14]. The main difference between our approaches is that we represent the set of states in DNF, while he presented it in CNF. A SAT based enumeration algorithm is used to compute a CNF formula equivalent to a given formula characterizing preimages. However, we use intermediate variables to convert DNF representation to CNF while running SAT. He used zero suppressed BDDs (zDDs) to store CNF clauses, and used a SAT conflict analysis based heuristic to enlarge the cubes. We did not compare the results reported in [14] as the set of benchmarks in [14] was not publicly available.

In [11], the authors used procedure similar to ours to compute preimages. They also intermediate variables to convert DNF cubes to CNF formula. However, they use offline Espresso [17] algorithm to reduce the number of cubes. Our cube storage procedure is on-line, in the sense that it processes the cubes as they are generated. Moreover, they do not have any algorithm to enlarge the cubes. We doubt their results. They also reported them only on two different circuits and for safety property checking only, which can be much easier than to do than computing reachability. Our attempts to contact them to get more information about the properties they checked failed.

In [18], the authors used ATPG instead of SAT to compute preimages. They used BDDS to store the resultant sets of states. ATPG allows reasoning directly on the circuits, hence they do not have any intermediate variables. They report results on only two circuits. These are known to be difficult circuits, however.

In [15], the only aim was to compute the sequential diameter, also called *diameter* of the circuit. They did not compute the reachable set of states at all. Their procedure was based on the Chaff SAT checker as well. Their procedures shared many similarities with bounded model checking (BMC) [1]. They build a SAT formula describing symbolic simulations of increasing length. In our approach, we explicitly compute the set of reachable states, and the SAT checker does not have to compute more than one step of symbolic simulation at a time. This we believe is a significant advantage that our method has over [15] and BMC. Using BMC for depth equal to circuit diameter is sufficient for $\mathbf{G}p$ kind of LTL properties. In [12], this notion is generalized to that of *completeness threshold* (CT). The authors describe a sorting network built on top of SAT formula for computing diameters.

In [10, 9], a mixed BDD and SAT based approach to image computation is described. They use SAT procedure to derive a top level disjunctive decomposition of image computation and use BDD based image computation for each leaf subproblem.

5 Experimental Results

We implemented our algorithms on top of the zChaff SAT solver. The SAT solver is modified to enumerate satisfying solutions by adding blocking clauses. We implemented our program in C++ as part of a larger high level verification system.

We conducted our experiments on a 1.53GHz dual AMD Athlon processor machine with 3GB of memory running Linux. The memory cutoff was set to 1.5GB of resident program size, and the time cut off was set to 1000 seconds. The results are summarized in table 1. For each circuit, we report the number of latches, the number of reachability steps that we could complete, the number of cubes stored in the representation for the reachable states, the number of cubes that were enumerated as blocking clauses (or how many times line 7 in algorithm of Figure 3 was called), the ratio of the number cubes v/s the number of blocking clauses added in percentage, and the total running time (user time+system time) in seconds. Note that the number of cubes in final set representation is much smaller than the total number of enumerated cubes, as evidenced by the %age size column. This asserts that the space saving data structure for storing cubes is effective. We compare our results with primarily that of [15]. We would like to emphasize that in [15], only the depth was computed, and the actual set of reachable states was not computed.

The circuits that we report come from mainly three sources, ISCAS'89 benchmarks, IU set of circuits from Synopsys, and some circuits that were translated from Verilog code available from various sources. The IU set of circuits are various abstractions of parts of a picoJava microprocessor implementation.

For a relatively small timeout value, we have been able to do reachability for many circuits. We have outperformed [15] by a large magnitude on all but one small completed benchmarks. They used a faster machine (2GHz v/s 1.53GHz) as well. The effectiveness of the cube merging procedure is evident from "%age space" column. The savings are dramatic for circuits that have counter like structures in them (iu**, s208.1, s420.1, s838.1, s635) and other circuits (s1512, s9234, s13207, s1423, s526, s526n). For other circuits, it can be inferred that the SAT checker and cube enlargement procedures generate many disjoint cubes that can not be merged with existing set of cubes. This may be dependent on

Circuit	# latches	# steps	Space Requirement			Time (sec)	Comparison with [15]	
			# cubes	# blocking clauses	%age space		Max. Depth	Time (sec)
decss	86	85*	655	131304	0.50	1000.00		
iu30	30	4*	3343	72037	4.64	1000.00		
iu35	35	3*	1479	94424	1.57	1000.00		
iu40	40	2*	20	33168	0.06	1000.00		
iu45	45	1*	2294	165192	1.39	1000.00		
s208.1	8	255	8	255	3.14	0.56		
s298	14	18	33	217	15.21	0.33	18	19.3
s344	15	6	558	2624	21.27	15.30		
s349	15	6	546	2624	20.81	14.82		
s382	21	150	337	8864	3.80	7.71		
s386	6	7	6	12	50.00	0.21	7	0.18
s400	21	150	336	8864	3.79	7.81		
s420.1	16	65535	16	65535	0.02	213.97		
s444	21	150	341	8864	3.85	8.00		
s499	22	21	21	21	100.00	1.74	21	1.07
s510	6	46	10	46	21.74	0.47	46	144.81
s526	21	150	381	8867	4.30	9.35		
s526n	21	150	372	8867	4.20	9.21		
s635	32	125528*	66	125528	0.05	1000.00		
s641	19	6	321	1543	20.80	2.24	6	97.03
s713	19	6	363	1543	23.53	2.53	6	126.94
s820	5	10	11	24	45.83	0.48	10	2.51
s832	5	10	11	24	45.83	0.47		
s838.1	32	155441*	26	155441	0.02	1000.00		
s953	29	10	189	503	37.57	2.01	10	102.23
s967	29	10	177	548	32.30	3.12		
s1196	18	2	802	2615	30.67	6.79	2	232.84
s1238	18	2	849	2615	32.47	7.26		
s1269	37	1*	2136	4339	49.23	1000.00	7*	5000
s1423	74	3*	2652	55568	4.77	1000.00	26*	5000
s1488	6	21	19	47	40.43	0.87	21	96.87
s1494	6	21	19	47	40.43	0.87		
s1512	57	4*	2035	178175	1.14	1000.00		
s9234	228	8*	507	6651	7.62	1000.00		
s13207	669	2*	76	1824	4.17	1000.00		
s15850	597	5*	362	2558	14.15	1000.00		
s38584	1452	2*	58	452	12.83	1000.00		

Table 1: Experimental results on a set of circuits from various sources including ISCAS'89 and Synopsys. The comparison is against [15]. Note: (*)-reachability was not complete. Empty boxes denote results N/A.

circuit structure.

In [13], our tool was used to enumerate symbolic solutions to certain predicate abstraction formulas. The characteristic of these formulas was that the number of variables to be quantified was much larger (an order of magnitude) than the number of variables representing the set of states. In our case, the number of variables to be quantified is $\mathbf{x}, \mathbf{I}$, and $\mathbf{i}_{int}$.

6 Conclusions and Future Work

We presented a completely SAT based image computation algorithm. The effectiveness of this algorithm is demonstrated for reachability analysis on many large circuits. This algorithm can be used for computing pre images equally well, hence it can be used in a general SAT based symbolic model checking algorithm. The novel features of our algorithm are an efficient data structure for storing sets of states as DNF cubes and a cube enlargement procedure based on static circuit analysis.

There are many directions for future research. The cube enlargement procedure can be improved by the use of BDD based symbolic simulation. One obvious extension that we are working on is to do full CTL model checking using this procedure. A related question to explore is whether techniques like iterative squaring can be used with SAT to speed up fixpoint detection in reachability. We want to extend this procedure for solving general quantified Boolean formulas (QBFs). Our algorithm was used in [13] for solving QBFs with one existential quantifier that arise in symbolic predicate abstraction. We are also identifying other problem domains where our method can be applied.

References

[1] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, M. Fujita, and Yunshan Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the Design Automation Conference (DAC'99)*, pages 317–320, 1999.

[2] Armin Biere, Alexandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proceedings of Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99)*, number 1579 in LNCS, 1999.

[3] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[4] 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, *Proceedings of the International Conference on Very Large Scale Int egration*, Edinburgh, Scotland, August 1991.

[5] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *27th ACM/IEEE Design Automation Conference*, 1990.

[6] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*. Springer-Verlag, Yorktown Heights, NY, May 1981.

[7] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[8] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[9] Aarti Gupta, Zijian Yang, Pranav Ashar, Lintao Zhang, and Sharad Malik. Partition-based decision heuristics for image computation using SAT and BDDs. In *Proc. of the Intl. Conf. on Computer-Aided Design (ICCAD)*, 2001.

[10] Aarti Gupta, Zijiang Yang, Pranav Ashar, and Anubhav Gupta. SAT-based image computation with application in reachability analysis. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Proceedings of FMCAD'00*, volume 1954 of *LNCS*, pages 354–371, November 2000.

[11] Hyeong-Ju Kang and In-Cheol Park. SAT-based unbounded symbolic model checking. In *Proceedings of Design Automation Conference (DAC'03)*, pages 840–843, 2003.

[12] Daniel Kroening and Ofer Strichman. Efficient computation of recurrence diameters. In L. Zuck, P. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *4th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 2575 of *Lecture Notes in Computer Science*, pages 298–309. Springer Verlag, January 2003.

[13] Shuvendu K. Lahiri, Randal E. Bryant, and Byron Cook. A symbolic approach to predicate abstraction. In *Proceedings of the International Conference on Computer-Aided Verification (CAV'03)*, 2003. To appear.

[14] Ken McMillan. Applying SAT methods in unbounded symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceedings of the International Conference on Computer-Aided Verification (CAV'02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2002.

[15] Maher Mneimneh and Karem Sakallah. SAT-based sequential depth computation. In *Proceedings of ASP-DAC*, January 2003.

[16] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference (DAC'01)*, pages 530–535, 2001.

[17] Richard Rudell. Espresso. Online. Available: http://www-cad.eecs.berkeley.edu/Software/software.html.

[18] Shuo Sheng and Michael Hsiao. Efficient preimage computation using a novel success-driven atpg. In *Proceedings of Design Automation and Test in Europe (DATE'03)*, 2003.

[19] J. P. Marques Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. Technical Report CSE-TR-292-96, Computer Science and Engineering Division, Department of EECS, Univ. of Michigan, April 1996.

[20] Hantao Zhang. SATO: An efficient propositional prover. In *Proceedings of the Conference on Automated Deduction (CADE'97)*, pages 272–275, 1997.