Memory Efficient All-Solutions SAT Solver and its Application for Reachability Analysis

Orna Grumberg Assaf Schuster Avi Yadgar

Computer Science Department, Technion, Haifa, Israel

Abstract

This work presents a *memory-efficient All-SAT engine* which, given a propositional formula over sets of *important* and *non-important* variables, returns the set of all the assignments to the *important* variables, which can be extended to solutions (satisfying assignments) to the formula. The engine is built using elements of modern SAT solvers, including a scheme for learning conflict clauses and non-chronological backtracking. Re-discovering solutions that were already found is avoided by the search algorithm itself, rather than by adding blocking clauses. As a result, the space requirements of a solved instance do not increase when solutions are found. Finding the next solution is as efficient as finding the first one, making it possible to solve instances for which the number of solutions is larger than the size of the main memory.

We show how to exploit our All-SAT engine for performing *image computation* and use it as a basic block in achieving full reachability which is purely SAT-based (no BDDs involved).

We implemented our All-SAT solver and reachability algorithm using the state-of-the-art SAT solver Chaff [23] as a code base. The results show that our new scheme significantly outperforms All-SAT algorithms that use blocking clauses, as measured by the execution time, the memory requirement, and the number of steps performed by the reachability analysis.

1 Introduction

This work presents a *memory-efficient All-SAT engine* which, given a propositional formula over sets of *important* and *non-important* variables, returns the set of all the assignments to the *important* variables, which can be extended to solutions (satisfying assignments) to the formula. The All-SAT problem has numerous applications in AI [25] and logic minimization [26]. Moreover, many applications require the ability to instantiate all the solutions of a formula, which differ in the assignment to only a subset of the variables. In [17] such a procedure is used for predicate abstraction. In [8] it is used for re-parameterization in symbolic simulation. In [22, 7] it is used for reachability analysis, and in [16] it is used for pre-image computation. Also, solving QBF is actually solving such a problem, as shown in [18].

Most modern SAT solvers implement the DPLL[10, 9] backtrack search. These solvers add clauses to the formula in order to block searching in subspaces that are known to contain no solution. All-SAT engines that are built on top of modern SAT solvers tend to extend this method by using additional clauses, called *blocking clauses*, to block solutions that were already found [22, 7, 16, 17, 8, 24]. However, while the addition of blocking clauses prevents repetitions in solution creation, it also significantly inflates the size of the solved formula. Thus, the engine slows down in accord with the number of solutions that were already found. Eventually, if too many solutions exist, the engine may saturate the available memory and come to a stop.

In this work we propose an efficient All-SAT engine which does not use blocking clauses. Given a propositional formula and sets of important and non-important variables, our engine returns the set of all the assignments to the important variables, which can be extended to solutions to the formula. Setting the non-important variables set to be empty yields all the solutions to the formula. Similar to previous works, our All-SAT solver is also built on top of a SAT solver. However, in order to block known solutions, it manipulates the backtracking scheme and the representation of the implication graph. As a result, the size of the solved formula does not increase when solutions are found. Moreover, since found solutions are not needed in the solver, they can be stored in external memory (disk or the memory of another computer), processed and even deleted. This saving in memory is a great advantage and enables us to handle very large instances with huge number of solutions. The memory reduction also implies time speedup, since the solver handles much less clauses. In spite of the changes we impose on backtracking and the implication graph, we manage to apply many of the operations that made modern SAT solvers so efficient. We derive conflict clauses based on conflict analysis, apply non-chronological backtracking to skip subspaces which contain no solutions, and apply conflict driven backtracking under some restrictions.

We show how to exploit our All-SAT engine for reachability analysis, which is an important component of model checking. Reachability analysis is often used as a preprocessing step before checking. Moreover, model checking of most safety temporal properties can be reduced to reachability analysis [1]. BDD-based algorithms for reachability are efficient when the BDDs representing the transition relation and the set of model states can be stored in memory [5, 6]. However, BDDs are quite unpredictable and tend to explode on intermediate results of image computation. When using BDDs, a great effort is invested in improving the variables order, which strongly influences the BDD's size. SAT-based algorithms, on the other hand, can handle models with larger number of variables. However, they are mainly used for Bounded Model Checking (BMC) [2].

Pure SAT-based methods for reachability [22, 7] and model checking of safety properties [16, 24] are based on All-SAT engines, which return the set of all the solutions to a given formula. The All-SAT engine receives as input a propositional formula describing the application of a transition relation T to a set of states S. The resulting set of solutions represents the image of S (the set of all successors for states in S). Repeating this step, starting from the initial states, results in the set of all reachable states.

Similar to [22, 7], we exploit our All-SAT procedure for computing an image for a set of states, and then use it iteratively for obtaining full reachability. Several optimizations are applied at that stage. Their goals are to reduce the number of found solutions by avoiding repetitions between images; to hold the found solutions compactly; and to keep the solved formula small.

An important observation is that for image computation, the solved formula is defined over variables describing current states $\overline{x}$, inputs $\overline{I}$, next states $\overline{x'}$, and some auxiliary variables that are added while transforming the formula to CNF. However, many solutions to the formula are not needed: the only useful ones are those which give different values to $\overline{x'}$. This set of solutions is efficiently instantiated by our algorithm by defining $\overline{x'}$ as the important variables. Since the variables in $\overline{x'}$ typically constitute just 10% of all the variables in the formula [28], the number of solutions we search for, produce, and store, is reduced dramatically. This was also done in [22, 7, 16, 24]. Note, that by defining $\overline{x}$ as the important variables, our engine can perform pre-image computation, and thus can be used whenever such a procedure is required. However, in our experiments, it was only employed for image computation.

We have built an All-SAT solver based on the state-of-the-art SAT solver Chaff [23]. Experimental results show that our All-SAT algorithm outperforms All-SAT algorithms based on blocking clauses. Even when discovering a huge number of solutions, our solver does not run out of memory, and does not slow down. Similarly, our All-SAT reachability algorithm also achieves significant speedups over blocking clauses-based All-SAT reachability, and succeeds to perform more image steps.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 gives the background needed for this work. Sections 4 and 5 describe our algorithm and its implementation. A proof for the algorithm's correctness is given in Section 6. Section 7 describes the utilization of our algorithm for reachability analysis. Section 8 shows our experimental results, and Section 9 includes conclusions.

2 Related Work

In [7] an optimization is employed to the blocking clauses method. The number of blocking clauses and the run time are reduced significantly by inferring from a newly found solution a set of related solutions, and blocking them all with a single clause. This, however, is insufficient when larger instances are considered. Moreover, the optimization is applicable only for formulae which originated from a model, and therefore, is not applicable for solving other QBF as described in the introduction.

In [19, 27] a success-driven learning is used. This method uses a hash table to store information about solutions found in sub-spaces which were already searched. While this information helps reducing the search time, it still implies growth of the solved instance. Moreover, this method depends on the fact that the CNF originates from a model, and is used for pre-image computation only, where the values of the 'next-state' variables are used as objectives for the SAT procedure to satisfy.

In [4, 15] all the solutions of a given propositional formula are found by repeatedly choosing a value to one variable, and splitting the formula accordingly, until all the clauses are satisfied. However, in this method, all the solutions of the formula are found, and distinguishing between *important* and *non-important* variables is not straight forward. This method can not be efficiently applied for quantifier elimination and image computation, since finding all the solutions for the formulae implies repetitive instantiation of the same 'next state'.

Other works also applied optimizations within a single image [7] and between images [22, 7]. These have similar strength to the optimization we apply between images. However, *within* an image computation we gain significant reductions in memory and time due to our new All-SAT procedure, and the ability to process the solutions outside the engine before the completion of the search. This gain is extended to the reachability computation as well, as demonstrated by our experimental results.

In [14], a hybrid method of SAT and BDD is proposed for image computation. This implementation of an All-SAT solver does not use blocking clauses. The known solutions are kept in a BDD which is used to restrict the search space of the All-SAT engine. While this representation might be more compact than blocking clauses, the

All-SAT engine still depends on the set of known solutions when searching for new ones. We believe that our algorithm can be used to enhance the performance of such hybrid methods as well.

3 Background

3.1 The SAT Problem

Let an assignment A to a set of Boolean variables $\overline{v}$ be a function $A : \overline{v} \to \{true, false\}$. Given a Boolean formula $\varphi(\overline{v})$, the *Boolean Satisfiability Problem* (SAT) is the problem of finding an assignment A to $\overline{v}$ such that $\varphi(\overline{v})$ will have the value 'true' under this assignment. Such an assignment is called a *satisfying assignment*, or a *solution*, for φ. A partial assignment A' is a partial function, for which the domain is $\overline{v'} \subseteq \overline{v}$. If $v' \in (\overline{v} \setminus \overline{v'})$ then $A'(v')$ is undefined. The SAT problem is an NP-Complete problem, thus believed to have exponential worst case complexity [12].

We shall discuss formulae presented in the conjunctive normal form (CNF). That is, φ is a conjunction of clauses, while each clause is a disjunction of literals over $\overline{v}$. A literal l is an instance of a variable or its negation: $l \in \{v, \neg v \mid v \in \overline{v}\}$. We define the value of a literal l under an assignment A to be
$A(l) = \begin{cases} A(v) & l = v \\ \neg A(v) & l = \neg v \end{cases}$. We shall consider a clause as a set of literals, and a formula as a set of clauses.

A clause cl is satisfied under an assignment A if and only if $\exists l \in cl, A(l) = true$. We denote this by $A \models cl$. For a formula φ given in CNF, and an assignment A, $A \models \varphi \Leftrightarrow \forall cl \in \varphi, A \models cl$. Hence, if, under an assignment (or a partial assignment) A, all of the literals of some clause in φ are false, then A does not satisfy φ. Formally:

$$\big(\exists cl \in \varphi \; such \; that \; \forall l \in cl, A(l) = false\big) \Rightarrow A \not\models \varphi \tag{1}$$

We such a situation a *conflict*, and say that A conflicts with φ. We can therefore look at the clauses of the formula as constraints which the assignments should satisfy. Each clause is a constraint, and satisfying all the clauses means satisfying the formula.

For a formula φ given in CNF, we can apply *resolution* on its clauses. For each two clauses c_1, c_2, if a variable v exists, such that $c_1 = A \cup \{v\}$ and $c_2 = B \cup \{\neg v\}$, where A and B are sets of literals, then $resolution(c_1, c_2) = A \cup B$. For $c_r = resolution(c_1, c_2), \varphi \Rightarrow c_r$, and $\varphi \equiv \varphi \wedge c_r$ [10]. That is, by using *resolution* we can add new clauses to φ, and the new formula will be equivalent to φ.

3.2 Davis-Putnam-Logemann-Loveland Backtrack Search (DPLL)

We begin by describing the *Boolean Constraint Propagation* (*bcp()*) procedure. Given a partial assignment A and a clause cl, if there is one literal $l \in cl$ such that $A(l)$ is undefined, while the rest of the literals are all false, then in order to avoid a conflict, A must be extended such that $A(l) = true$. cl is called a *unit clause* or an *asserting clause*, and the assignment to l is called an *implication*. An asserting clause which implies the literal l is its antecedent, referred to as *ante(l)*. An example for an asserting clause is given in Figure 1. Once the assignment is extended, other clauses may become asserting, and the assignment has to be extended again. The *bcp()* procedure finds all the possible implications in a given moment (iteratively) and returns CONFLICT if a conflict as described in Equation (1) occurs, and NO_CONFLICT otherwise. This procedure is efficiently implemented in [23, 13, 30, 21, 20].

$$A = \left\{ \begin{array}{l} x_0 = 0, \\ x_1 = 1, \\ x_3 = 1, \\ x_5 = 0 \end{array} \right\} \wedge \begin{array}{l} cl_1 = \{x_0, x_1, x_2, x_{10}\} \\ cl_2 = \{x_0, \neg x_1, \neg x_2, x_4\} \\ cl_3 = \{x_0 \underline{\neg x_2}, \neg x_3, x_5\} \end{array} \Rightarrow A' = \left\{ \begin{array}{l} x_0 = 0, \\ x_1 = 1, \\ \underline{x_2 = 0}, \\ x_3 = 1, \\ x_5 = 0 \end{array} \right\}$$

Figure 1: In clause cl_1, $A(x_1) = 1 \Rightarrow A \vDash cl_1$. Clause cl_2 is not asserting, not satisfied and is not a conflict according to Equation (1). Clause cl_3 is asserting since only x_2 is not assigned by A, and the rest of the literals are false. Therefore, A is extended to A' where $A' \vDash cl_3$. *ante*$(\neg x_2) = cl_3$.

```
bool satisfiability_decide() {
    mark all variables as not tried both ways
    while (true) {
        if (!decide())
            return SATISFIABLE;
        while (bcp() == CONFLICT) {
            if (!resolve_conflict())
                return UNSATISFAIABLE
        }
    }
}
```

(a) DPLL

```
bool resolve_conflict() {
    d ← the most recent decision
            not tried both ways
    if (d == NULL)
        return false
    flip the value of d
    mark d as tried both ways
    undo invalidated implication
    return true
}
```

(b) Resolve Conflict

Figure 2: DPLL Backtrack Search. The procedure *decide()* chooses one of the free variables and assigns a value to it. If no such variable exists, it returns 'false'. The procedure *resolve_conflict()* performs *chronological backtracking*.

The DPLL algorithm [10, 9] walks the binary tree which describes the variables space. At each step, a *decision* is made. That is, a value to one of the variables is chosen, thus reaching deeper in the tree. Each decision is assigned with a new *decision level*. After a decision is made, the algorithm uses the *bcp()* procedure to compute all its implications. All the implications are assigned with the corresponding decision level. If a conflict is reached, the algorithm backtracks in the tree, and chooses a new value to the most recent variable for which a decision was made, such that the variable was not yet tried both ways. This is called *chronological backtracking*. The algorithm terminates if one of the leaves is reached with no conflict, describing a satisfying assignment, or if the whole tree was searched and no satisfying assignment was found, meaning ϕ is unsatisfiable. The pseudo code of this algorithm is shown in Figure 2.

3.3 Optimizations

Current state of the art solvers use Conflict Analysis, Learning and Conflict Driven backtracking to optimize the DPLL algorithm as described below;

3.3.1 Implication Graphs

An *Implication Graph* represents the current partial assignment during the solving process, and the reason for the assignment to each variable. For a given assignment, the implication graph is not unique, and depends on the decisions and the order used by the *bcp()*. Recall that we denote the asserting clause that implied the value of l by $ante(l)$, and refer to it as the antecedent of l. If l is a decision, $ante(l) = NULL$. Given a partial assignment A:

- The implication graph is a directed acyclic graph $G(L, E)$.
- The vertices are the literals of the current partial assignment: $\forall l \in L, l \in \{v, \neg v \mid v \in V \wedge A(l) = true\}$.
- The edges are the reasons for the assignments: $E = \{(l_i, l_j) \mid l_i, l_j \in L, l_i \in ante(l_j)\}$. That is, for each vertex l, the incident edges represent the clause $ante(l)$. A decision vertex has no incident edge.
- Each vertex is also assigned with the decision level of the corresponding variable.

In an implication graph with no conflicts, there is at most one vertex for a variable. When a conflict occurs, there are both true and false vertices of some variable, denoted *conflicting variable*. In case of a conflict, we only consider the connected component of the graph which includes the conflict vertices, since the rest of the graph is irrelevant to the analysis.

A *Unique Implication Point* (UIP) in an implication graph is a vertex of the current decision level, through which all the paths from the decision vertex of this level to the conflict pass. There may be more than one UIP, and we order them starting from the conflict. The decision variable of a level is always a UIP, since it is the first vertex of all the paths in the corresponding level.

An implication graph with a conflict and 2 UIPs is presented in Figure 3(a).

3.3.2 Conflict Analysis and Learning

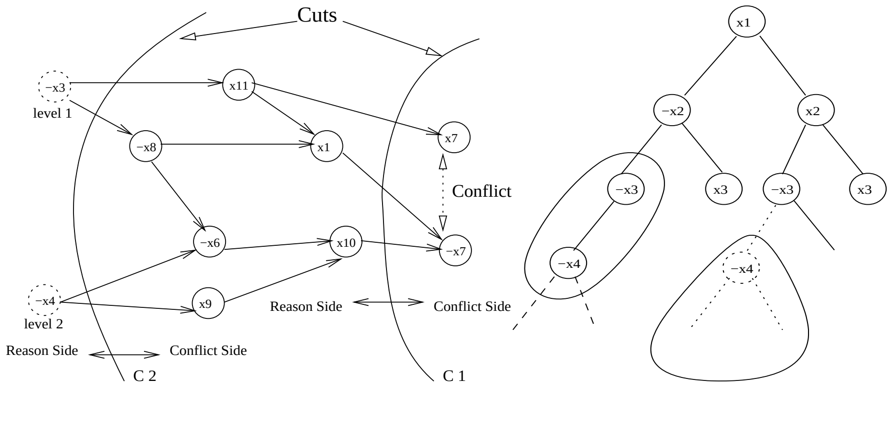
Upon an occurrence of a conflict, we perform conflict analysis, which finds the reason to it and adds constraints over the search space accordingly. The conflict analysis is performed over the implication graph as described next :

- The graph is bi-partitioned such that all the decision variables are on one side (the *reason* side) and the conflicting variable is on the other side (the *conflict* side). We call this bi-partitioning a *cut*. The reason for the conflict is defined by all the vertices on the reason side with at least one edge to the conflict side. These vertices are the *reason literals*.
- In order for the conflict not to re-occur, the current assignment to the reason variables should not occur. Therefore, a new clause is generated, which is comprised of the negation of the reason literals. This clause is called a *conflict clause*.
- The conflict clause is added to the problem's formula.

The conflict clauses added to the problem can be viewed as constraints over the search space, thus pruning the search tree by preventing the solver from reaching the subspace of the reason for the last conflict again. Note that the conclusion of the learning, which is expressed by the conflict clause, is correct regardless of the assignment to the variables. The generation of the new clause is actually applying *resolution* on the original clauses. Therefore, the new problem is equivalent to the original.

Figure 3(a,b) shows an implication graph which is used to generate a conflict clause, and the way it prunes the search tree.

Different cuts, corresponding to different learning schemes, will result in different conflict clauses. Moreover, more than one clause can be added when a single conflict



(a) Implication Graph (b) Pruning

Figure 3: (a) An implication graph and two cuts. The roots, $\neg x_3, \neg x_4$ are decisions. $\neg x_4, x_{10}$ are UIPs. C_1 matches the 1UIP scheme, and C_2 matches the decision variable scheme. C_2 is the cut which generated (x_3, x_4). (b) Adding the clause (x_3, x_4) prunes the search tree of the subspace defined by $\neg x_3 \wedge \neg x_4$.

is encountered. Choosing the cut is most significant for the effectiveness of the conflict clause and for the information it adds to the problem. We shall use two different cuts in our work that will generate clauses involving:

1. The first UIP (1UIP); i.e., The one which is closest to the conflict.
2. The decision variable of the current level, which is also a UIP.

We want a cut that will generate a conflict clause with only one variable from the highest decision level. Such a cut is achieved when a UIP is placed on the reason side, and all the vertices assigned after it are placed on the conflict side. The UIP (the negation of the literal in the graph) would then be the only variable from the highest decision level in the conflict clause. Figure 3(a) shows the cuts corresponding to the suggested schemes. This kind of a conflict clause is effective as we explain next.

Let c be the generated conflict clause and m the current decision level. In order to resolve the conflict, we backtrack one level and invalidate all of its assignments. The UIP was the only literal in c from level m, thus, it is the only variable in c with no value. Recall that the literals in c are the negation of the current values to the reason variables, making them all false under the current assignment. The result is that the conflict clause is now a unit clause, and therefore is asserting. The unit literal in the conflict clause is the negation of the UIP. Thus, flipping the value of the UIP variable (not necessarily the decision) is now an implication of the assignments in the lower levels and the added conflict clause.

3.3.3 Conflict Driven Backtracking

In order to emphasize the recently gained knowledge, *conflict driven backtracking* is used: Let l be the highest level of an assigned variable in the conflict clause. The solver discards some of its work by invalidating all the assignments above l. The implication of the conflict clause is then added to l [31]. This implies a new order of the variables in the search tree. The result of this backtrack is ‘jumping’ into a new search space, as shown in Figure 4. Note that the other variables for which the assignments were invalidated, may still assume their original values, and lead to a solution.

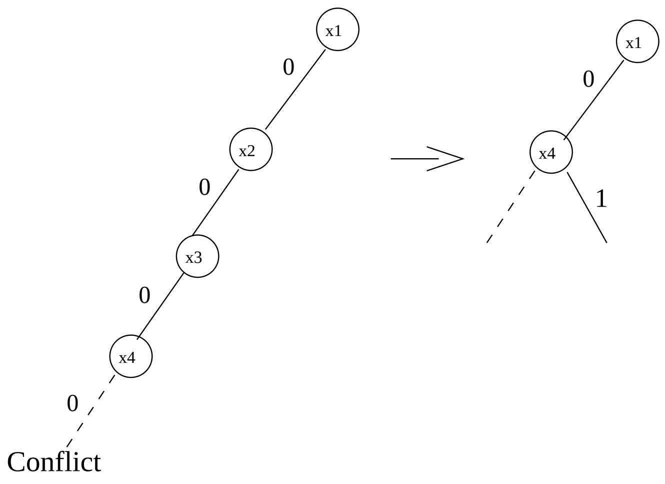


Figure 4: *Conflict driven* backtracking after adding the conflict clause (x_1, x_4) creates a new variables order. x_3 is now free to assume either true or false value.

3.4 The All-SAT Problem

Given a Boolean formula presented in CNF, the All-SAT problem is to find all of its solutions as defined by the SAT problem.

The Blocking Clauses Method

A straightforward method to find all of the formula’s solutions is to modify the DPLL SAT solving algorithm such that when a solution is found, a blocking clause describing its negation is added to the solver. This clause prevents the solver from reaching the same solution again. The last decision is then reversed, and the search is continued normally. At any given time, the solver holds a formula representing the original formula and the negation of all of the solutions found so far. Once all the solutions are found, there is no satisfying assignment to the current formula, and the algorithm terminates.

This algorithm suffers from rapid space growth as it adds a clause of the size of $\overline{v}$ for each solution found. Another problem is that the increasing number of clauses in the system will slow down the *bcp()* procedure which will have to look for implications in an increasing number of clauses.

An improvement for this algorithm can be achieved by generating a clause consisting of the negation of the decision literals only, since all the rest of the assignments are implications and are forced by these decisions. This reduces the size of the generated clauses dramatically (1-2 orders of magnitude) but does not affect their number.

The blocking clauses algorithm with the described optimization is shown in Figure 5. Variations of this algorithm were used in [22, 7, 16], where the All-SAT problem is solved for model checking. In the next section, we present a new All-SAT algorithm which is fundamentally different from the blocking clauses method.

```
Blocking Clauses All-SAT:
$A \leftarrow \phi$
while (true){
  if (!decide()) {
    if (handle_solution() == EXHAUSTED)
      return EXHAUSTED
  }
  while (bcp() == CONFLICT) {
    if (!resolve_conflict())
      return EXHAUSTED
  }
}
```

```
handle_solution() {
  $A \leftarrow A \cup \{$current assignment$\}$
  if (current level == 0)
    return EXHAUSTED
  $cl \leftarrow$ create a clause with the
    negation of all of the decisions
  add $cl$ to the solver
  backtrack one level
  return NOT_EXHAUSTED
}
```

Figure 5: All SAT algorithm using blocking clauses to prevent multiple instantiations of the same solution. The procedures *bcp()* , decide() and resolve_conflict() are the same as those defined for DPLL in section 2. At the end of the process, A is the set of all the solutions to the problem.

4 Memory efficient All-SAT Algorithm

4.1 Conventions

Assume we are given a partition of the variables to important and non-important variables. Two solutions to the problem which differ in the non-important variables only are considered the same solution. Thus, a solution is defined by a subset of the variables.

A *subspace* of assignments, defined by a partial assignment σ, is the set of all assignments which agree with σ on its assigned variables. A subspace is called *exhausted* if all of the satisfying assignments in it (if any) were already found. At any given moment, the current partial assignment defines the subspace which is now being investigated. However, it is enough to consider the decision variables only. This is correct because the rest of the assignments are implied by the decisions.

4.2 The All-SAT Algorithm

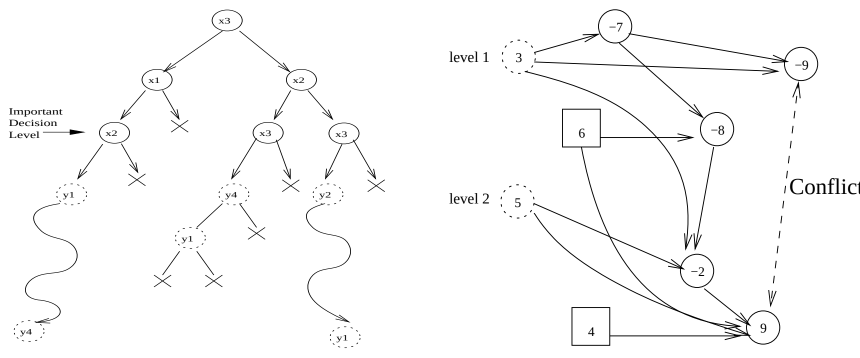
Our algorithm walks the search tree of the important variables. We call this tree the *important space*. Each leaf of this tree represents an assignment to the important variables which does not conflict with the formula. When reaching such a leaf, the algorithm tries to extend the assignment over the non-important variables. Hence, it looks for a solution within the subspace defined by the important variables. Note that the walk over the *important space* should be exhaustive, while only one solution for the non-important variables should be found. A pseudo code of this algorithm is given in Figure 6, and it is illustrated in Figure 7(a).

We incorporate these two behaviors into one procedure by modifying the decision and backtracking procedures of a conflict driven backtrack search, and the representation of the implication graph.

Important First Decision Procedure: We create a new decision procedure which looks for an unassigned variable from within the important variables. If no such variable is found, the usual decision procedure is used to choose a variable from the non-important set. The procedure returns 'true' if an unassigned variable was found, and 'false' otherwise. This way, at any given time, the decision variables are a sequence of important variables, followed by a sequence of non important variables. At any given

```
$All - SAT(\varphi, \overline{important})$
1)  $S_{alg} \leftarrow \phi$
2)  repeat {
3)    $s \leftarrow$ find the next partial assignment to $\overline{important}$
            which does not conflict with $\varphi$
4)    if $SAT(\varphi, s)$
5)      $S_{alg} \leftarrow S_{alg} \cup s$
5)  } until $(s = NULL)$
6)  return $S_{alg}$
```

Figure 6: All-SAT algorithm with important variables distinction. The input to the algorithm is φ, the Boolean formula, and $\overline{important}$, the set of important variables. The procedure $SAT(\varphi, s)$ solves the SAT problem for φ with a partial assignment s.



(a) Important First (b) Sub-Levels

Figure 7: (a)Important First Decision. x variables are important and y are non-importnat. (b) An implication graph in the presence of flipped decisions. 6 and 4 define new sub-levels.

time, the *important decision level* is the maximal level in which an important variable is a decision. An example is given in Figure 7(a).

Exhaustive Walk of the Important Space: An exhaustive walk of the *important space* is performed by extending the original DPLL algorithm with the following procedures: Chronological backtracking after a leaf of the *important space*, representing an assignments to the important variables, is handled; Learning a conflict clause and chronological backtracking upon an occurrence of a conflict; Non-chronological backtracking when a subspace of the problem is proved to be *exhausted*.

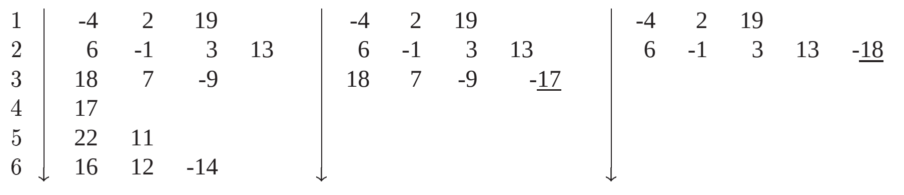
Chronological backtracking is performed when the current subspace is *exhausted*. It means that, under the previous decisions, the last decision variable *must* assume a new value. Hence, its reverse assignment can be seen as an implication of the previous decisions. We therefore flip the highest decision and assign it with the level below. This way, the highest decision is always the highest decision not yet flipped. Higher decisions which were already flipped are regarded as its implications. Note, though, that there is no clause implying the values of the flipped decisions. Therefore, a new definition for the implication graph is required.

We change the definition of the implication graph as follows: Root vertices in the graph still represent the decisions, but also decisions flipped because of previous chronological backtracking. Thus, for conflict analysis purposes, a flipped decision is considered as defining a new decision sub-level. The result is a graph in which the nodes represent the actual assignments to the variables, and the edges represent clauses. Therefore, this graph is an implication graph, as defined in Section 3.3.1. This graph describes the current assignment, though not the current decisions. An example for such a graph is given in Figure 7(b). Recall that generating a new conflict clause using an implication graph is equivalent to applying *resolution* on clauses of the formula, represented by the graph's edges. Such a new clause is derived from the formula, and is independent of the current decisions. Since our newly defined graph corresponds to the definition of an implication graph, we can use it in order to derive a new conflict clause, which can be added to the formula, without changing its set of solutions.

We perform the regular conflict analysis, which leads to a UIP in the newly defined sub-level. The generated conflict clause is added to the formula to prune the search tree, as for solving the SAT problem. Modifying the implication graph and introducing the new sub-levels may cause the conflict clause not to be asserting. However, since we do not use conflict driven backtracking in the *important space*, our solver does not require that the conflict clauses will be asserting. A non-asserting clause is added to the formula, so that it might prune the search tree in the future, and the search is then resumed.

We now consider the case in which a conflict clause is asserting. In this case, we extend the current assignment according to it. Let c_1 be a conflict clause, and lit its implication. Adding lit to the current assignment may cause a second conflict, for which lit is the reason. Therefore, we are able to perform conflict analysis again, using lit as a UIP. The result is a second conflict clause, c_2 which implies $\neg lit$. It is obvious now that neither of the assignments to lit will resolve the current conflict. The conclusion is that the reason for this situation lies in a lower decision level, and that a larger subspace is *exhausted*. Identifying the *exhausted* subspace is done by calculating $c_3 \leftarrow resolution(c_1, c_2)$ [10]. We skip it by backtracking to the decision level preceding the highest level of a variable in c_3. This is the level below the highest decision level of a variable in c_1 or c_2. Note that backtracking to any level higher than that would not resolve the conflict as both c_1 and c_2 would imply lit and $\neg lit$ respectively. Therefore, by this non-chronological backtracking, we skip a large *exhausted* subspace.

Assigning Non-Important Variables: After reaching a leaf in the *important space*, we have to extend the assignment to the non important variables. At this point, all the important variables are assigned with some value. Note, that since we only need one extension, we actually have to solve the SAT problem for the given formula with the current partial assignment. This is done by allowing the normal work of the optimized SAT algorithm, including decisions, conflict clause learning, and conflict driven backtracking. However, we allow backtracking down to the *important decision level* but not below it, in order not to interfere with the exhaustive search of the *important space*. If no solution is found, the current assignment to the important variables can not be extended to a solution for the formula, and should be discarded. On the other hand, if a solution to the formula is found, its projection over the important variables is a valid output. In both cases, we backtrack to the *important decision level* and continue the exhaustive walk of the *important space*.



(a) *Stack* (b) Backtrack (c) Chronological Backtrack

Figure 8: (a) A *stack*. $-4, 6, 18, 17, 22$ and 16 are the decisions. (b) Backtrack to level 3. -17 has NULL antecedent. (c) Chronological backtrack.

5 Implementation

We implemented our All-SAT engine using zChaff [23] as a base code. This SAT solver uses the VSIDS decision heuristic [23], an efficient *bcp()* procedure, conflict clause learning, and conflict driven backtracking. We modified the decision procedure to match our important-first decision procedure, and added a mechanism for chronological backtracking. We implemented the exhaustive walk over the *important space* using chronological backtracking, and by allowing non-chronological backtracking when subspaces without solutions are detected. We used the original optimized SAT solving procedures above the *important decision level*, where it had to solve a SAT problem. Next, we describe the modifications imposed on the solver in the *important space*.

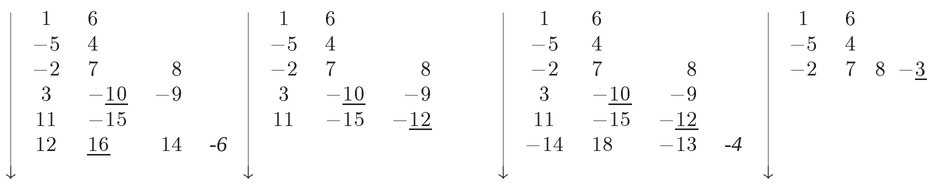
The original SAT engine represents the current implication graph by means of an *assignment stack*, hereafter referred to as *stack*. The *stack* consists of levels of assignments. The first assignment in each level is the decision, and the rest of the assignments in the level are its implications. Each implication is in the lowest level where it is implied. Thus, an implication is implied by some of the assignments prior to it, out of which at least one is of the same level. For each assigned variable, the *stack* holds its value and its antecedent, where the antecedent of a decision variable is NULL. An example for a *stack* is given in Figure 8(a).

In the following discussion, *backtrack to level* i refers to the following procedure: a) Flipping the decision in level $i + 1$. b) Invalidation of all the assignments in levels $i + 1$ and above, by popping them out of the *stack*. c) Pushing the flipped decision at the end of level i, with NULL antecedent. d) Executing *bcp()* to calculate the implications of the flipped decision. This is shown in Figure 8(a,b)

We perform *chronological backtracking* from level j within the *important space* by backtracking to level $j - 1$. This way, the flipped decisions appear as implications of prior ones, and the highest decision not yet flipped is always the highest in the *stack*. The assignments with NULL antecedents, which represent *exhausted* subspaces, remain in the *stack* until a whole subspace which includes them is *exhausted*. An example for this procedure is given in Figure 8(b,c).

Our *stack* now includes decision variables, implications, and flipped decisions, which appear as implications but with no antecedents. Using this *stack* we can construct the implication graph according to the new definition given in section 4.2. Thus we can derive a conflict clause whenever a conflict occurs.

Decisions about *exhausted* subspaces are made during the process of resolving a conflict, as described next. We define a *generated clauses stack*, which is used to



(a) Conflict (b) Chronological Backtrack (c) Implication (d) Backtrack

Figure 9: *Important Space* Resolve Conflict. (a) A conflict. ‘16’ has NULL antecedent, clause $c_1 = (-1, 2, -14)$ is generated. (b) Chronological backtrack. (c) ’-14’, the implication of clause c_1, is pushed into the *stack*, and leads to a conflict. Clause $c_2 = (-4, 2, -3, 14)$ is generated. $c_3 \leftarrow resolution(c_1, c_2) = (-1, 2, -3, -4)$. (d) Backtracking to the level preceding the highest level in c_3.

temporarily store clauses that are generated during conflict analysis[1]. In the following explanation, refer to Figure 9(a-d). (a) When a conflict occurs, we analyze it and learn a new conflict clause according to the 1UIP scheme [31]. This clause is added to the solver to prune the search tree, and is also pushed into the *generated clauses stack* to be used later. (b) We perform a chronological backtrack. A pseudo code for this procedure is shown in Figure 10(’chronological backtrack’). If this caused another conflict, we start the resolving process again. Otherwise, we start calculating the implications of the clauses in the *generated clauses stack*.

(c) Let c_1 be a clause popped out of the *generated clauses stack*. If c_1 is not asserting, we ignore it. If it implies lit, we push lit into the *stack* and run *bcp()*. For simplicity of presentation, lit is pushed to a new level in the *stack*. If a new conflict is found, we follow the procedure described in Section 4.2 to decide about *exhausted* subspaces: We create a new conflict clause, c_2, with lit as the UIP. c_1 and c_2 imply lit and $\neg lit$ respectively. We calculate $c_3 \leftarrow resolution(c_1, c_2)$ [10]. (d) We backtrack to the level preceding the highest level of an assigned variable in c_3. Thus, we backtrack higher, skipping a large *exhausted* subspace. A pseudo code of this procedure is given in Figure 10.

The complete algorithm for finding all the solutions for a given formula, with correspondence to a set of important variables, is given in Figure 11

6 Proof of Correctness

In this section we prove the correctness of our All-SAT procedure presented in Sections 4 and 5. We refer to the algorithm in Figure 11(a) as the *All-SAT algorithm*. This is a description of the exhaustive search of the *important space* and the SAT solving above the *important decision level*, required by the algorithm in Figure 6, combined into one procedure.

Given a boolean formula $\varphi(\overline{v})$, and a division of $\overline{v}$ to *important* and non-important variables, let S_φ be the set of all the assignments to the important variables, which can be extended to an assignment over $\overline{v}$ without conflicting with φ. Let S_{alg} be the set of assignments for the important variables returned by the *All-SAT algorithm*.

[1]We still refer to the decision stack as *stack*.

```
Important_Space_resolve_conflict() {
  while (conflict $\vee$ generated_clauses.size $>$ 0) {
    if (conflict) {
                       a)  if (current level == 0)
  chronological        b)      return FALSE
  backtrack            c)  $cl \leftarrow$ generate conflict clause with 1UIP, add $cl$ to $\varphi$
                       d)  generated_clauses.push($cl$)
                       e)  backtrack one level
    } else {
  Implicate             1)  $cl_1 \leftarrow$ generated_clauses.pop()
  Conflict              2)  if ($cl_1$ is asserting) {
  Clauses               3)      $lit \leftarrow$ unit literal in $cl_1$
                        4)      push $lit$ into the stack
                        5)      if (bcp() = CONFLICT) {
                        6)          $cl_2 \leftarrow$ generate conflict clause with $lit$ as UIP, add $c_2$ to $\varphi$
  Non-                  7)          $cl_3 \leftarrow resolution\ of\ (cl_1, cl_2)$, add $c_3$ to $\varphi$
  Chronological         8)          $generated\_clauses.push(cl_3)$
  Backtracking          9)          backtrack to the level preceding the highest level in $cl_3$
                        10)     }
                        11) }
    }
  }
  return TRUE
}
```

Figure 10: *Important Space* resolve conflict. φ is the formula being solved.

Theorem 6.1
1) $S_{alg} = S_{\varphi}$
2) $\forall s \in S_{alg}$*, the All-SAT algorithm adds s to S_{alg} exactly once.*

Proof:
Throughout this proof, we denote the current formula held by the solver at a given moment by φ'. If the algorithm adds a new clause to the solver, we denote the new formula by φ''.

The following lemma relates the implication graph defined in our algorithm to the implication graph defined by the SAT solving procedure.

Lemma 6.2 *For an implication graph $G(V, E)$, according to the new definition in Section 4.2, there exists an implication graph $G'(V', E')$, according to the definition in Section 3.3.1 where $V' = V$, $E' = E$ and all the roots of G' are decisions.*

Proof: Consider $r_1 \ldots r_k$, the roots in graph G in the order of their assignment. Each root r_i is either a decision or a flipped decision. We consider each such root as defining a sub-level, as illustrated in Figure 7(b).

We have to prove that there exists $G'(E', V')$ where $V' = V$ and $E' = E$, and all the roots in G' are decisions. In order to prove this, we have to show an implication graph G' where $r_1 \ldots r_k$ are roots, and with the same implications and in the same order in levels $1 \ldots k$, as the implications in sub-levels $1 \ldots k$ of graph G. We prove this by induction on r_i.
Base case. For $i = 1$, we have to show that there is an implication graph where r_1 is the root of level 1, and has the same implications in the same order as in sub-level 1 of graph G. Consider $a_1 \ldots a_n$, all the implications in the sub-level 1 of G. This is the set of all the implications of r_1. Hence, there is some SAT solving computation c, in which r_1 is the first decision, and the *bcp()* procedure assigns $a_1 \ldots a_n$ in this specific order.

```
$All - SAT(\varphi, \overline{important})$
1)  $S_{alg} \leftarrow \phi$
2)  while (true){
3)    if (!Important-First-decide()) {
4)      if (handle_solution() == EXHAUSTED)
5)        return $S_{alg}$
6)    }
7)    while (bcp() == CONFLICT) {
8)      if (current_decision_level > important decision level)
9)        $resolved$ ←SAT_resolve_conflict()
10)     else
11)       $resolved$ ← Important_Space_resolve_conflict()
12)     if (!$resolved$)
13)       return $S_{alg}$
12)   }
13) }
```

(a) All-SAT Algorithm

```
handle_solution()
1) $S_{alg} \leftarrow S_{alg} \cup \{$current assignment to
2)                  $\overline{important}\}$
3) if (important decision level == 0)
4)    return EXHAUSTED
5) backtrack to important decision level$-1$
6) return NOT_EXHAUSTED
```

```
SAT_resolve-conflict()
1)  $cl$ ← learn a new conflict clause using 1UIP
2)  add $cl$ to the solver
3)  $l$ ← highest level of a literal in $cl$
4)  if ($l <$ important decision level){
5)     if (important decision level == 0)
6)        return 'false'
7)     backtrack to important decision level
8)  } else {
9)     if ($l == 0$)
10)       return 'false'
11)    backtrack to level $l$
12) }
13) return 'true'
```

(b) Handle_Solution (c) SAT_Resolve_conflict

Figure 11: (a) All-SAT procedure. φ is the Boolean formula, and $\overline{important}$ is the set of important variables. The procedure Important-First_decide() is described in Section 4.2, and the procedure Important_Space_resolve_conflict() is described in Figure 10. (b) The procedure handle_solution(). (c) SAT_resolve_conflict(). This is the regular resolve_conflict() procedure, shown in Figure 2(b), limited to backtrack no lower than the *important decision level*.

The *bcp()* procedure assigns all the possible implications at a given time. Therefore, $a_1 \ldots a_n$ are the only implications assigned by c in level 1.

The implication graph corresponding to c has r_1 as a root, and $a_1 \ldots a_n$ as implications in the order they appear in G, and thus meets our requirements.

Induction step. Given that for $i = m < k$ there is an implication graph where the roots are $r_1 \ldots r_m$, and the implications in levels $1 \ldots m$ are the same, and in the same order, as in sub-levels $1..m$ of graph G, we have to show that there is an implication graph where the roots are $r_1 \ldots r_{m+1}$, and the implications in levels $1 \ldots (m+1)$ are the same, and in the same order, as in sub-levels $1..(m+1)$ of graph G

Since we have a graph where the roots are $r_1 \ldots r_m$, and the implications in levels

$1 \dots m$ are the same, and in the same order, as in sub-levels $1..m$ of graph G, we know that there is a SAT solving computation in which these decisions were made, and those implications were calculated in this specific order. We also know that there were no other implications which do not appear in the implication graph. Note that if r_{m+1} is a decision in G, then it is not implied by its preceding assignments. Also, if r_{m+1} is a flipped decision in G, it means that at some earlier point it was a decision with the same preceding assignments, and therefore is also not implied by them. The conclusion is that there is a SAT solving computation c, in which the decisions and implications are identical to those in sub-levels $1 \dots m$ of graph G, and in which r_{m+1} is not assigned. This computation can be extended by the decision r_{m+1}.

Consider $a_1 \dots a_n$, the implications in level $m+1$ in graph G. This is the set of implications of the assignments to the variables in sub-levels $1 \dots m$, and to r_{m+1} in the graph. We know that these variables are given the same assignments by the computations c. Therefore, $a_1 \dots a_n$ is the set of implications in level $m+1$ of the computation c. There is, then, an extension to c in which the *bcp()* procedure assigns $a_1 \dots a_n$ in this specific order. The *bcp()* procedure assigns all the possible implications at a given time. Thus, c assigns all the implications of r_{m+1} and its preceding assignments, and them only, and therefore $a_1 \dots a_n$ are the only implications calculated by c in level $m+1$.

The implication graph corresponding to c has $r_1 \dots r_{m+1}$ as roots, the same implications and in the same order in levels $1 \dots m$ as in sub-levels $1 \dots m$ of graph G, and $a_1 \dots a_n$ as implications in level $m+1$ in the order they appear in G, and thus meets our requirements.

$\square$

Lemma 6.3 *When running the All-SAT algorithm, at any given time, $\varphi' \equiv \varphi$.*

Proof: We prove Lemma 6.3 by induction.
Base case. At the beginning of the process, $\varphi' = \varphi \quad \Rightarrow \quad \varphi' \equiv \varphi$.
Induction step. Assume the current formula $\varphi' \equiv \varphi$, and a new clause cl is going to be added to the solver such that $\varphi'' = \varphi' \wedge cl$. Consider the *All-SAT algorithm*. cl was generated by one of the resolve_conflict() procedures, when executing line 9 or 11 of the algorithm. Therefore, the clause was either created by the procedure described in Figure 10 (corresponding to line 11), or by the procedure in Figure 11(c) (corresponding to line 9). We distinguish between the following cases:

1. cl was generated in line 7 of the algorithm in Figure 10. In this case, $cl =$ *resolution*(c_1, c_2), where c_1 and c_2 are clauses in φ'. Due to the nature of the *resolution* procedure, $\varphi' \wedge cl \equiv \varphi'$. Therefore, $\varphi'' \equiv \varphi' \equiv \varphi$.

2. cl was generated in line c or 6 of the algorithm in Figure 10, or in line 1 of the algorithm in Figure 11(c). In this case, cl was generated using our newly defined implication graph. According to Lemma 6.2, for each implication graph corresponding to our new definition, we can construct an implication graph with the same vertices and edges, which corresponds to the original definition used by the SAT solving procedure, given in Section 3.3.1. We know that conflict clauses generated from the original implication graphs can be added to the formula without changing its set of solutions [31]. Thus, we can use the newly defined graph to generate conflict clauses, and add them to the formula without changing its solutions. Therefore, $\varphi'' \equiv \varphi' \equiv \varphi$.

□

Lemma 6.3 shows that $\varphi' \equiv \varphi$ at any given time. In the following discussion we consider only the value of the formula, rather than its structure. Therefore, in the rest of the proof, we refer to the formula held by the solver at any given time during the solving process as φ.

Given $\varphi(\overline{v})$, a Boolean formula, and a set of important variables $\overline{v'} \subseteq \overline{v}$, let PS_φ be the set of all the assignments to $\overline{v'}$ which do not conflict with φ. Let PS_{alg} be the set of assignments to $\overline{v'}$ instantiated by the *All-SAT algorithm*. That is, for each $s \in PS_{alg}$, at some point, the partial assignment in the *stack* agreed with s on the values of $\overline{v'}$. Note that according to the definition of the sets, $S_\varphi \subseteq PS_\varphi$, and $S_{alg} \subseteq PS_{alg}$.

Lemma 6.4 $PS_{alg} = PS_\varphi$.

Proof:

- Given $s \in PS_{alg}$, s does not conflict with φ. This is ensured by the correctness of the *bcp()* procedure [10, 29]. Therefore, $s \in PS_\varphi$, and $PS_{alg} \subseteq PS_\varphi$.

- We now have to show that every $s \in PS_\varphi$ is instantiated by our algorithm. Our *important-first* decision procedure chooses important variables as long as there are unassigned ones, and then starts choosing non-important variables. The result is that the first variables in the search tree, starting with the root, are the important variables. We show that our algorithm performs a complete search over the important variables, and thus instantiate all the assignments in PS_φ. In the following discussion, a *solution* is an assignment to the important variables, which does not conflict with φ.

  Our algorithm walks the search tree defined by the important variables. When we reach a conflict or a *solution*, we backtrack in the search tree. If we use only chronological backtracking in the search tree, this walk is complete, as in DPLL [10, 9]. We now prove that the non-chronological backtracking performed by the *All-SAT algorithm* does not affect the completeness of the search.

  Non-chronological backtracking is performed either by the resolve_conflict() procedures in lines 9 and 11 of the *All-SAT algorithm*, or by the handle_solution() procedure in line 4.

  1. **Important_Space_resolve_conflict()**: In this procedure, presented in Figure 10, non-chronological backtracking is performed in line 9. The result of this line is backtracking to level $l - 1$, where l is the highest level of a literal in cl_3.

     Consider cl_1 and cl_2 in lines 1 and 6 of the procedure. cl_1 and cl_2 imply both lit and $\neg lit$, and thus are of the form $(A \vee lit)$ and $(B \vee \neg lit)$ respectively, where the literals of A and B are all 'false'. Let i and j be the highest levels of literals in A and B respectively.

     Assume, without loss of generality, that $i \geq j$. For each $k \geq i$, if we backtrack to level k, all the variables of A and B are still assigned 'false', thus still implying lit and $\neg lit$. This leaves us with a conflict. Therefore, by backtracking to level $i - 1$, we do not skip any subspace which includes a *solution*.

     cl_3, calculated in line 7, equals $(A \vee B)$. Therefore, l, the maximum level of a literal in cl_3, equals i. We showed that backtracking to level $i - 1$ does not cause loss of solutions. Therefore, the non-chronological backtracking to level $l - 1$, as done by this procedure, does not cause loss of *solutions*.

2. **SAT_resolve_conflict()**: In this procedure, presented in Figure 11(c), non-chronological backtracking is performed in lines 7 or 11. In line 7 the procedure backtracks to the *important decision level*, and in line 11 the procedure backtracks to a level higher than the *important decision level*.

   Consider the search tree created when using the *important-first* decision procedure. All the variables at levels higher than the *important decision level* are non-important variables. Thus, the backtrack is within a subspace defined by one *solution*, and can not skip other *solutions*.

   We conclude that the SAT_resolve_conflict() procedure does not affect the completeness of the search.

3. **handle_solution()**: This procedure backtracks to (*important decision level*-1). This implies a chronological backtrack in the *important space*, which does not affect the completeness of the search.

We showed that our search is complete, i.e, we reach every *solution*. Therefore, $PS_{alg} \supseteq PS_\varphi$.

- We proved both $PS_{alg} \subseteq PS_\varphi$ and $PS_{alg} \supseteq PS_\varphi$. Therefore, $PS_{alg} = PS_\varphi$, and we conclude our proof.

$\square$

Lemma 6.5 $S_{alg} = S_\varphi$

Proof:

- For each $s \in S_{alg}$, s was added to S_{alg} in line 1 of the handle_solution procedure described in Figure 11(b). This procedure is executed in line 4 of the *All-SAT algorithm*(Figure 11(a)), when all the variables are assigned without a conflict. Therefore, s is the projection of a solution to φ on the important variables. According to the definition of S_φ, $s \in S_\varphi$, and $S_{alg} \subseteq S_\varphi$.

- $\forall s \in S_\varphi$, $s \in PS_\varphi$. According to Lemma 6.4, $PS_{alg} = PS_\varphi$. Therefore, $s \in PS_{alg}$. It means that at some point, the *stack* was holding s without a conflict. At this point, all the important variables were assigned, and the *important-first* decision procedure started choosing non-important variables as decisions above the *important decision level*. The *All-SAT algorithm* uses regular SAT methods above this level. Therefore, at this point it solves a SAT problem for φ with a partial assignment s. If this problem is un-SAT, the algorithm tries to backtrack to a level lower than the *important decision level*.

  Since $s \in S_\varphi$, s is a part of an assignment A_s, which is a solution for φ. Therefore, φ, with the partial assignment s, is satisfiable. The correctness of the SAT procedures implies that the *All-SAT algorithm* will find a solution for φ when we use s as a partial assignment. In that case, the handle_solution() procedure will be executed, and s will be added to S_{alg}. Therefore, $\forall s \in S_\varphi, s \in S_{alg}$, and $S_{alg} \supseteq S_\varphi$.

- We showed that $S_{alg} \subseteq S_\varphi$ and $S_{alg} \supseteq S_\varphi$. Thus, $S_{alg} = S_\varphi$ and we conclude our proof.

□

Lemma 6.6 *$\forall s \in S_{alg}$, s is added to S_{alg} exactly once.*

Proof: Because of using the important-first decision procedure, the first variables in the search tree are the important variables. An assignment s to the important variables is added to S_{alg} when a leaf of this tree is reached. Whenever the *All-SAT algorithm* backtracks to level i which is lower than the *important decision level*, it flips the value of the decision variable in level $i+1$. Whenever a leaf is reached, and an assignment is added to S_{alg}, the *All-SAT algorithm* backtracks to the (*important decision level* - 1), and flips the value of the decision variable in the *important decision level*. This behavior ensures that each leaf which is reached during the search, has a unique path over the important variables.

When a leaf is reached (and only then), the corresponding assignment to the important variables is added to S_{alg}. Since each leaf originates in a different assignment to the important variables, two leaves reached never cause the addition of the same assignment to S_{alg}. Thus, each assignment is added to S_{alg} only once, and we conclude our proof. □

By proving Lemmas 6.5 and 6.6, we proved that $S_{alg} = S_{\varphi}$, and that $\forall s \in S_{alg}$, the *All-SAT algorithm* adds s to S_{alg} exactly once. Thus we complete the proof of Theorem 6.1. □

7 Reachability and Model Checking Using All-Solutions Solver

7.1 Reachability

We now present one application of our All-SAT algorithm. Given the set of initial states of a model, and its transition relation, we would like to find the set of all the states reachable from the initial states. We denote by $\overline{x}$ the vector of the model's state variables, and by $S_i(\overline{x})$ the set of states at distance i from the initial states. The transition relation is given by $T(\overline{x}, \overline{I}, \overline{x'})$ where $\overline{x}$ represents the current state, $\overline{I}$ represents the input and $\overline{x'}$ represents the next state. For a given set of states $S(\overline{x})$, the set of reachable states from them at distance 1 (i.e., their successors), denoted *Image*$(S(\overline{x}))$, is

$$Image(S(\overline{x})) = \{\overline{x'} \mid \exists \overline{I}, \exists \overline{x}, S(\overline{x}) \wedge T(\overline{x}, \overline{I}, \overline{x'})\} \tag{2}$$

Given S_0, the set of initial states, calculating the reachable states is done by iteratively calculating S_i, and adding them to the reachable set S^* until S_i contains no new states. The rechability algorithm is shown in Figure 12(a).

7.2 Image Computation Using All-SAT

We would like to use our All-SAT algorithm in order to implement line 5 in the reachability algorithm shown in Figure 12(a). We see that

$$Image(S(\overline{x})) \setminus S^* \equiv All-SAT(S(\overline{x}) \wedge T(\overline{x}, \overline{I}, \overline{x'}) \wedge \neg S^*(\overline{x'})) \tag{3}$$

Each solution to (3) represents a valid transition from one of the states in the set $S(\overline{x})$ to a new state $\overline{x'}$, which is not in $S^*(\overline{x'})$. Therefore, we have to find all the solutions for the following formula:

$$S(\overline{x}) \wedge T(\overline{x}, \overline{I}, \overline{x'}) \wedge \neg S^*(\overline{x'}) \tag{4}$$

By including $\neg S^*(\overline{x'})$ in the image computation we avoid finding the same state more than once. This was also done in [7, 22]. In order to use our All-SAT solver, we have to construct a CNF representation for formula (4).

$S(\overline{x})$ and $S^*(\overline{x})$ are sets of states. That is, sets of assignments to $\overline{x}$. We represent and store an assignment as a conjunction of the literals defining it. Let d_i be the conjunction describing $\overline{x_i}$, $S^*(\overline{x}) = (d_1 \vee d_2 \ldots)$. Since every d_i is a conjunction of literals, $S^*(\overline{x})$ is in DNF. Therefore, $\neg S^*(\overline{x})$ is in CNF. Generating it from the DNF clauses is linear in their size, simply by replacing each $\vee$ by $\wedge$ and vice versa, and by negating the literals.

The set $S(\overline{x}) = (d_1 \vee d_2 \ldots)$ is also in DNF. Generating the CNF clauses to describe it requires the following procedure: For each d_i in $S(\overline{x})$ we introduce a new variable Z_i. We create the CNF clauses which define the relations $\forall i[Z_i \Leftrightarrow d_i]$, and add the clause $(Z_1 \vee Z_2 \ldots)$. This way, each solution to Formula (4) must satisfy at least one of the variables Z_i, and thus is forced to satisfy the corresponding d_i. In order to force $Z_i \Rightarrow d_i$ we create, for each literal $l_j \in d_i$, the clause $(\neg Z_i, l_j)$, which is equivalent to $Z_i \Rightarrow l_j$. On the other direction, we add the clause $(Z_i, \neg l_1, \neg l_2 \ldots)$, which is equivalent to $Z_i \Leftarrow d_i$. This procedure is described in [3], and is also linear in the size of the solutions.

The procedure of representing T in CNF is described in [3] and is polynomial in the size of the original model. This procedure also requires the addition of variables other than $\overline{x}, \overline{I}$ and $\overline{x'}$. Unlike the procedures for representing $S(\overline{x})$ and $\neg S^*(\overline{x})$, this procedure is only used once, before starting the reachability analysis, as the transition relation does not change during the process.

When solving formula (4), each solution is an assignment to all of the variables in the CNF formula. The projection of a solution on $\overline{x'}$ defines the next state. We avoid repetitive finding of the same $\overline{x'}$ by setting the next state model variables to be the important variables in our All-SAT solver. Therefore, a state is found once, regardless of its predecessors, the input variables, or the assignment to the auxiliary variables added during the CNF construction procedure. In this way, we eliminate all the quantifiers in formula (4) simultaneously.

7.3 Minimization

7.3.1 Boolean Minimization

A major drawback of the implementation described above is the growth of S^* between iterations of the algorithm, and of S during the image computation. This poses a problem even though the solutions are held outside the solver. Representing each state by a conjunction of literals is pricy when their number increases. Therefore, we need a way to minimize this representation. We exploit the fact that a set of states is actually a DNF formula and apply Boolean minimization methods to find a minimal representation for it. For example, the solutions represented by $(x_0 \wedge x_1) \vee (x_0 \wedge \neg x_1)$, can be represented by (x_0).

In our tool, we use Berkeley's 'Espresso'[11], which receives as input a set of DNF clauses and returns their minimized DNF representation. Our experimental re-

```
$Reachability(S_0)$
1) $S^* \leftarrow S_0$
2) $S \leftarrow S_0$
3) $while\ (S \neq \phi)\ \{$
4)     $S^* \leftarrow S^* \cup S$
5)     $S \leftarrow Image(S) \setminus S^*$
6) $\}$
7) $return\ S^*$
```

(a) Reachability

```
$Reachability(T(\overline{x}, \overline{I}, \overline{x'}), S_0(\overline{x}))$
0) $S \leftarrow S_0$
1) $S^* \leftarrow S_0$
3) $while(S \neq \phi)\{$
4)     $P \leftarrow createCNF\ (S(\overline{x}) \wedge T(\overline{x}, \overline{I}, \overline{x'}) \wedge \neg S^*(\overline{x'}))$
5)     $S \leftarrow \phi$
5)     $repeat\{$
6)        $temp \leftarrow find\ next\ batch\ of\ solutions\ for\ (P)$
7)        $S \leftarrow S\ \cup\ temp$
8)        $minimize(S)$
9)        $S^* \leftarrow S^*\ \cup\ temp$
10)       $minimize(S^*)$
11)    $\}\ while((temp \neq \phi)$
12) $\}$
13) $return\ \ S^*$
```

(b) New Reachability

Figure 12: Reachability Algorithms. (a) Regular reachability algorithm. (b) Reachability using our All-SAT solver. Lines 5-11 are the implementation of the "on-the-fly" minimization.

sults show a reduction of up to 3 orders of magnitude in the number of clauses required to represent the sets of solutions when using this tool.

7.3.2 "On-The-Fly" Minimization

Finding all the solutions for the SAT problem (4), and then minimizing them, is not feasible for large problems, since their number would be too large to store and for the minimizer to handle. Therefore, during the solving process, whenever a pre-set number of solutions is found, the solver's work is suspended, and we use the minimizer to combine these solutions with the current S and S^*. The output is stored again in S and S^* respectively, to be processed with the next batch of solutions found by the All-SAT solver. This way we keep S^*, S and the input to the minimizer small. Note, also, that each batch of solutions includes new states only, due to the structure of Formula 4 and the behavior of the All-SAT solver. Employing logical minimization on-the-fly, before finding all the solutions, is possible since previous solutions are not required when searching for the next ones, and the work of the minimizer is independent of the All-SAT solver. Moreover, the minimization and the computation of the next batch of solutions can be performed concurrently. However, this is not implemented in our tool.

We now have a slightly modified reachability algorithm as shown in Figure 12(b).

8 Experimental Results

As the code base for our work, we use the zChaff SAT solver [23], which is one of the fastest a state of the art solver available. zChaff implements 1UIP conflict analysis with conflict driven backtracking, and uses efficient data structures. Since zChaff is an open source tool, we were able to modify the code according to our method.

For comparison with the All-SAT procedure, we used the same code base to implement the blocking clauses method. For fair comparison, we introduced the following improvement into the blocking clauses methods. In order to avoid repetition of the same assignments to the model variables, we constructed the blocking clauses from the model variables only. By using the important-first decision method described in 4.2, the blocking clauses can be constructed from only the subset of the model variables which are decision variables. This improvement reduced space requirements as well as solution times of the blocking clauses method substantially.

All experiments use dedicated computers with 1.7Ghz Intel Xeon CPU and 1GB RAM, running Linux operating system. The problem instances are from the ISCAS'89 benchmark.

8.1 All-SAT Solver

Figure 13 shows the performance of our new All-SAT solver. The problems are CNF representations of the transition relations of the models in the benchmark. In cases where a model name is followed by '*', the instance consists of multiple transitions and initial conditions. The important variables were chosen arbitrarily, and consist about half of the variables.

Group A in the table clearly shows a significant speedup for all the problems that the blocking clauses method could solve. A smaller number of clauses shortens the time of the *bcp()* procedure, and also allows more work to be performed in higher levels of the memory hierarchy (main memory and cache). The speedup increases with the hardness of the problem and the computation time.

Group B consists of larger instances, for which the blocking clauses method runs out of memory, yet our solver managed to complete. The number of solutions for these instances is simply too high to store in main memory as clauses. In contrast, using our new method, the solutions can be stored on the disk or in the memory of neighboring machines.

Group C in the table shows instances for which our solver timed out. In none of these cases, despite the huge number of solutions found (much larger than the size of the main memory in the machine employed), did the solver run out of memory. Our memory-efficient method clearly enables very long runs without memory blowup.

8.2 Reachability

Figure 14 shows the performance of our reachability analysis tool, calculating reachable states of the benchmark models. Since [7] and [22] are the only reachability analysis algorithms that, as far as we know, depend solely on SAT procedures, the table shows a comparison with the performance reported in [7]. Figure 15 shows the instances for which the reachability analysis did not complete. Here, again, a comparison to [7] is shown. The tables show significant speedup for the completed problems, and completion of a larger number of steps for the problems not completed.

9 Conclusions

In this work we presented an efficient All-SAT engine which, unlike current engines, does not add clauses, BDDs or any other form of data, in order to block solutions already found. By avoiding the addition of this data, we gain two main advantages.

First, the All-SAT algorithm is independent of the solutions already found. Therefore, their representation can be changed, they can be processed and even deleted, without affecting the completeness of the search, or its efficiency.

Second, the memory requirements are independent of the number of solutions. This implies that, during the computation, the number of solutions already found does not

	Name	# Clauses	# Variables	# Solutions	T1 (s)	T2 (s)	Speedup
A	s510	964	280	64	0.00	0.00	1.00
	s1488	983	286	64	0.00	0.00	1.00
	s1494	19153	4919	12	0.00	0.00	1.00
	s15850	166	48	2	0.00	0.00	1.00
	s208.1	47	22	11	0.00	0.00	1.00
	s23	303	97	5	0.00	0.00	1.00
	s298	437	137	8	0.00	0.00	1.00
	s382	462	140	8	0.00	0.00	1.00
	s400	462	96	2	0.00	0.00	1.00
	s420.1	498	153	5	0.00	0.00	1.00
	s444	1620	129	2	0.00	0.00	1.00
	s499	561	161	5	0.00	0.00	1.00
	s526	1528	192	3	0.00	0.00	1.00
	s838.1	1406	191	2	0.00	0.00	1.00
	s938	558	160	5	0.00	0.00	1.00
	s526n	479	138	36	0.00	0.00	0.50
	s208.1*	160	50	512	0.01	0.01	1.00
	s713	158	48	512	0.01	0.01	2.20
	s208.1*	190	61	512	0.01	0.02	2.67
	s832	454	136	183	0.01	0.02	1.29
	s820	404	129	344	0.01	0.03	3.11
	s208.1*	585	161	480	0.03	0.05	1.77
	s208.1*	610	167	960	0.06	0.10	1.64
	s641	10064	2338	224	0.03	0.16	5.81
	s953	1279	271	1188	0.07	0.24	3.52
	s1238	49699	13476	80	0.06	0.48	8.39
	s9234.1	239	77	640	0.35	0.55	1.57
	s967	959	277	6504	0.14	0.59	4.35
	s38584	1302	299	2272	0.13	0.81	6.31
	s1423	1884	560	28590	0.45	29.9	66.44
	s1269	2191	679	32768	0.82	50	60.75
	s13207*	18993	3890	24576	4.40	459	104.39
	s13207	37986	6613	97572	11.50	1542	134.09
B	s3271	4426	1349	6.70E+07	1020	M.O.	-
	s9234.1	10007	2317	3.50E+07	3220	M.O.	-
	s1512	2320	657	1.30E+08	4151	M.O.	-
	s3330	2496	775	1.50E+08	4375	M.O.	-
	s38417	48783	13261	8.30E+06	11379	M.O.	-
	s5378	1072	4031	5.00E+08	25102	M.O.	-
	s635*	1496	192	> 4E+09	98487	M.O.	-
	s6669	11963	3639	> 4E+09	157671	M.O.	-
C	s13207.1	18774	3847	> 1E+09	T.O.	M.O.	-
	s15850.1	18957	4850	> 1.2E+09	T.O.	M.O.	-
	s3384	5073	1784	> 4E+09	T.O.	M.O.	-
	s35932	55173	20155	> 7.5E+07	T.O.	M.O.	-
	s38584.1	49735	13449	> 2E+08	T.O.	M.O.	-
	s4863	10470	3001	> 1.3E+09	T.O.	M.O.	-
	s991	1635	560	> 5.5E+08	T.O.	M.O.	-

Figure 13: All-SAT solving time. T1 - The time required for our new All-SAT solver. T2 - The time required for the blocking clauses-based algorithm. M.O. - Memory Out. T.O. - Time out (48 hours) Speedup - The ratio T2/T1.

become a parameter of complexity in finding further solutions. It also implies that the size of the instance being solved fits in smaller and faster levels of the memory hierarchy. As a result, our method is faster than blocking clauses based methods, and can solve instances that produce solution sets too large to fit in the machine's memory.

Our All-SAT engine uses Chaff [23] as its base code, and experimental results show significant speedup compared to the blocking clauses method used so far for this purpose. Moreover, our tool can cope with large instances which caused memory blowup when using the blocking clauses method. We applied our tool on even harder problems, which it did not complete solving. These runs were stopped after a timeout

Model	# FLOPS	# steps	# states	T1 (sec)	T2 (sec)	Speedup
s386	6	8	13	0.1	0.21	2.10
s298	14	19	218	0.2	0.33	1.65
s832	5	11	25	0.1	0.47	4.70
s510	6	47	47	0.1	0.47	4.70
s820	5	11	25	0.2	0.48	2.40
s208.1	8	256	256	0.1	0.56	5.60
s1488	6	22	48	0.3	0.87	2.90
s1494	6	22	48	0.1	0.87	8.70
s499	22	22	22	0.3	1.74	5.80
s953	29	10	504	0.1	2.01	20.10
s641	19	7	1544	0.2	2.24	11.20
s713	19	7	1544	1	2.53	2.53
s967	29	10	549	0.2	3.12	15.60
s1196	18	3	2615	0.3	6.79	22.63
s1238	18	3	2615	0.2	7.26	36.30
s382	21	151	8865	4	7.7	1.93
s400	21	151	8865	4	7.8	1.95
s444	21	151	8865	4	8	2.00
s526n	21	151	8868	5	9.21	1.84
s526	21	151	8868	5	9.35	1.87
s349	15	7	2625	3	14.8	4.93
s344	15	7	2625	3	15.3	5.10

Figure 14: Reachability Analysis Performance. #FLOPS is the number of flip-flops in the model. #states is the total number of reachable states. T1 is the time required for our tool. T2 is the time as given in [7] using 1.5Ghz dual AMD Athlon CPU with 3GB RAM.

Model	#FLOPS	Depth 1 [7] (1000 sec')	Time to reach Depth 1 (sec)	Max depth Completed	Actual time for max depth (sec)
S1269	37	1	9	1	9
S1423	74	3	25	4	580
S13207	669	2	8	3	120
S1512	57	4	64	23	2980
S9234	228	8	291	111	23853
S15850	597	5	180	7	7793
S38584	1452	2	1	4	1921

Figure 15: Reachability Analysis Performance. 'Depth 1' is the maximal depth reached in [7] with timeout of 1000 seconds. 'Time to reach Depth 1' is the time required for our tool to complete the same depth. The 'Max depth' and the 'Actual time to reach max depth' are the maximal steps successfully completed by our tool, and the time required for it. The timeout is generally 3600 seconds (1 hour), with longer timeouts for s1512, s9234 and s15850.

of 48 hours. Our solver did not run out of memory in any of these cases.

We have demonstrated how to use our All-SAT engine for quantifiers elimination in image computation. By assigning the next state model variables to be the important variables in our All-SAT solver, we eliminate all the quantifiers on the current state and input variables. The result is a memory-efficient reachability computation, based solely on SAT procedures. This reachability tool proved to reach deeper in the reachability analysis compared to other tools based only on SAT procedures, and showed significant speedup on many problems.

References

[1] I. Beer, S. Ben-David, and A. Landver. On-the-fly model checking of RCTL formulas. In *10th Computer Aided Verification (CAV'98)*, volume 1427, pages 184–194, Vancouver, June 1998. Springer-Verlag.

[2] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *proceedings of the 36rd Design Automation Conference (DAC'99)*, New Orleans, June 1999. IEEE Computer Society Press.

[3] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. *Lecture Notes in Computer Science*, 1579:193–207, 1999.

[4] Elazar Birnbaum and Eliezer L. Lozinskii. The good old davis-putnam procedure helps counting models. *Journal of Artificial Intelligence Research*, 10:457–477, 1999.

[5] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE transactions on Computers*, C-35(8):677–691, 1986.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[7] P. Chauhan, E. M. Clarke, and D. Kroening. Using SAT based image computation for reachability analysis. Technical Report CMU-CS-03-151, Carnegie Mellon University, 2003.

[8] P. P. Chauhan, E. M. Clarke, and D. Kroening. A SAT-based algorithm for reparameterization in symbolic simulation. In *Design Automation Conference*, San Diego, June 2004. ACM.

[9] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *CACM*, 5(7), July 1962.

[10] M. Davis and H. Putnam. A computing procedure for quantification theory. *JACM*, 7(3):201–215, July 1960.

[11] Espresso, two level boolean minimizer. university of california, berkeley, 1990.

[12] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[13] E. Goldberg and Y. Novikov. Berkmin: A fast and robust SAT-solver. In *Design, Automation and Test in Europe Conference and Exposition (DATE'02)*, Paris, March 2002. IEEE Computer Society.

[14] Aarti Gupta, Zijiang Yang, Pranav Ashar, and Anubhav Gupta. SAT-based image computation with application in reachability analysis. In *Formal Methods in Computer-Aided Design, Third International Conference (FMCAD '00)*, volume 1954 of *LNCS 1954*, Austin, November 2000. Springer.

[15] Roberto J. Bayardo Jr. and Joseph Daniel Pehoushek. Counting models using connected components. In *Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI '00)*, pages 157–162, Austin, July 2000. AAAI Press / The MIT Press.

[16] H. J. Kang and I. C. Park. SAT-based unbounded symbolic model checking. In *Proceedings of the 40th Design Automation Conference (DAC '03)*, Anaheim, June 2003. ACM.

[17] S. K. Lahiri, R. E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In *Computer Aided Verification, 15th International Conference (CAV '03)*, volume 2725, Boulder, July 2003. Springer.

[18] R. Letz. Advances in decision procedures for quantified boolean formulas. In *International Joint Conference on Automated Reasoning (IJCAR '01)*, Siena, June 2001.

[19] B. Li, M. S. Hsiao, and S. Sheng. A novel SAT all-solutions solver for efficient preimage computation. In *Design, Automation and Test in Europe Conference and Exposition (DATE '04)*, pages 272–279, Paris, February 2004. IEEE Computer Society.

[20] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Fifteenth International Joint Conference on Artificial Intelligence (IJCAI '97)*, pages (1)366–371, Nagoya, Japan, August 1997. Morgan Kaufmann.

[21] J.P. Marques-Silva and K.A. Sakallah. Conflict analysis in search algorithms for propositional satisfiability. In *Eigth International Conference on Tools with Artificial Intelligence (ICTAI '96)*, Toulouse, November 1996. IEEE Computer Society.

[22] Ken L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Computer Aided Verification, 14th International Conference (CAV '02)*, volume 2404 of *Lecture Notes in Computer Science*, Copenhagen, July 2002. Springer.

[23] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *39th Design Aotomation Conference (DAC'01)*, 2001.

[24] D. Plaisted. Method for design verification of hardware and non-hardware systems. United States Patents, 6,131, 078, October 2000.

[25] D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1-2), 1996.

[26] S. Sapra, M. Theobald, and E. M. Clarke. SAT-based algorithms for logic minimization. In *21st International Conference on Computer Design (ICCD '03)*, San Jose, October 2003. IEEE Computer Society.

[27] S. Sheng and M. S. Hsiao. Efficient preimage computation using a novel success-driven ATPG. In *Design, Automation and Test in Europe Conference and Exposition (DATE '03)*, pages 822–827, Munich, March 2003. IEEE Computer Society.

[28] Ofer Shtrichman. Tuning SAT checkers for bounded model checking. In *Computer Aided Verification, 12th International Conference (CAV '00)*, volume 1855 of *Lecture Notes in Computer Science*, Chicago, July 2000. Springer.

[29] Ramin Zabih and David McAllester. A rearrangement search strategy for determining propositional satisfiability. In *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI '88)*, pages 155–160. AAAI Press / The MIT Press, August 21-26 1988.

[30] H. Zhang. SATO: An efficient propositional prover. In *Proc. of the 14th International Conference on Automated Deduction (CADE'97)*, 1997.

[31] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *International Conference on Computer-Aided Design (ICCAD '01)*, pages 279–285, San Jose, November 2001. ACM.