GRASP—A New Search Algorithm for Satisfiability

João P. Marques Silva
Karem A. Sakallah

CSE-TR-292-96

April 10, 1996

THE UNIVERSITY OF MICHIGAN

Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
Ann Arbor, Michigan 48109-2122
USA

GRASP—A New Search Algorithm for Satisfiability

João P. Marques Silva
Karem A. Sakallah

Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, Michigan 48109-2122

April 10, 1996

Abstract

This report introduces GRASP (Generic seaRch Algorithm for the Satisfiability Problem), an integrated algorithmic framework for SAT that unifies several previously proposed search-pruning techniques and facilitates identification of additional ones. GRASP is premised on the inevitability of conflicts during search and its most distinguishing feature is the augmentation of basic backtracking search with a powerful conflict analysis procedure. Analyzing conflicts to determine their causes enables GRASP to backtrack non-chronologically to earlier levels in the search tree, potentially pruning large portions of the search space. In addition, by "recording" the causes of conflicts, GRASP can recognize and preempt the occurrence of similar conflicts later on in the search. Finally, straightforward bookkeeping of the causality chains leading up to conflicts allows GRASP to identify assignments that are necessary for a solution to be found. Experimental results obtained from a large number of benchmarks, including many from the field of test pattern generation, indicate that application of the proposed conflict analysis techniques to SAT algorithms can be extremely effective for a large number of representative classes of SAT instances.

1 Introduction

The Boolean satisfiability problem (SAT) appears in many contexts in the field of computer-aided design of integrated circuits including automatic test pattern generation (ATPG), timing analysis, delay fault testing, and logic verification, to name just a few. Though well-researched and widely investigated, it remains the focus of continuing interest because efficient techniques for its solution can have great impact. SAT belongs to the class of NP-complete problems whose algorithmic solutions are currently believed to have exponential worst case complexity [11]. Over the years, many algorithmic solutions have been proposed for SAT, the most well known being the different variations of the Davis-Putnam procedure [5]. The best known version of this procedure is based on a backtracking search algorithm that, at each node in the search tree, elects an assignment and prunes subsequent search by iteratively applying the *unit clause* and the *pure literal* rules [30]. Iterated application of the unit clause rule is commonly referred to as *Boolean Constraint Propagation* (BCP) [30] or as *derivation of implications* in the electronic CAD literature [1].

Most of the recently proposed improvements to the basic Davis-Putnam procedure [2, 9, 10, 17, 28, 30] can be distinguished based on their decision making heuristics or their use of preprocessing or relaxation techniques. Common to all these approaches, however, is the chronological nature of backtracking. Only in [19] is a non-chronological backtracking procedure outlined for solving SAT, but it is only sketched and no experimental results are presented. Nevertheless, non-chronological backtracking techniques have been extensively studied and applied to different areas of Artificial Intelligence, particularly Truth Maintenance Systems (TMS) [7, 27], Constraint Satisfaction Problems (CSP) [6, 12, 13, 21] and Automated Deduction [3], in some cases with very promising experimental results [6, 13].

Interest in the direct application of SAT algorithms to electronic design automation (EDA) problems has been on the rise recently [4, 17, 20, 28]. In addition, improvements to the traditional structural (path sensitization) algorithms for some EDA problems, such as ATPG, include search-pruning techniques that are also applicable to SAT algorithms in general [14, 16, 25].

This report introduces GRASP (***Generic seaRch Algorithm for the Satisfiability Problem***), an integrated algorithmic framework for SAT that unifies several previously proposed search-pruning techniques and facilitates identification of additional ones. GRASP is premised on the inevitability of conflicts during search and its most distinguishing feature is the augmentation of basic backtracking search with a powerful ***conflict analysis*** procedure. Analyzing conflicts to determine their causes enables GRASP to backtrack non-chronologically to earlier levels in the search tree, potentially pruning large portions of the search space. In addition, by "recording" the causes of conflicts, GRASP can recognize and preempt the occurrence of similar conflicts later on in the search. Finally, straightforward bookkeeping of the causality chains leading up to conflicts allows GRASP to identify assignments that are necessary for a solution to be found. Experimental results obtained from a large number of benchmarks [8] indicate that application of the proposed conflict analysis techniques to SAT algorithms can be extremely effective for a large number of representative classes of SAT instances.

Several features distinguish the conflict analysis procedure in GRASP from others used in TMSs and CSPs. First, conflict analysis in GRASP is tightly coupled with BCP and the causes of conflicts need not necessarily correspond to decision assignments. Second, clauses can be added to the original set of clauses, and the number and size of added clauses is user-controlled. This is in explicit contrast with nogood recording techniques developed for TMSs and CSPs. Third, GRASP employs techniques to prune the search by analyzing the implication *structure* generated by BCP. Exploiting the "anatomy" of conflicts in this manner has no equivalent in other areas.

Some of the proposed techniques have also been applied in structural ATPG algorithms [14, 18, 26]. The GRASP framework, however, permits a unified representation of all known search-pruning methods and introduces several others. The basic SAT algorithm in GRASP is also customizable to take advantage of application-specific characteristics to achieve additional efficiencies [24, 25]. Finally, the framework is organized to allow easy adaptation of other algorithmic techniques, such as those in [4, 16], whose operation is orthogonal to those described here.

The remainder of this report is organized in four sections. In Section 2, we introduce the basics of backtracking search, particularly our implementation of BCP, and describe the overall architecture of GRASP. This is followed, in Section 3, by a detailed discussion of the procedures for conflict analysis and how they are implemented. Extensive experimental results on a wide range of benchmarks, including many from the field of ATPG, are presented and analyzed in Section 4. In particular, GRASP is shown to outperform two recent state-of-the-art SAT algorithms [9, 28] on most, but not all, benchmarks. The report concludes in Section 5 with some suggestions for further research.

2 Backtrack Search for CNF Satisfiability

2.1 Basic Definitions and Notation

A conjunctive normal form (CNF) formula φ on n binary variables $x_1, \ldots, x_n$ is the conjunction (AND) of m ***clauses*** $\omega_1, \ldots, \omega_m$ each of which is the disjunction (OR) of one or more literals, where a literal is the occurrence of a variable or its complement. A formula φ denotes a unique n-variable Boolean function $f(x_1, \ldots, x_n)$ and each of its clauses corresponds to an implicate of f [15, p. 288]. Clearly, a function f can be represented by many equivalent CNF formulas. A formula is complete if it consists of the entire set of prime implicates for the corresponding function. In general, a complete formula will have an exponential number of clauses. We will refer to a CNF formula as a ***clause database*** and use "formula," "CNF formula," and "clause database" interchangeably. The satisfiability problem (SAT) is concerned with finding an assignment to the arguments of $f(x_1, \ldots, x_n)$ that makes the function equal to 1 or proving that the function is equal to the constant 0.

A backtracking search algorithm for SAT is implemented by a ***search process*** that implicitly traverses the space of 2^n possible binary assignments to the problem variables. During the search, a variable whose binary value has already been determined is considered to be ***assigned***; otherwise it is ***unassigned*** with an implicit value of $X \equiv \{0, 1\}$. A ***truth assignment*** for a formula φ is a set of assigned variables and their corresponding binary values. It will be convenient to represent such assignments as sets of variable/value pairs; for example $A = \{(x_1, 0), (x_7, 1), (x_{13}, 0)\}$. Alternatively, assignments can be denoted as $A = \{x_1 = 0, x_7 = 1, x_{13} = 0\}$. Sometimes it is convenient to indicate that a variable x is assigned without specifying its actual value. In such cases, we will use the notation $\nu(x)$ to denote the binary value assigned to x. An assignment A is complete if $|A| = n$; otherwise it is partial. Evaluating a formula φ for a given a truth assignment A yields three possible outcomes: $\varphi|_A = 1$ and we say that φ is satisfied and refer to A as a ***satisfying assignment***; $\varphi|_A = 0$ in which case φ is unsatisfied and A is referred to as an ***unsatisfying assignment***; and $\varphi|_A = X$ indicating that the value of φ cannot be resolved by the assignment. This last case can only happen when A is a partial assignment. An assignment partitions the clauses of φ into three sets: satisfied clauses (evaluating to 1); unsatisfied clauses (evaluating to 0); and unresolved clauses (evaluating to X). The unassigned literals of a clause are referred to as its ***free literals***. A clause is said to be ***unit*** if the number of its free literals is one.

2.2 Formula Satisfiability

Formula satisfiability is concerned with determining if a given formula φ is satisfiable and with identifying a satisfying assignment for it. Starting from an empty truth assignment, a backtrack search algorithm traverses the space of truth assignments implicitly and organizes the search for a satisfying assignment by maintaining a ***decision tree***. Each node in the decision tree specifies an elective assignment to an unassigned variable; such assignments are referred to as ***decision assignments***. A ***decision level*** is associated with each decision assignment to denote its depth in the decision tree; the first decision assignment at the root of the tree is at decision level 1. The search process iterates through the steps of:

1. Extending the current assignment by making a decision assignment to an unassigned variable. This ***decision process*** is the basic mechanism for exploring new regions of the search space. The search terminates successfully if all clauses become satisfied; it terminates unsuccessfully if some clauses remain unsatisfied and all possible assignments have been exhausted.

2. Extending the current assignment by following the logical consequences of the assignments made thus far. The additional assignments derived by this ***deduction process*** are referred to as ***implication assignments*** or, more simply, ***implications***. The deduction process may also lead to the identification of one or more unsatisfied clauses implying that the current assignment is not a satisfying assignment. Such an occurrence is referred to as a ***conflict*** and the associated unsatisfying assignment is called a ***conflicting assignment***.

3. Undoing the current assignment, if it is conflicting, so that another assignment can be tried. This ***backtracking process*** is the basic mechanism for retreating from regions of the search space that do not correspond to satisfying assignments.

The decision level at which a given variable x is either electively assigned or forcibly implied will be denoted by $\delta(x)$. When relevant to the context, the assignment notation introduced earlier may be extended to indicate the decision level at which the assignment occurred. Thus, $x = v@d$ would be read as "x becomes equal to v at decision level d."

The average complexity of the above search process depends on how decisions, deductions, and backtracking are made. It also depends on the formula itself. The implications that can derived from a given partial assignment depend on the set of available clauses. In general, a formula consisting of more clauses will enable more implications to be derived and will reduce the number of backtracks due to conflicts. The limiting case is the complete formula that contains all prime implicates. For such a formula no conflicts can arise since all logical implications for a partial assignment can be derived. This, however, may not lead to shorter execution times since the size of such a formula may be exponential.

2.3 Function Satisfiability

Given an initial formula φ many search systems attempt to augment it with additional implicates to increase the deductive power during the search process. This is usually referred to as "learning" [22] and can be performed either as a preprocessing step (static learning) or during the search (dynamic learning)[1].

Our approach can be classified as a dynamic learning search mechanism based on diagnosing the causes of conflicts. It considers the occurrence of a conflict, which is unavoidable for an unsatisfiable instance unless the formula is complete, as an

1. Learning as defined in [17, 22] only yields implicates of size 2 (i.e. non-local implications) but the concept can be readily extended to implicates of arbitrary size.

opportunity to "learn from the mistake that led to the conflict" and introduces additional implicates to the clause database only when it stumbles. ***Conflict diagnosis*** produces three distinct pieces of information that can help speed up the search:

1. New implicates that did not exist in the clause database and that can be identified with the occurrence of the conflict. These clauses may be added to the clause database to avert future occurrence of the same conflict and represent a form of ***conflict-based equivalence*** (CBE).

2. An indication of whether the conflict was ultimately due to the most recent decision assignment or to an earlier decision assignment.

   a. If that assignment was the most recent (i.e. at the current decision level), the opposite assignment (if it has not been tried) is immediately implied as a necessary consequence of the conflict; we refer to this as a ***failure-driven assertion*** (FDA).

   b. If the conflict resulted from an earlier decision assignment (at a lower decision level), the search can backtrack to the corresponding level in the decision tree since the subtree rooted at that level corresponds to assignments that will yield the same conflict. The ability to identify a backtracking level that is much earlier than the current decision level is a form of non-chronological backtracking that we refer to as ***conflict-directed backtracking*** (CDB), and has the potential of significantly reducing the amount of search.

These conflict diagnosis techniques are discussed further in Section 3.

2.4 Structure of the Search Process

The basic mechanism for deriving implications from a given clause database is Boolean constraint propagation (BCP) [9, 30]. Consider a formula φ containing the clause $\omega = (x + \neg y)$ and assume $y = 1$. For any satisfying assignment to φ, ω requires that x be equal to 1, and we say that $y = 1$ implies $x = 1$ due to ω. In general, given a unit clause $(l_1 + \ldots + l_k)$ of φ with free literal l_j, consistency requires $l_j = 1$ since this represents the only possibility for the clause to be satisfied. If $l_j = x$, then the assignment $x = 1$ is required; if $l_j = \neg x$ then $x = 0$ is required. Such assignments are referred to as ***logical implications*** (implications, for short) and correspond to the application of the unit clause rule proposed by M. Davis and H. Putnam [5]. BCP refers to the iterated application of this rule to a clause database until the set of unit clauses becomes empty or one or more clauses become unsatisfied.

Let the assignment of a variable x be implied due to a clause $\omega = (l_1 + \ldots + l_k)$. The ***antecedent assignment*** of x, denoted as $A(x)$, is defined as the set of assignments to variables other than x with literals in ω. Intuitively, $A(x)$ designates those variable assignments that are directly responsible for implying the assignment of x due to ω. For example, the antecedent assignments of x, y and z due to the clause $\omega = (x + y + \neg z)$ are, respectively, $A(x) = \{y = 0, z = 1\}$, $A(y) = \{x = 0, z = 1\}$, and $A(z) = \{x = 0, y = 0\}$. Note that the antecedent assignment of a decision variable is empty.

The sequence of implications generated by BCP is captured by a directed ***implication graph I*** defined as follows (see Figure 2-1):

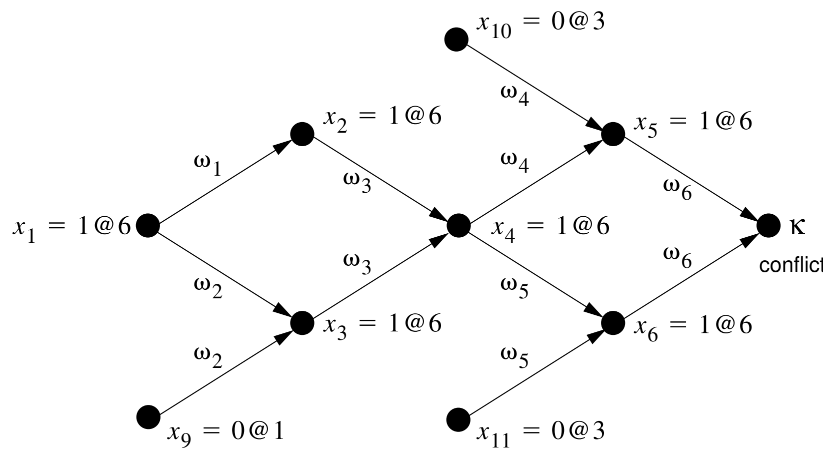
1. Each vertex in I corresponds to a variable assignment $x = \nu(x)$.

Current Truth Assignment: $\{x_9 = 0@1, x_{10} = 0@3, x_{11} = 0@3, x_{12} = 1@2, x_{13} = 1@2\}$

Current Decision Assignment: $\{x_1 = 1@6\}$

$$\begin{aligned}
\omega_1 &= (\neg x_1 + x_2) \\
\omega_2 &= (\neg x_1 + x_3 + x_9) \\
\omega_3 &= (\neg x_2 + \neg x_3 + x_4) \\
\omega_4 &= (\neg x_4 + x_5 + x_{10}) \\
\omega_5 &= (\neg x_4 + x_6 + x_{11}) \\
\omega_6 &= (\neg x_5 + \neg x_6) \\
\omega_7 &= (x_1 + x_7 + \neg x_{12}) \\
\omega_8 &= (x_1 + x_8) \\
\omega_9 &= (\neg x_7 + \neg x_8 + \neg x_{13})
\end{aligned}$$

Clause Database φ



Implication Graph *I* for Current Decision Assignment

Figure 2-1: Example of clause database and partial implication graph

2. The predecessors of vertex $x = \nu(x)$ in *I* are the antecedent assignments $A(x)$ corresponding to the unit clause ω that led to the implication of *x*. The directed edges from the vertices in $A(x)$ to vertex $x = \nu(x)$ are all labeled with ω. Vertices that have no predecessors correspond to decision assignments.

3. Special conflict vertices are added to *I* to indicate the occurrence of conflicts. The predecessors of a conflict vertex κ correspond to variable assignments that force a clause ω to become unsatisfied and are viewed as the antecedent assignment $A(\kappa)$. The directed edges from the vertices in $A(\kappa)$ to κ are all labeled with ω.

The decision level of an implied variable *x* is related to those of its antecedent variables according to:

$$\delta(x) = \max\{\delta(y) | (y, \nu(y)) \in A(x)\} \tag{2.1}$$

2.5 Search Algorithm Template

The general structure of the GRASP search algorithm is shown in Figure 2-2. We assume that an initial clause database φ and an initial assignment *A*, at decision level 0, are given. This initial assignment, which may be empty, may be viewed as an additional problem constraint and causes the search to be restricted to a subcube of the *n*-dimensional Boolean space. As the search proceeds, both φ and *A* are modified. The recursive search procedure consists of four major operations:

1. `Decide()`, which chooses a decision assignment at each stage of the search process. Decision procedures are commonly based on heuristic knowledge. For the results given in Section 4, the following greedy heuristic is used:

```
// Global variables:                     Clause database
//                                       Partial variable assignment A
// Return value:                         FAILURE or SUCCESS
// Auxiliary variables:                  Backtracking decision level β
//
GRASP()
{
     return ((Search (0, β) != SUCCESS) ? FAILURE : SUCCESS);
}

// Input argument:                       Current decision level d
// Output argument:                      Backtracking decision level β
// Return value:                         CONFLICT or SUCCESS
//
Search (d, &β)
{
     if (Decide (d) == SUCCESS)
         return SUCCESS;
     while (TRUE) {
         if (Deduce (d) != CONFLICT) {
           if (Search (d + 1, β) == SUCCESS) return SUCCESS;
           else if (β != d) { Erase(); return CONFLICT;}
         }
         if (Diagnose (d, β) == CONFLICT) {Erase(); return CONFLICT;}
         Erase();
     }
}
```

Figure 2-2: Description of GRASP

At each node in the decision tree evaluate the number of clauses directly satisfied by each assignment to each variable. Choose the variable and the assignment that directly satisfies the largest number of clauses.

Other decision making procedures have been implemented in the GRASP algorithmic framework, particularly those described in [9]. For these heuristics, preference is given to assignments that simplify the clauses the most, and can lead to more implications due to BCP. This is in explicit contrast with our heuristic which always attempts to satisfy the largest number of clauses.

2. `Deduce()`, which implements BCP and (implicitly) maintains the resulting implication graph. The pseudo-code for this procedure is shown in Figure 2-3.

3. `Diagnose()`, which identifies the causes of conflicts and can augment the clause database with additional implicates. Realization of different conflict diagnosis procedures is the subject of Section 3.

4. `Erase()`, which deletes the assignments at the current decision level.

```
// Global variables:                Implication graph I
// Input argument:                  Current decision level d
// Return value:                    CONFLICT or SUCCESS
//
Deduce (d)
{
    while (unit clauses in φ or clauses unsatisfied) {
        if (exists unsatisfied clause ω) {
            add conflict vertex κ to I;
            record A(κ);
            return CONFLICT;
        }
        if (exists unit clause ω with free literal l = x or l = ¬x) {
            record A(x);
            δ(x) = d;
            set x = 1 if l = x or x = 0 if l = ¬x;
        }
    }
    return SUCCESS;
}
```

Figure 2-3: Description of the deduction engine

We refer to `Decide()`, `Deduce()` and `Diagnose()` as the *Decision*, *Deduction* and *Diagnosis* ***engines***, respectively. Different realizations of these engines lead to different SAT algorithms. For example, the Davis-Putnam procedure can be emulated with the above algorithm by defining a decision engine, requiring the deduction engine to implement BCP and the pure literal rule, and organizing the diagnosis engine to implement chronological backtracking.

3 Conflict Analysis Procedures

When a conflict arises during BCP, the *structure* of the implication sequence converging on a conflict vertex κ is analyzed to determine those (unsatisfying) variable assignments that are directly responsible for the conflict. The conjunction of these conflicting assignments is an implicant that represents a sufficient condition for the conflict to arise. Negation of this implicant, therefore, yields an implicate of the Boolean function *f* (whose satisfiability we seek) that does not exist in the clause database φ. This new implicate[2], referred to as a ***conflict-induced clause***, provides the primary mechanism for implementing failure-driven assertions, non-chronological conflict-directed backtracking, and conflict-based equivalence (see Section 2.3).

We denote the conflicting assignment associated with a conflict vertex κ by $A_C(\kappa)$ and the associated conflict-induced clause by $\omega_C(\kappa)$. The conflicting assignment is determined by a backward traversal of the implication graph starting at κ. Besides the decision assignment at the current decision level, only those assignments that occurred at previous decision levels

2. Conditions similar to these implicates are referred to as "nogoods" in TMS [7, 27] and in some algorithms for CSP [21]. Nevertheless, the basic mechanism for creating conflict-induced clauses differs.

are included in $A_C(\kappa)$. This is justified by the fact that the decision assignment at the current decision level is directly responsible for all implied assignments at that level. Thus, along with assignments from previous levels, the decision assignment at the current decision level is a sufficient condition for the conflict. To facilitate the computation of $A_C(\kappa)$ we partition the antecedent assignments of κ as well as those for variables assigned at the current decision level into two sets. Let x denote either κ or a variable that is assigned at the current decision level. The partition of $A(x)$ is then given by:

$$\begin{aligned} \Lambda(x) &= \{(y, \nu(y)) \in A(x) \mid \delta(y) < \delta(x)\} \\ \Sigma(x) &= \{(y, \nu(y)) \in A(x) \mid \delta(y) = \delta(x)\} \end{aligned} \tag{3.1}$$

For example, referring to the implication graph of Figure 2-1, $\Lambda(x_6) = \{x_{11} = 0@3\}$ and $\Sigma(x_6) = \{x_4 = 1@6\}$. Determination of the conflicting assignment $A_C(\kappa)$ can now be expressed as:

$$\boldsymbol{A}_C(\kappa) = \text{causesof}(\kappa) \tag{3.2}$$

where $\text{causesof}(\cdot)$ is defined by:

$$\text{causesof}(x) = \begin{cases} (x, \nu(x)) & \text{if } A(x) = \varnothing \\ \Lambda(x) \cup \left[\bigcup_{(y, \nu(y)) \in \Sigma(x)} \text{causesof}(y) \right] & \text{otherwise} \end{cases} \tag{3.3}$$

The conflict-induced clause corresponding to $A_C(\kappa)$ is now determined according to:

$$\omega_C(\kappa) = \sum_{(x, \nu(x)) \in A_C(\kappa)} x^{\nu(x)} \tag{3.4}$$

where, for a binary variable x, $x^0 \equiv x$ and $x^1 \equiv \neg x$. Application of (3.1)-(3.4) to the conflict depicted in Figure 2-1 yields the following conflicting assignment and conflict-induced clause at decision level 6:

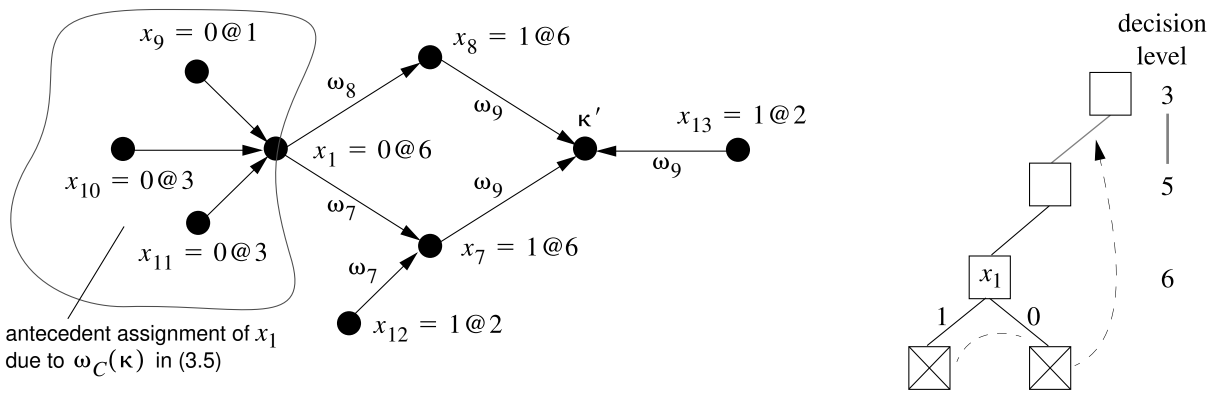
$$\begin{aligned} A_C(\kappa) &= \{x_1 = 1@6, x_9 = 0@1, x_{10} = 0@3, x_{11} = 0@3\} \\ \omega_C(\kappa) &= (\neg x_1 + x_9 + x_{10} + x_{11}) \end{aligned} \tag{3.5}$$

3.1 Standard Conflict Diagnosis Engine

The identification of a conflict-induced clause $\omega_C(\kappa)$ enables the derivation of further implications that help prune the search. Immediate implications of $\omega_C(\kappa)$ include asserting the current decision variable to its opposite value and determining a backtracking level for the search process. Such immediate implications do not require that $\omega_C(\kappa)$ be added to the clause database. Augmenting the clause database with $\omega_C(\kappa)$, however, has the potential of identifying future implications that are not derivable without $\omega_C(\kappa)$. In particular, adding $\omega_C(\kappa)$ to the clause database insures that the search engine will not regenerate the conflicting assignment that led to the current conflict.

3.1.1 Failure-Driven Assertions

If $\omega_C(\kappa)$ involves the current decision variable, erasing the implication sequence at the current decision level makes $\omega_C(\kappa)$ a unit clause and causes the immediate implication of the decision variable to its opposite value. We refer to such



(a) Conflicting implication sequence (b) Decision tree

Figure 3-1: Implication sequence and backtracking due to asserting $x_1 = 0$

assignments as failure-driven assertions (FDAs) to emphasize that they are implications of conflicts and not decision assignments. We note further that their derivation is automatically handled by our BCP-based deduction engine and does not require special processing. This is in contrast with most search-based SAT algorithms that treat a second branch at the current decision level as another decision assignment. Using our running example (see Figure 2-1) as an illustration, we note that after erasing the conflicting implication sequence at level 6, the conflict-induced clause $\omega_C(\kappa)$ in (3.5) becomes a unit clause with $\neg x_1$ as its free literal. This immediately implies the assignment $x_1 = 0$ and x_1 is said to be asserted.

3.1.2 Conflict-Directed Backtracking

If all the literals in $\omega_C(\kappa)$ correspond to variables that were assigned at decision levels that are *lower* than the current decision level, we can immediately conclude that the search process needs to backtrack. This situation can only take place when the conflict in question is produced as a direct consequence of diagnosing a previous conflict and is illustrated in Figure 3-1 (a) for our working example. The implication sequence generated after asserting $x_1 = 0$ due to conflict κ leads to another conflict κ'. The conflicting assignment and conflict-induced clause associated with this new conflict are easily determined to be

$$\begin{aligned} A_C(\kappa') &= \{x_9 = 0@1, x_{10} = 0@3, x_{11} = 0@3, x_{12} = 1@2, x_{13} = 1@2\} \\ \omega_C(\kappa') &= (x_9 + x_{10} + x_{11} + \neg x_{12} + \neg x_{13}) \end{aligned} \tag{3.6}$$

and clearly show that the assignments that led to this second conflict were all made prior to the current decision level.

In such cases, it is easy to show that no satisfying assignments can be found until the search process backtracks to the highest decision level at which assignments in $A_C(\kappa')$ were made. Denoting this ***backtrack level*** by β, it is simply calculated according to:

$$\beta = \max\{\delta(x) | (x, \nu(x)) \in A_C(\kappa')\} \tag{3.7}$$

When $\beta = d - 1$, where d is the current decision level, the search process backtracks ***chronologically*** to the immediately preceding decision level. When $\beta < d - 1$, however, the search process may backtrack ***non-chronologically*** by jumping back over several levels in the decision tree. It is worth noting that all truth assignments that are made after decision level β will

```
// Global variables:              Implication graph I
//                                Clause database φ
// Input variable:                Current decision level d
// Output variable:               Backtracking decision level β
// Return value:                  CONFLICT or SUCCESS
//
Diagnose (d, &β)
{
     ωC(κ) = Create_Conflict_Induced_Clause();       // Using (3.4)
     Update_Clause_Database ( ωC(κ) );
     β = Compute_Max_Level();                         // Using (3.7)
     if (βL != d) {
         add new conflict vertex κ to I;
         record A(κ);
         return CONFLICT;
     }
     return SUCCESS;
}
```

Figure 3-2: Description of the standard diagnosis engine

force the just-identified conflict-induced clause $\omega_C(\kappa')$ to be unsatisfied. A search engine that backtracks chronologically may, thus, waste a significant amount of time exploring a useless region of the search space only to discover after much effort that the region does not contain any satisfying assignments. In contrast, the GRASP search engine jumps *directly* from the current decision level back to decision level β. At that point, $\omega_C(\kappa')$ is used to either derive a FDA at decision level β or to calculate a new backtracking decision level.

For our example, after occurrence of the second conflict the backtrack decision level is calculated, from (3.6), to be 3. Backtracking to decision level 3, the deduction engine creates a conflict vertex corresponding to $\omega_C(\kappa')$. Diagnosis of this conflict leads to a FDA of the decision variable at level 3 (see Figure 3-1 (b)).

The pseudo-code for the diagnosis engine in GRASP is shown in Figure 3-2 and illustrates the main features of standard conflict diagnosis described above. The GRASP search algorithm described in this report is sound and complete. General proofs of this claim can be found in [13, 26].

3.2 Variations on the Standard Diagnosis Engine

The standard conflict diagnosis, described in the previous section, suffers from two drawbacks. First, conflict analysis introduces significant overhead which, for some instances of SAT, can lead to large run times. Second, the size of the clause database grows with the number of backtracks; in the worst case such growth can be exponential in the number of variables.

The first drawback is inherent to the algorithmic framework we propose. Fortunately, the experimental results presented in Section 4 clearly suggest that, for specific instances of SAT, the performance gains far outweigh the procedure's additional overhead.

One solution to the second drawback is a simple modification to the conflict diagnosis engine that guarantees the worst case growth of the clause database to be polynomial in the number of variables. The main idea is to be selective in the choice of clauses to add to the clause database. Assume that we are given an integer parameter k. Conflict-induced clauses whose size (number of literals) is no greater than k are marked *green* and handled as described earlier by the standard diagnosis engine. Conflict-induced clauses of size greater than k are marked *red* and kept around only while they are unit clauses. Implementation of this scheme requires a simple modification to procedure `Erase()`, which must now delete red clauses with more than one free literal, and to the diagnosis engine, which must attach a color tag to each conflict-induced clause. With this modification the worst case growth becomes a kth-order polynomial in the number of variables.

Further enhancements to the conflict diagnosis engine involve generating stronger implicates (containing fewer literals) by more careful analysis of the structure of the implication graph. Such implicates correspond to the dominators [29] of the conflict vertex κ. These dominators, referred to as *unique implication points* (UIPs), can be identified in linear time with a single traversal of the implication graph.

4 Experimental Results

In this section we present experimental results for GRASP. Several benchmarks are used and GRASP is compared with other state-of-the-art and publicly available SAT programs. In particular, we compare GRASP with TEGUS [28] and POSIT [9]. TEGUS is included in SIS [23]. It was adapted to read CNF formulas and augmented to continue searching when all its default options were exhausted[3]. No changes were required with POSIT.

GRASP and POSIT are implemented in the C++ programming language, whereas TEGUS is implemented in the C programming language. All programs were compiled with GCC 2.7.2 and run on a SUN SPARC 5/85 machine with 64 MByte of RAM. In order to evaluate the three programs, two different sets of benchmarks were tested:

- The UCSC benchmarks, available from [8], that include instances of SAT commonly encountered in test pattern generation of combinational switching circuits for bridging and stuck-at faults.
- The DIMACS challenge benchmarks, also available from [8], that include instances of SAT from several authors and from different application areas.

While GRASP has a large number of configuration options, for the experimental results given below, it was configured to use the decision engine described in Section 2.5, to allow the generation of clauses based on UIPs, and to limit the size of clauses added to the clause database to 20 or fewer literals. All SAT programs were run with a CPU time limit of 10,000 seconds (about three hours).

For the tables of results the following definitions apply. A benchmark suite is partitioned into classes of related benchmarks. In each class, **#M** denotes the total number of class members; **#S** denotes the number of class members for which the program terminated in less than 10,000 CPU seconds; and **Time** denotes the total CPU time, in seconds, taken to process all members of the class.

3. Otherwise TEGUS would abort almost all benchmark examples.

Benchmark Class	#M	GRASP		TEGUS		POSIT	
		#S	Time	#S	Time	#S	Time
BF-0432	21	21	47.6	19	53,852	21	55.8
BF-1355	149	149	125.7	53	993,915	64	946,127
BF-2670	53	53	68.3	25	295,410	53	2,971
SSA-0432	7	7	1.1	7	1,593	7	0.2
SSA-2670	12	12	51.5	0	120,000	12	2,826
SSA-6288	3	3	0.2	3	17.5	3	0.0
SSA-7552	80	80	19.8	80	3,406	80	60.0

Table 4-1: Results on the UCSC Benchmarks

4.1 UCSC Benchmark Results

The results obtained for the UCSC benchmarks are shown in Table 4-1. The BF and SSA benchmark classes denote, respectively, CNF formulas for bridging and stuck-at faults[4]. These results are separated into benchmark classes according to the given benchmark circuit (which are taken from the ISCAS'85 benchmark circuits). GRASP performs much better than any other program on these benchmarks. The other two algorithms abort an extremely large number of problem instances, whereas GRASP aborts none. Furthermore, both the CPU times and the number of decisions of GRASP are extremely small when compared with the results of other programs. These benchmarks are characterized by extremely sparse CNF formulas for which BCP-based conflict analysis works particularly well. The performance difference between GRASP and TEGUS, a very efficient ATPG tool, illustrates the power of the search-pruning techniques included in GRASP.

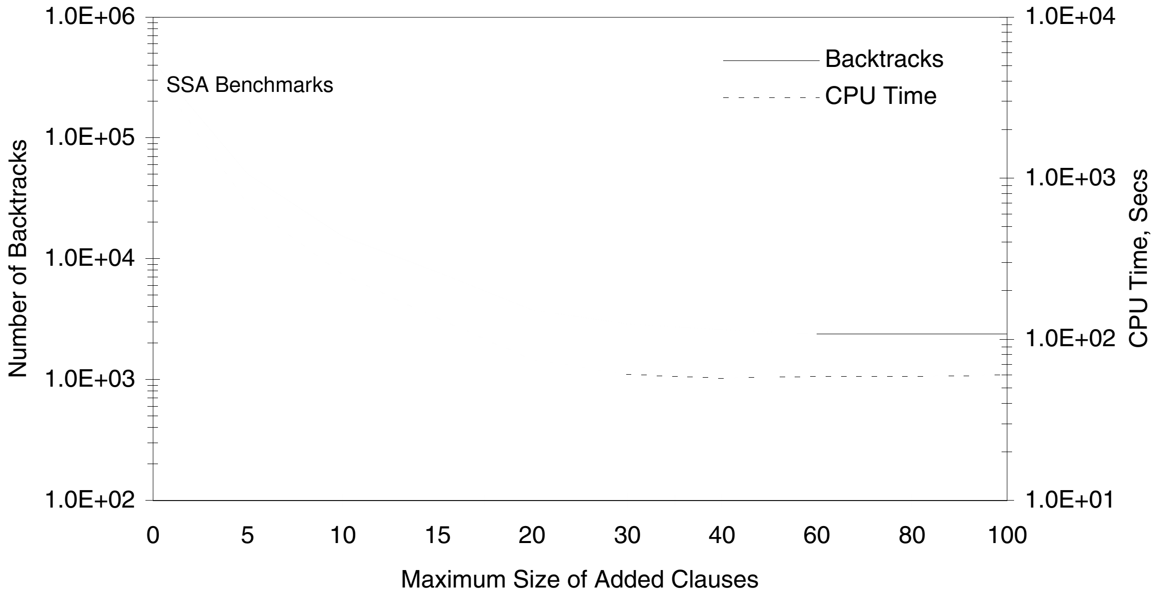
Despite the extremely good results of GRASP when compared with the other algorithms, some caution is required. GRASP performs well in separate instances of SAT without knowledge of the circuit structure. In ATPG systems the structure of the circuit is known and can be used to improve efficiency. This is what is done with TEGUS when applied to ATPG [28]. Given the greater overhead of GRASP, TEGUS is likely to perform better on a larger number of easy faults. However, GRASP is ideal for the hard faults, for which TEGUS applies learning techniques and iterates over several decision making orderings. As a result the optimum organization for ATPG is expected to be a combination of TEGUS for the simple faults, followed by GRASP for the hard faults.

4.1.1 Database Growth Versus CPU Time

It is interesting to evaluate how the growth of the clause database affects the amount of search and the CPU time. For this purpose the UCSC benchmark suites are used. The same decision making procedure is used and GRASP is run allowing clauses of size at most 0, 5, 10, 15, 20, 30, 40, 60, 80 and 100 to be added to the clause database in each experiment. The CPU time and the number of backtracks for the SSA and BF benchmarks are shown in Figure 4-1.

As the maximum size of added clauses grows, the number of backtracks decreases and the CPU time decreases accordingly. Eventually, this tendency is reversed, and even though the number of backtracks continues to decrease, the CPU time begins to increase. We can thus conclude that adding larger clauses leads to additional overhead for conducting the search pro-

4. These CNF formulas were developed at the University of California, Santa Cruz.



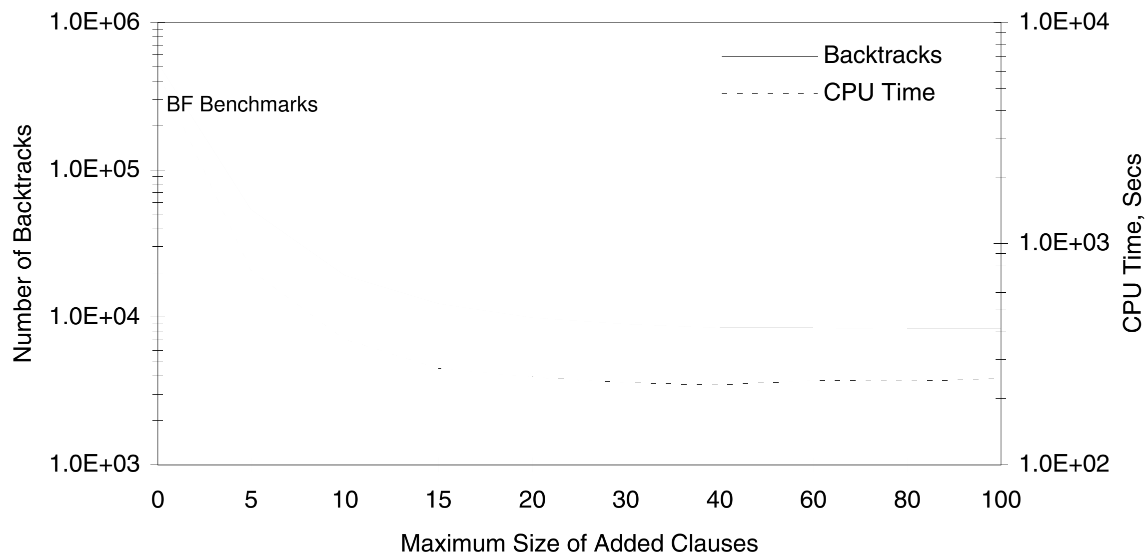


Figure 4-1: UCSC benchmarks with different growths of the clause database

cess and hence it eventually costs more than what it saves in terms of backtracks. These results also suggest that it may possible to experimentally identifying optimal growth rates for different classes of problem instances. For example, for the SSA and BF benchmarks the optimal bound is near 30.

4.2 DIMACS Benchmark Results

Finally, we illustrate the application of GRASP to other benchmarks, not directly related with EDA applications. For this purpose the DIMACS benchmarks are used [8]. The results of running GRASP and the other algorithms are shown in Table 4-2 and again are separated into classes of benchmarks. GRASP performs better than any of the other programs for the AIM-100,

Benchmark Class	#M	GRASP		TEGUS		POSIT	
		#S	Time	#S	Time	#S	Time
AIM-100	24	24	1.8	24	107.9	24	1,290
AIM-200	24	24	10.8	23	14,059	13	117,991
BF	4	4	7.2	2	26,654	2	20,037
DUBOIS	13	13	34.4	5	90,333	7	77,189
II-32	17	17	7.0	17	1,231	17	650.1
PRET	8	8	18.2	4	42,579	4	40,691
SSA	8	8	6.5	6	20,230	8	85.3
AIM-50	24	24	0.4	24	2.2	24	0.4
II-8	14	14	23.4	14	11.8	14	2.3
JNH	50	50	21.3	50	6,055	50	0.8
PAR-8	10	10	0.4	10	1.5	10	0.1
PAR-16	10	10	9,844	10	9,983	10	72.1
II-16	10	9	10,311	10	269.6	9	10,120
H	7	5	27,184	4	32,942	6	11,540
F	3	0	30,000	0	30,000	0	30,000
G	4	0	40,000	0	40,000	0	40,000
PAR-32	10	0	100,000	0	100,000	0	100,000

Table 4-2: Results on the DIMACS Benchmarks

AIM-200, BF, DUBOIS, PRET and SSA benchmark classes, whereas POSIT performs better than GRASP for the II-8, JNH, PAR-8 and PAR-16 benchmarks. It can be concluded that for benchmarks where GRASP performs better the other programs either take a very long time to find a solution or are unable to find a solution in less than 10, 000 seconds. We have also observed that benchmarks for which POSIT performs better than GRASP are also handled by GRASP with a similar amount of search; only the overhead inherent to GRASP becomes apparent. TEGUS and POSIT each perform better on different classes of instances. In general, however, POSIT seems to be slightly more efficient than TEGUS.

It is also interesting to measure how well conflict analysis works in practice. For this purpose statistics regarding some DIMACS benchmarks are shown in Table 4-3, where **#B** denotes the number of backtracks, **#NCB** denotes the number of non-chronological backtracks, **Largest jump** is the size of the largest non-chronological backtrack, **#UIP** indicates the number of unique implication points found, **% of Growth** denotes the variation in size of the clause database, and **Time** is the CPU time in seconds. For these examples several conclusions can be drawn. First, the number of non-chronological backtracks can be a significant percentage of the total number of backtracks. Second, the jumps in the decision tree can save a large amount of search work. As can be observed, in some cases the jumps taken potentially save searching millions of nodes in the decision tree. Third, the growth of the clause database is not necessarily large. Fourth, UIPs do occur in practice and for some benchmarks a reasonable number is found given the number of backtracks. Finally, for most of these examples conflict analysis causes GRASP to be much more efficient than POSIT. Nevertheless, POSIT and TEGUS can be more efficient in specific benchmarks, as the examples of the last three rows indicate. TEGUS performs particularly well on these instances because they are satisfiable and because TEGUS iterates several decision orderings. Hence, for these instances, iterating different decision making orderings is likely to lead to a solution.

Benchmark	#B	#NCB	Largest jump	#UIP	% of Growth	GRASP Time	TEGUS Time	POSIT Time
aim-200-2_0-yes1-2	109	50	13	25	152.63	0.38	2.80	7,990.71
aim-200-2_0-yes1-3	74	35	16	15	99.67	0.31	0.64	> 10,000
aim-200-2_0-no-1	29	20	12	5	22.9	0.13	69.93	> 10,000
aim-200-2_0-no-2	39	20	37	4	43.6	0.19	87.53	> 10,000
bf0432-007	335	124	17	32	47.99	5.18	6,648.68	11.79
bf1355-075	40	20	24	2	6.50	1.25	4.83	> 10,000
bf1355-638	11	7	8	4	1.11	0.32	> 10,000	> 10,000
bf2670-001	16	8	22	2	3.02	0.40	> 10,000	25.64
dubois30	233	72	16	21	465.83	0.68	> 10,000	> 10,000
dubois50	485	175	26	51	631.92	2.80	> 10,000	> 10,000
dubois100	1438	639	67	150	1033.54	26.22	> 10,000	> 10,000
pret60_40	147	98	17	8	407.08	0.41	652.30	175.49
pret60_60	131	83	16	10	353.54	0.35	639.27	173.12
pret150_25	428	313	38	35	588.17	4.84	> 10,000	> 10,000
pret150_75	388	257	49	20	446.75	3.85	> 10,000	> 10,000
ssa0432-003	37	6	5	1	30.80	0.15	221.71	0.01
ssa2670-130	130	45	34	10	17.26	2.07	> 10,000	14.23
ssa2670-141	377	97	16	28	65.71	3.42	> 10,000	70.82
ii16a1	110	19	13	0	0.03	13.61	5.99	> 10,000
ii16b2	2664	120	9	39	63.46	175.85	6.94	16.38
ii16b1	88325	2588	41	624	131.94	> 10,000	21.65	16.73

Table 4-3: Statistics of Running GRASP on Representative DIMACS Benchmarks

5 Conclusions and Research Directions

This report introduces a procedure for conflict analysis in satisfiability algorithms and describes a configurable algorithmic framework for solving SAT. Experimental results indicate that conflict analysis and its by-products, non-chronological backtracking and identification of equivalent conflicting conditions, can contribute decisively for efficiently solving a large number of classes of instances of SAT. As a result, the proposed SAT algorithm is shown to be more efficient than other state-of-the-art algorithms for a large number of SAT instances.

The natural evolution of this research work is to apply GRASP to different EDA applications, in particular test pattern generation, timing analysis, delay fault testing and logic verification, among others. Despite being a fast SAT algorithm, GRASP introduces noticeable overhead that can become a liability for some of these applications. Consequently, besides the algorithmic organization of GRASP, special attention must be paid to the implementation details. One envisioned compromise is to use GRASP as the second choice SAT algorithm for the hard instances of SAT whenever other simpler, but with less overhead, algorithms fail to find a solution in a small amount of CPU time.

Future research work will emphasize heuristic control of the rate of growth of the clause database. Another area for improving GRASP is related with the deduction engine. Improvements to the BCP-based deduction engine are described in [26] and consist of different forms of probing the CNF formula for creating new clauses. This approach naturally adapts and extends other deduction procedures, e.g. recursive learning [16] and transitive closure [4], since it completes the clause database with additional implicates, in addition to being able to identify as many necessary assignments.

The actual practical usefulness of improved deduction engines needs to be experimentally validated. Finally, we propose to undertake a comprehensive experimental characterization of the instances of SAT for which conflict analysis provides significant performance gains.

Acknowledgments

This work was supported in part by NSF under grant MIP-9404632.

References

[1] M. Abramovici, M. A. Breuer and A. D. Friedman, *Digital Systems Testing and Testable Design,* Computer Science Press, 1990.

[2] C. E. Blair, R. G. Jeroslow and J. K. Lowe, "Some Results and Experiments in Programming Techniques for Propositional Logic," *Computers and Operations Research*, vol. 13, no. 5, pp. 633-645, 1986.

[3] M. Bruynooghe, "Analysis of Dependencies to Improve the Behaviour of Logic Programs," in *Proceedings of the 5th Conference on Automated Deduction*, pp. 293-305, 1980.

[4] S. T. Chakradhar, V. D. Agrawal and S. G. Rothweiler, "A Transitive Closure Algorithm for Test Generation," *IEEE Transactions on Computer-Aided Design*, vol. 12, no. 7, pp. 1015-1028, July 1993.

[5] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory," *Journal of the Association for Computing Machinery*, vol. 7, pp. 201-215, 1960.

[6] R. Dechter, "Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition," *Artificial Intelligence*, vol. 41, pp. 273-312, 1989/90.

[7] J. de Kleer, "An Assumption-Based TMS," *Artificial Intelligence*, vol. 28, pp. 127-162, 1986.

[8] DIMACS Challenge benchmarks in ftp://Dimacs.Rutgers.EDU/pub/challenge/sat/benchmarks/cnf. UCSC benchmarks in /pub/challenge/sat/contributed/UCSC.

[9] J. W. Freeman, *Improvements to Propositional Satisfiability Search Algorithms*, Ph.D. Dissertation, Department of Computer and Information Science, University of Pennsylvania, May 1995.

[10] G. Gallo and G. Urbani, "Algorithms for Testing the Satisfiability of Propositional Formulae," *Journal of Logic Programming*, vol. 7, pp. 45-61, 1989.

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

[12] J. Gaschnig, *Performance Measurement and Analysis of Certain Search Algorithms*, Ph.D. Dissertation, Department of Computer Science, Carnegie-Mellon University, CMU-CS-79-124, May 1979.

[13] M. L. Ginsberg, "Dynamic Backtracking," *Journal of Artificial Intelligence Research*, vol. 1, pp. 25-46, August 1993.

[14] J. Giraldi and M. L. Bushnell, "Search State Equivalence for Redundancy Identification and Test Generation," in *Proceedings of the International Test Conference*, pp. 184-193, 1991.

[15] J. P. Hayes, *Introduction to Digital Logic Design*, Addison-Wesley, 1993.

[16] W. Kunz and D. K. Pradhan, "Recursive Learning: An Attractive Alternative to the Decision Tree for Test Generation in Digital Circuits," in *Proceedings of the International Test Conference*, pp. 816-825, 1992.

[17] T. Larrabee, *Efficient Generation of Test Patterns Using Boolean Satisfiability*, Ph.D. Dissertation, Department of Computer Science, Stanford University, STAN-CS-90-1302, February 1990.

[18] S. Mallela and S. Wu, "A Sequential Circuit Test Generation System," in *Proceedings of the International Test Conference*, pp. 57-61, 1985.

[19] D. A. McAllester, “An Outlook on Truth Maintenance,” AI Memo 551, MIT AI Laboratory, August 1980.

[20] P. C. McGeer, A. Saldanha, P. R. Stephan, R. K. Brayton and A. L. Sangiovanni-Vincentelli, “Timing Analysis and Delay-Fault Test Generation Using Path Recursive Functions,” in *Proceedings of the International Conference on Computer-Aided Design*, pp. 180-183, 1991.

[21] T. Schiex and G. Verfaillie, “Nogood Recording for Static and Dynamic Constraint Satisfaction Problems,” in *Proceedings of the International Conference on Tools with Artificial Intelligence*, pp. 48-55, 1993.

[22] M. H. Schulz and E. Auth, “Improved Deterministic Test Pattern Generation with Applications to Redundancy Identification,” *IEEE Transactions on Computer-Aided Design*, vol. 8, no. 7, pp. 811-816, July 1989.

[23] E. M. Sentovich et. al, “SIS: An Environment to Sequential Circuit Synthesis,” Memorandum no. UCB/ERL M92/41, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, May 1992.

[24] J. P. M. Silva and K. A. Sakallah, “Efficient and Robust Test Generation-Based Timing Analysis,” in *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 303-306, June 1994, London, England.

[25] J. P. M. Silva and K. A. Sakallah, “Dynamic Search-Space Pruning Techniques in Path Sensitization,” in *Proc. IEEE/ACM Design Automation Conference (DAC)*, pp. 705-711, June 1994, San Diego, California.

[26] J. P. M. Silva, *Search Algorithms for Satisfiability Problems in Combinational Switching Circuits*, Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, University of Michigan, May 1995.

[27] R. M. Stallman and G. J. Sussman, “Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis,” *Artificial Intelligence*, vol. 9, pp. 135-196, October 1977.

[28] P. R. Stephan, R. K. Brayton and A. L. Sangiovanni-Vincentelli, “Combinational Test Generation Using Satisfiability,” Memorandum no. UCB/ERL M92/112, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, October 1992.

[29] R. E. Tarjan, “Finding Dominators in Directed Graphs,” *SIAM Journal on Computing*, vol. 3, no. 1, pp. 62-89, March 1974.

[30] R. Zabih and D. A. McAllester, “A Rearrangement Search Strategy for Determining Propositional Satisfiability,” in *Proceedings of the National Conference on Artificial Intelligence*, pp. 155-160, 1988.