GRASP: A Search Algorithm
for Propositional Satisfiability

João P. Marques-Silva, *Member, IEEE,* and Karem A. Sakallah, *Fellow, IEEE*

Abstract—This paper introduces GRASP (Generic seaRch Algorithm for the Satisfiability Problem), a new search algorithm for Propositional Satisfiability (SAT). GRASP incorporates several search-pruning techniques that proved to be quite powerful on a wide variety of SAT problems. Some of these techniques are specific to SAT, whereas others are similar in spirit to approaches in other fields of Artificial Intelligence. GRASP is premised on the inevitability of conflicts during the 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 nonchronologically 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 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.

Index Terms—Satisfiability, search algorithms, conflict diagnosis, conflict-directed nonchronological backtracking, conflict-based equivalence, failure-driven assertions, unique implication points.

---

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 [13]. Over the years, many algorithmic solutions have been proposed for SAT, the most well-known being the different variations of the Davis-Putnam procedure [7]. 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* [39]. Iterated application of the unit clause rule is commonly referred to as *Boolean Constraint Propagation* (BCP) [39] or as *derivation of implications* in the electronic CAD literature [1].

Most of the recently proposed improvements to the basic Davis-Putnam procedure [3], [6], [11], [12], [22], [30], [36], [39] 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 [28] is a nonchronological backtracking procedure outlined for solving problems in Logic Truth Maintenance Systems (LTMS), but it is only sketched and no experimental results are presented. Nevertheless, nonchronological backtracking techniques have been extensively studied and applied to different areas of Artificial Intelligence, particularly Truth Maintenance Systems (TMS) [9], [35], Constraint Satisfaction Problems (CSP) [8], [14], [15], [31], and Logic Programming [4], in some cases with very promising experimental results [8], [15]. In recent years, extensive research work has been dedicated to the development of local search algorithms for SAT [33]. These algorithms are, in general, incomplete, i.e., they may not find a solution and cannot prove unsatisfiability. Nevertheless, local search algorithms have been shown to be extremely effective on specific classes of *satisfiable* instances of SAT.

Interest in the direct application of SAT algorithms to electronic design automation (EDA) problems has been on the rise recently [5], [22], [29], [36]. 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 [16], [21], [25].

This paper introduces a new procedure for conflict analysis in satisfiability algorithms and describes its use in a configurable algorithmic framework for solving SAT problems. Titled GRASP[1] (Generic seaRch Algorithm for the Satisfiability Problem), this framework is premised on the inevitability of conflicts during search. By noting that conflicts arise when certain clauses are missing from the problem specification, GRASP views conflict occurrence as an opportunity to augment the problem description with such conflict-induced clauses. The addition of these clauses

---

- *J.P. Marques-Silva is with Cadence European Laboratories, IST/INESC, R. Alves Redol, 9, 1000 Lisboa, Portugal.*
- *K.A. Sakallah is with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122. E-mail: karem@eecs.umich.edu.*

helps to prune the search for a solution in three complementary ways. First, annotation of the literals in a conflict-induced clause by the decision level at which their values were assigned enables GRASP to backtrack nonchronologically to earlier levels in the search tree. Second, by "recording" these clauses, GRASP can recognize and preempt the occurrence of similar conflicts later on in the search. And third, 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 [18] provide ample evidence 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.

The remainder of this paper 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. (In the Appendix, we prove that the GRASP SAT algorithm is both correct and complete.) Extensive experimental results on a wide range of benchmarks are presented and analyzed in Section 4. In particular, GRASP is shown to outperform several state-of-the-art SAT algorithms [2], [6], [10], [11], [30], [33], [36], [19] on most, but not all, benchmarks. Furthermore, the experimental results strongly suggest that, for several practical classes of SAT instances, local search algorithms may be inadequate. This is particularly significant whenever the SAT instances are likely to be unsatisfiable, as is typical in Automated Theorem Proving and in several Electronic Design Automation tasks. The paper 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 $x_1, \cdots, x_n$ is the conjunction (AND) of m *clauses* $\omega_1, \cdots, \omega_m$ 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 $f(x_1, \cdots, x_n)$ and each of its clauses corresponds to an implicate of f [17, 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 [17, p. 288] 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 $f(x_1, \cdots, x_n)$ 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 it is unresolved and 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 be 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.[2] 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, a search system can attempt to *augment* it with additional implicates to increase the deductive power during the search process. We propose a search mechanism that identifies additional implicates by diagnosing the causes of conflicts. Our approach considers the occurrence of a conflict, which is unavoidable for an unsatisfiable instance unless the formula is complete, as an 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 nonchronological backtracking that we refer to as *conflict-directed backtracking* (CDB),[3] 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) [11], [39]. 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 $(l_1 + \cdots + l_k)$ 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 Davis and Putnam [7]. 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 $\omega = (l_1 + \cdots + l_k)$. The *antecedent assignment* of x, denoted as $A^\omega(x)$, is defined as the set of assignments to variables other than x with literals in ω. Intuitively, $A^\omega(x)$ 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, $A^\omega(x) = \{y = 0, z = 1\}$, $A^\omega(y) = \{x = 0, z = 1\}$, and $A^\omega(z) = \{x = 0, y = 0\}$. Note that the antecedent assignment of an electively assigned variable is empty.

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

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

2. The predecessors of vertex $x = \nu(x)$ in I are the antecedent assignments $A^\omega(x)$ corresponding to the unit clause ω that led to the implication of x. The directed edges from the vertices in $A^\omega(x)$ 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 $\mathcal{K}$ correspond to variable assignments that force a clause ω to become unsatisfied and are viewed as the antecedent assignment $A^\omega(\kappa)$. The

---

2. This assertion is proven in Theorem 3 in the Appendix.

3. The designation CDB is used instead of *dependency-directed backtracking* [35] because the backtracking procedure is tightly associated with BCP.

directed edges from the vertices in $A^\omega(x)$ to $\mathcal{K}$ are all labeled with ω.

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

$$\delta(x) = \max\{\delta(y)|(y, \nu(y)) \in A^\omega(x)\}. \qquad (1)$$

2.5 Search Algorithm Template

The general structure of the GRASP search algorithm is shown in Fig. 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() function 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:

    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 [11], [26]. For most of 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. We chose to employ this heuristic in our experimental evaluation because of its simplicity and to highlight the effectiveness of conflict analysis.

2. Deduce(), which implements BCP and (implicitly) maintains the resulting implication graph. The pseudocode for this procedure is shown in Fig. 3. The algorithm repeatedly applies the unit clause rule [7] while unit clauses exist. It returns with a SUCCESS indication unless one or more clauses become unsatisfied. In that case, a conflict vertex is added to the implication graph and a CONFLICT indication is returned.

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.

The Search() function starts by calling Decide() to choose a variable assignment at decision level d. It then determines the consequences of this elective assignment by calling Deduce(). If this assignment does not cause any clauses to become unsatisfied, Search() is called recursively at decision level $d + 1$. If, on the other hand, a conflict

arises due to this assignment, the Diagnose() function is called to analyze this conflict and to determine an appropriate decision level for backtracking the search. When Search() encounters a conflict, it returns with a CONFLICT indication and causes the elective assignment made on entry to the function to be erased. 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 $\mathcal{K}$ 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, referred to as a *conflict-induced clause*,[4] provides the primary mechanism for implementing failure-driven assertions, nonchronological conflict-directed backtracking, and conflict-based equivalence (see Section 2.3).

We denote the conflicting assignment associated with a conflict vertex $\mathcal{K}$ by $A^{\omega_C}(\kappa)$ 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 $\mathcal{K}$. Besides the decision assignment at the current decision level, only those assignments that occurred at previous decision levels are included in $A^{\omega_C}(\kappa)$. 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 $A^{\omega_C}(\kappa)$, we partition the antecedent assignments of $\mathcal{K}$, as well as those for variables assigned at the current decision level into two sets. Let x denote either $\mathcal{K}$ or a variable that is assigned at the current decision level. The partition of $A(x)$ is then given by:[5]

$$\Lambda(x) = \{(y, \nu(y)) \in A(x)|\delta(y) < \delta(x)\}$$
$$\Sigma(x) = \{(y, \nu(y)) \in A(x)|\delta(y) = \delta(x)\}. \qquad (2)$$

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

$$A^{\omega_C}(\kappa) = \text{causes\_of}(\kappa)$$

where causes_of(.) is defined by:

---

4. Conditions similar to these implicates are referred to as "nogoods" in TMS [9], [35] and in some algorithms for CSP [31]. Nevertheless, the basic mechanism for creating conflict-induced clauses differs.

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: $\{x_1 = 1 @ 6\}$

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

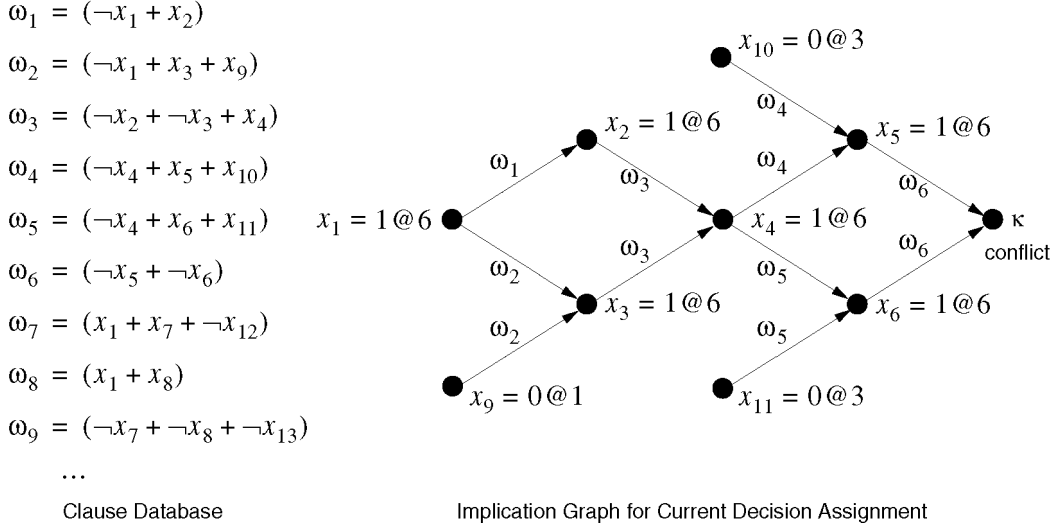Clause Database   Implication Graph for Current Decision Assignment

Fig. 1. Example of clause database and partial implication graph.

$$\text{causes\_of}(\kappa) =
\begin{cases}
(x, \nu(x)) & \text{if } A(x) = \emptyset \\
\Lambda(x) \cup \left[\bigcup_{(y,\nu(y)) \in \Sigma(x)} \text{causes\_of}(y) \right] & \text{otherwise.}
\end{cases}
\tag{3}$$

The conflict-induced clause corresponding to $A^{\omega_C}(\kappa)$ is now determined according to:

$$\omega_C(\kappa) = \sum_{(x,\nu(x)) \in A^{\omega_C}(\kappa)} x^{\nu(x)}, \tag{4}$$

where, for a binary variable x, $x^0 \equiv x$, and $x^1 \equiv \neg x$. Application of (2)-(4) to the conflict depicted in Fig. 1 yields the following conflicting assignment and conflict-induced clause at decision level 6:

$$A^{\omega_C}(\kappa) = \{x_1 = 1 @ 6, x_9 = 0 @ 1, x_{10} = 0 @ 3, x_{11} = 0 @ 3\}$$
$$\omega_C(\kappa) = (\neg x_1 + x_9 + x_{10} + x_{11}). \tag{5}$$

We note that our method for deriving implicates by analyzing the causes of conflicts has its foundations in [26]. It is also similar in spirit to the approaches of Freeman [11, chapter 8] and McAllester [28]. However, unlike the precise computations of the conflicting assignment $A^{\omega_C}(\kappa)$ in (3) and conflict-induced clause $\omega_C(\kappa)$ in (4), the procedures in these related works were only informally described.

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

5. To reduce clutter, we omit the superscripts denoting the clauses that lead to these antecedent assignments and assume them to be understood from context.

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 ensures 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 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 Fig. 1) as an illustration, we note that after erasing the conflicting implication sequence at level 6, the conflict-induced clause $\omega_C(\kappa)$ in (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

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

Fig. 2. Description of GRASP.

illustrated in Fig. 4a for our working example. The implication sequence generated after asserting $x_1 = 0$ due to conflict $\mathcal{K}$ leads to another conflict κ'. The conflicting assignment and conflict-induced clause associated with this new conflict are easily determined to be

$$
\begin{aligned}
A^{\omega_C}(\kappa') =& \\
& \{x_9 = 0 @ 1, x_{10} = 0 @ 3, x_{11} = 0 @ 3, \\
& \quad x_{12} = 1 @ 2, x_{13} = 1 @ 2\} \\
\omega_C(\kappa') =& (x_9 + x_{10} + x_{11} + \neg x_{12} + \neg x_{13})
\end{aligned}
\tag{6}
$$

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 $A^{\omega_C}(\kappa')$ were made. Denoting this *backtrack level* by β, it is simply calculated according to:

$$
\beta = \max\{\delta(x) | (x, \nu(x)) \in A^{\omega_C}(\kappa')\}.
\tag{7}
$$

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 *nonchronologically* 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 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 an 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 (7) applied to (6), to be 3. Backtracking to decision level 3, the deduction engine creates a conflict vertex corresponding to $\omega_C(\kappa')$.

```
// 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^ω(κ);
            return CONFLICT;
          }
          if (exists unit clause ω with free literal l = x or l = ¬x) {
            record A^ω(x);
            δ(x) = d;
            set x = 1 if l = x or x = 0 if l = ¬x;
          }
      }
      return SUCCESS;
}
```

Fig. 3. Description of the deduction engine.
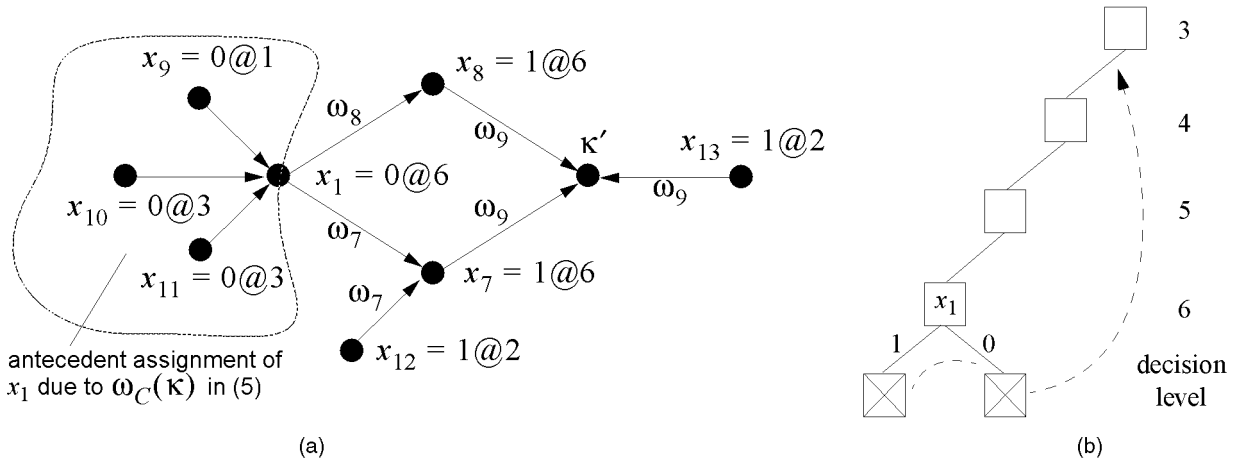
Fig. 4. Implication sequence and backtracking due to asserting $x_1 = 0$. (a) Conflicting implication sequence. (b) Decision tree.

Diagnosis of this conflict leads to an FDA of the decision variable at level 3 (see Fig. 4b).

The pseudocode for the diagnosis engine in GRASP is shown in Fig. 5 and illustrates the main features of standard conflict diagnosis described above. The procedure starts with an analysis of what caused the conflict and the creation of a new conflict-induced clause. This clause is added to the clause database and used to calculate the backtracking decision level β. If backtracking is necessary (indicated by $\beta \neq d$), a new conflict vertex $\mathcal{K}$ is added to the implication graph and its antecedent assignments $A(\kappa)$ are recorded.

3.2 Variations on the Standard Diagnosis Engine

This section describes two improvements to the standard diagnosis engine described above. The first is concerned with ways of controlling the growth of the clause database. The second provides techniques that utilize the structure of the implication sequences to reduce the size of identified implicates. Both of these improvements represent novel contributions to search-based SAT algorithms.

3.2.1 Space-Bounded Diagnosis Engines

Standard conflict diagnosis, described in the previous section, suffers from two drawbacks. First, conflict analysis introduces significant overhead which, for some instances

```
// 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)
{
    ω_C(κ) = Create_Conflict_Induced_Clause();        // Using (4)
    Update_Clause_Database (ω_C(κ));
    β = Compute_Max_Level();                           // Using (7)
    if (β != d) {
        add new conflict vertex κ to I;
        record A(κ);
        return CONFLICT;
    }
    return SUCCESS;
}
```

Fig. 5. Description of the standard diagnosis engine.

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 SAT instances, 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 satisfied, unsatisfied, or are unit clauses. Red clauses are used for defining failure-driven assertions and are deleted as soon as they become unresolved with more than one free literal, since, in such a situation, these clauses no longer define failure-driven assertions. Implementation of this scheme requires a simple modification to procedure `Erase()`, which must now delete unresolved 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 polynomial in the number of variables as a function of the fixed integer k. The major drawback of the proposed solution is that the ability to apply conflict-based equivalence decreases, since some large conflict-induced clauses can now be deleted as the search proceeds.

3.2.2 Unique Implication Points

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 I. An implication sequence at a given decision level d corresponds to the subgraph in I whose vertices are all annotated by d. Assuming a conflict is detected, let $U = \{(u_1, \nu(u_1)), \cdots, (u_k, \nu(u_k))\}$ denote the set of *dominators* [37] of $\mathcal{K}$ which are defined with respect to the variable assignment at decision level d. Each $(u_i, \nu(u_i))$ is referred to as a *unique implication point* (UIP) and can be viewed as triggering an implication sequence at decision level d that leads to the same conflict. To illustrate the application of UIPs, consider again the implication sequence of Fig. 1. The set of dominators of $\mathcal{K}$ with respect to x_1 is $\{(x_1, 1), (x_4, 1)\}$. Together with the earlier assignments $x_{10} = 0$ and $x_{11} = 0$, the assignment $x_4 = 1$ is, thus, a sufficient condition for triggering an implication sequence leading to the same conflict. Hence, the clause $(\neg x_4 + x_{10} + x_{11})$ is an implicate of the function that did not exist in the clause database. Moreover, since $(x_1 = 1) \wedge (x_9 = 0) \Rightarrow (x_4 = 1)$, the clause[6] $(\neg x_1 + x_9 + x_4)$ is another implicate of the function. Both of these implicates are stronger than the single conflict-induced clause identified earlier in (5) and can potentially provide additional implications in the presence of partial assignments. It is also interesting to note that neither implicate can be identified from the original clause database with just BCP.

This procedure for constructing strong implicates can be generalized for an arbitrary number of UIPs. Given a set of UIPs $U = \{(u_1, \nu(u_1)), \cdots, (u_k, \nu(u_k))\}$ and a conflict vertex $\mathcal{K}$, replace (3) with:

$$\text{causes\_of}(x, u_i) =$$

$$\begin{cases} (u_i, \nu(u_i)) & \text{if } x = u_i \\ \Lambda(x) \cup \left[\bigcup_{(y, \nu(y)) \in \Sigma(x)} \text{causes\_of}(y, u_i) \right] & \text{otherwise,} \end{cases} \quad (8)$$

where $(u_i, \nu(u_i)) \in U$ and $\text{causes\_of}(x, u_i)$ are now interpreted as the set of antecedent assignments of x due to u_i and any other relevant assignments from earlier decision levels. Conflict-induced clauses can now be created for every pair of adjacent UIPs (u_{i-1}, u_i) for $i = 2, \cdots, k$ as well as for the last UIP u_k and the conflict vertex $\mathcal{K}$:

$$\omega_C(u_{i-1}, u_i) = \left[\sum_{(x, \nu(x)) \in \text{causes\_of}(u_{i-1}, u_i)} x^{\nu(x)} \right] + u_i^{\neg \nu(u_1)}$$
$$\omega_C(u_k, \kappa) = \sum_{(x, \nu(x)) \in \text{causes\_of}(u_k, \kappa)} x^{\nu(x)}. \quad (9)$$

Clearly, the size of each of these conflict-induced clauses will be less than or equal to the size of the single conflict-induced clause created without identification of UIPs. Furthermore, the diagnosis engine of Fig. 5 can easily identify UIPs in linear time with one 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 POSIT [11], C-SAT [10], H2R [30], 2CL [38], TABLEAU [6] (whose latest version is named NTAB), SATO [19], TEGUS [36], a recent implementation of the Davis-Putnam procedure, DPL [2], and GSAT [33]. For all these algorithms, either the source code or the executable was provided by the respective author.[7]

GRASP is implemented in the C++ programming language and was compiled with GCC 2.7.2. The CPU times for all programs were scaled to the equivalent CPU times on a SUN SPARC 5/85 machine.[8] In order to evaluate the different programs, two different sets of benchmarks were tested:

- The DIMACS challenge benchmarks, available from [18], that include instances of SAT from several authors and from different application areas.
- The UCSC benchmarks, also available from [18], that include instances of SAT commonly encountered in test pattern generation of combinational switching circuits. These benchmarks represent one practical application of SAT algorithms to the field of

---

6. Note that $(x \Rightarrow y) \iff (\neg x + y)$.

7. These comparisons were conducted in 1995 and 1996 [26], [27] and do not reflect enhancements (some adapted from GRASP) that were made to several of these programs since then. In particular, conflict-directed backtracking was only available in GRASP at the time.

8. We deployed all available computing resources in order to complete this experiment in a reasonable amount of time and used the test programs explicitly provided in [18] to normalize the results obtained from different machine architectures. The other machines that were used included SUN Sparc2/40 and SUN Sparc20/50. All machines had 64MB of physical memory.

Electronic Design Automation, thus being of key significance for experimentally evaluating SAT algorithms.

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.

For the tables of results presented below the following definitions apply. A benchmark suite is partitioned into classes of related benchmarks, e.g., for the DIMACS benchmarks, class *AIM-100* includes all benchmarks with name *aim-100-**. In each class, **#M** denotes the total number of class members. Some tables of results contain CPU times in seconds, whereas the remaining tables contain the number of class members for which the program terminates in less than 10,000 CPU seconds, being referred to as the number of *successes*. Ideally, an algorithm should have a number of successes equal to the number of class members for all classes of benchmarks.

4.1   DIMACS Benchmark Results

The CPU times of running GRASP and the other algorithms on the DIMACS benchmarks are shown in Table 1.[9] The number of successes for each algorithm are shown in Table 2. For GSAT, which cannot be used to prove unsatisfiability, experimental results are only available for benchmark classes for which *all* members are satisfiable; entries corresponding to benchmark classes that have unsatisfiable instances are indicated with n/a. From the results, it can be concluded that GRASP performs better than any of the other programs for the AIM-100, AIM-200, BF, DUBOIS, PRET, and SSA benchmark classes, whereas POSIT performs better than GRASP for the II-8, JNH, PAR-8, PAR-16, and HANOI benchmarks classes. It can also 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 most 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. Among the other tools, POSIT performs better in four classes of benchmarks, whereas SATO performs better on two, TEGUS on one, and GSAT on one. Even though GSAT has been shown to be extremely effective on random instances of SAT [33], for specific benchmark classes, it performs significantly worse than deterministic algorithms (e.g., II8, II16, II32, HANOI). Still, GSAT is the most efficient tool for the class of benchmarks G. Finally, none of the evaluated algorithms was able to find a solution to any problem instance of the benchmark classes F and PAR32.

It is also interesting to measure how well conflict analysis works in practice. For this purpose, statistics regarding some DIMACS benchmarks are shown in

---

9. For H2R, entries with § indicate benchmark classes where segmentation faults occurred for some problem instances; in these cases and for each such instance, the CPU time assumed is 10,000 seconds.

TABLE 1
CPU Times for the DIMACS Benchmarks

Benchmark Class	#M	GRASP	POSIT	H2R	C-SAT	2CL	NTAB	SATO	TEGUS	DPL	GSAT
AIM-100	24	1.8	1,290	21,571	4.0	57.9	39,569	60,390	107.9	58,510	n/a
AIM-200	24	10.8	117,991	150,004	4.2	53,266	69,410	150,095	14,059	156,196	n/a
BF	4	7.2	20,037	10,200	30,634	24.6	27,900	35,695	26,654	40,000	n/a
DUBOIS	13	34.4	77,189	§73,729	95,485	21,005	47,952	71,528	90,333	96,977	n/a
II32	17	7.0	650.1	§36,029	454.7	10,803	697.0	10,004	1,231	21,520	83,814
PRET	8	18.2	40,691	40,342	41,398	40,035	80,000	40,430	42,579	41,429	n/a
SSA	8	6.5	85.3	20,006	15,781	668.5	20,024	30,092	20,230	80,000	n/a
AIM-50	24	0.4	0.4	2.3	47.2	2.4	24.3	12.7	2.2	10.7	n/a
II8	14	23.4	2.3	§30,005	18,119	58,949	11,411	0.4	11.8	84,189	27,647
JNH	50	21.3	0.8	5.8	10.3	13.4	10.9	11.0	6,055	40.0	n/a
PAR8	10	0.4	0.1	0.6	3.2	2.0	0.7	0.2	1.5	0.8	50,005
PAR16	10	9,844	72.1	264.8	824.4	351.4	591.5	10,447	9,983	11,741	100,000
II16	10	10,311	10,120	75,940	50,545	62,849	10,126	85,522	269.6	83,933	11,670
HANOI	2	14,480	10,117	§10,733	16,550	20,175	15,840	20,000	11,641	20,000	20,000
HOLE	5	12,704	937.9	11,182	1,025	1,270	1,244	362.2	21,301	11,404	n/a
F	3	30,000	30,000	§30,000	30,000	30,000	30,000	30,000	30,000	30,000	30,000
G	4	40,000	40,000	40,000	40,000	40,000	40,000	40,000	40,000	40,000	20,079
PAR32	10	100,000	100,000	100,000	100,000	100,000	100,000	100,000	100,000	100,000	100,000

Table 3 for GRASP and for POSIT, which is the next best performing SAT algorithm besides GRASP. In this table, **#B** denotes the number of backtracks, **#NCB** denotes the number of nonchronological backtracks, **Largest jump** is the size of the largest nonchronological backtrack, **#UIP** indicates the number of unique implication points found, **%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 nonchronological 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 can be more efficient for specific benchmarks, as the examples of the last two rows indicate.

4.2 UCSC Benchmark Results

The results obtained for the UCSC benchmarks are shown in Table 4 and in Table 5. The BF and SSA benchmark classes denote, respectively, CNF formulas for bridging and stuck-at faults. These results are divided into benchmark classes according to each benchmark circuit number. GRASP performs significantly better than any other

program on these benchmarks. With the exception of 2CL, all other algorithms abort on a large number of problem instances, whereas GRASP aborts on none. Moreover, the CPU times of GRASP are extremely small when compared with the CPU times of the other programs. The UCSC benchmarks are characterized by extremely sparse CNF formulas for which the BCP-based conflict analysis procedure of GRASP works particularly well. In addition, it should be noted that a direct comparison of the results of each algorithm with the results of DPL illustrates how effective search-pruning techniques can be for these classes of instances of SAT. Finally, it should be emphasized the performance difference between GRASP and TEGUS, a very efficient test-pattern generation tool [36], that further illustrates the power of the search-pruning techniques included in GRASP.

4.2.1 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 Fig. 6.

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

TABLE 2
Number of Successes on the DIMACS Benchmarks

Benchmark Class	#M	GRASP	POSIT	H2R	C-SAT	2CL	NTAB	SATO	TEGUS	DPL	GSAT
AIM-100	24	24	24	23	24	24	18	20	24	21	n/a
AIM-200	24	24	13	9	24	20	11	9	23	9	n/a
BF	4	4	2	3	1	4	2	1	2	0	n/a
DUBOIS	13	13	7	7	4	11	5	7	5	5	n/a
II32	17	17	17	14	17	16	17	16	17	15	9
PRET	8	8	4	4	4	4	0	4	4	4	n/a
SSA	8	8	8	6	7	8	6	5	6	0	n/a
AIM-50	24	24	24	24	24	24	24	24	24	24	n/a
II8	14	14	14	11	13	10	13	14	14	7	12
JNH	50	50	50	50	50	50	50	50	50	50	n/a
PAR8	10	10	10	10	10	10	10	10	10	10	5
PAR16	10	10	10	10	10	10	10	10	10	10	0
II16	10	9	9	3	5	4	6	7	10	2	9
HANOI	2	1	1	1	1	1	1	0	1	0	0
HOLE	5	4	5	4	5	5	5	5	3	4	n/a
F	3	0	0	0	0	0	0	0	0	0	0
G	4	0	0	0	0	0	0	0	0	0	2
PAR32	10	0	0	0	0	0	0	0	0	0	0

decrease, the CPU time begins to increase. We can thus conclude that adding larger clauses leads to additional overhead for conducting the search process and, hence, it eventually costs more than what it saves in terms of backtracks. These results also suggest that it may possible to experimentally identify optimal growth rates for different classes of problem instances. For example, for the SSA and BF benchmarks the optimal bound is near 30.

5 Conclusions and Research Directions

This paper 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, nonchronological 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 [21] and transitive closure [5], 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.

APPENDIX

CORRECTNESS AND COMPLETENESS OF GRASP

The purpose of this appendix is to prove that the GRASP SAT algorithm finds a solution to a given instance of SAT if and only if a solution exists. In particular, the standard conflict analysis procedure described in Section 3.1 is assumed. (A more thorough discussion and proof of the

TABLE 3
Statistics of Running GRASP on Representative DIMACS Benchmarks

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

TABLE 4
CPU Times for the UCSC Benchmarks

Benchmark Class	#M	GRASP	POSIT	H2R	C-SAT	2CL	NTAB	SATO	TEGUS	DPL
BF0432	21	47.6	55.8	101.4	1,861	46.7	1,863	28,174	53,852	210,000
BF1355	149	125.7	946,127	352,175	914,863	3,555	975,862	1286,584	993,915	1490,000
BF2670	53	68.3	2,971	143,357	154,565	8,408	253,336	360,009	295,410	530,000
SSA0432	7	1.1	0.2	2.2	32.0	1.8	103.9	651.5	1,593	70,000
SSA2670	12	51.5	2,826	120,000	95,355	6,735	120,000	120,000	120,000	120,000
SSA6288	3	0.2	0.0	7.9	15.9	39.7	13.3	0.7	17.5	30,000
SSA7552	80	19.8	60.0	97.5	20,217	53.0	166.0	242,250	3406	800,000

correctness and completeness of the algorithm and its variations can be found in [26].) We shall start by introducing a few basic definitions and by proving that GRASP is correct. Afterward, we will establish a few preliminary formal results, which will be used for proving that GRASP is complete. Finally, for completeness, we prove, in Theorem 3, the assertion that conflicts do not arise when the clause database contains all of the function's prime implicates.

Definition 1. *Given an instance of SAT, a complete assignment is said to be a* solution *to that instance if no clauses are unsatisfied.*

Definition 2. *A SAT algorithm is said to be* correct *if, for each instance of SAT, any assignment identified as solution is indeed a solution to that instance.*

Definition 3. *A SAT algorithm is said to be* complete *if, for each instance of SAT, a solution is found if a solution exists.*

Theorem 1. *The GRASP SAT algorithm is correct.*

TABLE 5
Number of Successes on the UCSC Benchmarks

Benchmark Class	#M	GRASP	POSIT	H2R	C-SAT	2CL	NTAB	SATO	TEGUS	DPL
BF0432	21	21	21	21	21	21	21	21	19	0
BF1355	149	149	64	121	61	149	58	22	53	0
BF2670	53	53	53	41	43	53	30	17	25	0
SSA0432	7	7	7	7	7	7	7	7	7	0
SSA2670	12	12	12	0	6	12	0	0	0	0
SSA6288	3	3	3	3	3	3	3	3	3	0
SSA7552	80	80	80	80	78	80	80	57	80	0

Proof. GRASP, as described in Fig. 2, terminates with a SUCCESS indication whenever function `Decide()` eventually returns itself a SUCCESS indication, which signifies that all variables are assigned. Since this situation can only occur whenever no unsatisfied clause is found with BCP, then the complete assignment thus defined is indeed a solution to the given instance of SAT. Hence, GRASP is correct. □

Lemma 1. *A deduction engine based on BCP (as described in Fig. 3) only identifies assignments necessary for a partial assignment to be contained in a solution of a given instance of SAT.*

Proof. BCP, as described in Fig. 3, is solely based on the unit-clause rule, which identifies assignments that are necessary to guarantee that unit clauses will not become unsatisfied. Hence, BCP only identifies those assignments that are necessary for a partial assignment to be extended to a complete assignment representing a solution of a given instance of SAT. □

Before establishing the next few formal results, we need to define failure-driven assertions (FDAs).

Definition 4. *An assignment $x = \nu_x @ \delta_x$ is said to be a failure-driven assertion whenever $\delta(y) < \delta_x$ for all $(y, \nu_y) \in A^\omega(x)$.*

Lemma 2. *Let the conflict assignment $A^{\omega_C}(\kappa)$ be computed according to (3). Then, the following holds:*

1. *Any partial assignment A, such that $A \supseteq A^{\omega_C}(\kappa)$ cannot be contained in a solution to the given instance of SAT.*
2. *Each conflict-induced clause $\omega_C(\kappa)$ (given by (4)) identifies an implicate of the Boolean function $f(x_1, \cdots, x_n)$ associated with the initial CNF formula φ.*

Proof. There are only two situations under which a conflict $\mathcal{K}$ can be identified. Either a decision assignment yields a conflict or a set of existing failure-driven assertions (FDAs) at a given decision level d yields a conflict. We analyze each case separately and, for each case, we prove that no partial assignment A, such that $A \supseteq A^{\omega_C}(\kappa)$, can be extended to a solution of the given instance of SAT.

First, consider the case of a decision assignment that yields a conflict with conflicting assignment $A^{\omega_C}(\kappa)$. Since conflict analysis of the conflict computes $A^{\omega_C}(\kappa)$, then BCP applied to a the clause database under any assignment $A \supseteq A^{\omega_C}(\kappa)$ necessarily yields a conflict. Since $f(x_1, \cdots, x_n)$ assumes value 0 for any assignment $A \supseteq A^{\omega_C}(\kappa)$, then $\omega_C(\kappa)$ is an implicate of $f(x_1, \cdots, x_n)$.

Suppose now the situation under which j FDAs yield the conflict $\mathcal{K}$. For each FDA i, $1 \le i \le j$, we can associate a variable x_i, a definition of the assertion $x_i = \nu_i @ d$ and the antecedent of x_i, $A(x_i)$. In addition, each FDA i is necessarily the result of a given conflict κ_i such that

$$A^{\omega_C}(\kappa_i) = A(x_i) \cup \{(x_i, 1 - \nu_i)\}.$$

Let us now assume a partial assignment A, with $A \supseteq A^{\omega_C}(\kappa)$. Then, we must also necessarily have,

$$A \supseteq A^{\omega_C}(\kappa_i) - \{(x_i, 1 - \nu_i)\} \quad i = 1, \cdots, j.$$

Hence, given A and the assignment $x_i = 1 - \nu_i$, BCP necessarily yields a conflict for $i = 1, \cdots, j$. On the other hand, the assignment $(x_1 = \nu_1) \wedge (x_2 = \nu_2) \wedge \cdots \wedge (x_j = \nu_j)$ necessarily yields conflict $\mathcal{K}$ by hypothesis. Therefore, given an assignment $A \supseteq A^{\omega_C}(\kappa)$, for any combination of assignments to variables $x_1, x_2, \cdots, x_j$, BCP yields a conflict. Thus, A cannot be extended to a solution to the given instance of SAT and $\omega_C(\kappa)$ is necessarily an implicate of $f(x_1, \cdots, x_n)$. □

Lemma 3. *Let β be the backtracking decision level computed with (7). Furthermore, let A_β denote the partial assignment containing all variable assignments with decision levels no greater than β. In this situation, a solution cannot be found for any assignment A such that $A \supseteq A_\beta$.*

Proof. From Lemma 2 we know that $\omega_C(\kappa)$ is an implicate of the Boolean function being evaluated. Since this conflict-induced clause is now included in the clause database, it will remain unsatisfied for any assignment $A \supseteq A_\beta$ and, so, a solution will not be found. □

Corollary 1. *Let d be the current decision level and β be the computed backtracking decision level. In this situation, a*
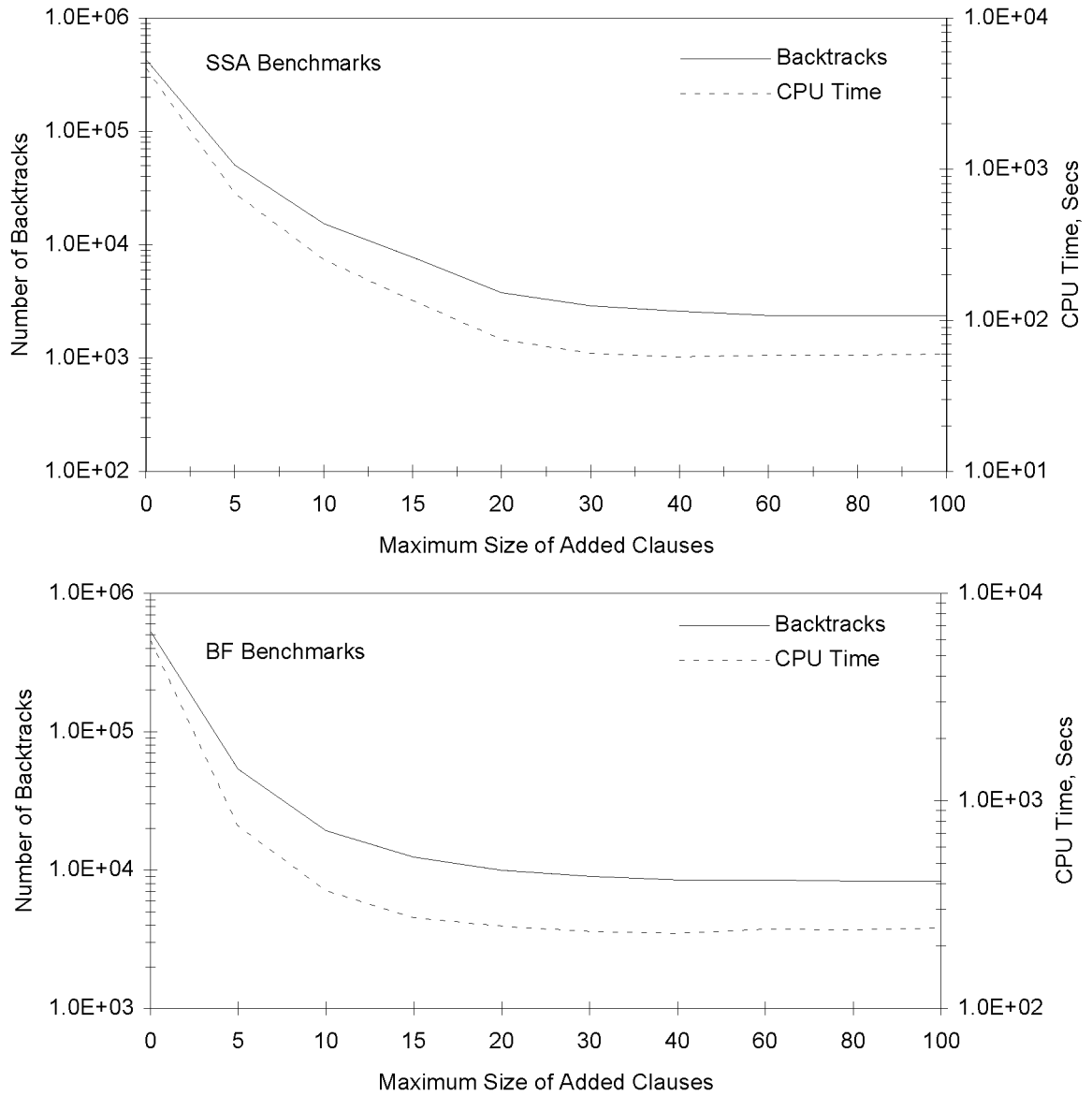
Fig. 6. UCSC benchmarks with different growths of the clause database.

solution to the given instance of SAT cannot be found until the search process backtracks to decision level β.

Theorem 2. *The GRASP SAT algorithm is complete.*

Proof. It is well-known that chronological backtracking is a complete search procedure [20]. Backtracking search extends partial assignments until either a solution is found or an inconsistent assignment is identified. In the event of an inconsistent assignment being found, the most recent decision assignment yet untried is considered and the search proceeds. Hence, if a solution exists, it will eventually be enumerated. Furthermore, for the case of SAT, and since BCP only identifies necessary assignments (from Lemma 1), then backtracking search with BCP is also a complete search procedure. Given these facts, we just have to prove that nonchronological backtracks do not jump over partial assignments that can be extended to solutions of the given instance of SAT. Let us suppose that the current decision level is d and the

computed backtracking decision level is β. Then, by Lemma 3 and Corollary 1, we can conclude that a solution cannot be found by extending the partial assignment defined by decision levels no greater than β, i.e., A_β. It then follows that if a solution exists, it will eventually be enumerated and, consequently, the algorithm is complete.   □

Theorem 3. *The GRASP SAT algorithm does not generate inconsistent assignments (conflicts) when applied to a complete formula, i.e., a formula that contains all the prime implicates of the underlying function, unless the formula is unsatisfiable.*

Proof. Let φ be an arbitrary CNF formula and let $P(\varphi)$ be its complete product. Viewed as a clause database, $P(\varphi)$ is, thus, the set of all of φ's prime implicates. Let $\omega \in P(\varphi)$. Then, by definition [17, p. 288], φ implies ω but does not imply any clause v that in turn implies ω. Consider two cases.

1. φ is unsatisfiable. In this case, the set $P(\varphi)$ consists of a single empty clause (a clause with no literals): The constant 0 function has no prime implicates. To show this, assume otherwise, i.e., let $\omega = (l_1 + l_2 + \cdots, l_k)$, where each l_i is a literal, be a prime implicate of 0, and let $\upsilon = l_i$. Thus, $0 \Rightarrow \omega$, $0 \Rightarrow \upsilon$, and $\upsilon \Rightarrow \omega$, i.e., ω is not prime. When the GRASP SAT algorithm is applied to a clause database consisting of a single empty clause, it returns immediately with a FAILURE indication proving unsatisfiability.

2. φ is satisfiable. Without loss of generality, suppose that the sequence of decision assignments $x_1 = 1, x_2 = 1, \cdots, x_k = 1$ was taken and that it led to a conflict. GRASP's conflict diagnosis procedure would then yield the conflict-induced clause $\upsilon = (\neg x_1 + \neg x_2 + \cdots + \neg x_k)$ as an implicate of φ that is not in the set $P(\varphi)$, i.e., υ is a nonprime implicate of φ (GRASP may actually generate a clause that is "stronger" than υ, i.e., one that does not involve all k literals.) Let $\upsilon \Rightarrow \omega$, where $\omega \in P(\varphi)$. In particular, assume, without loss of generality, that $\omega = (\neg x_1 + \neg x_2 + \cdots + \neg x_{k-1})$. Then, at decision level $k - 2$, when the partial assignment is $\{x_1 = 1, x_2 = 1, \cdots, x_{k-2} = 1\}$, ω becomes a unit clause and BCP causes x_{k-1} to be implied to 0. In other words, the assignment $\{x_1 = 1, x_2 = 1, \cdots, x_{k-1} = 1, x_k = 1\}$ would never be generated and the conflict corresponding to it would never arise.   □

ACKNOWLEDGMENTS

REFERENCES

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

[2] P. Barth, "A Davis-Putnam Based Enumeration Algorithm for Linear Pseudo-Boolean Optimization," Technical Report MPI-I-95-2-003, Max-Planck-Institut für Informatik, 1995.

[3] 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.

[4] M. Bruynooghe, "Analysis of Dependencies to Improve the Behaviour of Logic Programs," *Proc. Fifth Conf. Automated Deduction,* pp. 293-305, 1980.

[5] S.T. Chakradhar, V.D. Agrawal, and S.G. Rothweiler, "A Transitive Closure Algorithm for Test Generation," *IEEE Trans. Computer-Aided Design,* vol. 12, no. 7, pp. 1,015-1,028, July 1993.

[6] J. Crawford and L. Auton, "Experimental Results on the Cross-Over Point in Satisfiability Problems," *Proc. 11th Nat'l Conf. Artificial Intelligence (AAAI-93),* pp. 22-28, 1993.

[7] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory," *J. ACM,* vol. 7, pp. 201-215, 1960.

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

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

[10] O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier, "SAT versus UNSAT," *Second DIMACS Implementation Challenge,* D.S. Johnson and M.A. Trick, eds., 1993.

[11] J.W. Freeman, "Improvements to Propositional Satisfiability Search Algorithms," PhD dissertation, Dept. of Computer and Information Science, Univ. of Pennsylvania, May 1995.

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

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

[14] J. Gaschnig, "Performance Measurement and Analysis of Certain Search Algorithms," PhD dissertation, Dept. of Computer Science, Carnegie-Mellon Univ., CMU-CS-79-124, May 1979.

[15] M.L. Ginsberg, "Dynamic Backtracking," *J. Artificial Intelligence Research,* vol. 1, pp. 25-46, Aug. 1993.

[16] J. Giraldi and M.L. Bushnell, "Search State Equivalence for Redundancy Identification and Test Generation," *Proc. Int'l Test Conf.,* pp. 184-193, 1991.

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

[18] D.S. Johnson and M.A. Trick eds. *Second DIMACS Implementation Challenge,* 1993. DIMACS benchmarks available at ftp://Dimacs.-Rutgers.EDU/pub/challenge/sat/benchmarks/cnf. UCSC benchmarks available at ftp://Dimacs.Rutgers.EDU/pub/challenge/sat/contributed/UCSC.

[19] S. Kim and H. Zhang, "ModGen: Theorem Proving by Model Generation," *Proc. 12th Nat'l Conf. Am. Assoc. Artificial Intelligence (AAAI-94),* pp. 162-167, 1994.

[20] D.E. Knuth, "Estimating the Efficiency of Backtrack Programs," *Mathematics of Computation,* vol. 29, no. 129, pp. 121-136, Jan. 1975.

[21] W. Kunz and D.K. Pradhan, "Recursive Learning: An Attractive Alternative to the Decision Tree for Test Generation in Digital Circuits," *Proc. Int'l Test Conf.,* pp. 816-825, 1992.

[22] T. Larrabee, "Efficient Generation of Test Patterns Using Boolean Satisfiability," PhD dissertation, Dept. of Computer Science, Stanford Univ., STAN-CS-90-1302, Feb. 1990.

[23] S. Mallela and S. Wu, "A Sequential Circuit Test Generation System," *Proc. Int'l Test Conf.,* pp. 57-61, 1985.

[24] J.P. Marques-Silva and K.A. Sakallah, "Efficient and Robust Test Generation-Based Timing Analysis," *Proc. IEEE Int'l Symp. Circuits and Systems (ISCAS),* pp. 303-306, London, June 1994.

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

[26] J.P. Marques-Silva, "Search Algorithms for Satisfiability Problems in Combinational Switching Circuits," PhD dissertation, Dept. of Electrical Eng. and Computer Science, Univ. of Michigan, May 1995.

[27] J.P. Marques-Silva and K.A. Sakallah, "GRASP—A New Search Algorithm for Satisfiability," *Digest of IEEE Int'l Conf. Computer-Aided Design (ICCAD),* pp. 220-227, San Jose, Calif., Nov. 1996.

[28] D.A. McAllester, "An Outlook on Truth Maintenance," AI Memo 551, MIT AI Laboratory, Aug. 1980.

[29] 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," *Proc. Int'l Conf. Computer-Aided Design,* pp. 180-183, 1991.

[30] D. Pretolani, "Efficiency and Stability of Hypergraph SAT Algorithms," *Second DIMACS Implementation Challenge,* D.S. Johnson and M.A. Trick, eds., 1993.

[31] T. Schiex and G. Verfaillie, "Nogood Recording for Static and Dynamic Constraint Satisfaction Problems," *Proc. Int'l Conf. Tools with Artificial Intelligence,* pp. 48-55, 1993.

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

[33] B. Selman, H.J. Levesque, and D. Mitchell, "A New Method for Solving Hard Satisfiability Problems," *Proc. 10th Nat'l Conf. Artificial Intelligence (AAAI-92),* 1992.

[34] E.M. Sentovich et al., "SIS: An Environment to Sequential Circuit Synthesis," Memorandum no. UCB/ERL M92/41, Dept. of Electrical Eng. and Computer Sciences, Univ. of California at Berkeley, May 1992.

[35] 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, Oct. 1977.

[36] P.R. Stephan, R.K. Brayton, and A.L. Sangiovanni-Vincentelli, "Combinational Test Generation Using Satisfiability," Memorandum no. UCB/ERL M92/112, Dept. of Electrical Eng. and Computer Sciences, Univ. of California at Berkeley, Oct. 1992.

[37] R.E. Tarjan, "Finding Dominators in Directed Graphs," *SIAM J. Computing,* vol. 3, no. 1, pp. 62-89, Mar 1974.

[38] A. Van Gelder and Y.K. Tsuji, "Satisfiability Testing with More Reasoning and Less Guessing," *Second DIMACS Implementation Challenge,* D.S. Johnson and M.A. Trick, eds., 1993.

[39] R. Zabih and D.A. McAllester, "A Rearrangement Search Strategy for Determining Propositional Satisfiability," *Proc. Nat'l Conf. Artificial Intelligence,* pp. 155-160, 1988.

João P. Marques-Silva (M'95) received the BS and MS degrees in Computer Engineering from Instituto Superior Técnico, Lisbon, Portugal, and the PhD degree from the University of Michigan, Ann Arbor, in 1988, 1991, and 1995, respectively. He is currently an assistant professor at the Informatics Department of Instituto Superior Técnico, Lisbon, Portugal. His research interests include design and analysis of algorithms for computationally hard problems and their application to electronic design automation. Dr. Marques-Silva is a member of the IEEE and the ACM.

Karem A. Sakallah (S'76-M'81-SM'92-F'98) received the BE degree (with distinction) in electrical engineering from the American University of Beirut, Beirut, Lebanon, in 1975, and the MSEE and PhD degrees in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, Pennsylvania, in 1977 and 1981, respectively. In 1981, he joined the Department of Electrical Engineering at CMU as a visiting assistant professor. From 1982 to 1988, he was with the Semiconductor Engineering Computer-Aided Design Group at Digital Equipment Corporation in Hudson, Massachusetts, where he headed the Analysis and Simulation Advanced Development team. Since September 1988, he has been at the University of Michigan, Ann Arbor, as a professor of electrical engineering and computer science. From September 1994 to March 1995, he was on a six month sabbatical leave at the Cadence Berkeley Laboratory in Berkeley, California. He was associate editor of the *IEEE Transactions on Computer-Aided Design* during 1995-1997, and has served on the program committees of ICCAD, DAC, ICCD, and numerous other workshops. He has published more than 90 papers and has presented seminars and tutorials at many professional meetings and various industrial sites. His research interests are primarily in the area of computer-aided design, with particular emphasis on simulation, timing verification and optimal clocking, modeling, synthesis, knowledge abstraction, and design environments. Dr. Sakallah is a fellow of the IEEE and a member of the ACM and Sigma Xi.