Robust Search Algorithms for Test Pattern Generation

João P. Marques Silva
Cadence European Laboratories
IST/INESC
1000 Lisboa, Portugal

Karem A. Sakallah
Department of EECS
University of Michigan
Ann Arbor, Michigan 48109-2122

Abstract

In recent years several highly effective algorithms have been proposed for Automatic Test Pattern Generation (ATPG). Nevertheless, most of these algorithms too often rely on different types of heuristics to achieve good empirical performance. Moreover, there has not been significant research work on developing algorithms that are robust, in the sense that they can handle most faults with little heuristic guidance. In this paper we describe an algorithm for ATPG that is robust and still very efficient. In contrast with existing algorithms for ATPG, the proposed algorithm reduces heuristic knowledge to a minimum and relies on an optimized search algorithm for effectively pruning the search space. Even though the experimental results are obtained using an ATPG tool built on top of a Propositional Satisfiability (SAT) algorithm, the same concepts can be integrated on application-specific algorithms.

1 Introduction

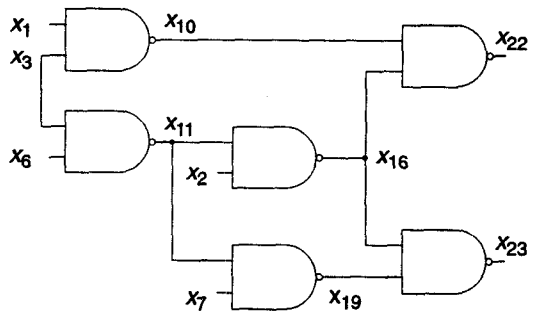
During the last decade a large number of algorithms for deterministic ATPG have been proposed [3-6, 8-11, 12-14, 18-19], many of which are extremely effective on most existing benchmarks, and permit very high fault coverages. Most of these ATPG algorithms are based on implicit enumeration [7] and incorporate different search pruning techniques to effectively reduce the amount of search in most practical cases. The most well-known search-pruning concepts include *head lines* [5], *non-local implications* [4, 12, 13], *recursive learning* [9, 10], *E-frontiers* [6], *transitive closure* [3], *non-chronological backtracking* [14] among several others. Despite this continued research effort on the effectiveness of ATPG algorithms, they still significantly rely on heuristic techniques. For example, preprocessing is often applied after hard faults have been identified (which were identified in a previous stage of the algorithm). In addition, different heuristics for decision making are commonly available and are used in different phases of the ATPG algorithm. Examples of application of these techniques can be found in [6, 9, 12-14, 18-19].

In contrast, little research effort has been spent on developing ***robust*** ATPG algorithms. We define a robust ATPG algorithm as one that reduces the amount of heuristic knowledge to a minimum and relies almost exclusively on pruning techniques for effectively reducing the amount of search. Robust ATPG algorithms can be crucial for circuits with a large number of hard faults and where existing heuristic techniques fail. Furthermore, some techniques commonly applied in ATPG will not scale for larger circuits. For example, preprocessing by identifying non-local implications takes quadratic [11] or cubic [18] worst-case time in the size of the circuit. Hence, for large circuits non-local implications and other forms of preprocessing may become impractical. On the other hand, a robust ATPG algorithm, that does not require preprocessing for handling all faults, can then become the algorithm of choice.

In this paper we describe different pruning techniques that can be used for improving the robustness of ATPG algorithms. Some of these techniques have been embedded in the GRASP [15] algorithm for Propositional Satisfiability (SAT), and hence the proposed ATPG tool is built on top of GRASP. Nevertheless, all the techniques we propose can be integrated in a dedicated ATPG tool, which would then avoid the overhead of mapping fault detection problems into instances of satisfiability as the input to the SAT solver.

The paper is organized as follows. In the next section we describe how to represent circuits and fault detection problems as Conjunctive Normal Form (CNF) formulas. Afterwards, we briefly describe the GRASP SAT algorithm for CNF formulas, that includes several powerful search-pruning techniques. The next section is dedicated to studying the integration of SAT algorithms in the ATPG process. Several techniques can be applied with the goal of reducing the complexity of fault detection problems. Furthermore, we propose several methods to improve the CNF representation of fault detection problems. We note that some of the concepts proposed in this section can in general be applied to other EDA tasks that can be solved with SAT algorithms. Section 5 includes experimental

0731-3071/97 $10.00



(a) Circuit

Number of stuck-at faults	34
Collapsed fault set size [2]	17

(b) Stuck-at faults

Node x	x_{11}
$O(x)$	$\{x_{16}, x_{19}\}$
$O^*(x)$	$\{x_{16}, x_{19}, x_{22}, x_{23}\}$
$I(x)$	$\{x_3, x_6\}$
$I^*(x)$	$\{x_3, x_6\}$
$K_O(x)$	$\{x_2, x_7, x_{10}, x_{16}, x_{19}, x_{22}, x_{23}\}$
$K_I(x)$	$\{x_{10}, x_2, x_7, x_1, x_3, x_6\}$

(c) Topological data for x_{11}

Figure 1: Example circuit — C17 [2]

results on the well-known ISCAS'85 [2] benchmarks as well as other benchmarks. Finally, Section 6 highlights directions for future research work.

2 Definitions

2.1 Combinational Circuits

We start by introducing unified representations for circuits and fault detection problems. These representations are used throughout the paper. A combinational circuit C is represented as a directed acyclic graph $C = (V_C, E_C)$, where the elements of V_C, i.e. the circuit nodes, are either primary inputs or gate outputs, with $|V_C| = N$. The set of edges $E_C \subseteq V_C \times V_C$ identifies gate input-output connections. We shall assume gates with bounded fanin, and so $|E_C| = O(|N|)$. For every circuit node x in V_C, the following definitions apply:

- $O(x)$ denotes the ***fanout*** nodes of node x, i.e. nodes y in V_C such that $(x, y) \in E_C$.
- $O^*(x)$ denotes the ***transitive fanout*** of node x, i.e. the set of all nodes y such that there is a path connecting x to y.
- $I(x)$ denotes the ***fanin*** nodes of node x, i.e. nodes y in V_C such that $(y, x) \in E_C$.
- $I^*(x)$ denotes the ***transitive fanin*** of node x, i.e. the set of all nodes y such that there is a path connecting y to x.
- $K_O(x)$ denotes ***immediate fanout cone of influence*** of x, being defined as follows:

$$K_O(x) = \{y | y \in O^*(x) \vee y \in I(w) \wedge w \in O^*(x)\} . \quad (1)$$

- $K_I(x)$ denotes ***immediate fanin cone of influence*** of x, being defined as follows:

$$K_I(x) = \left[\bigcup_{y \in O^*(x)} I^*(y) \right] - (O^*(x) \cup \{x\}) . \quad (2)$$

The set of primary inputs can also be referred to as *PI*, and the set of primary outputs as *PO*. Simple gates are assumed: AND, NAND, OR, NOR, NOT and BUFF. Finally, the number of stuck-at faults in the circuit is M, with $M = O(N)$, since $|E_C| = O(|N|)$, and are numbered $1, \ldots, M$. The example in Figure 1 illustrates the previous definitions.

2.2 Conjunctive Normal Form Formulas

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 x_i or its complement x_i'. 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. When appropriate we refer to a CNF formula φ as a ***clause database***.

A backtracking search algorithm for Propositional Satisfiability (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 corre-

Gate type	Gate function	φ_x
AND	$x = \text{AND}(w_1, \ldots, w_j)$	$\left[\prod_{i=1}^{j}(w_i + x')\right] \cdot \left(\sum_{i=1}^{j} w_i' + x\right)$
NAND	$x = \text{NAND}(w_1, \ldots, w_j)$	$\left[\prod_{i=1}^{j}(w_i + x)\right] \cdot \left(\sum_{i=1}^{j} w_i' + x'\right)$
OR	$x = \text{OR}(w_1, \ldots, w_j)$	$\left[\prod_{i=1}^{j}(w_i' + x)\right] \cdot \left(\sum_{i=1}^{j} w_i + x'\right)$
NOR	$x = \text{NOR}(w_1, \ldots, w_j)$	$\left[\prod_{i=1}^{j}(w_i' + x')\right] \cdot \left(\sum_{i=1}^{j} w_i + x\right)$
NOT	$x = \text{NOT}(w_1)$	$(x + w_1) \cdot (x' + w_1')$
BUFFER	$x = \text{BUFFER}(w_1)$	$(x' + w_1) \cdot (x + w_1')$

Table 1: CNF formulas for simple gates

sponding binary values. Truth assignments will be represented as sets of variable/value pairs; for example $A = \{(x_1, 0), (x_7, 1), (x_{13}, 0)\}$. Alternatively, assignments can be denoted as $A = \{x_1 = 0, x_7 = 1, x_{13} = 0\}$. An assignment A is complete if $|A| = n$; otherwise it is partial. Evaluating a formula φ for a given 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.

The CNF formula of a circuit is the conjunction of the CNF formulas for each gate output, where the CNF formula of each gate denotes the valid input-output assignments to the gate. (For simple gates, generalized CNF formulas are shown in Table 1 [14].) If we view a CNF formula as a set of clauses, the CNF formula φ for the circuit is defined by the set union of the CNF formulas for each gate with output x, φ_x:

$$\varphi = \bigcup_{x \in V_C} \varphi_x \tag{3}$$

The Boolean function $f : \{0, 1\}^N \rightarrow \{0, 1\}$, where $N = |V_C|$, associated with φ is referred to as the ***circuit consistency function***.

For Automatic Test Pattern Generation (ATPG), the following definitions apply. The single stuck-at line (SSF) fault model is assumed [1]. We say that a stuck-at fault is ***detectable*** if and only if there exists an assignment of logic values to the circuit primary inputs such that the effect of discrepancy caused by the fault can be observed on at least one of the circuit primary outputs.

2.3 Test Pattern Generation

The application of CNF representations of circuits and fault detection problems in ATPG has been extensively studied [3, 11, 18]. In this section we provide very simple and non-optimized CNF representations of circuits and fault detection problems, which will be assumed in the remainder of the paper.

In the context of test pattern generation, and for capturing the fault detection problem, each node x is characterized by three propositional variables:

- x^G denotes the logic value assumed by the node in the ***good*** circuit.
- x^F denotes the logic value assumed by the node in the ***faulty*** circuit.
- x^S denotes whether x^G and x^F assume different logic value [11]. We shall refer to this variable as the ***sensitization status*** of node x. (Other semantic definitions

Sub-formula/Condition	Clause Set
Good Circuit	$\varphi^G = \bigcup_{x \in V_C} \varphi_x^G$
Faulty Circuit	$\varphi^F = \bigcup_{x \in O^*(z)} \varphi_x^F$
Node Sensitization	$\varphi^S = \bigcup_{x \in O^*(z)} \varphi_x^S$
Propagation Blocking Conditions	$\varphi^B = (\neg x^S) \qquad x \in K_O(z) - O^*(z)$
Side Input Equivalence	$\varphi^E = (\neg x^G + x^F) \cdot (x^G + \neg x^F) \qquad x \in K_O(z) - O^*(z)$
Fault Activation Conditions	$\varphi^A = (z^S) \cdot (\neg z^G) \cdot (z^F) \qquad$ if $v = 1$ $\varphi^A = (z^S) \cdot (z^G) \cdot (\neg z^F) \qquad$ if $v = 0$
Fault Detection Requirement	$\varphi^R = \left(\sum_{x \in PO \wedge x \in O^*(z)} x^S \right)$
Detection of Fault z s-a-v	$\varphi^D = \varphi^G \cup \varphi^F \cup \varphi^S \cup \varphi^B \cup \varphi^E \cup \varphi^A \cup \varphi^R$

Table 2: Definition of the fault detection problem for the stem fault z s-a-v

of the sensitization status have been proposed [3, 18], which are more stringent.)

Given the definition of variable x^S, the following relationship must hold:

$$[(x^G \neq x^F) \leftrightarrow x^S] \Leftrightarrow (x^G + \neg x^F + x^S) \cdot (\neg x^G + x^F + x^S) \cdot (\neg x^S + x^G + x^F) \cdot (\neg x^S + \neg x^G + \neg x^F) \quad (4)$$

which basically states that the logic values of x^G and x^F differ if and only if x^S assumes logic value 1.

Let φ_x denote the CNF formula associated with gate output x. The notation φ_x^G denotes the CNF formula for x in the good circuit, i.e. using y^G variables, whereas φ_x^F denotes the CNF formula for x in the faulty circuit, i.e. using y^F variables. For a ***stem*** fault z-a-v[1], the CNF representation of the associated fault detection problem contains the following components:

- CNF formula for the circuit, denoting the good circuit.
- CNF formula for the circuit, denoting the faulty circuit. This formula only needs to contain the CNF formulas for the nodes that are relevant for detecting the given fault, i.e. nodes in the transitive fanout of node z.
- CNF formulas for defining the sensitization status of every node in the transitive fanout of the fault site, i.e. node z. Hence, for each of these nodes,

$$\varphi_x^S = (x^G + \neg x^F + x^S) \cdot (\neg x^G + x^F + x^S) \cdot (\neg x^S + x^G + x^F) \cdot (\neg x^S + \neg x^G + \neg x^F) \quad (5)$$

  which states that $x^S = 1$ if and only if $x^G \neq x^F$.
- Clauses that prevent each node x from being sensitized, by having $x^S = 0$, whenever x is not in the transitive fanout of z but at least one fanout node of x is in the transitive fanout of z.
- Clauses requiring $x^G = x^F$ on each node x such that x is not in the transitive fanout of z but at least one fanout node of x is in the transitive fanout of z. (Observe that this condition and the previous one permit restricting the number of x^F and x^S variables that must actually be used.)
- Clauses capturing conditions for ***activating*** the fault, i.e. by requiring $z^G \neq z^F$ and by forcing a suitable logic value on z^G.

The formula φ^D for detecting a fault z s-a-v is summarized in Table 2 and will henceforth be referred to as the ***fault detection formula***. Similarly, we define the ***fault-specific formula***, φ^{FS}, as follows,

1. See [1] for ATPG definitions used throughout the paper.

Sub-formula/Condition	Clause Set	Example
Faulty Circuit	$\varphi^F = \{\varphi^F_{x_{16}} \cup \varphi^F_{x_{19}} \cup \varphi^F_{x_{22}} \vee \varphi^F_{x_{23}}\}$	$\varphi^F_{x_{16}} = (x_2^F + x_{16}^F) \cdot (x_{11}^F + x_{16}^F) \cdot (\neg x_2^F + \neg x_{11}^F + \neg x_{16}^F)$
Node Sensitization	$\varphi^S = \{\varphi^S_{x_{16}} \cup \varphi^S_{x_{19}} \cup \varphi^S_{x_{22}} \cup \varphi^S_{x_{23}}\}$	
Propagation Blocking Conditions	$\varphi^B = (\neg x_2^S) \cdot (\neg x_7^S) \cdot (\neg x_{10}^S)$	
Side Input Equivalence	$\varphi^E = (\neg x_2^G + x_2^F) \cdot (x_2^G + \neg x_2^F) \cdot (\neg x_7^G + x_7^F) \cdot (x_7^G + \neg x_7^F) \cdot (\neg x_{10}^G + x_{10}^F) \cdot (x_{10}^G + \neg x_{10}^F)$	
Fault Activation Conditions	$\varphi^A = (x_{11}^S) \cdot (\neg x_{11}^G) \cdot (x_{11}^F)$	
Fault Detection Requirement	$\varphi^R = (x_{22}^S + x_{23}^S)$	

Figure 2: Fault-specific formula and fault detection requirement for fault x_{11} s-a-1

$$\begin{aligned}\varphi^{FS} &= \varphi^D - (\varphi^G \cup \varphi^R) \\ &= \varphi^F \cup \varphi^S \cup \varphi^B \cup \varphi^E \cup \varphi^A\end{aligned} \tag{6}$$

Fault-specific formulas contain only the clauses associated with propagating the error signal to the primary outputs and can be defined independently of the circuit formula. The fault-specific CNF formula for fault x_{11} s-a-1 is given in Figure 2.

Fanout-branch faults require additional information for dealing with setting specific values on the fanout branch. For a given fanout-branch fault (z, y) s-a-v[1], the CNF formula of Table 2 needs to be modified as follows:

- For all sub-formulas in Table 2, replace node z by node y.
- Replace the fault activation formula φ^A with the following formulas. First, add clauses requiring $y^G \neq y^F$,

$$(y^G \neq y^F) \Leftrightarrow (y^G + y^F) \cdot (\neg y^G + \neg y^F) \tag{7}$$

This condition causes the creation of the fault effect. Second, require $z^G = 1$ or $z^G = 0$ depending on whether the fault is stuck at 0 or 1, respectively.

- If the gate with output y has a non-controlling value [1], $nc(y)$, require the side inputs of y with respect to z to assume the non-controlling value of y,

$$\begin{aligned}&\bigcup_{w \in I(y) - \{z\}} (w) && \text{if } nc(y) = 1 \\ &\bigcup_{w \in I(y) - \{z\}} (\neg w) && \text{if } nc(y) = 0\end{aligned} \tag{8}$$

These clauses allow propagation of the fault effect from node z to node y if the gate y has a controlling value. We refer to the sub-formula obtained from (8) as φ^N, and thus the fault-specific formula becomes

$$\varphi^{FS} = \varphi^F \cup \varphi^S \cup \varphi^B \cup \varphi^E \cup \varphi^A \cup \varphi^N \tag{9}$$

Given the proposed CNF formulations for the fault detection problem, we have the following formal results:

Proposition 1. Given a stuck-at stem fault z s-a-v, or a fanout-branch fault (z, y) s-a-v, the fault is detectable if and only if the associated fault detection formula $\varphi^D = \varphi^{FS} \cup \varphi^G \cup \varphi^R$ is satisfiable.

Proposition 2. For any fault in a combinational circuit composed of simple gates, the size of the associated fault detection formula φ^D is $O(N)$ (clauses or literals), where N is the number of circuit nodes.

As will be shown in the remainder of the paper, the proposed CNF formulation can be simplified and improved. For example, nodes that do not affect the fault detection problem need not be included in the circuit formula φ^G. In contrast, other conditions can be added, which permit pruning the

1. The fanout-branch from node z to node y is denoted by edge (z, y), which can be stuck at a given logic value.

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

// Input argument:        Current decision level d
// Output argument:       Backtracking 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 3: Description of GRASP

amount of search associated with the satisfiability problem.

3 The Propositional Satisfiability Algorithm

In this section we briefly review the GRASP (Generic seaRch Algorithm for Satisfiability Problems) SAT algorithm, developed by Silva and Sakallah [15][1]. GRASP can be used as a library of search algorithms and hence it can integrated as the back-end search engine of other applications. GRASP is a backtrack search algorithm organized as shown in Figure 3. Each stage of the search process is characterized by a decision level. 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. During the search, we say that a ***conflict*** occurs when one or more unsatisfied clauses result from a given partial assignment to the variables. 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.
2. `Deduce()`, which implements Boolean Constraint Propagation as described in [15, 18]. This procedure is equivalent to the derivation of implications in digital circuits [1].
3. `Diagnose()`, which identifies the causes of conflicts and can augment the clause database with additional implicates. These implicates are referred to as ***conflict-induced clauses***.
4. `Erase()`, which deletes the assignments at the current decision level.

The distinguishing feature of GRASP is the ability to diagnose conflicts, and to record the causes of conflicts as conflict-induced clauses. These clauses provide a unified mechanism for implementing the following search-pruning techniques:

- A ***non-chronological backtracking*** search strategy. Non-chronological backtracking permits jumping over parts of the decision tree where a solution cannot be found.
- Early identification of conflicts associated with ***equivalent conflicting conditions***. This technique is provided automatically by adding conflict-induced clauses to the clause database.
- ***Unique implication points*** permit finding necessary assignments to prevent the occurrence of known conflicting conditions.

The basic procedure for conflict-induced clause identification consists of recording dependencies associated with variable assignments while tracing implication sequences from a given unsatisfied clause to the decision assignment causing the conflict. (Further details of the algorithm and a description of the conflict diagnosis procedure can be found in [15].)

4 Integration of SAT Algorithms in ATPG

The design of a SAT-based ATPG tool must take into account two key issues. First, the tool must help the SAT algorithm in reducing the amount of search for each fault. Second, the tool ought to overcome possible drawbacks

1. A detailed description of GRASP can be found in [15].

inherent to the algorithmic framework chosen.

Regarding the first issue, two main search reduction techniques can be applied:

1. Incorporate in the CNF formulation additional information that can be used to reduce the amount of search. This is the case, for example, of the structural information associated with fault detection problems.
2. Extend the pruning ability of the search algorithm by taking into account any specific properties of the ATPG process. For example, since ATPG involves a sequence of fault detection problems, we can simplify subsequent fault detection problems by *learning from* difficulties encountered in previous fault detection problems.

The second issue involves developing mechanisms aimed at reducing the effects of potential drawbacks of the approach chosen. Current SAT algorithms are known to exhibit the following major drawbacks when used for fault detection:

1. SAT-based ATPG algorithms are known to over-specify test patterns. Hence, a SAT-based ATPG algorithm ought to include mechanisms for preventing the overspecification of test patterns.
2. Decision-making procedures in SAT algorithms cannot exploit structural information for deciding assignments. Nevertheless, relating decision assignments with relevant circuit nodes can be particularly helpful in reducing the amount of search.

In the following subsections we propose solutions to each of the above drawbacks, which have been incorporated into TG-GRASP, a SAT-based ATPG tool developed on top of the GRASP SAT algorithm [15].

4.1 Including Structural Information

Most, if not all, of recent structural ATPG algorithms have included structural information in order to prune the amount of search. A paradigmatic example of this fact are the identification of ***unique sensitization points*** (USPs), originally introduced by Fujiwara and Shimono in [5], and further generalized by Schulz and others in [12-14]. A unique sensitization point is a node that must propagate the error signal for the error signal to be observed at a primary output. Unique sensitization points permit identifying necessary assignments, thus constraining the search process. In TG-GRASP structural information is identified while constructing the CNF formula for a given fault, and can be used for either defining necessary assignments or adding additional clauses which permit pruning the amount of search during the search process.

TG-GRASP can include structural information associated with USPs under two different perspectives. While generating the CNF formula, ***static*** USPs are identified using the linear-time algorithm described in [14]. Each static USP node u leads to setting $u^S = 1$. Furthermore, if node u has a controlling value, then every fanin node v of u that cannot propagate the error signal must assume the non-controlling value of u.

Another way to include structural information in the CNF formula is to establish conditions denoting implied variable assignments under the assumption that a node becomes a unique sensitization point. For the example circuit of Figure 1, if x_6 becomes a USP and x_2 cannot propagate the error signal, then x_2 must assume the non-controlling value of x_6. This condition can be captured by the clause $(\neg x_6^S + x_2^S + x_2^G)$, because the non-controlling value of x_6 is 1.

We can thus conclude that by identifying static USPs and by adding conditions defining implied assignments due to dynamic USPs, the proposed CNF formulas are necessarily more constrained versions of the CNF formulas proposed in [3, 11, 18]. Nevertheless, we note that a SAT-based ATPG algorithm cannot use truly dynamic structural information for pruning the search, the way it is done in some structural algorithms [13, 14].

4.2 Increased Pruning Ability

The GRASP SAT algorithm provides several powerful pruning techniques. When GRASP is used within another application, some of these pruning techniques can be naturally generalized, thus further extending the pruning power of the resulting tool. One of the key features of GRASP is its ability to record conflict-induced clauses and use them to prevent similar conflicts from being identified subsequently during the search process. For ATPG it is plain that some of these conflict-induced clauses are independent of the given target fault and depend only on the function and structure of the circuit. Hence, such conflict-induced clauses can be *re-used* for other target faults, and so need only be derived once. We refer to such clauses as ***pervasive*** conflict-induced clauses.

Proposition 3. Let there be a conflict during the search

process, such that any clause ω involved in the conflict is included in the circuit formula, i.e. $\omega \in \varphi^G$. Then the resulting conflict-induced clause is an implicate of the circuit consistency function.

In TG-GRASP, and after detecting a given fault, all created conflict-induced clauses are analyzed. If a clause is not pervasive it is discarded. If a clause is pervasive and its size is below a user-specified maximum allowed threshold size, then the clause is kept for subsequent faults. Otherwise the clause is also discarded. Consequently, the clause recording mechanisms of GRASP can be naturally extended for ATPG. Pervasive clauses that are kept in the clause database permit simplifying the search for subsequent faults.

An interesting side result is that any of the pervasive clauses created under Proposition 3 can also be *re-used* in other circuit analysis tasks besides ATPG, e.g. Delay-Fault Testing, Path Delay Computation, Combinational Equivalence Checking, among others. Moreover, an open issue is how to extend the definition of pervasive conflict-induced clause in order to include other conflict-induced clauses that are derived from clauses that are not necessarily contained in φ^G.

4.3 Reducing Test Pattern Overspecification

By definition an instance of SAT is satisfied when all clauses are satisfied. This requirement may lead, in ATPG to the overspecification of test patterns. This problem does not arise in structural ATPG algorithms where the termination conditions are much less stringent. In fact, as soon as the error signal reaches a primary output and the justification frontier [1] becomes empty, most structural ATPG algorithms declare the fault to be detected, thus potentially allowing many primary inputs to remain unassigned.

The procedure used in TG-GRASP for reducing the overspecification problem hinges on the following observation. It is plain that there are assignments that satisfy φ^G, since we can always find consistent assignments in a combinational circuit. Furthermore, from the definition of the fault detection problem in Table 2, we can also conclude that consistent assignments can always be found for the formula $\varphi^G \cup \varphi^F \cup \varphi^S \cup \varphi^B \cup \varphi^E$.

In general, clauses in $\varphi^A \cup \varphi^R$ are declared as requiring being satisfied. Each time a variable y becomes assigned, all clauses containing literals in y are also said to require being satisfied. Hence, the search process can terminate when all clauses that require being satisfied are indeed satisfied. This termination condition implies that we can terminate the search process even when some clauses are not satisfied, since we know beforehand that those clauses can be satisfied. This modified termination condition for SAT is referred to as ***syntactic satisfiability***.

4.4 Decision Making Procedures

One clear advantage of structural ATPG algorithms is that decision assignments can be made on the primary inputs and heuristically related with goals of the ATPG process. Such goals may include the controllability of a line or the observability of another. SAT algorithms cannot directly exploit such structural information in order to guide the search process. Nevertheless, and as noted by Stephan et al. in [18], the variables can be reordered so that decisions will be first made with respect to primary inputs close to the objectives being satisfied. TG-GRASP incorporates such techniques. Hence, each time a fault is being targeted, variables are reordered so that decisions will be first made with respect to the primary inputs close to the site of the fault. Note, however, that this decision making procedure orders variables statically, and consequently it is not necessarily as effective as dynamic decision making procedures used by structural algorithms.

5 Experimental Results

The TG-GRASP ATPG algorithm has been implemented as a new software layer on top of GRASP. The ISCAS'85 [2] benchmarks were used to evaluate the algorithm and, in order to fully evaluate the robustness of the proposed algorithmic organization, every possible fault in each circuit is targeted. A single and fixed decision making procedure was used. In addition, *no* preprocessing was used. These experimental conditions are in explicit contrast with most algorithmic organizations in ATPG, because each fault becomes harder to detect. Thus we are able to evaluate in greater detail how robust the proposed algorithm is.

Both GRASP and TG-GRASP have been implemented in the C++ programming language, and compiled with GCC 2.7.2. TG-GRASP was run on a SUN 5/85 machine with 64 MByte of RAM. The experimental results of running TG-GRASP on the ISCAS'85 benchmarks are shown

Circuit	#F	#D	#R	#A	CNF Time	SAT Time	Total Time
C432	524	520	4	0	71.0	27.3	98.5
C499	758	750	8	0	61.5	15.6	77.0
C880	942	942	0	0	29.7	8.6	38.3
C1355	1574	1566	8	0	121.7	105.1	226.8
C1908	1879	1870	9	0	133.0	32.5	165.5
C2670	2747	2630	117	0	95.2	30.8	126.0
C3540	3428	3291	137	0	236.2	78.8	315.0
C5315	5350	5291	59	0	104.5	27.3	131.8
C6288	7744	7708	34	0	497.2	107.2	604.4
C7552	7550	7419	131	0	141.2	44.3	185.5

Table 3: Results on the ISCAS'85 benchmark circuits

in Table 3. In this table all the CPU times denote average CPU times per fault in milliseconds (msec). **#F**, **#D**, **#R** and **#A** denote, respectively, the total number of faults, and the number of detected, redundant and aborted faults. The GRASP SAT solver was run with the default set of options described in [15]. The interface between TG-GRASP and GRASP was instructed to keep pervasive clauses of size no greater than 10. Furthermore, decision making followed the procedure described in Section 4.4.

As can be concluded, even under particularly adverse experimental conditions, TG-GRASP is able to detect or prove redundant every fault in all ISCAS'85 circuits. Moreover, the average running time for each fault is competitive with the most efficient ATPG algorithms. One other conclusion is that a significant percentage of the CPU time is spent generating the CNF formulas. This data agrees with experimental data from other authors [11, 18].

We can also conclude that the behavior of TG-GRASP improves as the circuit size increases. This is in contrast with other ATPG tools, and justifies using robust ATPG algorithms. In addition, and to our best knowledge, TG-GRASP is the first ATPG tool that is able to detect or prove redundant every fault in the ISCAS benchmark suite without using any form of preprocessing and under a fixed decision-making strategy.

Finally, we note that additional experimental results, obtained on circuits synthesized using power management techniques, also support the robustness claim of TG-GRASP [17].

6 Conclusions and Ongoing Work

In this paper we describe a SAT-based ATPG algorithm, TG-GRASP, that is targeted at being robust, i.e. not requiring much heuristic knowledge for successfully detecting all faults. The algorithm is built on top of a highly efficient SAT algorithm and further extends the pruning ability of the SAT algorithm by exploiting intrinsic properties of the ATPG problem. Experimental results, obtained with the ISCAS'85 benchmark circuits, confirm that the proposed algorithm is indeed robust and still competitive with other existing ATPG algorithms.

Despite the promising results the proposed approach has important drawbacks. First, the structural information inherent to fault detection can only be exploited under very limited conditions. Second, very noticeable overhead is introduced by using a general SAT solver. Clearly, much better experimental results could be attained if the same search-pruning concepts could be used without having to map fault detection problems into CNF formulas, and thus without having to use the Propositional Satisfiability algorithmic framework. Nevertheless, the proposed version of TG-GRASP strongly suggests that a highly efficient and robust ATPG tool can be developed. Such tool can incorporate powerful search-pruning techniques and still have reduced computational overhead.

Future research work includes the application of the algorithmic framework described in the paper, but bypassing the need for a SAT solver. Hence, the search procedure will be done on top of circuits instead of CNF formulas. Ideally, such search algorithm will include the search-pruning techniques of GRASP, but will avoid the significant overhead of mapping fault detection problems into CNF formulas. Another relevant research topic is the evaluation of the practical usefulness of pervasive conflict-induced clauses. Moreover, stronger conditions, under which conflict-induced clauses can still be declared pervasive, need also to be developed.

References

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

[2] F. Brglez and H. Fujiwara, "A Neutral List of 10

Combinational Benchmark Circuits and a Target Translator in FORTRAN," in *Proceedings of the International Symposium on Circuits and Systems*, 1985.

[3] 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.

[4] H. Cox and J. Rajski, "On Necessary and Nonconflicting Assignments in Algorithmic Test Pattern Generation," *IEEE Transactions on Computer-Aided Design*, vol. 13, no. 4, pp. 515-530, April 1994.

[5] H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms," *IEEE Transactions on Computers*, vol. 32, no. 12, pp. 1137-1144, December 1983.

[6] 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.

[7] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Transactions on Computers*, vol. 30, no. 3, pp. 215-222, March 1981.

[8] T. Kirkland and M. Ray Mercer, "A Topological Search Algorithm for ATPG," in *Proceedings of the 24th Design Automation Conference*, pp. 502-508, 1987.

[9] 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.

[10] W. Kunz and D. K. Pradhan, "Accelerated Dynamic Learning for Test Pattern Generation in Combinational Circuits," *IEEE Transactions on Computer-Aided Design*, vol. 12, no. 5, pp. 684-694, May 1993.

[11] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability," *IEEE Transactions on Computer-Aided Design*, vol. 11, no. 1, pp. 4-15, January 1992.

[12] M. H. Schulz, E. Trischler and T. M. Sarfert, "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System," *IEEE Transactions on Computer-Aided Design*, vol. 7, no. 1, pp. 126-137, January 1988.

[13] 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.

[14] J. P. M. Silva and K. A. Sakallah, "Dynamic Search-Space Pruning Techniques in Path Sensitization," in *Proceedings of the 31st Design Automation Conference*, pp. 705-711, 1994.

[15] J. P. M. Silva and K. A. Sakallah, "GRASP—A New Search Algorithm for Satisfiability," in *Proceedings of the International Conference on Computer-Aided Design*, November 1996.

[16] J. P. M. Silva and K. A. Sakallah, "Robust Search Algorithms for Test Pattern Generation," Technical Report RT/02/97, INESC, Portugal, January 1997.

[17] J. P. M. Silva, J. C. Monteiro and K. A. Sakallah, "Test Pattern Generation for Circuits Using Power Management Techniques," in *Proceedings of the European Test Workshop*, Italy, May 1997.

[18] P. R. Stephan, R. K. Brayton and A. L. Sangiovanni-Vincentelli, "Combinational Test Generation Using Satisfiability," *IEEE Transactions on Computer-Aided Design*, vol. 15, no. 9, pp. 1167-1176, September 1996.

[19] M. Teramoto, "A Method for Reducing the Search Space in Test Pattern Generation," in *Proceedings of the International Test Conference*, pp. 429-435, 1993.