Chapter 12

SAT AND ATPG: ALGORITHMS FOR BOOLEAN DECISION PROBLEMS

Wolfgang Kunz

João Marques-Silva

Sharad Malik

Abstract The problems of *Boolean satisfiability* (SAT) and *automatic test pattern generation* (ATPG) are strongly related - both in terms of application areas (pre-manufacturing design validation and post-manufacturing testing), as well as in terms of techniques used in their practical solutions (searching large combinatorial spaces through efficient pruning). However, historically these domains have evolved somewhat independently with limited interaction. While ATPG has been primarily driven by reasoning based on circuit structure, SAT has focussed on reasoning using *conjunctive normal form* (CNF) representations of Boolean formulas. In this chapter, we introduce these problems, describe key techniques used to solve them in practice, and highlight the common themes and differences between them.

12.1 INTRODUCTION

The problems of *Boolean satisfiability* (SAT) and *automatic test pattern generation* (ATPG) have been investigated intensively for many decades. Boolean satisfiability serves as an important reference problem in complexity theory of Computer Science. It has been shown that many difficult decision problems from various application domains can be reduced to the Boolean satisfiability problem. Algorithms for solving this problem have been an active field of research and steady progress has been achieved over many years.

The problem of ATPG for combinational circuits is closely related to Boolean satisfiability. ATPG deals with the problem of efficiently generating test stimuli for testing chips after fabrication. As with SAT, ATPG has also been well studied.

SAT and ATPG algorithms have received even stronger interest more recently, since they have been used successfully as Boolean engines for equivalence checking (see Chapter 13) and logic synthesis (see Chapter 2). In this chapter, we ignore the physical aspects of testing and do not consider testing of sequential circuits. Our interest is exclusively in the combinatorial search problems in SAT and ATPG.

Traditionally, the research domains of SAT and ATPG developed quite independently of each other. With a broad spectrum of applications as background, the research in SAT typically focused on general concepts useful in diverse fields. Due to the more specialized problem formulation, research in ATPG focused on techniques that are fine-tuned to dealing with digital circuits. In general, however, there are a lot of commonalities between the notions and methods developed in the two research domains. This chapter describes the basic concepts of SAT and combinational ATPG. Where possible, it attempts to bridge the gap between the terminology commonly used in the two areas.

12.2 SAT AND ATPG PROBLEM FORMULATIONS

Algorithms for automatic test pattern generation in combinational circuits are traditionally based on a gate netlist description of the *circuit under test* (CUT). SAT algorithms typically represent the circuit by a *conjunctive normal form* (CNF) formula. This distinction is not very sharp, however. SAT and ATPG are closely related problems and there also exist various ATPG tools based on a CNF representation of the circuit. Note that a gate netlist description can always be translated into a CNF formula, as described in the next section, but not vice versa. In the following, we present the problem formulations for Boolean satisfiability and combinational stuck-at fault testing based on the conventional CNF- and gate netlist representations, respectively.

12.2.1 CONJUNCTIVE NORMAL FORM AND SATISFIABILITY

Instances of SAT are most often represented as CNF formulas. A CNF formula is a *conjunction* (product) of *clauses*, where a clause is a *disjunction* (sum) of *literals*, and a literal is either a variable or its complement. For example $\varphi = (x_1 + \neg x_2)(\neg x_1 + x_3)$, denotes a CNF formula with two clauses and three variables.

The Boolean satisfiability problem for a CNF formula is formulated as follows: Given a CNF formula, φ, representing a Boolean function, $f(x_1, \ldots, x_n)$, the *satisfiability problem* consists of identifying a set of assignments to the formula variables, $\{x_1 = v_1, \ldots, x_n = v_n\}$, such that all clauses are satisfied, i.e., $f(v_1, \ldots, v_n) = 1$, or proving that no such assignment exists.

Combinational circuits can easily be represented using CNF formulas. A CNF formula is associated with each gate, and captures the consistent assignments between the gate inputs and output. The CNF formula for the circuit is the conjunction of the CNF formulas for the gates in the circuit; assignments to the circuit nodes are consistent if and only if the assignments are consistent for the inputs and output of each gate in the circuit.

The derivation of the CNF formula for a gate $x = f(w_1, \ldots, w_j)$ is straightforward. First, a new Boolean function $\xi_x = \overline{x \oplus f(w_1, \ldots, w_j)}$ is defined. Observe that ξ_x only assumes value 1 provided x and $f(w_1, \ldots, w_j)$ assume the same value. Next, ξ_x is represented as a product-of-sums form, φ_x. Hence, the CNF formula φ_x assumes value 1 if and only if the value of x is equal to the value of $f(w_1, \ldots, w_j)$. As an example consider a 2-input AND gate, $x = a \cdot b$. The resulting formula becomes $\xi_x(x, a, b) = \overline{x \oplus f(a, b)} = \overline{x \cdot (\overline{a} + \overline{b}) + \overline{x} \cdot a \cdot b} = (\overline{x} + a) \cdot (\overline{x} + b) \cdot (x + \overline{a} + \overline{b}) = \varphi_x$.

12.2.2 GATE NETLIST AND STUCK-AT FAULT TESTING

Techniques of test generation most often rely on a gate netlist description of the circuit. A gate netlist can be modeled as a *Boolean network*. A Boolean network is a graph, with vertices (referred also to as nodes, variables or signals) representing gates and directed edges representing connections between gates. Each vertex has a function associated with it which corresponds to the function of the gate it represents.

In the following, we always assume that each function in the Boolean network is very simple such that it can be implemented by one of the primitive gate types AND, OR, NOT, NAND or NOR. Each of these gates can have an arbitrary number of inputs. Extending the ATPG techniques of this chapter to XOR, XNOR or other more complex functions is possible but will not be further considered.

The goal of testing is to detect physical defects on a chip inflicted by the fabrication process or those occurring later during the operation of the chip. Physical defects are described at the gate level by certain *fault models*. The *single stuck-at fault* is a widely accepted model. It assumes that a *single* line in the combinational circuit fails to change its logic value and is "stuck" at a constant value of 0 or 1.

Techniques of test generation can be roughly divided into *random* and *deterministic* methods. In random methods, a set of *random* input stimuli is generated for the circuit under test (CUT). A *fault simulator* can be used to measure the quality of the random stimuli with respect to detecting a given set of faults. For a survey of fault simulation techniques and the use of random testing in the context of *design for testability*, see, e.g. [1]. In *deterministic* ATPG, the

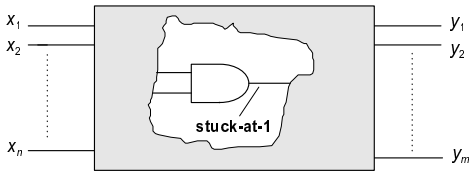

Figure 12.1. Circuit under test (CUT).

starting point of all operations is a list of faults that must be detected by the tests to be determined. The faults in the fault list are targeted one after the other and for every fault a test is derived by the ATPG algorithm. In this chapter, we exclusively consider *deterministic* ATPG. For reasons of simplicity the word "deterministic" is often omitted.

The problem of test generation for single stuck-at faults in combinational circuits is illustrated in Figure 12.1. For a given gate netlist C there is a fault list $\Phi = \{\phi_1, \phi_2, \ldots, \phi_k\}$ containing all faults for which a test has to be generated. These faults are called *target faults*. Let C have n primary input signals and m primary output signals and let $\boldsymbol{y} = (y_1 = v_1, y_2 = v_2, \ldots, y_m = v_m)$ be the response of the *fault-free* circuit and $\boldsymbol{z} = (y_1 = v'_1, y_2 = v'_2, \ldots, y_m = v'_m)$ be the response of the *faulty* circuit to some input stimulus $\boldsymbol{x} = (x_1 = w_1, x_2 = w_2, \ldots, x_n = w_n)$ with $v_i, v'_j, w_k \in \{0, 1\}$. The input stimulus $\boldsymbol{x}$ is called a *test* for a fault ϕ if and only if the circuit response $\boldsymbol{z}$ in the presence of ϕ is different from the response $\boldsymbol{y}$ of the fault-free circuit. There may not exist a test for every fault in a circuit. Untestable faults are also called *redundant*. The task of a deterministic ATPG algorithm is to calculate a test $\boldsymbol{x}$ for a given fault ϕ if a test exists, or to prove its untestability otherwise. An input stimulus $\boldsymbol{x}$ that detects a fault ϕ is also called a *test vector* or *test pattern* for fault ϕ.

12.3 COMBINATIONAL DETERMINISTIC ATPG

After introducing the basic problem formulations for SAT and ATPG we now consider the algorithms for solving these problems. The basic SAT procedure will be described in Section 12.4. This section gives an introduction to the general combinational ATPG procedure. It will become apparent how ATPG and SAT are generally related.

12.3.1 LOGIC ALPHABETS

A combinational circuit with n primary inputs and m primary outputs is specified at the gate level by an m-ary function $f : B^n \rightarrow B^m$, where B is the chosen domain of logic values. For fault-free circuits it is common to use the logic alphabet $B_2 = \{0, 1\}$. Often, for specific problems other logic alpha-

AND	0	1	X	D	$\overline{\text{D}}$
0	0	0	0	0	0
1	0	1	X	D	$\overline{\text{D}}$
X	0	X	X	X	X
D	0	D	X	D	0
$\overline{\text{D}}$	0	$\overline{\text{D}}$	X	0	$\overline{\text{D}}$

OR	0	1	X	D	$\overline{\text{D}}$
0	0	1	X	D	$\overline{\text{D}}$
1	1	1	1	1	1
X	X	1	X	X	X
D	D	1	X	D	1
$\overline{\text{D}}$	$\overline{\text{D}}$	1	X	1	$\overline{\text{D}}$

f	$\overline{\text{f}}$
0	1
1	0
X	X
D	$\overline{\text{D}}$
$\overline{\text{D}}$	D

Table 12.1 . AND-, OR-, NOT-operation in the 5-valued logic alphabet B_5.

bets are required. In order to describe the faulty behavior of a circuit, Roth's *D-calculus* [37] has become widely accepted. In Roth's notation a signal is assigned the logic value D if it assumes 1 in the fault-free and 0 in the faulty circuit. In the opposite case, if the signal is 0 in the fault-free and 1 in the faulty circuit, it is denoted by $\overline{\text{D}}$. For logic values being equal in the fault-free and faulty cases, namely 0 or 1, the signal value is denoted 0 or 1, respectively. With these notations we obtain the logic alphabet $B_4 = \{0, 1, \text{D}, \overline{\text{D}}\}$. For test generation it is of advantage to introduce a fifth logic value, X, describing the case where no unique logic value has been assigned. This value is usually referred to as the *don't care* or *unknown* value. Including X into B_4 results in the five-valued logic alphabet $B_5 = \{0, 1, X, \text{D}, \overline{\text{D}}\}$. A 5-valued algebra over B_5 can be obtained by straightforward extensions of the usual operations of conjunction, disjunction, and negation for B_2; the truth tables for these extensions are shown in Table 12.1. Although some applications especially in delay testing require more sophisticated logic alphabets [2, 14] Roth's D-alphabet, to this date, forms the basis for many modern ATPG algorithms.

12.3.2 ATPG ALGORITHMS: AN OVERVIEW

It is the great merit of Roth's D-calculus that it allows us to completely describe the faulty behavior of the circuit on its structural gate-level description paving the way for efficient heuristics. In this section, we describe a conventional procedure of test generation based on the D-alphabet operating on a gate netlist representation of the circuit.

For illustration of the general ATPG procedure consider the circuit of Figure 12.2. Consider the single stuck-at-1 (s-a-1) fault at signal j. Obviously, an input vector can only be a test for this fault if it exhibits a faulty logic value or *fault signal* at the fault line j. This process is called *fault excitation, fault setup* or *fault injection*. In our example, signal j has to be *controlled* from the primary inputs in such a way that it assumes a logic 0 in the fault-free case. If signal j is 0 in the fault-free case and 1 in the faulty case, then it assumes the logic value $\overline{\text{D}}$ in Roth's notation.

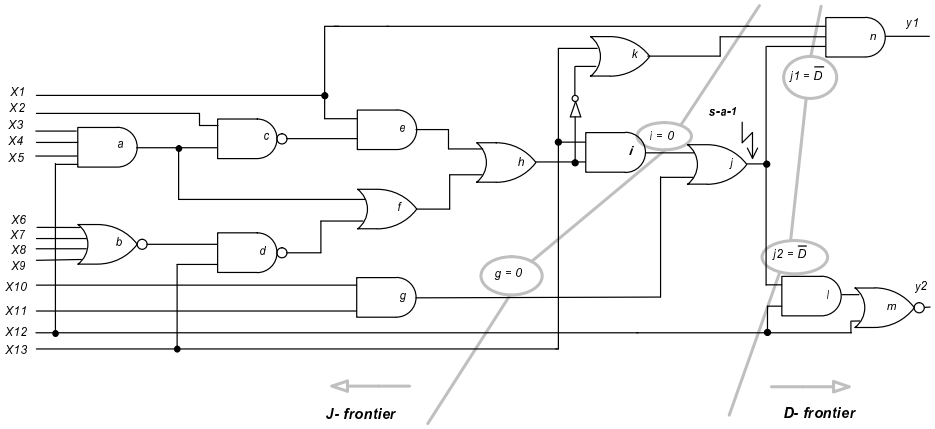

Figure 12.2. Fault injection.

The problem of identifying a set of value assignments at the primary inputs such that a faulty logic value is produced at the fault line is called the *controllability problem* in ATPG. Further, it must be ensured that a faulty signal at fault line f can propagate to at least one primary output of the circuit. Identifying a set of value assignments at the primary inputs such that a signal change at f from 0 to 1 or from 1 to 0 can propagate to at least one primary output is called the *observability problem* for f. Test generation for stuck-at faults requires the simultaneous solution of the controllability and the observability problem for a given fault line f.

After fault injection all common test generators perform logic implications, i.e., they make value assignments that can be derived from fault injection. In our example, we assume that the implication procedure produces the new value assignments $g = 0$, $i = 0$, $j_1 = \overline{\mathrm{D}}$ and $j_2 = \overline{\mathrm{D}}$ as shown in Figure 12.2. This is the starting point for all subsequent steps. Among the signal assignments generated during the ATPG process, there are two sets of signals that are of particular interest, called the D-frontier and the J-frontier.

The *D-frontier* F consists of all fault signals, i.e., signals being assigned either D or $\overline{\mathrm{D}}$, which are input signals of logic gates whose output signal is unspecified. The *D-frontier* indicates how far the faulty signals have propagated from the fault location towards the primary outputs. In Figure 12.2 fault injection has produced the D-frontier $F = \{j_1, j_2\}$.

The *J-frontier* U consists of all output signals of gates being assigned a value of 0 or 1, where this logic value cannot be implied from the logic values at the gate inputs. More commonly, the signals of the J-frontier are called *unjustified lines*. They represent signals in the circuit, where value assignments have produced fixed logic values at the outputs of internal gates which, however, are

not completely *justified* by the value assignments at the gate inputs. In Figure 12.2 two unjustified lines $U = \{i, g\}$ have been created. For a valid test, they have to be justified in subsequent steps of test generation.

Using these notations the task of a test generator can be reformulated as follows: find a set of binary value assignments at the primary inputs of the circuit, such that

- the *D-frontier* reaches the primary outputs, i.e., at least one primary output assumes a faulty value (solve the observability problem),
- the *J-frontier* reaches the primary inputs, i.e., there exist no unjustified lines in the interior of the circuit (solve the controllability problem).

Note that the controllability problem is essentially a SAT problem, and is solved by identifying a set of assignments for the inputs of the circuit such that all signals of the J-frontier are *satisfied*. Therefore, not surprisingly, many algorithmic concepts for combinational test generation are closely related to those of SAT solving methods and SAT solvers can be used to solve the ATPG problem in parts or entirely.

To accomplish the above two tasks most algorithms for deterministic test generation proceed step by step, assigning appropriate logic values at well-selected signals in the circuit such that the D-frontier is moved towards the primary outputs and the J-frontier is moved towards the primary inputs.

Along the process of test generation we distinguish two types of value assignments. First, we consider *necessary assignments*. Necessary assignments are uniquely determined by the current situation of value assignments in the circuit and any test vector that can be generated starting from the current situation *must* contain this value assignment. In Figure 12.2 fault injection has produced the necessary value assignments $i = 0$, $g = 0$, $j_1 = \overline{\mathrm{D}}$, $j_2 = \overline{\mathrm{D}}$. Besides necessary assignments every test generator makes *optional assignments*. Optional assignments are assignments that *can* be made in order to reach the goal of generating a test vector. The existence of optional assignments results from the fact that there usually exists more than just one test vector for a given fault. By making optional assignments we continue to restrict the number of possible test vectors until we finally end up with exactly one. The choice of optional value assignments is subject to *heuristics*.

Test generation can be understood as a sequence of making optional and necessary assignments. Figure 12.3 shows a general procedure that outlines the steps for combinational test generation typical of many popular tools. In the following, we discuss these concepts briefly - they will be treated in more detail in later sections.

A) Fault injection (fault set-up)

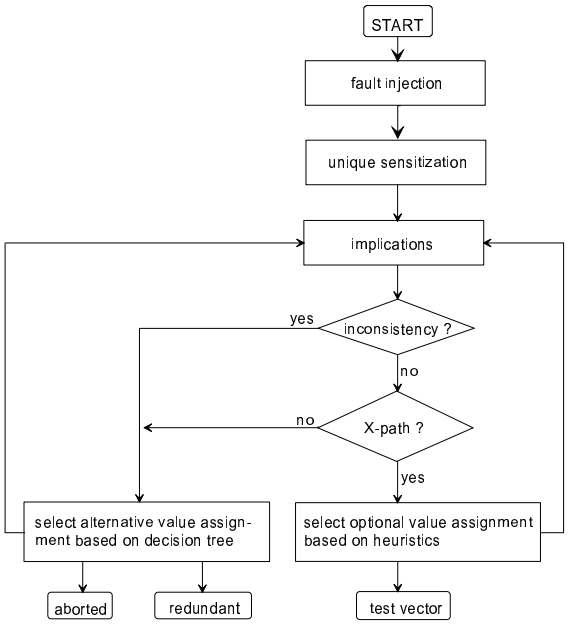

Figure 12.3. General procedure of test generation.

As explained above, the fault signal at the fault location immediately results from the fault value (stuck-at-1 or stuck-at-0) and the definition of the underlying logic alphabet. In Figure 12.2 this is illustrated for 5-valued logic.

B) Unique sensitization

After fault set-up a topological analysis examines the paths along which the fault signal can propagate. Certain topological concepts are used to derive value assignments that are necessary for the propagation of the fault signal. This is called *unique sensitization* and is discussed in Section 12.5.1.4.

C) Implications

For the previously made value assignments, implications are performed in order to restrict the search space. At this point, tools may differ in their ability to perform different types of implications. Implication techniques will be the subject of Section 12.5.1.

D) Checking

Before taking further steps it has to be examined whether the current situation still allows for the generation of a test vector. This is generally not an easy task. However, there are two conditions that must be fulfilled at any point during the test generation process and these can easily be checked:

- logic consistency
- existence of an X-path

First, the logic value assignments must be consistent, i.e. there must be no logic contradictions between signal values in the circuit. This is usually verified during the implication process. Second, there has to exist at least one path in the circuit along which the fault signal can propagate to at least one of the primary outputs. This condition is fulfilled if there is a path from at least one signal of the D-frontier to one of the primary outputs along which all signals are unspecified, i.e., they have the logic value X. We say that the current situation passes the *X-path-check*. Generally, we speak of a *conflict* if either the X-path-check has failed or logic inconsistencies have occurred.

E) Optional value assignments

If no conflict has occurred and no test vector has been obtained yet, heuristic methods are employed to suggest optional value assignments. Importantly, only *one* and not several *optional* value assignments are made at a time. (An exception is the test generator of [17] where under certain conditions several assignments can be made at once.) After each optional value assignment, implications have to be performed again. Heuristics to select optional value assignments will be discussed in Section 12.5.4.1.

F) Decision tree (backtracking)

In most practical algorithms, the search consists of making decisions that are added to a *decision tree*. Optional value assignments constitute decisions. Each decision is associated with a node in the decision tree. Importantly, necessary assignments are not decisions and they are therefore not represented in the decision tree. If a conflict occurs during test generation, previous decisions and the necessary assignments associated with them have to be reversed, and as will be explained in Section 12.4.1 systematic backtracking is performed. The decision tree guarantees that the search is *complete*. In Section 12.5.2.2 we will present an enhancement to the basic backtracking scheme.

Finally, it should be noted that the notions described in this section are also useful with those ATPG approaches that map the stuck-at testing problem to a Boolean satisfiability problem and use a CNF representation of the circuit, e.g. [27], [43]. CNF-based ATPG methods can exploit many basic concepts of the netlist-based approaches, and vice versa. As pointed out before, controllability is a SAT problem by nature and observability can be mapped to SAT using the notion of *Boolean difference* [27]. Constraints like unique sensitization conditions can be expressed in terms of clauses that are added to the CNF. In this way, structural circuit information such as given by the D-frontier

becomes available to the underlying SAT algorithm. Conversely, concepts developed in the context of SAT solving can be used to solve controllability and observability in ATPG.

12.4 SAT ALGORITHMS: A TAXONOMY

The are two main classes of algorithms for solving instances of SAT: *complete algorithms* and *incomplete algorithms*. Complete algorithms can prove unsatisfiability, given enough resources of memory and time. Incomplete algorithms cannot prove unsatisfiability.

Complete algorithms entail the following approaches:

- Algorithms based on backtrack search, where the search space is implicitly enumerated (see Section 12.4.1).
- Algorithms based on different forms of deduction, that allow deriving all necessary consequences (i.e., all necessary assignments or establishing unsatisfiability) for a given CNF formula (see Section 12.4.2.1).
- Algorithms based on function representation. Examples include the various families of *decision diagrams* (see Chapter 11).

Most incomplete algorithms utilize local search, and consequently are based on applying some form of variable assignment flipping [41]. We will not focus on this class of algorithms as they find limited use in synthesis and verification.

In the next few sections the first two classes of SAT algorithms are described in more detail.

12.4.1 BACKTRACK SEARCH

We have already introduced backtracking as the basic searching scheme in ATPG. Also in the domain of SAT, backtrack search is most widely used and is known as the *Davis-Logemann-Loveland procedure* [11]. Before describing backtrack search for Boolean satisfiability, it is important to distinguish between the original Davis-Putnam procedure [12], that is based on the consensus/resolution operation and not on backtrack search, and the more well-known Davis-Logemann-Loveland procedure (DLL) that implements a form of backtrack search [11]. The simplest form of backtrack search is illustrated in Figure 12.4. A partial assignment A to the problem instance variables is extended at each step. Each time an unsatisfied clause is identified, the search procedure backtracks and attempts a different assignment. The process is repeated until either a solution is found, i.e., all clauses become satisfied, or no more variables can be toggled, in which case the problem instance is deemed unsatisfiable. Despite this procedure's simplicity a few drawbacks are apparent. Consider two clauses $\omega_1 = (x_1 + \neg x_2 + x_{k+2})$ and $\omega_2 = (x_1 + \neg x_2 + \neg x_{k+2})$,

```
BacktrackSearch(A)
  if (A unsatisfies one or more clauses) return false
  if (A satisfies all clauses) return true
  else select target variable x and value v
  A = A ∪ {(x, v)}
  if ( not BacktrackSearch(A))
     A = (A − {(x, v)}) ∪ {(x, v̄)}
     if ( not BacktrackSearch(A))
        A = A − {(x, v̄)}
        return false
  return true
```

Figure 12.4. Naïve backtrack search.

and the assignment $A = \{(x_1, 0), (x_2, 1)\}$. Clearly, by inspection one can conclude that assignment A cannot be extended to an assignment that satisfies the two clauses (i.e. a satisfying assignment). Assuming that in the backtrack search procedure unassigned variables are picked in lexicographical order, then it can take up to 2^k decisions (on variables $x_3, \ldots, x_{k+2}$) to conclude that the assignment A cannot be extended to a satisfying assignment.

All backtrack search SAT algorithms address the above problem by implementing the so called *unit-clause ruleunit clause rule* [12]: if a partial variable assignment causes a clause to be unit (i.e. with a single unassigned literal), then the remaining unassigned literal must be assigned value 1. For example, assume clause $\omega = (x_1 + \neg x_2 + x_3)$, and the assignment $A = \{(x_1 = 0), (x_3 = 0)\}$. Under this assignment, ω is unit, and so x_2 must be assigned value 0 for ω to be satisfied. The iterated application of the unit-clause rule is often referred to as *Boolean constraint propagation* (BCP) and corresponds to the implication process in ATPG as will be explained in more detail in Section 12.5.1.1.

Another less utilized simplification technique is the *pure literal rule* (PLR) [12]. If for a given variable x all of its literals are either all positive or all negative, then the variable can be assigned the value that satisfies all of its clauses.

The first backtrack search algorithm for Boolean satisfiability, the backtrack search version of the Davis-Logemann-Loveland procedure [11], implements backtrack search augmented with Boolean constraint propagation and the pure literal rule. The original Davis-Putnam procedure, despite being based on the iterated application of the consensus operation, also incorporates BCP and PLR.

The standard backtrack search algorithm is shown in Figure 12.5. The basic philosophy is the same as with the ATPG procedure of Figure 12.3. Function `doBCP()` implements BCP and has the task of performing implications after each decision. Function `undoBCP()` erases all assignments implied due to the most recent application of BCP.

```
BacktrackSearch(A)
  if (A unsatisfies one or more clauses) return false
  ApplyPLR(A)
  if (A satisfies all clauses) return true
  else
    select target variable x and value v
  A = A ∪ {(x, v)}
  if ( doBCP(A) ≠ CONFLICT and BacktrackSearch(A))
    return true
  undoBCP(A)
  A = A − {(x, v)} ∪ {(x, v̄)}
  if ( doBCP(A) ≠ CONFLICT and BacktrackSearch(A))
    return true
  undoBCP(A)      /* Undo implied assignments */
  A = A − {(x, v̄)}  /* Undo decision assignment */
  return false
```

Figure 12.5. Standard Davis-Logemann-Loveland procedure (backtrack search).

12.4.2 DERIVING ALL NECESSARY CONSEQUENCES

We now consider a completely different approach for checking the satisfiability of a CNF-formula or the testability of a stuck-at fault. While backtracking, as described in Section 12.4.1, systematically makes decisions in order to generate a satisfying vector, the approaches considered now generate necessary conditions and prove satisfiability by the absence of a conflict.

12.4.2.1 RESOLUTION

The resolution method is a classical method for CNF-based SAT solving that *deduces all* logical consequences from a given formula in the form of additional clauses that are added to the CNF. *Resolution* is a fundamental deduction mechanism. In the Boolean domain the principle of resolution is given by

$$(x + y)(\overline{y} + z) = (x + y)(\overline{y} + z)(x + z)$$

The term $x + z$ is called the *resolvent* of $x + y$ and $\overline{y} + z$ and we denote $x + z = res(x + y, \overline{y} + z, y)$.

This law is also known as the *consensus rule*. This name is often used in the field of logic synthesis where the disjunctive form of the law,

$$xy + \overline{y}z = xy + \overline{y}z + xz$$

is more common. Resolution or consensus can be used to check satisfiability of a CNF-formula using the procedure known as *iterated resolution* or *iterated consensus* shown in Figure 12.6.

```
IteratedResolution(φ)
  for (each variable x ∈ φ)
    Let Ω_x be the set of clauses having x as a literal
    Let Ω_x̄ be the set of clauses having ¬x as a literal
    for (every pair of clauses (ω_x, ω_x̄), with ω_x ∈ Ω_x, ω_x̄ ∈ Ω_x̄)
      ω_r = res(ω_x, ω_x̄, x)
      if (ω_r = ∅) return false
      add ω_r to φ
    delete from φ all clauses in Ω_x̄ and in Ω_x
  return true
```

Figure 12.6. The iterated resolution procedure.

The procedure basically applies resolution operations between pairs of clauses with the goal of eliminating one variable at each step of the algorithm. If the empty clause is derived, then the problem instance is unsatisfiable. Otherwise, the empty set of clauses is derived, and the problem instance is declared satisfiable. At each step of the iterated resolution procedure, one can utilize the iterated unit clause rule (i.e. BCP) and the pure literal rule. The resulting algorithm is the original Davis-Putnam procedure [12].

Note an important difference with the backtrack search algorithm in Section 12.4.1. If the formula is satisfiable the resolution method will not immediately yield a satisfying vector. Satisfiability is proved by the absence of the empty clause. This principle has already been employed to check the validity of Boolean formulas ever since the early days of Boolean algebra (see also [7]). Later, in the 1950s, it was discovered by Quine and McCluskey that the method of iterated consensus can be used when minimizing two-level circuits. If iterated consensus is run to completion and all subsumed terms are removed we obtain the *prime implicants* (if they are product terms) or *prime clauses* (if they are sum terms) of a Boolean function. They play an important role in almost all exact two-level minimization procedures.

When solving the Boolean satisfiability problem, exhaustive deduction as given by the above procedure is only of limited use as it may lead to the generation of an exponential number of clauses. Therefore, resolution is usually only applied to generate a limited amount of additional clauses. These can help to prune the search space in a backtrack search based on a decision tree.

12.4.2.2 AND/OR SEARCH

A different approach for deduction, applicable especially in multi-level circuit representations, has been suggested in [25, 26] and is known as *recursive learning* or *AND/OR reasoning*. With ATPG as background, it was originally developed to run directly on a gate netlist description but is also applicable to CNF-based SAT solving. The basic philosophy is similar to that of resolution methods. Instead of exploring all "options" to fulfill a certain requirement

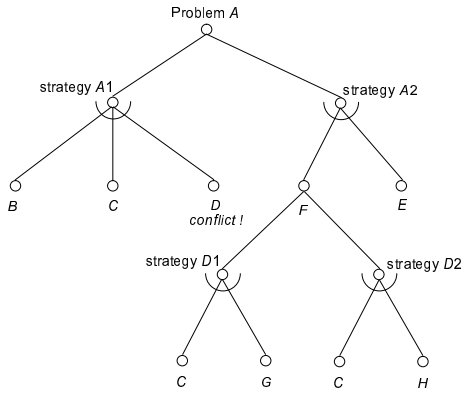

Figure 12.7. Problem solving with AND/OR trees.

as in backtracking, all "necessities" are enumerated. This leads to a general searching scheme sometimes categorized as AND/OR search in the AI literature [36]. Unlike the decision tree of Section 12.4.1 the AND/OR search tree has *two* types of nodes, AND-nodes and OR-nodes.

Consider Figure 12.7 . The initial problem to be solved corresponds to the root node in the AND/OR tree. Several strategies are possible. If one strategy is successful the problem is solved. Therefore, the root node is an *OR*-node. In our example, problem A can be solved by strategy $A1$ or $A2$. In strategy $A1$ three sub-problems B, C and D arise. *All* need to be solved, therefore $A1$ corresponds to an AND-node. Actually, it turns out that D leads to a conflict. Hence, strategy $A1$ is not successful and we consider $A2$. This leads to problems E and F where two strategies exist for problem F. This process continues until the given problems can no longer be decomposed into subproblems. A closer analysis reveals that no matter what strategy we choose, problem C needs to be solved. This yields the implications $A \Rightarrow C$ and $\bar{C} \Rightarrow \bar{A}$. Another implication that is perhaps less evident is $\bar{G}\bar{H} \Rightarrow \bar{A}$.

In [26] an AND/OR searching scheme was formulated to derive all necessary conditions for the satisfiability of a Boolean function in a Boolean network. In contrast to backtrack search (based on a decision tree) that can sometimes finish quickly when a sufficient solution exists, AND/OR reasoning can sometimes terminate early when no solution exists. The original motivation was therefore to prove the untestability of stuck-at faults in combinational circuits.

Other applications result from the fact that for an arbitrary node y in the Boolean network, AND/OR reasoning can derive product terms of the form $x_1 \cdot x_2 \cdot \ldots x_n$ with $n \geq 1$ such that $(x_1 \cdot x_2 \cdot \ldots x_n = 1) \Rightarrow (y = 1)$ or $(x_1 \cdot x_2 \cdot \ldots x_n = 1) \Rightarrow (y = 0)$. These product terms have been referred

to in [26] as *1-implicants* and *0-implicants*, respectively. In the special case of $n = 1$, the technique is called *recursive learning* and can be used to derive all implications in a Boolean network (see also Section 12.5.1.1). Besides pruning the search space, implications and implicants in multi-level circuits can be very useful in multi-level logic synthesis [26]. This is based on the same philosophy as for two-level minimization where deduction methods to derive implicants such as iterated consensus have played an important role for a long time.

12.5 SEARCH ACCELERATION TECHNIQUES

12.5.1 DEDUCING CONSTRAINTS

This section provides additional detail on techniques to identify simple logical relationships and constraints that permit us to simplify the problem instances.

12.5.1.1 BOOLEAN CONSTRAINT PROPAGATION AND IMPLICATIONS

As already pointed out in previous sections, SAT and ATPG algorithms make extensive use of techniques that evaluate the given set of value assignments to the variables (signals) of the CNF-formula or the Boolean network in order to derive further assignments that are uniquely determined by the current situation. In the context of test generation, this process is usually referred to as *performing implications*. In the terminology of SAT solving methods it is called *Boolean constraint propagation* (BCP).

It is helpful to use the following classification of implication techniques in Boolean networks.

In a Boolean network, a value assignment at an arbitrary input or output signal of a logic gate g follows by *simple implication* if it is uniquely determined by previous logic value assignments at other input or output signals of g and by the function of g (AND, OR, NOT, NAND, NOR). A simple implication is a local evaluation of a given gate. Additionally, it is common to distinguish between *forward* and *backward* implication, depending on whether the new value assignment is implied at the outputs or the inputs of the considered gate. Figure 12.8 a) and b) show examples of simple forward and backward implications. Simple implications have an equivalent counterpart in the domain of evaluating CNF-formulas. Simple implications in a Boolean network correspond to the application of the *unit-clause rule* as described in Section 12.4.1 if the circuit is modeled by a CNF-formula.

A logic value assignment is said to be obtained by *direct implication*, if it is determined by a sequence of simple implications. While simple implications are restricted to locally evaluating single gates, direct implications can make

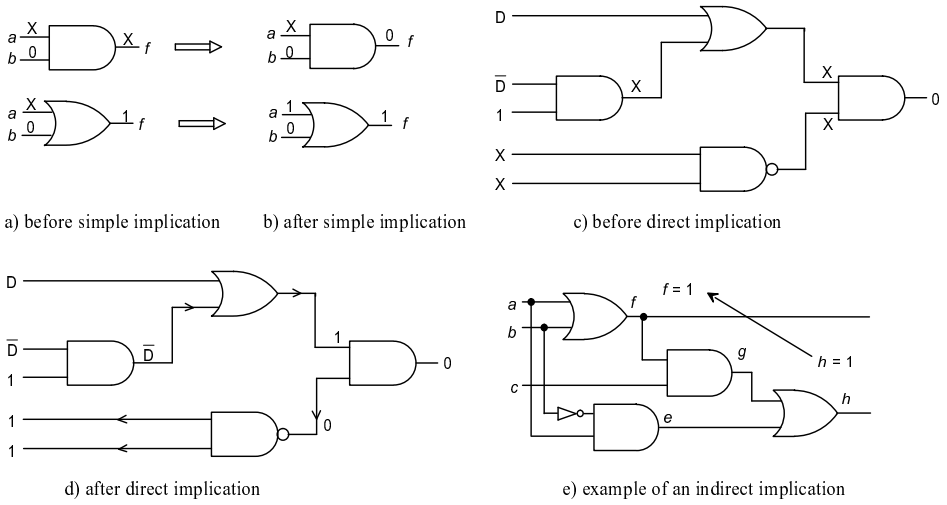

Figure 12.8. Types of implications in a combinational circuit.

value assignments across several gates along the paths in a circuit; see Figure 12.8 c) and d). Direct implications in a Boolean network correspond to BCP based on the unit-literal rule for a CNF-formula. Note also that the pure-literal rule of Section 12.4.1 has a counterpart in ATPG. Applying the pure-literal rule corresponds to making assignments to so called *head objectives* [16] that can be identified in Boolean networks by *backtracing* techniques such as those described in Section 12.5.4.1.

If a value assignment at a given signal is uniquely determined by previous value assignments in the circuit and cannot be implied directly, then it is said to be determined by *indirect* implication. Figure 12.8 e) shows an example of an indirect implication. Indirect implications can only be identified by more sophisticated deduction schemes such as the ones already described in Sections 12.4.2.1 and 12.4.2.2, as well as additional ones we describe next. SOCRATES [39] was the first test generator to identify some indirect implications.

12.5.1.2 VARIABLE PROBING

The goal of identifying logical constraints in ATPG has motivated researchers to study variable probing techniques [39, 40], also known as as *static* and *dynamic learning*. The objective of variable probing is to identify relationships among variables. Consider again the circuit in Figure 12.8e. If the assignment $f = 0$ is "probed", and all direct (forward and backward) implications are performed, then the implied assignment $h = 0$ is identified. As a result, we can

```
ProbeVariables(φ, level)
  ( φ, C) = simple_rules(φ)
  if (is_satisfied(φ))  return (C,  true)
  if (is_unsatisfied(φ))  return (∅,  false)
  if (level ≥ 1)
     for (every x ∈ φ)
        (C_x, status) =  ProbeVariables (φ ∧ C ∧ (x), level − 1)
        if (status = true)  return (C, status)
        (C_x̄, status) =  ProbeVariables (φ ∧ C ∧ (¬x), level − 1)
        if (status = true)  return (C, status)
        C = C ∪ (C_x ∩ C_x̄)
        if (∅ ∈ C)  return (∅,  false)
  return (C,  false)
```

Figure 12.9. Variable probing (Stålmarck's method).

say that $f = 0 \rightarrow h = 0$. Moreover, we can also conclude by applying the law of contraposition that $h = 1 \rightarrow f = 1$. Observe that the latter implication is not readily obtained by a sequence of simple implications but requires an additional concept, in this case, the law of contraposition. Such implications have been referred to as indirect in Section 12.5.1.1 and were called *global* in [39]. Indirect implications can be identified during a preprocessing phase (i.e. static learning) or during the search phase (i.e. dynamic learning). Note that it is not obvious what signals should be probed in order to identify additional value assignments. Therefore, in static and dynamic learning probing is performed for all unassigned signals of the circuit.

In SAT an example of using variable probing is Stålmarck's method (SM) [42, 21]. SM is a complete proof procedure for Boolean satisfiability. The core technique of this proof procedure can be interpreted as the recursive application of variable assignment probing. While static and dynamic learning of [39, 40] apply probing only to all *single* variables of the circuit SM extends the probing process recursively to all pairs, all triples and so forth.

A recursive version of the variable probing algorithm (adapted from [21]) is illustrated in Figure 12.9. At each depth of the recursive procedure, a set C of conclusions (e.g., unit clauses denoting necessary assignments) is maintained. For every variable x the two possible assignments are evaluated and, depending on the outcome, the problem instance can be deemed satisfied, or currently unsatisfiable.

It is plain to conclude that variable probing with arbitrary depth identifies all necessary assignments. Moreover, if the problem instance is unsatisfiable, variable probing is able to conclude so.

12.5.1.3 RESOLUTION AND AND/OR REASONING

Resolution and AND/OR reasoning solve satisfiability by deducing logical consequences rather than enumerating the variable space. The example in

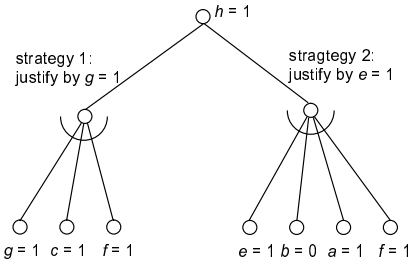

Figure 12.10. AND/OR reasoning tree for $h = 1$.

Figure 12.8e can be used to compare the two techniques. Suppose we want to perform the implications for $h = 1$. First, we consider resolution being applied to the characteristic CNF-formula of the circuit as given by

$$\begin{aligned} z \;=\; & (a + b + \bar{f})_1 \cdot (\bar{a} + f)_2 \cdot (\bar{b} + f)_3 \cdot (f + \bar{g})_4 \cdot (c + \bar{g})_5 \cdot (\bar{c} + \bar{f} + g)_6 \cdot \\ & (a + \bar{e})_7 \cdot (\bar{b} + \bar{e})_8 \cdot (\bar{a} + b + e)_9 \cdot (e + g + \bar{h})_{10} \cdot (\bar{e} + h)_{11} \cdot (\bar{g} + h)_{12} \end{aligned}$$

For better readability of this example the clauses have been numbered as indicated by the index at each clause. Performing the implications for $h = 1$ means that we would like to identify all clauses of type $(\bar{h} + x)$ where x is some signal of the circuit. Applying resolution to all pairs of clauses results in the following resolvents:

1-4:	$(a + b + \bar{g})_{13}$	1-8:	$(a + \bar{e} + \bar{f})_{14}$	1-9:	$(b + e + \bar{f})_{15}$
2-6:	$(\bar{a} + \bar{c} + g)_{16}$	2-7:	$(\bar{e} + f)_{17}$	3-6:	$(\bar{b} + \bar{c} + g)_{18}$
3-9:	$(\bar{a} + e + f)_{19}$	4-10:	$(e + f + \bar{h})_{20}$	5-10:	$(c + e + \bar{h})_{21}$
6-12:	$(\bar{c} + \bar{f} + h)_{22}$	7-10:	$(a + g + \bar{h})_{23}$	8-10:	$(\bar{b} + g + \bar{h})_{24}$
9-10:	$(\bar{a} + b + h)_{25}$				

Clauses that are covered by other clauses can be eliminated. Next, the pairwise comparisons have to be performed for the new clauses too. This procedure has to be continued until no more new clauses are generated. For reasons of brevity we omit these steps but note that the clauses 17 and 20 yield the clause $(f + \bar{h})_{26}$. This corresponds to the implication $(h = 1) \Rightarrow (f = 1)$ which is indirect.

We now examine how the same implication can be identified by AND/OR reasoning. Figure 12.10 shows the AND/OR reasoning tree for the assignment $h = 1$. There are two possible strategies to reach this assignment called *justifications* in [25]. We can justify $h = 1$ by $e = 1$ or $g = 1$ (or both). For each justification we perform direct implications as described in Section 12.5.1.1. Note that in either case we immediately obtain $f = 1$. Hence, $f = 1$ is a

necessary assignment and the implication $(h = 1) \Rightarrow (f = 1)$ is obtained. For a detailed description of recursive learning and AND/OR reasoning see [26].

The example illustrates some differences between resolution and AND/OR reasoning. Resolution produces all prime clauses for a given CNF-formula. If the CNF-formula represents a multi-level Boolean network we obtain all implications and implicants between all nodes in the network. There is usually a tremendous number of such relationships so that only a subset of all clauses can be generated in practice. It is usually hard to develop heuristics such that only those clauses are generated which are relevant for the problem being considered. In the above example, where we are only interested in the implications for $h = 1$ it would be sufficient to only generate clauses 17, 20 and 26.

AND/OR-reasoning solves a more specific problem than resolution. It generates all implicants for a particular target line in a multi-level circuit, such as h in the above example, i.e., it only generates all clauses of the form $(h + \ldots)$ or $(\overline{h} + \ldots)$. Therefore, it can be more efficient than resolution if our interest is in implicants for a specific target line or in implications for a specific set of value assignments in a multi-level circuit.

In general, however, both, resolution and AND/OR-reasoning are fairly expensive methods. They should only be used in SAT or ATPG when other less expensive methods fail.

12.5.1.4 TOPOLOGICAL ANALYSIS FOR UNIQUE SENSITIZATION

In ATPG, the requirement of fault observability can result in additional search constraints called *unique sensitization*. This is based on a topological analysis conducted on a representation of the circuit C as a *directed acyclic graph* (DAG).

Note that a fault signal can only propagate through a specific gate if the other inputs of that gate are set to specific values (1 for AND-gates, 0 for OR-gates). This is called *sensitization* of a gate. A topological analysis of the circuit graph can often identify sets of gates through which the fault signal *must* propagate in order to reach an output [16], [24].

As an example, consider the circuit in Figure 12.2. For a target fault at signal b it is obvious that the fault signal must propagate through gates d, f and h. In the DAG representation of the circuit these nodes are called *dominators*. For the circuit in Figure 12.2 unique sensitization results in the necessary assignments $x_{13} = 1$, $a = 0$ and $e = 0$.

12.5.2 SEARCH PRUNING BY CONFLICT ANALYSIS

There exist a variety of dedicated techniques for pruning the search space in backtrack search algorithms. Most of these techniques come into play whenever conflicting conditions are identified and are equally applicable in SAT and ATPG tools.

To explain these techniques, it is useful to note the following additional definitions relating to the backtrack search process (see Section 12.4.1). A *decision level* is associated with each variable selection and assignment. The first variable selection corresponds to decision level 1. For each new decision, the decision level is incremented by 1. The application of the unit clause rule yields variables whose value is implied. The unit clause ω_x that implies the assignment of a variable x to value $\nu(x)$ is referred to as the *antecedent* of x, $\alpha(x)$. The decision level of each implied variable x, $\delta(x)$, is given by the highest decision level of the variables that directly make ω a unit clause (before x is assigned). The notation $x = \nu(x)@\delta(x) : \alpha(x)$ is used to denote that variable x is assigned value $\nu(x)$ with decision level $\delta(x)$ and antecedent $\alpha(x)$. When a variable w is a decision variable, and so with no antecedent, then $\alpha(w) = \text{NIL}$.

For example, to compute the decision level of a variable x, implied due to a unit clause ω_x, the following formula can be applied:

$$\delta(x) = max\{\delta(y) : (y \in \omega_x) \wedge (y \neq x)\}$$

Observe that the unit clause ω_x becomes satisfied as soon as x is assigned its implied value $\nu(x)$. Moreover, observe that decision levels, although strictly not necessary for implementing the techniques described below, significantly simplify the process of explaining those techniques.

12.5.2.1 CLAUSE RECORDING

At the core of most techniques for pruning the search space is *clause recording*, which has become one of the most often utilized techniques in state-of-the-art SAT solvers [33, 5, 47, 35]. Nevertheless, it is important to observe that clause recording (also known as *nogood recording*) has been a widely used technique in different areas of Artificial Intelligence [38]. Clause recording entails techniques for adding *new* clauses to the CNF formula that explain identified conflicting conditions or help prevent future potential conflicting conditions. For SAT, a formal explanation of one possible approach to clause recording during backtrack search is described in [33]. In this, as in other approaches, the basic idea supporting clause recording is to identify a *sufficient* set of causes for a given conflicting condition to take place during search.

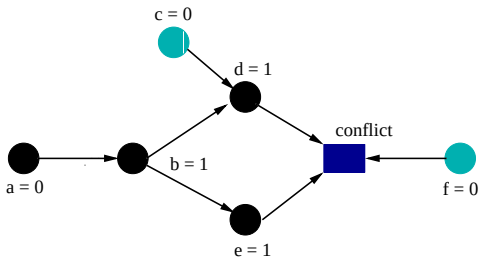

Figure 12.11. Example of clause recording.

Consider the following example CNF formula:

$$\varphi = (a + b) \cdot (\neg b + c + d) \cdot (\neg b + e) \cdot (\neg d + \neg e + f) \cdots$$

Moreover assume the sequence of decision assignments $c = 0$ and $f = 0$. In this case, the order of decisions is $c = 0$ first (hence, $c = 0 \ @ \ 1 : \text{NIL}$), and $f = 0$ next (hence, $f = 0 \ @ \ 2 : \text{NIL}$). Let us further assume a few other decision assignments, to be used in the next section, and finally assume the decision assignment $a = 0 \ @ \ 5 : \text{NIL}$. The obtained implication sequence resulting in a conflict is shown in Figure 12.11.

By inspection it is straightforward to conclude that $(c = 0) \wedge (f = 0) \wedge (a = 0) \Rightarrow (\varphi = 0)$. Since the objective is to have $\varphi = 1$, then one immediately concludes that $\varphi \Rightarrow (c + f + a)$. Hence $(c + f + a)$ denotes a new clause that can be added to the CNF formula (i.e. it can be recorded). Moreover, $(c + f + a)$ can be viewed as an explanation of this conflict, and so enables avoiding the same conflict to occur again during the search.

As the example illustrates, relevant assignments are split into assignments at the *current* decision level and assignments at *lower*, i.e. earlier, decision levels. Assignments at lower decision levels are recorded, whereas assignments at the current decision level are used to trace the causes of the conflict to the decision assignment. The resulting set of assignments (all at lower decision levels with the exception of the decision variable) are used to specify the clause to be recorded [33].

In general it is not feasible to record the explanations for all conflicting conditions identified during the search. As a result, it is often the case that *large* clauses with a number of literals above a certain size are eventually deleted during the subsequent search [33]. Large clauses can be deleted whenever they become unresolved during the subsequent search.

There is no equivalent development to clause recording in the ATPG domain. The most related concepts are probably the techniques in [17, 15] that use the notion of an E-frontier in netlist-based ATPG with the goal of capturing equivalent and dominant states during the search process. The recorded infor-

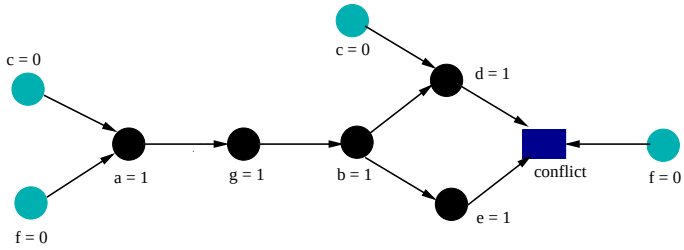

Figure 12.12. Example of non-chronological backtracking.

mation is used to reduce the amount of search for the current and subsequent target faults.

12.5.2.2 NON-CHRONOLOGICAL BACKTRACKING

The backtrack search procedure, as described in Section 12.4.1, implements *chronological backtracking*, where backtracking is always performed to the most recent decision assignment which has not yet been toggled by a previous backtrack step. In general, for real-world problem instances, chronological backtracking can perform poorly when compared with alternative approaches, namely non-chronological backtracking, where the search algorithm can backtrack to decision assignments which are in fact deemed relevant for each identified conflict.

It is interesting to observe that the utilization of clause recording (described in the previous section) provides the necessary information for implementing non-chronological backtracking. The remainder of this section illustrates one possible approach for implementing a non-chronological backtracking search strategy. A detailed explanation of the implementation of non-chronological backtracking is available in [33].

In order to illustrate how non-chronological backtracking can be implemented assume an augmented version of the example in the previous section,

$$\begin{aligned}\varphi \;=\; & (a+b)\cdot(\neg b+c+d)\cdot(\neg b+e)\cdot(\neg d+\neg e+f)\cdot(\neg a+g)\cdot \\ & (\neg g+b)\cdot(\neg h+j)\cdot(\neg i+k)\cdot(a+c+f),\end{aligned}$$

where the clause recorded as a result of the conflict analyzed in the previous section is already shown. Moreover, assume the decision assignments $c = 0$, $f = 0$, $h = 0$ and $i = 0$. The implication sequence that results from this scenario is shown in Figure 12.12. Note that the previous decision variable a is now implied by the just derived conflict clause $(a + c + f)$, and that the new implication sequence leads to another conflict. Analysis of this new conflict readily yields a new clause $(c + f)$, which indicates that backtracking is required and that the backtracking step can be non-chronological. The newly derived clause $(c + f)$ indicates that the decisions $h = 0$ and $i = 0$ are irrel-

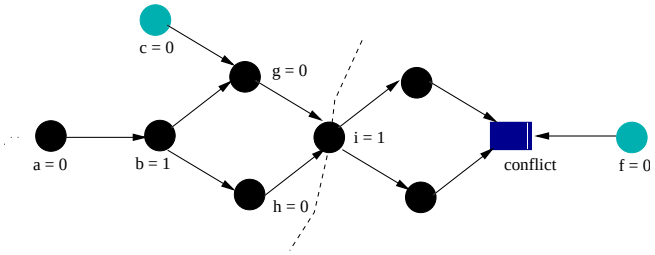

Figure 12.13. Example of unique implication point.

evant for the two most recent conflicts, and that backtracking to one of these two decision assignments would not eliminate these conflicts. To remove the conflicting conditions the backtrack search algorithm needs to backtrack to one of the decisions indicated by $(c + f)$. In general the backtracking step is performed to the most recent decision, in this case $f = 0$.

In ATPG the utilization of non-chronological backtracking search strategies was initially outlined in [29, 30], and detailed and implemented in [31]. Interestingly, the work of [15, 17] did not utilize non-chronological backtracking, and the work of [29, 30, 31] did not consider search equivalence or dominance conditions.

12.5.2.3 RELEVANCE-BASED LEARNING AND UNIQUE IMPLICATION POINTS

Besides the creation of conflict clauses, the ability to backtrack non-chronologically, and the deletion of large clauses, a few additional search pruning techniques have been proposed that also build upon clause recording.

Relevance-based learning [5] consists of extending the life-span of clauses, by deleting unresolved recorded clauses only after a certain number of literals becomes unassigned. We should note that relevance-based learning is often integrated with deletion of large clauses, and so large clauses are only deleted when a certain number of literals becomes unassigned.

Unique implication points denote dominators [45] in the graph of implications of decision assignments with respect to identified conflicts. Unique implication points represent variable assignments which, by themselves, can trigger sequences of implied assignments that yield the same conflicts. Consider the following CNF formula,

$$(a+b)\cdot(\neg b+c+\neg g)\cdot(\neg b+\neg h)\cdot(g+h+i)\cdot(\neg i+d)\cdot(\neg i+e)\cdot(\neg d+\neg e+f)$$

and the sequence of decision assignments $c = 0$, $f = 0$ and $a = 0$. The resulting implied assignments are shown in Figure 12.13. The clause recording procedure outlined in section 12.5.2.1 would identify the new clause $(a+c+f)$. However, noting that the implied assignment $i = 1$ yields by itself the same

conflict, one can create instead two clauses, namely $(\neg i + f)$ and $(a + c + i)$. After erasing the most recent sequence of implied assignments and decision assignment $(a = 0)$, a new unit clause is identified $(\neg i + f)$ and the resulting implied assignments include $i = 0$ and $a = 1$, that result from the two recorded clauses. Observe that the original procedure would only imply $a = 1$ from the sole recorded clause.

12.5.3 OTHER PRUNING TECHNIQUES

The backtrack search procedure can be augmented with different techniques described in previous sections, namely resolution, AND/OR reasoning and variable probing [32]. The integration of these techniques with other search pruning techniques used in backtrack search is technically challenging, because each derived necessary assignment *must* be explained in terms of newly added clauses. Interestingly, the identification of explanations for the necessary assignments identified with resolution, AND/OR reasoning (recursive learning) and variable probing (Stålmarck's method), provides another mechanism for clause recording, and consequently allows establishing new pruning techniques that differ fundamentally (due to clause recording) from the original ones.

Recent work in SAT has involved formula simplification techniques [34] and search strategies based on randomization with restarts [19, 4]. Formula simplification establishes conditions for removing variables and inferring new clauses. Randomization with restarts allows the search process to consider alternative search paths after a pre-determined number of backtracks is reached.

12.5.4 BRANCHING HEURISTICS

12.5.4.1 SEARCH PATHS IN ATPG

The performance of test generators can be greatly influenced by the use of *backtracing* procedures as has been shown in the *FAN-algorithm* [16]. Their goal is to effectively drive the test generation process towards certain *objectives* [16]. The objectives result from the general strategy to move the D-frontier towards the primary outputs and the J-frontier towards the primary inputs.

In [16] an objective is defined as a triple $(f, n_0(f), n_1(f))$ where f is a signal in the Boolean network, $n_0(f)$ is an integer number expressing how strongly the logic value 0 is required at f and $n_1(f)$ is an integer number expressing how strongly the logic value 1 is required at f. The initial objectives are derived from the signals in the J- and D-frontier and often exploit *testability measures*, see e.g. [1]. Backtracking means that these initial objectives are propagated backwards in the circuit according to simple rules. A few examples are shown in Figure 12.14.

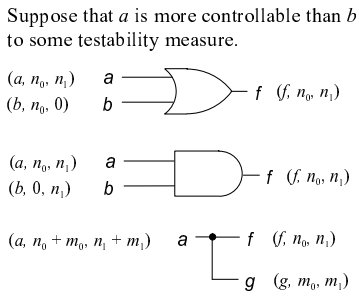

Figure 12.14. Backpropagation of objectives in FAN.

The main goal of backtracing is the early identification of conflicts. If a fanout point f is reached with contradictory requirements, i.e., $n_0(f) \geq 0$ and $n_1(f) \geq 1$, the backtracing process stops and an optional value assignment is made at f, i.e., a decision is made at at this fanout point and inserted in the decision tree. If no conflict occurs backtracing continues until the primary inputs are reached and the numbers $n_0(f)$ and $n_1(f)$ indicate what value assignment should be made to reach the objectives of the test generation process. Importantly, as long as backtracing produces fanout points with contradictory requirements these are always chosen first for decision making so that conflicts are detected as early as possible and backtracks are avoided. An improvement to this scheme based on single path backtracing has recently been proposed in [22].

Note that this is related to the general strategy of selecting the *non-unate* variables in two-level circuit representations as decision points in SAT solving or tautology checking. If we formulated the objectives of a FAN-based test generation procedure by a Boolean function to be satisfied, the conflicting variables would correspond to non-unate variables of this characteristic function. This explains why backtracing is very important. It is the basic instrument to incorporate a strategy also known as *unate recursive paradigm* [6] into a Boolean decision making procedure operating on *multi-level* Boolean networks. Therefore, backtrace procedures represent an integral part of almost all modern ATPG tools.

The above backtrace branching heuristics are essentially based on the circuit structure and try to generate early conflicts by focusing on the fault justification and propagation paths. In contrast to this, since CNF based SAT does not have structural information, its branching procedures must rely on different heuristics based on the distribution of literals in the clauses.

12.5.4.2 BRANCHING HEURISTICS IN CNF-SAT

One of the most important but least understood aspects of SAT algorithms is how to choose the next decision variable at each branching point. Various authors [28, 23, 8, 13] have experimented with many different branching schemes with different degrees of success. Gomes *et al.* provide an excellent review of this subject [20].

Despite the effort spent on decision schemes, it is still unknown what properties make a particular decision scheme better than others. Current decision schemes are mostly based on intuition and empirical results. Usually, a good decision scheme for a certain class of problems may not be good for others. Thus, it is very hard to determine which decision scheme should be used for a certain class of problems.

Traditional decision schemes generally emphasize the quality of the decision, e.g., minimization of the total number of decisions required to solve certain problems. Although the number of decisions is a straightforward metric to compare different strategies – fewer decisions ought to mean smarter decisions – it is not the only metric, since not all decisions yield an equal number of BCP operations. As a result, a shorter sequence of decisions may actually lead to more BCP operations than a longer sequence of decisions. Besides, traditional decision schemes usually involve updating some statistics dynamically, and thus are computationally expensive. As the deduction process becomes more and more optimized, the time spent on decision making in a SAT solver becomes non-negligible. Therefore, it is important to keep the decision-making overhead to a minimum, while maintaining the quality of the branching heuristic. Recently Chaff has introduced a very low overhead decision scheme referred to as *variable state independent decision strategy* (VSIDS) [35]. The emphasis here is on minimizing computation overhead for decision state maintenance in change of variable state during branching and backtracking.

12.6 IMPLEMENTATION ISSUES

Algorithms for ATPG and SAT are especially sensitive to variations in implementation - particularly to the basic data structures used. Netlist based ATPG tools are usually built on ad-hoc data structures. They can directly exploit the physical and structural information represented in the netlist. On the other hand, the logic operations needed are usually more complex than for CNF-based tools. In contrast, in a CNF-based framework it is more difficult to exploit structural circuit information, however, highly efficient data structures have been proposed for BCP and conflict analysis. In the following, we describe some of these data structures.

12.6.1 CLAUSE DATABASE STORAGE

12.6.1.1 SPARSE MATRIX REPRESENTATION

The simplest and currently most widely used data structure is to represent the clauses in a sparse matrix. Each clause stores a vector of the literals in it, and some statistical data, e.g., the number of total literals, the number of free literals, etc. Each variable may keep a list of all its occurrences (as in GRASP [33]) or some of its occurrence (as in SATO [47] or Chaff [35]) in the clause database.

Counter based BCP. The most straightforward BCP method operating on a sparse matrix clause database is to keep three counters for each clause: the total number of literals, the number of literals evaluating to 1, and the number of literals evaluating to 0. The first counter is constant during the entire lifetime of the clause. The other two counters are updated whenever the related variable is set/unset to a value. The status of a clause - conflict, satisfied or unit - can be determined by these three counters (e.g., GRASP [33]). An efficient variation of this method eliminates the need to store the number of literals that evaluate to 1 (e.g., relsat [5]).

Pointer based BCP. A more efficient way to implement the BCP operation is proposed by SATO [46]. Although the data structure in SATO is also based on sparse matrices, it determines the status of the clauses in a different way. Each clause has two pointers: *Head* and *Tail* to point to its first and last unassigned literals. Instead of keeping track of *all* the literal occurrences - for each variable, the *Head List* is a collection of its occurrences as the first *unassigned* literal in the clauses, and the *Tail List* is a collection of its occurrences as the last *unassigned* literal in the clauses. Starting from a variable, one needs to only visit the clauses that have this variable as their *Head/Tail*. Unit clauses and conflict clauses are detected by examining the relative positions of the *Head* and *Tail* pointer of the clauses.

The key advantage of this algorithm is that whenever a variable changes state, only the clauses in the Head/Tail list for this variable will be visited. When we move the *Head/Tail* literals, it is possible that we will encounter a value 1 literal very early if the clause is satisfied. Whenever a search for the next *Head* or *Tail* encounters a literal with value 1, the *Head* or *Tail* of the clause will stop to be active, thus eliminating the need to update the status of the clause in the future.

Chaff [35] proposes an even more efficient implementation of the BCP procedure than that used in SATO. Chaff's BCP is based on the observation that if a clause has more than 1 unassigned literal, it can be neither a unit clause nor a conflict clause. In Chaff, for each clause two literals are being "watched". The

"watched" literals have the property of being the last to be assigned to value 0 if the clause is not satisfied. Each variable only keeps a list of all its "watched" literals. In practice, the "watched" literals of the two different phases (complemented and uncomplemented) are kept in separate arrays. Whenever a variable becomes assigned, Chaff traverses through all the "watched" literals of this variable evaluating to 0.

Chaff's BCP algorithm has all the advantages of SATO's algorithm during the deduction procedure. Moreover, during backtracking, undoing assignments does not require a change in the states of the clauses, i.e. there is no need to change the "watched" literals. The reason is that the "watched" literals have the property of being the last to be assigned value 0 if the clause is not satisfied, therefore, they will be the first to become unassigned if the clause is free again. Therefore, un-assigning variable assignments in backtracking is very cheap. This is a very significant performance advantage.

12.6.1.2 IMPLICIT DATA STRUCTURES

Besides the sparse matrix clause database representation, there are some other data structures that have been proposed in the literature for storing the clauses.

A version of SATO uses a data structure called a *trie* to store the clause database. A trie is a ternary tree. Each internal node in the trie structure is a variable index, and its three leaf edges are labeled Pos, Neg and DC respectively for positive, negative and don't care. A leaf node in a trie is either True or False. Each path from root of the trie to a True leaf represents a clause. A trie is said to be ordered if for every internal node V, Parent(V) has a smaller variable index than the index of variable V. The ordered trie structure has the nice property of being able to detect duplicate and tail subsumed clauses of a database quickly. A clause is said to be tail subsumed by another clause if its first portion of the literals (a prefix) is also a clause in the clause database. For example, $(a + b + c)$ is tail subsumed by $(a + b)$.

Intuitively an ordered trie has some similarity with binary decision diagrams, i.e. easy sharing of common parts. This has naturally led to the exploration of other decision diagram style set representations. Chatalic [10] and Aloul [3] have both experimented with using *zero-suppressed binary decision diagrams* (ZBDD) to represent the clause database. A ZBDD representation of the clause database can detect not only tail subsumption but also head subsumption. Both authors report significant compression of the clause database for certain classes of problems.

It is unclear if the irredundancy advantages of the trie and ZBDD data structure are sufficient to justify the additional maintenance overhead of these data structures compared to the sparse matrix representation.

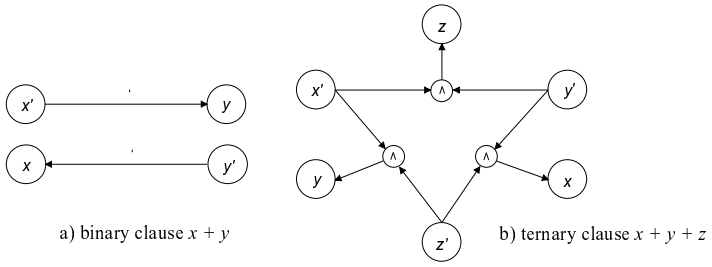

Figure 12.15. Implication graph for clauses with two and three literals.

12.6.2 GRAPH DATA STRUCTURES FOR BCP

In the context of CNF-based ATPG algorithms *implication graphs*[9, 44] have been proposed as specific data structures for efficient implementations of implication techniques. The implication graph of [44] is a directed graph $G_I = (V, E)$ where the vertex set is partitioned into a set of *signal nodes* V_S and a set of $\wedge$*-nodes* $V_\wedge$. Signal nodes represent the encoding bits of a signal x. For simplicity, we assume a three-valued logic alphabet $\{0, 1, \mathrm{X}\}$. In this case, for each clause variable x two nodes, x and x' are introduced in G_I. If we assign in the circuit $x = 1$ or $x = 0$ then in G_I node x or x', respectively, is said to be *set*. Each n-ary clause in the characteristic CNF of the circuit is represented by n associated $\wedge$-nodes in G_I and n^2 edges connecting the $\wedge$-nodes with the signal nodes for the clause variables. The implication graph for a binary and ternary clause are shown in Figure 12.15.

Direct implications can be performed in G_I by *setting* signal nodes according to the following rule:

Starting from an initial set of set signal nodes, all immediate successor nodes v_i are set if node v_i is an $\wedge$-node and *all* its immediate predecessors are set, or if node v_i is a signal node and *at least one* immediate predecessor is set.

This rule is applied repeatedly until no additional node can be set. A logic inconsistency occurs if there is a signal x in the graph such that both x and x' are set.

The simplicity of the above implication rule facilitates an efficient encoding of logic alphabets and a fast bit-parallel implementation of the implication engine. In [44] a reconvergence analysis is proposed to identify indirect implications and it is shown how AND/OR-reasoning can be realized efficiently on this data structure.

12.7 HISTORICAL PERSPECTIVES AND OPEN PROBLEMS

The first complete method to generate test vectors for single stuck-at faults was the famous D-algorithm proposed by Roth [37]. Goel [18] formulated test generation as backtrack search for the input variables of the circuit. In his test generator PODEM he introduced the general ATPG procedure as was discussed in Section 12.3. Important refinements to this procedure were proposed by Fujiwara and Shimono [16]. Their ATPG tool FAN was the first to incorporate a sophisticated backtracing procedure as well as unique sensitization. Another significant step was made in SOCRATES [39] where it was shown by Schulz et al. that the ATPG process can greatly benefit from the identification of indirect implications.

With respect to the SAT domain, the first algorithm was the well-known Davis-Putnam procedure [12], which is based on the application of iterated resolution. Shortly afterwards, the first backtrack search algorithm was proposed [11], which is commonly known as the Davis-Logemann-Loveland (DLL) procedure. Recent years have seen significants improvements being made to the basic DLL procedure, with the introduction of non-chronological backtracking, clause recording and several other very effective search pruning techniques [33, 5]. More recently, the implementation of efficient SAT solvers has been subject to key improvements [35].

One may expect that further progress in the area of SAT and ATPG is very hard after decades of research. However, innovation could result particularly from combinations of search pruning techniques that have so far been considered independently from each other. As already pointed out in Section 12.5.3, the concept of clause recording that has been applied in backtrack search and AND/OR reasoning could be extended to other reasoning schemes like resolution or variable probing, or even approaches using a mix of backtrack search and reasoning schemes.

ATPG and SAT continuously conquer new application domains. Specific applications require specific heuristics. As an example, SAT-based property checking recently became an intensive field of research and it is a challenging task to adapt SAT algorithms to traversing *sequential* circuits. Therefore, as with SAT and combinational ATPG today, in the near future, we may see a unification process of SAT and sequential ATPG, promising not only a unified terminology but also substantially more powerful tools.

References

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

[2] S. Akers, "A logic system for fault test generation," *IEEE Trans. on Comp.*, vol. 25, pp. 620–630, June 1976.

[3] F. A. Aloul and K. A. Sakallah, "SAT using ZBDDs," in *Dagstuhl Seminar 01051 on Computer Aided Design and Test – BDDs versus SAT, Schloss Dagstuhl, Germany*, Jan 28-Feb 2 2001.

[4] L. Baptista and J. P. Marques-Silva, "Using randomization and learning to solve hard real-world instances of satisfiability," in *Proc. Intl. Conf. on Principles and Practice of Constraint Programming*, September 2000.

[5] R. Bayardo Jr. and R. Schrag, "Using CSP look-back techniques to solve real-world SAT instances," in *Proc. Natl. Conf. on Artificial Intelligence*, pp. 203–208, 1997.

[6] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[7] F. Brown, *Boolean Reasoning – The Logic of Boolean Equations*. Kluwer Academic Publishers, 1990.

[8] M. Buro and H. Kleine-Buning, "Report on a SAT competition," in *Technical report, University of Paderborn*, 1992.

[9] S. Chakradhar, V. D. Agrawal, and S. Rothweiler, "A transitive closure algorithm for test generation," *IEEE Trans. on CAD*, vol. 12, pp. 1015–1028, July 1993.

[10] L. S. P. Chatalic, "Multi-resolution on compressed set of clauses," in *Proc. SAT 2000, Third Workshop on the Satisfiability Problem*, 2000.

[11] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Communications ACM*, vol. 5, pp. 394–397, July 1962.

[12] M. Davis and H. Putnam, "A computing procedure for quantification theory," *Journal of the Association for Computing Machinery*, vol. 7, pp. 201–215, 1960.

[13] J. W. Freeman, "Improvements to propositional satisfiability search algorithms," in *PhD thesis, University of Pennsylvania*, 1995.

[14] K. Fuchs, F. Fink, and M. Schulz, "Dynamite: An efficient automatic testpattern generation system for path delay faults," *IEEE Trans. on CAD*, vol. 10, pp. 1323–1335, October 1991.

[15] T. Fujino and H. Fujiwara, "An efficient test generation algorithm based on search state dominance," in *Proc. Intl. Symposium on Fault-Tolerant Comp.*, pp. 246–253, 1992.

[16] H. Fujiwara and T. Shimono, "On the acceleration of test generation algorithms," *IEEE Trans. on Comp.*, 1983.

[17] J. Giraldi and M. Bushnell, "Search state equivalence for redundancy identification and test generation," in *Proc. Intl. Test Conference*, pp. 184–193, 1991.

[18] P. Goel, "An implicit enumeration algorithm to generate tests for combinational logic circuits," *IEEE Trans. on Comp.*, vol. C-30, pp. 215–222, March 1981.

[19] C. P. Gomes, B. Selman, and H. Kautz, "Boosting combinatorial search through randomization," in *Proc. Natl. Conf. on Artificial Intelligence*, July 1998.

[20] C. P. Gomes, B. Selman, N. Crato, and H. Kautz, "Heavy-tailed phenomena in satisfiability and constraint satisfaction problems," in *Journal of Automated Reasoning*, 1999.

[21] J. Groote and J. Warners, "The propositional formula checker heerhugo," in *SAT 2000* (I. Gent, H. van Maaren, and T. Walsh, eds.), pp. 261–281, IOS Press, 2000.

[22] I. Hamazaoglu and J. H. Patel, "New techniques for deterministic test pattern generation," *Journal of Electronic Testing*, 1999.

[23] R. Jeroslow and J. Wang, "Solving propositional satisfiability problems," in *Annals of Mathematics and AI*, pp. 167–187, 1990.

[24] T. Kirkland and R. Mercer, "A topological search algorithm for ATPG," in *Proc. Design Automation Conference*, vol. 24, pp. 502–508, 1987.

[25] W. Kunz and D. Pradhan, "Recursive learning: An attractive alternative to the decision tree for test generation in digital circuits," in *Proc. Intl Test Conference*, pp. 816–825, September 1992.

[26] W. Kunz and D. Stoffel, *Reasoning in Boolean Networks - Logic Synthesis and Verification Using Testing Techniques*. Boston: Kluwer Academic Publishers, 1997.

[27] T. Larrabee, "Efficient generation of test patterns using Boolean difference," in *Proc. Intl. Test Conference*, pp. 795–801, 1989.

[28] C.-M. Li and Anbulagan, "Heuristics based on unit propagation for satisfiability problems," in *Proc. 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, pp. 366–371, Morgan Kaufmann Publishers, ISBN 1-55860-480-4, 1997.

[29] S. Mallela and S. Wu, "A sequential circuit test generation system," in *Proc. Intl. Test Conf.*, pp. 57–61, 1985.

[30] R. Marlett, "An effective test generation system for sequential circuits," in *Proc. Design Automation Conf.*, pp. 250–256, 1986.

[31] J. P. Marques-Silva and K. A. Sakallah, "Dynamic search-space pruning techniques in path sensitization," in *Proc. Design Automation Conf.*, pp. 705–711, June 1994.

[32] J. P. Marques-Silva and T. Glass, "Combinational equivalence checking using satisfiability and recursive learning," in *Proc. Design, Automation and Test in Europe Conf.*, pp. 145–149, March 1999.

[33] J. P. Marques-Silva and K. A. Sakallah, "GRASP-A search algorithm for propositional satisfiability," *IEEE Trans. on Comp.*, vol. 48, pp. 506–521, May 1999.

[34] J. P. Marques-Silva, "Algebraic simplification techniques for propositional satisfiability," in *Proc. Intl. Conf. on Principles and Practice of Constraint Programming*, September 2000.

[35] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Engineering an efficient SAT solver," in *Proc. Design Automation Conf.*, 2001.

[36] E. Rich, *Artificial Intelligence*. McGraw-Hill, 1983.

[37] J. P. Roth, "Diagnosis of automata failures: A calculus and a method," *IBM Journal of Research and Development*, vol. 10, pp. 278–291, July 1966.

[38] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 1995.

[39] M. Schulz, E. Trischler, and T. Sarfert, "SOCRATES: A highly efficient automatic test pattern generation system," in *Proc. Intl. Test Conference*, pp. 1016–1026, 1987.

[40] M. Schulz and E. Auth, "Improved deterministic test pattern generation with applications to redundancy identification," *IEEE Trans. on CAD*, vol. 8, pp. 811–816, July 1989.

[41] B. Selman and H. Kautz, "Domain-independent extensions to GSAT: Solving large structured satisfiability problems," in *Proc. Intl. Joint Conf. on Artificial Intelligence*, pp. 290–295, 1993.

[42] G. Stålmarck, "A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula," 1989. Swedish Patent 467 076, US Patent 5 276 897, European Patent 0 403 454.

[43] P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli, "Combinational test generation using satisfiability," *IEEE Trans. on CAD*, 1996.

[44] P. Tafertshofer, A. Ganz, and K. Antreich, "Igraine - an implication graph based engine for fast implication, justification, and propagation," *IEEE Trans. on CAD*, vol. 19, pp. 907–927, August 2000.

[45] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, 1972.

[46] H. Zhang and M. Stickel, "Implementing Davis-Putnam's method," in *Technical Report, The University of Iowa*, 1994.

[47] H. Zhang, "SATO: An efficient propositional prover," in *Proc. Intl. Conf. on Automated Deduction*, pp. 272–275, July 1997.