UNIVERSITY OF Southampton

University of Southampton Research Repository
ePrints Soton

Copyright © and Moral Rights for this thesis are retained by the author and/or other copyright owners. A copy can be downloaded for personal non-commercial research or study, without prior permission or charge. This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the copyright holder/s. The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the copyright holders.

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given e.g.

AUTHOR (year of submission) "Full thesis title", University of Southampton, name of the University School or Department, PhD Thesis, pagination

http://eprints.soton.ac.uk

# ABSTRACT

SEARCH ALGORITHMS FOR SATISFIABILITY PROBLEMS
IN COMBINATIONAL SWITCHING CIRCUITS

by
João Paulo Marques da Silva

Chair: Karem A. Sakallah

A number of tasks in computer-aided analysis of combinational circuits, including test pattern generation, timing analysis, delay fault testing and logic verification, can be viewed as particular formulations of the satisfiability problem (SAT). The first purpose of this dissertation is to describe a configurable search-based algorithm for SAT that can be used for implementing different circuit analysis tools. Several methods for reducing the amount of search are detailed and integrated into a general algorithmic framework for solving SAT. Special emphasis is given to the description of methods for diagnosing the causes of conflicts that may be identified while searching for a solution to each instance of SAT. These methods allow the implementation of non-chronological backtracking, conflict identification based on equivalence relations and logic value assertions derived from conflicts.

Path sensitization in combinational circuits is often used to solve test pattern generation, timing analysis and delay fault testing problems. While path sensitization can be cast as an instance of SAT, such an approach can conceal desirable structural properties of the problem and may lead to exponential size representations. Another purpose of this dissertation is to introduce a new model for path sensitization that permits modeling test pattern generation and timing analysis with linear size representations. In addition, this formulation for path sensitization permits the adaptation of all the pruning methods developed for the general SAT problem.

The proposed SAT algorithms and path sensitization model form an initial kernel for the development of tools for the analysis of combinational circuits. Their practical applicability is supported by experimental results obtained with test pattern generation and timing analysis tools.

# SEARCH ALGORITHMS FOR SATISFIABILITY PROBLEMS IN COMBINATIONAL SWITCHING CIRCUITS

**by**

**João Paulo Marques da Silva**

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Electrical Engineering)
in The University of Michigan
1995

Doctoral Committee:

Associate Professor Karem A. Sakallah, Chair
Professor Richard B. Brown
Professor John P. Hayes
Professor Trevor N. Mudge
Professor Quentin F. Stout

© João Paulo Marques da Silva 1995.
All Rights Reserved

For my family.

# ACKNOWLEDGEMENTS

Many people have helped and encouraged me during my work on this research and dissertation. First and foremost, I would like to thank Prof. Karem Sakallah, my research advisor, for all his support throughout my graduate studies at the University of Michigan. Karem's wisdom and knowledge of the field have inspired and guided me over the years. His reviews, suggestions and constant encouragement are gratefully acknowledged.

I am also grateful to each of the other members of my dissertation committee: Prof. Rich Brown, Prof. John Hayes, Prof. Trevor Mudge and Prof. Quentin Stout. At different stages of my research work, each has been invaluable for their technical advice. In particular, I would like to thank Prof. Hayes for motivating my interest in testing, and Prof. Stout for shaping my knowledge of algorithms.

I also would like to thank all friends and colleagues: Matthew Jones, Mike Riepe, Tim Burks, Ajay Daga, Wei-Han Lien, Krish Chakrabarty, Hakan Yalcin, Mark Hansen, John-David Wellman, Jeff Bell, Vaidyanathan Chandramouli, Chuan-Hua Chang and others who have shared many discussions and technical advice. In particular, I would like to thank Matthew Jones and Ajay Daga for listening and debugging many ideas, Hakan Yalcin for reading early drafts of the dissertation and providing useful comments, and Mike Riepe, Jeff Bell and Vaidyanathan Chandramouli for proofreading a complete draft of the dissertation. I thank Tim Burks and Ayman Kayssi for having supplied the FrameMaker document formats for this dissertation.

I would like to thank the financial support, at different stages of my research work, of the Fulbright Committee and the I.I.E, and the Portuguese 'Junta Nacional the Investigação Científica e Tecnológica'. Their support is gratefully acknowledged.

I would like to thank my family for all their support and encouragement over the years. To my mother, Maria Marques, for all her constant support and love. To my brothers, José and António, for the greatest friendship of all. Finally, I would like to thank my wife, Rosário, for all

her love and understanding, for all her strength in the difficult moments, and for having always encouraged and believed in me; and I would like to thank my son, João Nuno, for the years to come.

# CHAPTER I

# INTRODUCTION

## 1.1 Statement of the Problem

Satisfiability problems are ubiquitous to the computer-aided analysis of combinational switching circuits. The identification of circuit input assignments that satisfy some circuit property can be viewed as a satisfiability problem. The identification of circuit input assignments that satisfy a given circuit output objective is an example of a satisfiability problem. Logic verification is a satisfiability problem. Path sensitization for test pattern generation is a satisfiability problem. Path sensitization for timing analysis is yet another satisfiability problem. Path sensitization for delay fault testing is also a satisfiability problem.

Given this state of affairs, an *efficient*[1] algorithm to solve general satisfiability problems can solve efficiently any of the problems above. The problem is that the satisfiability problem (SAT), in several distinct formulations, is known to be *NP*-complete [34, 65, 90], and it is commonly accepted that any algorithmic solution for solving SAT requires at least worst-case exponential time in the size of each problem instance description. The aforementioned circuit analysis tasks are also known to be algorithmically hard [63, 65, 84, 118] and no known efficient algorithm exists for them. Despite these negative facts, circuit designs have to be validated prior to fabrication and have to be tested after fabrication, and thus acceptable algorithmic solutions must be

[1] By efficient algorithm we mean an algorithm that runs in worst-case polynomial time in the size of the problem instance [65, pp. 6-9]. Conversely an inefficient algorithm runs in worst-case exponential time on the size of each problem instance.

devised. Common algorithmic solutions attempt to be *effective*, and thus perform well on a large number of problem instances, even though the worst-case behavior is still exponential in the size of the problem instance representation.

Two main algorithmic approaches exist for solving satisfiability problems in circuit analysis tasks; search and set of solutions construction. Search can be organized in many different ways, but the most often used is derived from backtracking search algorithms. Construction of sets of solutions entails encoding all solutions to a given problem instance in some effective manner, most often in canonical form. For example binary decision diagrams (BDDs) [3, 22] can be used to encode solutions of different circuit analysis problems. The main drawback of encoding all solutions is that the size of the representation can become exponential in the size of the problem instance representation.

Search is most often the best compromise for solving circuit analysis problems, and is the focus of the present work. Several search algorithms have been developed over the years for solving different circuit analysis tasks. Several search algorithms have also been developed to solve satisfiability problems in different problem domains; for example conjunctive normal form (CNF) satisfiability, constraint satisfaction problems, truth maintenance systems, among others. However, algorithms for circuit analysis tasks have seldom been influenced by algorithmic techniques developed for other application domains. Furthermore, few attempts have been made to reduce the amount of search in algorithms for circuit analysis, by using knowledge from other problem domains and the specific structure of circuit analysis problems.

**The Questions**

The development of search-based algorithms for SAT should attempt to answer several relevant questions. Given that search is based on ordered sequences of decision assignments, can these assignments reveal facts that reduce the amount of search? How can these facts be inferred? Given that conflicts are intrinsic to search, can conflicts provide facts that reduce the amount of search? How can these facts be inferred? Can the structure of instances of satisfiability be used for reducing the amount of search? Can the structure of the search be used to reduce the amount of search?

In the more specific domain of path sensitization, several questions should also be

answered when developing algorithmic solutions. How to model path sensitization? Are different instances of path sensitization somehow related? Can the representation of instances of path sensitization be unified? What insights can such unification provide? Can SAT algorithms be adapted for solving path sensitization?

Finally, from a more practical perspective, a few questions can also be formulated. How to organize search algorithms for SAT and for path sensitization? How to configure those algorithms for different circuit analysis tasks? What design tradeoffs must be considered?

**The Thesis**

In the present dissertation we endeavor to answer the above questions. We show that at several stages of the search useful inferences can be identified which reduce the total amount of search. These inferences can either be the consequence of decision assignments (referred to as *forward reasoning*) or the consequence of understanding the causes of conflicts (referred to as *backward reasoning*). Mechanisms for reducing the amount of search are devised primarily for SAT, but are shown to be applicable to the path sensitization domain.

We propose a model for path sensitization that is independent of any target application, and which can represent different circuit analysis tasks that involve path sensitization. In particular, we describe how fault detection in test pattern generation and circuit delay computation in timing analysis can be represented with the proposed path sensitization model. One key advantage of the model is allowing the search algorithms proposed for SAT to be extended to path sensitization. We describe how these search algorithms can be adapted to different target applications involving path sensitization.

In all cases our thrust is to understand the structure of the problems being solved, and how that structure can be used for reducing the amount of search. In addition, the structure of how the search is conducted is important and can be used to reduce the amount of search. Most important, search-based algorithms for circuit analysis problems have often considered the existence of conflicts as negative and as a potential indication that a solution might not be identified in a reasonable amount of time. Throughout the dissertation we show that conflicts can be helpful. Conflicts can identify useful facts related to the problem instance or the structure of the search. Conflicts *can* reduce the amount of search.

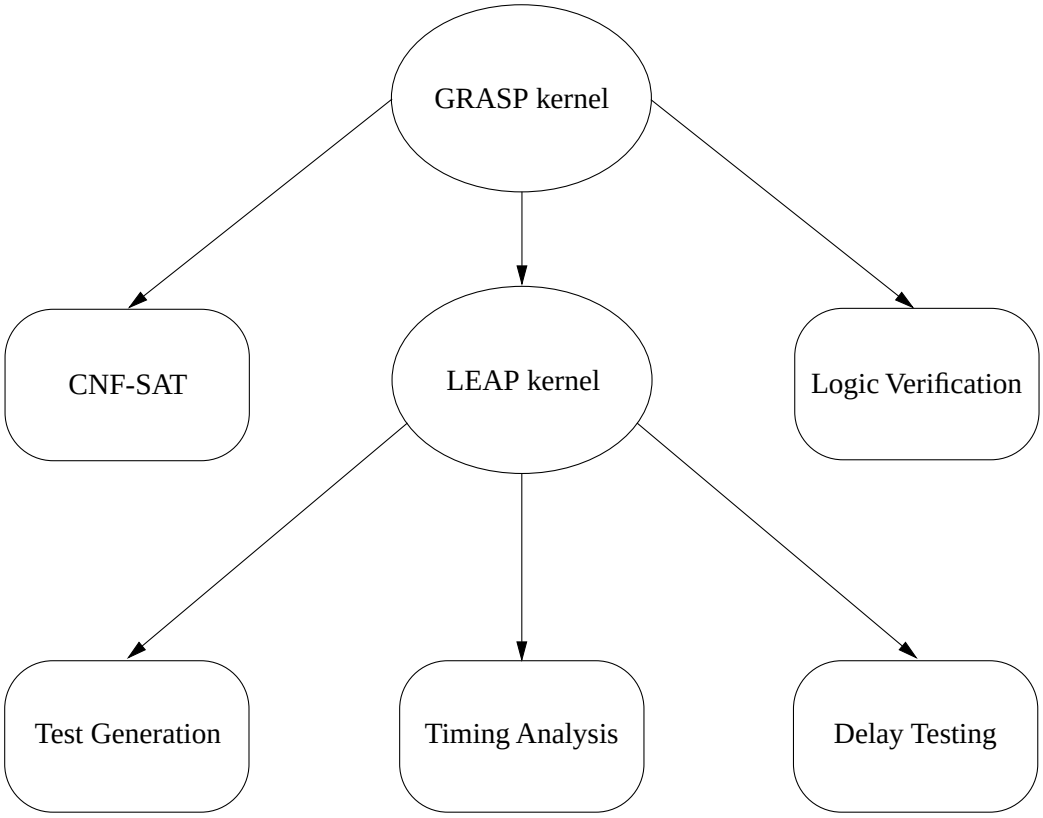



**Figure 1.1:** The GRASP+LEAP toolset

One significant advantage of *diagnosing* conflicts is that no computational effort is spent if no conflicts are identified. For time-critical applications such as test pattern generation or timing analysis, the ability to invest computational resources as needed is significant, and may represent an alternative and better solution over algorithms that spend computational resources trying to avoid conflicts, even in situations where conflicts are not to be found.

**The Practical Applications**

The practical contribution of this dissertation is a toolset for the analysis of combinational switching circuits, referred to as GRASP (General seaRch Algorithm for Satisfiability Problems), and which is depicted in Figure 1.1. For solving path sensitization, GRASP is completed with LEAP (LEvel-dependent Analysis in Path sensitization) which manipulates the information intrinsic to path sensitization. The kernel of GRASP consists of a satisfiability algorithm, which can be used for solving distinct satisfiability problems, e.g. logic verification or satisfiability of conjunctive normal form (CNF) formulas, and which can also be used as a component of LEAP. The ker-

nel of LEAP implements the proposed path sensitization model and is used for solving the problem of path sensitization in different applications.

Different circuit analysis tools can be developed within the GRASP+LEAP toolset. In this dissertation we describe tools for test pattern generation (TG-LEAP) and for timing analysis (TA-LEAP), but the underlying algorithmic framework can be readily extended to other applications.

## 1.2 Search Algorithms

The purpose of the present section is to briefly review definitions associated with search algorithms and strategies for organizing those algorithms. We explicitly assume that the search algorithm is intended to solve some form of satisfiability problem. Other more general formulations can be developed which also include solving optimization problems.

### 1.2.1 Basic Definitions

The process of searching for a solution of a given satisfiability problem is referred to as a *search process* (or *decision procedure*). The search process implements a systematic enumeration of a given search space and, as a result, a *decision tree* (or *search tree*) is maintained. The decision tree accounts for portions of the search space being searched, and implicitly identifies those portions of the search space already searched and those yet to be searched.

A search algorithm for solving a satisfiability problem is said to be *sound* whenever a solution computed for a given problem instance is indeed a solution to that problem instance. A search algorithm is said to be *complete* if it identifies a solution to a problem instance if such a solution exists [169, p. 31].

### 1.2.2 Search Strategies

Different strategies exist for organizing search algorithms. Different organizations lead to different methods for constructing and traversing the decision tree, and to different space complexities.

#### Backtracking

For satisfiability, the most commonly used search strategy is *backtracking*. The backtrack-

```
Backtracking (k)
{
   if (k == n) return SUCCESS;                    // Solution found — return
   S_k = { values of x_{k+1} | P_{k+1}(x_1, ..., x_k, x_{k+1}) is true };
   for (each value y in S_k) {
      set x_{k+1} to y;                           // Define next value of x_{k+1}
      status = Backtracking (k+1);
      if (status == SUCCESS) return SUCCESS;      // Solution found
   }
   return CONFLICT;                               // All values of x_{k+1} tested
}
```

**Figure 1.2:** The backtracking procedure

ing procedure was originally applied to solving several computational problems in the 1950's. The name backtracking is due to R. J. Walker, who first described backtracking in its most general form [175][2].

The following description of backtracking follows, with minor modifications, the one in [97]. Let us assume a problem description with $n$ variables, $x_1$, …, $x_n$, each with several possible values. For each ordered sequence of variables $x_1$, …, $x_k$ a property $P_k(x_1, \ldots, x_k)$, that can assume values true or false, is defined such that:

$$P_{k+1}(x_1, \ldots, x_k, x_{k+1}) \Rightarrow P_k(x_1, \ldots, x_k), \text{ for } 0 \leq k < n \tag{1.1}$$

These predicates are used to define the implementation of the backtracking procedure. The organization of backtracking is shown in Figure 1.2. The procedure recursively extends a set of value assignments to the problem variables. Whenever all variables are assigned and $P_n(x_1, \ldots, x_n)$ holds, then a solution has been identified. Clearly, property $P_n(x_1, \ldots, x_n)$ must be defined to hold true if and only if a solution to the problem has indeed been identified.

**Example 1.1.** In order to illustrate the application of backtracking, we consider the $N$-queens problem [128, 169]. The $N$-queens problem entails the placing of $N$ queens on an $N \times N$ board such that no queen attacks any other queen. One approach for solving the $N$-queens problem is to create

[2] Some authors [11, 74] have attributed the name backtracking to D. H. Lehmer. However, D. H. Lehmer in [107, p. 26], attributes the origin of the name to R. J. Walker.

$N$ variables $x_1, \ldots, x_N$ each taking a value in the set { 1, …, $N$ }, i.e. variable $x_i$ with value $j$ indicates that the queen of the $i^{\text{th}}$ column is placed in the $j^{\text{th}}$ row. Property $P_k(x_1, \ldots, x_k)$ can be defined as follows:

$$P_k(x_1, \ldots, x_k) = \text{“No two of the first } k \text{ queens attack each other”}$$

Consequently, given the definition of the variables $x_1, \ldots, x_N$ and their possible values, invoking `Backtracking`(0) computes a solution to the $N$-queens problem if and only if a solution exists. Predicate $P_k$ is defined informally, but could readily be formalized given the definition of the problem variables and associated values. □

Let us assume a problem instance with $n$ variables, $x_1, \ldots, x_n$ where each variable can take values in domain $D_1, \ldots, D_n$, respectively. Then, the worst-case space required for implementing backtracking (as described in Figure 1.2) is,

$$O(|D_1| + |D_2| + \ldots + |D_n|) \tag{1.2}$$

which is optimal, since all values for each variable must be made available. On the other hand, the worst case time required for finding a solution with backtracking is,

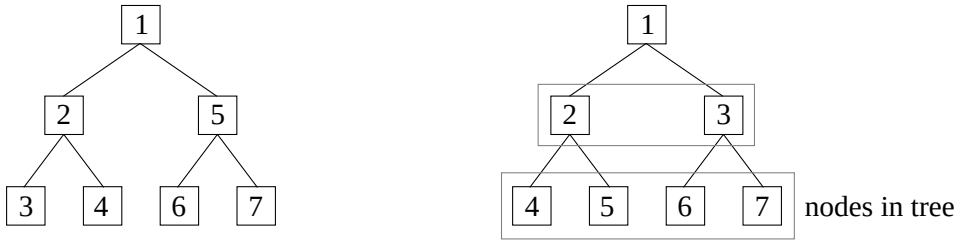
$$O(|D_1| \cdot |D_2| \cdot \ldots \cdot |D_n|) \tag{1.3}$$

For example if $|D_1| = |D_2| = \ldots = |D_n| = 2$, then the worst-case time becomes $O(2^n)$, i.e. a worst-case exponential time procedure.

Backtracking has been extensively studied. D. E. Knuth, in [97], shows that the backtracking procedure is complete and describes a method to estimate the average complexity of computing all solutions to a given problem instance.

**Other Strategies**

*Depth-first* search is a strategy commonly used in artificial intelligence [128]. Its operation is equivalent to backtracking. (J. Pearl [128] claims a minor organization difference between backtracking and depth-first search, related to how at a given stage of the search process the possible extensions are identified. The organization of backtracking in [11, 97] and in Figure 1.2 is equiva-




(a) Traversal order in backtracking (b) Traversal order in breadth-first search

**Figure 1.3:** Difference between backtracking and breadth-first search

lent to the organization of depth-first search in [128] in that all possible extensions are identified prior to entering a new recursion level.)

While depth-first search and backtracking implement a LIFO (last-in-first-out) organization of how nodes in the search tree are visited, *breadth-first* search [128] implements a FIFO (first-in-first-out) organization of how to visit nodes in the search tree. The differences between backtracking and breadth-first search are illustrated in Figure 1.3. While in backtracking the number of active nodes corresponds to a path in the decision tree, in breadth-first search all nodes at the same depth in the decision tree are active. For a satisfiability problem instance with $n$ variables, which requires assignments on all $n$ variables, the space and time complexity of breadth-first search is necessarily exponential in $n$, since all nodes at all levels in the decision tree have to be visited before visiting level $n$ where a solution can be identified.

Other search strategies exist. For satisfiability problems, *iterative deepening* [98] and *iterative broadening* [67] are of hypothetical interest. Iterative deepening exhaustively searches, using depth-first search, increasing depths of the decision tree until a solution is identified, thus guaranteeing that the shortest solution is computed. Iterative broadening imposes a cutoff limit on the number of backtracks to each node in the decision tree and backtracks further when that limit is reached. In order to ensure completeness, the search is executed for increasing values of the cutoff limit until a solution is found. Even though both procedures have been shown to exhibit interesting theoretical properties, their practical application to satisfiability problems is questionable. Iterative deepening enumerates all decision assignments of length less than the length of the computed solution. Iterative broadening becomes impractical if a large number of cutoff limits needs to be considered and it is less useful if the number of branches at each decision node is small.

Search-based algorithms for satisfiability problems are often based on the backtracking search algorithm, since other search strategies can be less effective. Breadth-first requires exponential space, whereas iterative deepening is exponential in the size of the computed solution. Iterative broadening can be useful for problem instances with large variable domains. For the satisfiability problems addressed in this dissertation, variable domains are of size two (i.e. variables assume values in the set { 0, 1 }), and thus the potential advantages of iterative broadening cannot be exploited.

## 1.3 Solving Satisfiability Problems

A satisfiability problem is defined as the task of identifying assignments for variables $x_1$, … , $x_n$ such that a given set of constraints must be satisfied, and where each variable domain is the set { 0, 1 }. A further restriction is to require each constraint to be specified in *clausal* form in which case the satisfiability problem is referred to as SAT [65]. This definition of satisfiability problem corresponds to a restriction of the formulation of constraint satisfaction problems (CSPs) [99, 111, 169], but which is sufficient to capture circuit analysis tasks in combinational switching circuits. By restricting the variable domains, specific algorithmic techniques can be developed, as will be described in the following chapters.

Several algorithmic approaches can be applied to the solution of satisfiability problems. We distinguish search-based, non-search based and algorithms from other problem domains.

### 1.3.1 Using Search-Based SAT Algorithms

The best-known search-based algorithm for SAT is the Davis-Putnam procedure [39, 110][3] which implements a backtracking search procedure completed with several rules for simplifying the CNF formula. Several algorithmic variations of the Davis-Putnam procedure have been proposed in the past [64, 125]. Monien and Speckenmeyer [125] propose a backtracking algorithm, based on the Davis-Putnam procedure, whose main distinct feature is that decision assignments always ensure that a clause is satisfied. Consequently, each node in the decision tree can contain several branches, each of which denotes a solution to a chosen clause. The decision proce-

[3] The Davis-Putnam procedures [38, 110] are studied in Section 2.5.3 and Section 2.5.4.

dure also ensures that redundant sets of assignments are not considered. Gallo and Urbani [64] propose to generate Horn[4] relaxations of a CNF formula, and then conduct the search process in such a way that the sub-formula of Horn clauses is always satisfied. The organization of the algorithm is similar to the Davis-Putnam procedure described in [110], in that the backtracking search strategy is not explicitly enforced.

Test pattern generation algorithms can be used for solving satisfiability problems in combinational circuits. Furthermore, in the context of test pattern generation several SAT algorithms have been proposed in recent years [104, 105, 162], which can be viewed as variations of the Davis-Putnam procedure in terms of how the search is conducted.

### 1.3.2 Using Non-Search Based SAT Algorithms

Several algorithms have been proposed for solving SAT which are not based on systematic search. Recent work has focused on local (non-systematic) search algorithms, with very promising results [76, 77, 146, 147]. In these algorithms the search for a solution consists in iteratively modifying components of an initial complete but invalid assignment in an attempt to eventually identify a valid solution to the problem instance. The major drawback of most local search algorithms is that they are not complete, and thus they cannot be used to prove unsatisfiability. Gu [77] has suggested complete algorithms based on local search and plain backtracking, but does not describe the implementation in detail and gives no experimental results. An example of the application of optimization techniques for solving SAT is the use of Boltzmann machines [4], but these algorithms are also not complete. Ginsberg [68] has recently proposed a new search paradigm where conflict handling methods are integrated in a search algorithm that implements local search; the same approach can be applied to SAT.

Satisfiability problems can be solved with algebraic methods, that are briefly reviewed in Section 2.5.3. Mixed algorithmic solutions also exist. For example, Billionet and Sutter in [10] propose an algorithm based on search, which uses restricted forms of consensus operations for constraining the search process.

Another algorithmic approach is to *estimate* whether a solution exists. Iwama in [86] pro-

[4] A Horn clause contains at most one positive literal (see for example [64, 131]).

poses counting the number of maxterms covered by a given CNF formula; if the Boolean space is covered, then the formula is unsatisfiable. This algorithm does not compute a solution assignment in the case the formula is satisfiable. In addition, counting maxterms involves considering all subsets of clauses not involving the same literals, which in the worst-case can be in exponential number in the number of such subsets. No experimental results are presented in [86]. In a related work, Tanaka [165] provides conditions under which it is possible to test satisfiability. However, the proposed algorithm is not complete, since it may be possible to estimate a formula as satisfiable when it actually is not. As with Iwama's algorithm, Tanaka's will not provide a solution assignment that satisfies the formula.

### 1.3.3 Using Other Problem Domains

Algorithms for solving constraint satisfaction problems (CSPs) can be applied to solving satisfiability problems, which represent a restriction of CSPs. A large body of work has been dedicated to the development of algorithmic techniques for solving CSPs [41, 42, 61, 111, 126, 127, 133, 143, 169]. These techniques can be categorized as consistency methods [111, 126, 169] and as search-based methods [41, 42, 127, 143, 169]. Integrations of the two types of methods has also been reported [127]. However, CSPs are a very general formulation, that is unable to deal effectively with the specific structure of satisfiability problems. Evidence of this fact will become apparent in subsequent chapters.

Satisfiability problems can also be solved with truth maintenance systems (TMSs) [43-46, 54, 60, 69, 115, 116]. A truth maintenance system identifies a general framework for maintaining a knowledge database and interface procedures for testing the validity of the knowledge database given a set of assumptions with respect to objects on the database. The formulation of TMSs and their variations is not directed towards efficiency; in most cases the objective is to complete the knowledge database with information deemed of interest to the application domain. Notwithstanding, in the following chapters we will adapt several concepts commonly used in TMSs, in particular logical TMSs [60, 115], in describing algorithms for satisfiability problems.

Other fields of research have developed algorithms that, with adequate adaptations of the problem representation, can be used for solving satisfiability problems. This is the case, for exam-

ple, of logic programming and constraint logic programming (CLP) [88]. An example of applying constraint logic programming to the satisfiability problem of propositional formulas can be found in [157]. Another example is the application of integer programming methods for solving SAT [12, 82].

Even though satisfiability problems can be solved with algorithms from several different areas, existing experimental results suggest that knowledge intrinsic to each problem domain can be applied in developing application-specific algorithms that perform better than mapping satisfiability problems to more general problem domains. As a result, we conjecture that for time-critical applications such as circuit analysis tasks, the development of algorithmic solutions specific to these applications provides the most effective algorithmic solutions.

## 1.4 Application of Satisfiability Algorithms to Circuit Analysis

The most common satisfiability problem in the analysis of a combinational switching circuit is to answer the query: *identify a logic assignment at the circuit inputs that is consistent with a logic objective (set of logic objectives) at a given circuit output (set of circuit outputs)*. The circuit by itself may already encode the representation of another fairly different satisfiability problem. Other more elaborated satisfiability problems can be formulated in circuit analysis tasks.

Logic verification entails the comparison of two (possibly structural) descriptions of a combinational circuit. Let $E$ and $F$ be two circuit descriptions each with $|PI|$ primary inputs and $|PO|$ primary outputs. (If the number of primary inputs or of primary outputs of $E$ and $F$ were different, then the two circuit descriptions would certainly be different.) Let $n$ identify the common primary inputs to both circuits, and let $o^E$ and $o^F$ respectively denote the primary outputs of $E$ and $F$. In this situation, define the equation:

$$\sum_{i=1}^{|PO|} [o_i^E(n_1, \ldots, n_{|PI|}) \oplus o_i^F(n_1, \ldots, n_{|PI|})] = 1 \tag{1.4}$$

If the above equation is satisfiable, then there exists at least one logic assignment to the primary inputs such that at least one of the outputs of $E$ and $F$ differ. The graphical representation of (1.4) is shown in Figure 1.4, and is referred to as a *miter* in [16]. As a result, we can apply any satisfi-

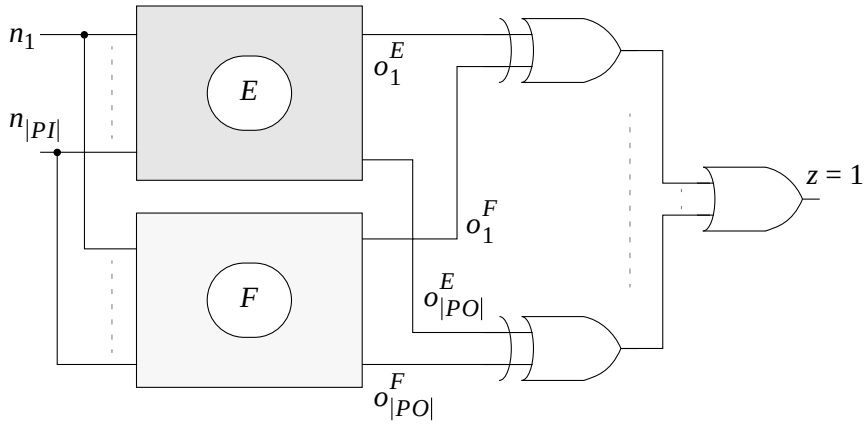



**Figure 1.4:** The logic verification problem

ability algorithm to solve the logic verification problem.

A large body of work has been dedicated to the logic verification of combinational circuits, which involves knowledge from different areas of Computer Science [16, 22, 27, 100, 112, 142, 172, 173, 176]. Currently, practical solutions to the logic verification problem in combinational circuits are either based on BDD representation and manipulation [22, 112], or on algorithms using test pattern generation techniques [16, 100, 142, 176].

In recent years, SAT algorithms have been used for solving the path sensitization problem in test pattern generation [24, 104, 105, 162], timing analysis [119, 120, 150, 152] and delay fault testing [120]. In all cases a set of logical conditions, that captures the conditions of the path sensitization problem, is created and a satisfiability objective is specified. The resulting problem formulation is tested for satisfiability with a dedicated SAT algorithm.

SAT algorithms can be applied to other tasks of the computer-aided analysis and design of digital circuits. For example placement and routing [48] and asynchronous circuit synthesis [136].

## 1.5 Path Sensitization

As mentioned earlier, path sensitization in a combinational circuit entails the identification of circuit input assignments that permit some form of relevant information to be made observable at the circuit outputs. For example, path sensitization can be used to solve the problem of fault detection in test pattern generation and circuit delay computation in timing analysis.

In recent years, extensive research work has been done on developing effective algorith-

mic techniques to solve path sensitization in different application domains. In the following sections we define the goals of path sensitization in two of these applications and briefly review proposed algorithmic solutions.

### 1.5.1 Test Pattern Generation

Test pattern generation concerns the identification of circuit input patterns that permit detecting failures in digital circuits after fabrication. Failures are assumed to be caused by fabrication defects, and circuits subject to failures are said to be *faulty.* Different fabrication defects can occur, which are the source of different types of incorrect behavior. For example, a fabrication defect may cause a circuit node $x$ to always assume the same logic value $v$. Node $x$ is then said to be *stuck-at* $v$ (referred to as $x$ s-a-$v$). If fabrication defects cause two nodes $x$ and $y$ to be connected, then a logic function between the two nodes is defined, which affects other nodes in the transitive fanout of the two nodes. These defects are referred to as *bridging faults* [1, pp. 289-292].

Fault models define which types of improper behavior can be modeled. For circuits described at the gate abstraction level, the *single-stuck fault* (SSF) [1, pp. 110-118] model is the most commonly used and is the model assumed in this dissertation. Experimental evidence suggests that tests computed with the SSF model usually provide good coverage of other types of faults [1, pp. 110-118]. In the SSF model, each node $x$ in a circuit is characterized by two faults, $x$ stuck-at 0 and $x$ stuck-at 1. The interconnection between two nodes $x$ and $y$, $(x, y)$ that denotes the fanout branch between $x$ and $y$, can also be subject to the same two stuck-at faults.

Assuming the SSF model, and for a combinational circuit, the fault detection problem is defined as the identification of circuit input logic values that permit the effect of a given stuck-at fault to be observed at the circuit outputs. If this problem is satisfiable, the circuit input logic values define a *test* $\boldsymbol{T}$ that *detects* the fault. The partial paths along which the effect of the stuck-at fault reaches the circuit outputs are said to be *sensitizable* under $\boldsymbol{T}$. If the problem is not satisfiable, then the fault is said to be *redundant*.

#### The D-Calculus

Algorithmic solutions for fault detection are most often based on the D-calculus [141] or on its algebraic derivations [2, 23, 33, 37]. The D-calculus augments the two-value Boolean alge-

| **NOT** | OUT |
|---|---|
| 0 | 1 |
| 1 | 0 |
| D | $\overline{D}$ |
| $\overline{D}$ | D |
| *X* | *X* |

| **AND** | 0 | 1 | D | $\overline{D}$ | *X* |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | D | $\overline{D}$ | *X* |
| D | 0 | D | D | 0 | *X* |
| $\overline{D}$ | 0 | $\overline{D}$ | 0 | $\overline{D}$ | *X* |
| *X* | 0 | *X* | *X* | *X* | *X* |

| **OR** | 0 | 1 | D | $\overline{D}$ | *X* |
|---|---|---|---|---|---|
| 0 | 0 | 1 | D | $\overline{D}$ | *X* |
| 1 | 1 | 1 | 1 | 1 | 1 |
| D | D | 1 | D | 1 | *X* |
| $\overline{D}$ | $\overline{D}$ | 1 | 1 | $\overline{D}$ | *X* |
| *X* | *X* | 1 | *X* | *X* | *X* |

**Figure 1.5:** Definition of the D-calculus

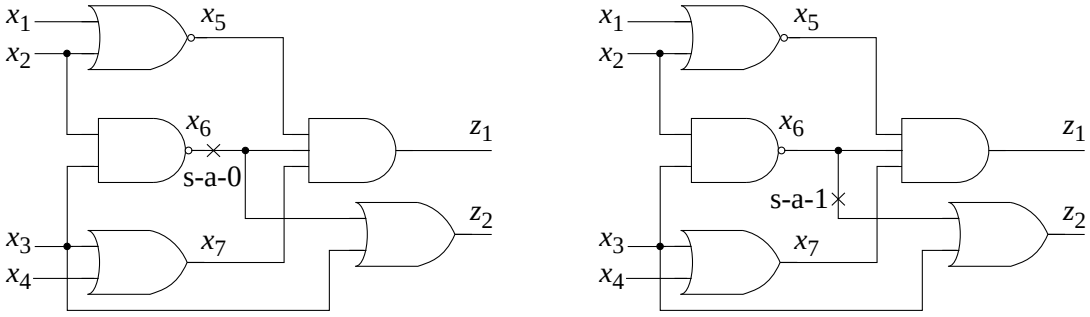



**Figure 1.6:** Example of detectable and redundant faults

bra with two discrepancy values, D and $\overline{D}$, that denote a difference between the logic values of the correct and faulty circuits. If the logic value of a node is $v$ in the correct circuit and $v_f$ in the faulty circuit, then D represents $v/v_f = 1/0$, and $\overline{D}$ represents $v/v_f = 0/1$. The D-calculus has been shown to be valid for the fault detection problem; if the D-calculus is used to model the behavior of a stuck-at fault, then a test $\boldsymbol{T}$ detects that fault if and only if under the D-calculus at least one circuit output assumes value D or $\overline{D}$ [141]. The algebraic definition of the D-calculus is given in Figure 1.5 (where *X* denotes the unassigned value).

**Example 1.2.** Examples of stuck-at faults are shown in Figure 1.6. Each fault needs to be *activated* and so the node associated with the fault must assume value opposite to the value forced by the fault. For fault $x_6$ s-a-0, the test $x_1 = 0$, $x_2 = 0$, $x_3 = 1$ and $x_4 = 0$ enables the effect of the fault to be observed at circuit output $z_1$. For fault ($x_6$, $z_2$) s-a-1 to be activated, it is necessary to have $x_6 = 0$, which implies $x_2 = x_3 = 1$, which then imply $z_1 = 0$ and $z_2 = 1$. Hence the fault cannot be detected and is said to be *redundant*. □

Other approaches for fault detection exist. For example algebraic formulations based on the boolean difference can be used to capture fault detection in test pattern generation. However, it is generally accepted that algebraic formulations for fault detection are not adequate for practical algorithmic implementations.

In recent years, other models for fault detection in test pattern generation have been proposed which formulate the path sensitization problem in terms of a satisfiability problem [24, 104, 162]. For example, the approach of [104] consists in creating XORs between each primary output of the correct and faulty circuits, where the faulty circuit exhibits the effect of the target fault. A test $\boldsymbol{T}$ that detects the fault must set the output of at least one XOR to 1.

**Algorithms for Test Pattern Generation**

The D-algorithm [141] represents one of the first complete algorithms for test pattern generation. It implements a decision procedure that at each state of the search either attempts to propagate an error signal to a circuit output or attempts to justify internal circuit node assignments. For some practical circuits, the organization of the D-algorithm may lead to large decision trees, since a large number of internal nodes may participate in the decision process.

PODEM [72] is the first test pattern generation algorithm to propose implicit enumeration of the circuit inputs as an effective technique to reduce the complexity of the D-algorithm for several problem instances. In addition, the practical implementation can be much simpler than that of the D-algorithm since the search process is restricted to enumeration of the circuit inputs. For example, PODEM does not implement justification, which significantly facilitates the process of backtracking. Decision making in PODEM is guided by simple backtracing [160] that traces objectives through the combinational circuit and decides assignments on the circuit inputs.

FAN [62] proposes several improvements with respect to PODEM. First, the notion of unique sensitization point (USP) is introduced, that identifies logic assignments that constrain the search. In FAN, USPs are only viewed statically (without considering the effects of logic assignments) and preprocessing methods are suggested for their identification. FAN introduces the concept of head line, i.e. the output of a fanout-free subcircuit. Head lines permit reducing the total number of decision variables with respect to PODEM. The authors of FAN observed that the absence of justification affected negatively the overall performance of PODEM, and so FAN

implements justification. Another contribution of FAN is multiple backtracing, that extends simple backtracing by tracing multiple objectives to the circuit inputs and deciding assignments based on the objectives traced.

TOPS [92] and SOCRATES [144, 145] represent evolutions of FAN. TOPS formalizes the notion of unique sensitization points and proposes preprocessing algorithms for their identification. In addition, TOPS extends the notion of head line to basis nodes, i.e. nodes of complete reconvergence. SOCRATES introduces the concept of static [144] and dynamic learning [145], i.e. the identification of non-local implications. In [144] several variations on the identification of static USPs are proposed. Moreover, in [145] the concept of dynamic USPs is introduced (that consider the effects of logic assignments) and a procedure for their identification is described.

EST [70, 71] proposes recording state information associated with conflicts or with identified solutions. This recorded information can then be used to simplify the search for subsequent faults. Information recording takes place at each stage of the search process and consists of a cut of logic values (driving unassigned nodes) as well as a copy of the D-frontier. EST has been used to improve TOPS in [70] and SOCRATES in [71] with promising experimental results.

Other algorithms for test pattern generation have been proposed. QUEST [37] and recursive learning [101] describe methods targeted at hard-to-detect faults, but which may not be practical in practice. The same holds true for TRAN [24], which proposes using transitive closure algorithms for identifying implications. TRAN is directed at hard-to-detect faults, and relies on random test pattern generation and fault simulation for detecting most faults. In TRAN, NEMESIS [104] and TAGUS [162], the test pattern generation problem is formulated as a SAT problem and solved with a SAT algorithm. Experimental results for TAGUS suggest that this approach can be particularly efficient, even though it must resort to different decision making procedures for detecting all faults.

### 1.5.2 Timing Analysis

Timing analysis of digital circuits is concerned with identifying the timing properties of circuits that limit circuit performance. Each circuit component (or interconnect wire) causes signal transitions to be delayed, and so signal transitions are subject to a non-zero propagation delay

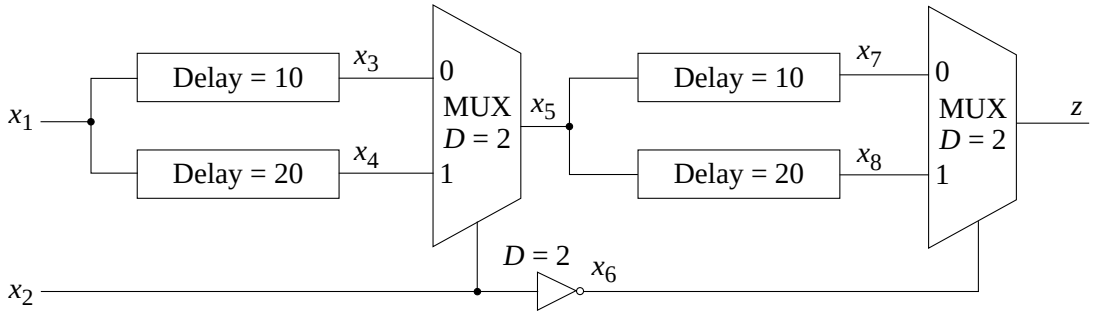



**Figure 1.7:** Example of a circuit with a false path

between the inputs and outputs of the circuit. Propagation delays are defined by the sum of element and interconnect wire delays along paths in the circuit, and the largest propagation delay determines the maximum frequency at which a circuit can operate when it drives memory and synchronization elements.

Early timing analyzers computed circuit delay solely based on topological information, where the longest topological path in a circuit defined the circuit delay [81, 93, 122]. However, there can be paths along which signal transitions cannot propagate under any circuit input assignment. These paths are referred to as *false paths*.

**Example 1.3.** An example of a circuit with a false path (adapted from [9]) is shown in Figure 1.7. A signal transition in $x_1$ can propagate to $z$. However, there are only two propagation paths over which a signal transition can propagate, $\langle x_1, x_4, x_5, x_7, z\rangle$ and $\langle x_1, x_3, x_5, x_8, z\rangle$. Note that $x_6$ assumes the complemented value of $x_2$. Hence, a signal transition propagates from $x_4$ to $x_5$ if $x_2 = 1$, which immediately causes $x_6 = 0$, and so a signal transition can no longer propagate from $x_8$ to $z$. As a result the largest propagation delay from $x_1$ to $z$ is 34 time units and not 44 time units as is defined by the longest topological path delay. □

Most of the early timing analyzers provided mechanisms for either discarding some paths [81] or forcing logical conditions to determine whether a signal transition could propagate along the path [122]. This latter procedure was referred to as *case analysis*. Both solutions for removing false paths have serious drawbacks. First the designer must have extremely good knowledge of how a circuit operates in order to avoid inadvertently removing true paths. Second, case analysis must be applied to all false paths, which may be an extremely large number.

The notion of false path can be traced to [83], where it was first shown that the circuit delay can be less than the delay of the longest topological path[5]. First attempts to identify false paths and compute better estimates of the circuit delay were described in [8, 15] and involve different models of *path sensitization*, i.e. definition of valid conditions for a signal transition to propagate along a path. Since then extensive research work has been done on delay computation for combinational circuits with two main purposes: accurate timing modeling and effective algorithmic procedures.

It is commonly accepted that algorithmic complexity for path sensitization is directly related to the accuracy of the assumed timing model. In the following review of methods for circuit delay computation, we emphasize methods based on simple timing models, since path sensitization is a particularly hard satisfiability problem, even for the most simple timing models.

**Single Path Sensitization**

Early work on path sensitization for circuit delay computation was dedicated to a path-by-path analysis, where each potential longest path was individually tested for sensitization. Different approaches for enumerating paths were proposed. In [55] each path is individually tested using a path extraction algorithm first described in [177]. Other approaches attempt to extend a sensitizable path to a primary output so that the propagation delay is maximized [8, 117]. Strategies for extending partial paths can be based on depth-first or best-first search [128].

The most important aspect of early solutions for circuit delay computation was the assumed timing model. Proposed models range from very precise representations of circuit timing behavior [103, 149] to models targeted to simplify algorithmic implementations [8, 15, 32, 118, 129]. The models describe different criteria for establishing whether a signal transition propagates along a path, and reviews of the different criteria can be found in [32, 118, 151, 153]. In recent years, the most popularized path sensitization criteria assume floating-mode operation, in which the state of each circuit node is assumed to be unknown, though probably going through several signal transitions, before its stabilizes to a known logic value. These criteria include static sensitization [8], viability [117] and the floating-mode criterion [31] and trade off some accuracy on the

5. V. Hrapcenko [83] formalized the notion of a false path. The fact that the circuit delay could be less than the longest topological delay was known before that, and justified the design techniques for high-speed adders.

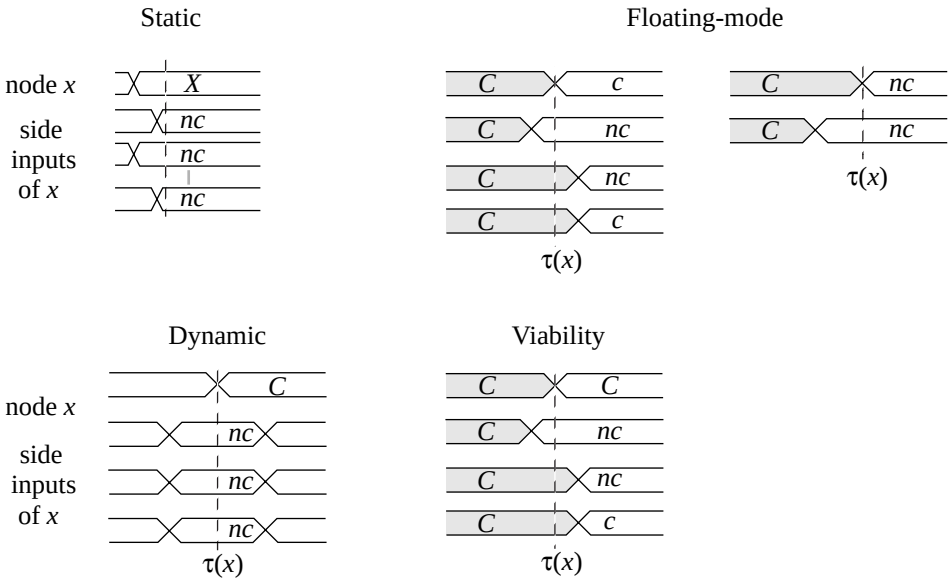



**Figure 1.8:** A characterization of path sensitization criteria

computed circuit delay estimate by being more amenable to algorithmic implementation. While static sensitization can underestimate and overestimate the circuit delay, viability and floating-mode sensitization provide the same upper bound to the circuit delay [31, 32, 153].

A characterization of the different criteria for floating-mode operation is shown in Figure 1.8 (adapted from [153]), and identifies logical and temporal constraints on the side inputs to each node $x$ in a path. $\tau(x)$ denotes the propagation delay of a signal transition to node $x$ along a given path. The side inputs values can either be controlling (*c*) or non-controlling (*nc*). Symbol *C* indicates that a given circuit node value is unknown and can experience changes in time. For floating-mode operation, the primary input stimuli assumes that the initial value of each primary input is unknown and changes to a known logic value at the specified arrival time. In the floating-mode sensitization criterion, a node $y$ in the fanout of a node $x$ stabilizes as a direct consequence of node $x$ stabilizing if $x$ is either the *earliest* controlling value to stabilize or all fanin nodes assume non-controlling values and $x$ is the *latest* node to stabilize. Dynamic sensitization [117, 118, 151, 153] is also included in Figure 1.8, since it computes the exact circuit delay under the assumption of fixed gate delays and one transition (between two known logic values) primary input stimuli.

**Concurrent Path Sensitization**

Single path sensitization faces two complex problems. First, even for a single path, the

validation of the sensitization conditions corresponds to a satisfiability problem. Second, the number of long false paths can be extremely large, and even approaches based on extending partial sensitizable paths are unable to handle circuits with a large number of false paths.

In recent years several procedures have been proposed that are based on *concurrent path sensitization* [30, 50, 120, 152, 156]. A delay range is specified, and the satisfiability problem is then to identify valid conditions for sensitizing any path with delay within the given range. For most practical approaches the delay range considered is $[\Delta, \text{LTP}]$, where LTP denotes the longest topological path delay in a circuit. Concurrent path sensitization was first proposed by S. Devadas et al. in [49].

While for single path sensitization the major emphasis has been the accuracy of the path sensitization criterion, for concurrent path sensitization the emphasis is the algorithmic implementation, and most approaches uniformly assume floating-mode operation. (Other approaches based on more precise timing models [51] use floating-mode operation delay as an initial estimate to a more precise computation of the circuit delay.)

Concurrent path sensitization is characterized by two main algorithmic approaches:

1. *Logic-based,* that consists in defining logical conditions for each node to be part of a sensitizable path with delay no less than $\Delta$ [5, 119, 120, 150, 152]. The search procedure then operates on the established logical conditions, and all approaches are either based on SAT algorithms or test pattern generation algorithms [5].
2. *Delay-based,* that consists in some form of guided timed simulation, based on implicit enumeration of the primary inputs, in which propagation delay estimates to each circuit node are maintained. Different delay estimates can be used, which distinguishes the different algorithmic approaches [29, 30, 50, 52, 156]. (Note that for [156] structural and logical information is maintained, which is used to prune the search. This approach is described in Chapter IV and in Chapter VI.)

**Other Approaches**

The algorithmic solutions for circuit delay computation described above implement some form of search process. Other solutions not based on search have been developed. For example, [7] proposes using ADDs (Algebraic Decision Diagrams). The circuit delay computation problem can

be solved with ADDs by creating an algebraic decision diagram that encodes the logical conditions for each sensitizable path delay. This approach faces two major drawbacks. First, the number of paths delays can be exponential in the size of the circuit. Second, the representation of the logical conditions may also require an exponential size ADD. Results reported in [7] indicate that ADDs work well for some classes of regular circuits (e.g. carry-skip adders) but are considerably inefficient for other more general classes of circuits.

## 1.6 Dissertation Organization

The dissertation is divided into three main parts. In the first part, we lay the foundations for solving SAT with search-based algorithms. An algorithm for solving satisfiability of conjunctive normal form (CNF) formulas, GRASP, is described and compared with other algorithms for CNF SAT. Afterwards, we address models and algorithms for path sensitization and describe their applications. Finally, the third part includes experimental results for circuit analysis tools and delineates future research work.

We start, in Chapter II, by defining a formal framework for representing combinational circuits as CNF formulas. Algebraic methods for solving SAT are reviewed, since they are exploited in later chapters for improving search-based SAT algorithms. Chapter III concludes the first part of this dissertation; it describes GRASP and the main ideas on how to organize search-based SAT algorithms.

Chapter IV begins the second part of the dissertation, and describes a new model for path sensitization, the *perturbation propagation* model. The major objective of Chapter IV is to describe LEAP, a search-based algorithm for path sensitization based on the perturbation propagation model, and which follows the organization of GRASP. Subsequent chapters describe the application of the perturbation propagation model to target applications. Chapter V details the model and algorithm for test pattern generation, and Chapter VI details the model and algorithm for timing analysis.

Experimental results for test pattern generation and timing analysis tools are analyzed in Chapter VII. Chapter VIII concludes the dissertation with an overview of the contributions and a discussion of directions for future research work.

# CHAPTER II

# FUNDAMENTAL CONCEPTS

## 2.1 Introduction

The purpose of this chapter is to introduce the mathematical framework that will be used throughout the remaining chapters. Emphasis is given to the definition and algebraic manipulation of *conjunctive normal form* (CNF) formulas.

We start, in Section 2.2, by defining CNF formulas which provide a unified representation for instances of *satisfiability problems* (SAT). CNF formulas are interchangeably referred to as *clause databases*. We then illustrate, in Section 2.3, how other problem representations of SAT, described either as propositional formulas or as combinational circuits, can be mapped into CNF formulas. These mappings were originally proposed by G. S. Tseitin [170], in the context of propositional formulas, and guarantee CNF formulas of size linear in the size of the original representation.

The next step is to formalize the derivation of *logical implications* of CNF formulas by relating this concept to the *unit clause rule* originally proposed by Davis and Putnam [38]. The iterated application of the unit clause rule is referred to as *Boolean constraint propagation* [116] which is examined in some detail.

Algebraic techniques to manipulate CNF formulas are described in Section 2.5. We review some well-known concepts, e.g. *consensus* and *resolution*, briefly analyze the generation of *prime implicates*, and conclude by studying different algebraic and search-based techniques for solving SAT. These algebraic techniques are shown to be analogous to procedures for generating prime

implicates and some of their simplifications.

The chapter concludes by formalizing how satisfiability tests are executed on CNF formulas. These tests are defined in terms of queries on a clause database and some conditions that characterize valid queries are examined.

## 2.2 Basic Definitions

In this section we define the basic formal framework that will be used to describe satisfiability algorithms in the following chapters. We propose to uniformly represent different instances of satisfiability problems as conjunctive normal form (CNF) formulas. The composition of these formulas can be modified by satisfiability algorithms. Consequently, CNF formulas will be referred to as *clause databases*, over which SAT algorithms can operate.

In the following definitions, and throughout the remainder of this dissertation, the two-element Boolean algebra (also known as switching algebra) is assumed [79, pp. 160-171].

### 2.2.1 Variables and Literals

The definitions introduced in the sequel assume a set of *variables* $\boldsymbol{V}$, each of which is identified by a symbol in the set of symbols $[rstuwxyz]_{[0-9]^*}$. The size of $\boldsymbol{V}$ is interchangeably identified by $|\boldsymbol{V}|$ or $N$.

Each variable $y \in \boldsymbol{V}$ is characterized by a *logic value*, denoted by $\nu(y)$, with $\nu(y) \in \{ 0, 1, X \}$. When clear from the context, $\nu(y)$ is also referred to by $y$. Whenever $y = X$ we say that $y$ is *unassigned*; otherwise $y$ is *assigned*.

**Example 2.1.** An example of a set of variables is $\{ x_1, y_2, t_1 \}$, with $N = 3$. Assuming $x_1 = 1$, then $x_1$ is assigned. $y_2 = X$ indicates that $y_2$ is unassigned. □

The *assignment* of a logic value $v_y \in \{ 0, 1, X \}$ to a variable $y \in \boldsymbol{V}$ is denoted by $y \leftarrow v_y$ and identifies the action of setting $\nu(y)$ to value $v_y$. $y \leftarrow v_y$ is defined as a predicate that always evaluates to 1. Predicate $y = v_y$ evaluates to 1 or 0 depending on whether $\nu(y)$ is equal to or different from $v_y$. It is always true that,

$$(y \leftarrow v_y) \Rightarrow (y = v_y) \tag{2.1}$$

An *assignment set* $\boldsymbol{A}$ is defined as a mapping from $\boldsymbol{U} \subseteq \boldsymbol{V}$ into { 0, 1 } and is always treated as a set $\boldsymbol{A} \subseteq \boldsymbol{V} \times \{ 0, 1 \}$ of variable-value pairs $(y, v_y)$. We say that an assignment set $\boldsymbol{A}$ is *committed* whenever each variable-value pair $(y, v_y) \in \boldsymbol{A}$ denotes an assignment of value $v_y$ to variable $y$; otherwise the assignment set is *uncommitted*. Variables not specified by a committed assignment set $\boldsymbol{A}$ assume $X$ as a default value. A committed assignment set $\boldsymbol{A}$ is said to be a *partial variable assignment* whenever $|\boldsymbol{A}| < |\boldsymbol{V}|$; otherwise the committed assignment set is said to be *complete*. Unless stated otherwise, assignment sets are always assumed to be committed.

**Example 2.2.** Let $y \in \boldsymbol{V}$. $y \leftarrow 0$ indicates that 0 is assigned to $y$. Subsequent to the assignment, proposition $(y = 0)$ holds true. Similarly, let $\boldsymbol{V} = \{ x_1, x_2, x_3, x_4, x_5, z_1 \}$ and $\boldsymbol{A} = \{ (x_1, 0), (x_3, 0), (z_1, 1) \}$. Then, given $\boldsymbol{A}$ the following holds true,

$$(x_1 = 0) \wedge (x_2 = X) \wedge (x_3 = 0) \wedge (x_4 = X) \wedge (x_5 = X) \wedge (z_1 = 1) \qquad \square$$

A *literal* $l$ is defined by $l = y^i$, $y \in \boldsymbol{V}$ and $i \in \{ 0, 1 \}$, where $i = 1$ corresponds to the complemented literal, also referred to as $l = \neg y$, and $i = 0$ corresponds to the uncomplemented literal, also referred to as $l = y$. $y^0$ is said to be a *positive* literal, and $y^1$ is said to be a *negative* literal. Given an partial variable assignment $\boldsymbol{A}$, the value of a literal $y^i$ under $\boldsymbol{A}$ is defined by,

$$y^i\big|_{\boldsymbol{A}} = y \oplus i \tag{2.2}$$

which is equal to $X$ if $(y, v_y) \notin \boldsymbol{A}$.

### 2.2.2 Clauses and CNF Formulas

A *clause* is defined as a disjunction of literals,

$$\omega = \sum_{i=1}^{|\omega|} l_i \tag{2.3}$$

where each $l_i$ is a literal, and the clause contains $|\omega|$ literals. A clause can also be viewed as a set of literals $\omega = \{ l_1, \ldots, l_{|\omega|} \}$ and this representation is used when convenient. Given a partial variable assignment $\boldsymbol{A}$, the value of a clause $\omega$ under $\boldsymbol{A}$ is defined as follows:

$$\omega\big|_{\boldsymbol{A}} = \sum_{i=1}^{|\omega|} l_i\big|_{\boldsymbol{A}} \tag{2.4}$$

If $\omega\big|_{\boldsymbol{A}} = 1$ then $\omega$ is said to be *satisfied*. If $\omega\big|_{\boldsymbol{A}} = 0$ then $\omega$ is said to be *unsatisfied*. Otherwise $\omega$ is said to be *unresolved*. Under a partial variable assignment, the unassigned literals of an unresolved clause $\omega$ are called the *free literals* of $\omega$. An unresolved clause is said to be a *unit clause* whenever it contains only one free literal.

**Example 2.3.** An example of a clause is $\omega = (y + \neg w + z)$. Let us consider the assignment set $\boldsymbol{A} = \{(w, 1), (y, 0)\}$. Then $\omega\big|_{\boldsymbol{A}}$ is given by,

$$\omega\big|_{\boldsymbol{A}} = X \oplus 0 + 1 \oplus 1 + 0 \oplus 0 = X \qquad \square$$

A clause $\omega_1$ is said to *subsume* another clause $\omega_2$ if $\omega_1$ logically implies $\omega_2$ [28]. In particular, $\omega_1 \subseteq \omega_2$ if and only if $\omega_1$ subsumes $\omega_2$.

**Example 2.4.** For example, $\omega_1 = (y + \neg w)$ subsumes $\omega_2 = (y + \neg w + z)$ because whenever $\omega_1$ is satisfied, then $\omega_2$ is also satisfied, and if $\omega_2$ is unsatisfied, then $\omega_1$ is necessarily unsatisfied. Conversely, the two clauses can be represented by $\omega_1 = \{ y, \neg w \}$ and $\omega_2 = \{y, \neg w, z \}$, hence $\omega_1 \subseteq \omega_2$ and so $\omega_1$ subsumes $\omega_2$. $\square$

In general, two clauses $\omega_1$ and $\omega_2$ defined on $\boldsymbol{V}$ may not exhibit a containment relation. In those cases, we can still relate the two clauses. We say that $\omega_1$ is *stronger* than $\omega_2$ if and only if $|\omega_1| < |\omega_2|$, i.e. stronger clauses have more complete variable assignments that unsatisfy them.

A *CNF formula* $\varphi$ is defined as a conjunction of clauses,

$$\varphi = \prod_{j=1}^{|\varphi|} \omega_j \tag{2.5}$$

where each $\omega_j$ is a clause, and the formula contains $|\varphi|$ clauses. The number of literals in $\varphi$ is referred to as the *size of* $\varphi$ and is given by,

$$\|\varphi\| = \sum_{j=1}^{|\varphi|} |\omega_j| \tag{2.6}$$

A formula can also be viewed as a set of clauses $\varphi = \{ \omega_1, \ldots, \omega_{|\varphi|} \}$ and this representation is used when convenient. Given a partial variable assignment $\boldsymbol{A}$, the value of a formula $\varphi$ under $\boldsymbol{A}$ is defined as follows:

$$\varphi|_{\boldsymbol{A}} = \prod_{j=1}^{|\varphi|} \omega_j|_{\boldsymbol{A}} \tag{2.7}$$

If $\varphi|_{\boldsymbol{A}} = 1$ then $\varphi$ is said to be *satisfied*. If $\varphi|_{\boldsymbol{A}} = 0$ then $\varphi$ is said to be *unsatisfied*. Otherwise $\varphi$ is said to be *unresolved*.

**Example 2.5.** An example of a CNF formula is,

$$\varphi = (\neg x + \neg y + z) \cdot (x + \neg z) \cdot (y + \neg z)$$

Let $\boldsymbol{A} = \{ (x, 0), (z, 1) \}$. Then $\varphi|_{\boldsymbol{A}} = 0$, because $(x + \neg z)|_{\boldsymbol{A}} = 0$. □

### 2.2.3 Consistency Functions and Satisfiability

The *consistency function* $\xi$ of a CNF formula $\varphi$ is defined by the values of $\varphi$ given each assignment set $\boldsymbol{A}$ defined in $\boldsymbol{V}$, i.e. $\xi|_{\boldsymbol{A}} = \varphi|_{\boldsymbol{A}}$ for all $\boldsymbol{A}$. As a result, each clause $\omega \in \varphi$ defines an *implicate*[1] of $\xi$. A clause $\omega$ is said to be a *prime implicate* of $\xi$ if and only if no other implicate of $\xi$ subsumes $\omega$. Observe that several CNF formulas can correspond to the same consistency function $\xi$.

In general, a CNF formula $\varphi$ does not necessarily include all the implicates of the associated consistency function $\xi$. A formula that includes all implicates of $\xi$ is referred to a $\hat{\varphi}$. A for-

[1] An *implicate* of a switching function $\xi$ is defined as a disjunction of literals $\omega$ such that $\xi$ implies $\omega$ (see for example [79, p. 288]).

```
// Input arguments:    Initial clause database φ_i
//                     Initial assignments A_i
// Output arguments:   status ∈ { SUCCESS, FAILURE }
// Return values:      Final clause database φ_f
//                     Final assignment set A_f
SAT(φ_i, A_i, &status)
{
   …                                  // Implementation of SAT algorithm
   return (φ_f, A_f);
}
```

**Figure 2.1:** Interface to SAT algorithms

mula that contains all the prime implicates of $\xi$ is said to be a complete product of sums (POS) representation of $\xi$, and is referred to as $\varphi^P$.

In the next few chapters, while studying search algorithms for SAT, we allow implicates of $\xi$ to be identified and used to complete the original CNF formula. As a result, we shall refer to $\varphi$ as a *clause database*. Acceptable transformations on the clause database include adding clauses or sets of clauses, and removing clauses or sets of clauses. The precise meanings of these transformations to the clause database depend on how clauses are added or removed, as is described in the following chapters.

Given a CNF formula $\varphi$, the *satisfiability problem*, denoted by SAT, consists in the identification of an assignment set $\boldsymbol{A}$ such that $\xi|_{\boldsymbol{A}} = 1$. If such an assignment set exists, then $\varphi$ is said to be *satisfiable*, and $\boldsymbol{A}$ is referred to as a *satisfying assignment*. Otherwise, $\varphi$ is said to be *unsatisfiable*. The CNF formula $\varphi$ is said to be *consistent* for any assignment set $\boldsymbol{A}$ such that $\xi|_{\boldsymbol{A}} = 1$. An assignment set $\boldsymbol{A}$ is said to yield a *conflict*, or to make $\varphi$ *inconsistent*, if and only if $\xi|_{\boldsymbol{A}} = 0$, i.e. the value of at least one clause $\omega$ of $\varphi$ under $\boldsymbol{A}$ is 0. SAT has been shown to be *NP*-complete in [34], and it is currently conjectured (and generally accepted) that any algorithm for SAT requires at least worst-case exponential time in the size of the CNF formula [65].

The algorithms for satisfiability described in this dissertation provide the interface specified in Figure 2.1. Given an initial clause database $\varphi_i$ and an initial assignment set $\boldsymbol{A}_i$, the SAT algorithm implements the following computation:

$$(\varphi_f, \boldsymbol{A}_f) \leftarrow \text{SAT}(\varphi_i, \boldsymbol{A}_i, \texttt{status}) \tag{2.8}$$

that yields an updated clause database $\varphi_f$ and a resulting assignment set $\boldsymbol{A}_f$. The above computation can be invoked from other procedures in a program. The output argument *status* can take values FAILURE and SUCCESS, which indicate respectively whether the clause database is unsatisfiable or satisfiable given the initial assignment set $\boldsymbol{A}_i$.

It is important to note that several restrictions of SAT can be solved in polynomial time. This is the case of 2SAT (i.e. SAT restricted to CNF formulas with at most 2 literals per clause) and HSAT (i.e. SAT restricted to Horn clauses[2]). 2SAT can be solved in linear time in the size of the clause database [6, 58]. HSAT was shown to be solvable in polynomial time in [80, 89]. Moreover, several polynomial time algorithms for solving HSAT have been proposed in the past (see for example [53, 64, 80, 124]). The algorithms of [53, 64, 124] have linear time complexity. These restrictions of SAT and associated algorithms have been used to solve the more general SAT problem [64, 105, 106].

## 2.3 Conjunctive Normal Form Representations

In some situations the formulations of instances of SAT are not in conjunctive normal form. Accordingly, a unified treatment of CNF-based satisfiability algorithms must provide methods to represent such instances of SAT as CNF formulas. In this section we discuss how to represent the satisfiability problem of propositional formulas and of combinational switching circuits as CNF formulas.

### 2.3.1 Representation of Propositional Formulas

Let $\boldsymbol{V}$ denote a set of propositional variables. A well-formed propositional formula is defined as follows (adapted from [40, p. 231]):

1. Any propositional variable $x \in \boldsymbol{V}$ is a well-formed formula.
2. If $\rho$ is a well-formed formula, then so is $\neg\rho$.
3. If $\rho$ and $\sigma$ are well-formed formulas, then so are $(\rho \wedge \sigma)$, $(\rho \vee \sigma)$ and $(\rho \leftrightarrow \sigma)$. (With the sole purpose of simplifying the discussion, we disallow logical implication as a valid propositional operator.)

[2] A Horn clause contains at most one positive literal (see for example [64, 131]).

There are several ways to represent a propositional formula in CNF. A straightforward procedure to map a propositional formula $\psi$ into CNF is defined as follows (adapted from [40, pp. 236-237]):

1. Expand logical equivalence operations: $(x \leftrightarrow y) \equiv (\neg x \wedge \neg y) \vee (x \wedge y)$.
2. Repeatedly apply De Morgan's laws to all negation operators involving formulas other than single variables. Remove duplicate negations. After this step, all negations are associated only with propositional variables.
3. Repeatedly apply the distributive law $(l_1 \wedge l_2) \vee (l_3 \wedge l_4) \equiv (l_1 \vee l_3) \wedge (l_1 \vee l_4) \wedge (l_2 \vee l_3) \wedge (l_2 \vee l_4)$. After this step a CNF formula is obtained.

**Example 2.6.** The expression $\psi = [\neg(\neg x \vee y) \wedge \neg(w \vee z) \vee \neg(x \wedge \neg w)]$ is a well-formed propositional formula. Step 1 of the above procedure need not be applied to this formula. The application of step 2 yields the formula $[(x \wedge \neg y \wedge \neg w \wedge \neg z) \vee \neg x \vee w]$. Finally, step 3 computes the CNF formula $[(\neg y \vee \neg x \vee w) \wedge (\neg z \vee \neg x \vee w)]$. □

**Example 2.7.** Consider the propositional formula $\psi = ((\ldots(x_1 \leftrightarrow x_2)\ldots) \leftrightarrow x_n)$. There are $2^{n-1}$ complete variable assignments for which $\psi$ holds true. Expanding $\psi$ results in $2^{n-1}$ clauses that cannot be further simplified, since no two adjacent complete variable assignments to variables $x_1$, …, $x_n$ yield the same propositional value for $\psi$. □

As the last example suggests, the above procedure can produce a CNF formula of exponential size even though the size of the original propositional formula is polynomial in $N$. The solution to this problem was originally proposed by G. Tseitin in 1968 [170]. Let $\psi$ define a propositional formula. Associate a new propositional variable $\eta$ with each subformula contained in $\psi$, such that $\eta$ and the associated subformula of $\psi$ always assume the same propositional value. If $\eta = \rho \wedge \sigma$, then define clauses $(\neg\eta \vee \rho) \wedge (\neg\eta \vee \sigma) \wedge (\eta \vee \neg\rho \vee \neg\sigma)$. If $\eta = \rho \vee \sigma$, then define the clauses $(\eta \vee \neg\rho) \wedge (\eta \vee \neg\sigma) \wedge (\neg\eta \vee \rho \vee \sigma)$. If $\eta = \rho \leftrightarrow \sigma$, then define clauses $(\neg\eta \vee \rho \vee \neg\sigma) \wedge (\neg\eta \vee \neg\rho \vee \sigma) \wedge (\eta \vee \rho \vee \sigma) \wedge (\eta \vee \neg\rho \vee \neg\sigma)$. If $\eta = \neg\rho$, then define clauses $(\neg\eta \vee \neg\rho) \wedge (\eta \vee \rho)$. (Note that in [170], Tseitin describes the representation of logical implication instead of *not* and logical equivalence, but both the *not* and the equivalence operations can be easily derived from logical implication, disjunction and conjunction.) Finally, define clause $\psi$, to test

the satisfiability of $\psi$ to true. The set of defined clauses is satisfiable if and only if propositional formula $\psi$ is satisfiable.

Tseitin's transformation guarantees that the derived CNF formula is linearly related to the size of the original propositional formula. Note, however, that it requires $\boldsymbol{V}$ to include the auxiliary variables created by the transformation.

**Example 2.8.** For the propositional formula of Example 2.6, let $t_1 = \neg x$, $t_2 = \neg w$, $t_3 = (t_1 \vee y)$, $t_4 = \neg t_3$, $t_5 = (w \vee z)$, $t_6 = \neg t_5$, $t_7 = (t_4 \wedge t_6)$, $t_8 = (x \wedge t_2)$, $t_9 = \neg t_8$, and $\psi = (t_7 \vee t_9)$. The resulting CNF formula becomes:

$$
\begin{aligned}
&(\neg t_1 \vee \neg x) \wedge (t_1 \vee x) \wedge (\neg t_2 \vee \neg w) \wedge (t_2 \vee w) \wedge \\
&(t_3 \vee \neg t_1) \wedge (t_3 \vee \neg y) \wedge (\neg t_3 \vee t_1 \vee y) \wedge (\neg t_4 \vee \neg t_3) \wedge (t_4 \vee t_3) \wedge \\
&(t_5 \vee \neg w) \wedge (t_5 \vee \neg z) \wedge (\neg t_5 \vee w \vee z) \wedge (\neg t_6 \vee \neg t_5) \wedge (t_6 \vee t_4) \wedge \\
&(\neg t_7 \vee t_4) \wedge (\neg t_7 \vee t_6) \wedge (t_7 \vee \neg t_4 \vee \neg t_6) \wedge (\neg t_8 \vee x) \wedge (\neg t_8 \vee t_2) \wedge (t_8 \vee \neg x \vee \neg t_2) \wedge \\
&(\neg t_9 \vee \neg t_8) \wedge (t_9 \vee t_8) \wedge (\psi \vee \neg t_7) \wedge (\psi \vee \neg t_9) \wedge (\neg \psi \vee t_7 \vee t_9) \wedge \psi
\end{aligned}
$$

which can be used to test whether $\psi$ can be satisfied to true. Observe that no attempt was made to simplify the resulting CNF formula. For example, all clauses that result from negations could be removed if templates of negated formulas were considered (e.g. $\eta = \neg(\rho \wedge \sigma)$ mapped into $(\eta \vee \rho) \wedge (\eta \vee \sigma) \wedge (\neg\eta \vee \neg\rho \vee \neg\sigma)$). □

**Example 2.9.** For the formula of Example 2.7, we can create a formula $\eta_i$ for each partial $\sigma \leftrightarrow x_i$, i.e. $\eta_i = \sigma \leftrightarrow x_i$, by adding the following clauses: $(\neg\eta_i \vee x_i \vee \neg\sigma) \wedge (\neg\eta_i \vee \neg x_i \vee \sigma) \wedge (\eta_i \vee x_i \vee \sigma) \wedge (\eta_i \vee \neg x_i \vee \neg\sigma)$. Hence, $n - 1$ additional variables are generated, and the resulting CNF formula contains $4 \times (n - 1) + 1$ clauses. Consequently, the size of the CNF formula is linear in the size of the original propositional formula. □

Besides Tseitin's transformation, other polynomial size mappings of propositional formulas into CNF formulas have since been developed (see for example [12, 75, 132]).

### 2.3.2 Representation of Combinational Circuits

#### 2.3.2.1 Combinational Circuits

A well-formed combinational circuit is defined as follows (adapted from [78]):

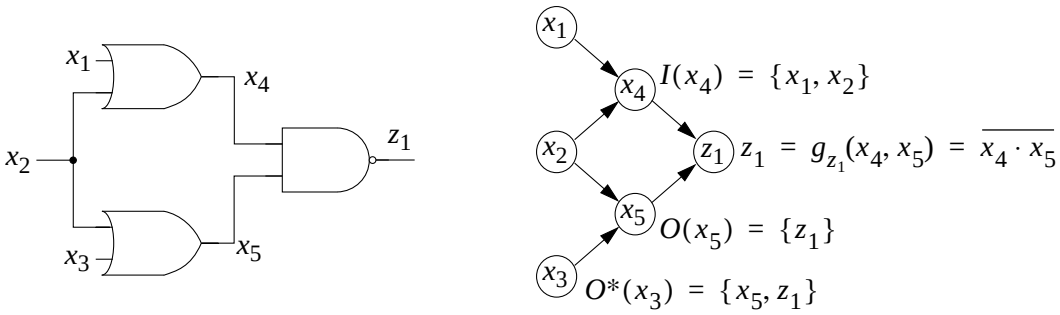
1. A single gate, either simple (i.e. AND, OR, NAND, NOR, NOT, BUFFER) or XOR/XNOR is a well-formed circuit.
2. Let $C_1$ and $C_2$ be two disjoint well-formed circuits. Then their juxtaposition is a well-formed circuit. By connecting a primary output of $C_1$ to a primary input of $C_2$ a well-formed circuit is obtained.
3. If $C_1$ is a well-formed circuit, then by joining input lines of $C_1$ a well-formed circuit is obtained.

In the sequel, a well-formed combinational circuit is referred to as a combinational circuit. Thus, only *acyclic* combinational circuits are considered. Each circuit is characterized by a set *PI* of primary inputs and a set *PO* of primary outputs[3]. The set of circuit nodes is referred to as $\boldsymbol{V}$, as defined in Section 2.2.1. Hence, we may interchangeably refer to the elements of $\boldsymbol{V}$ as either circuit nodes or variables. Whenever $\boldsymbol{V}$ identifies nodes in a circuit, a partial variable assignment is also referred to as a *partial node assignment*. The function of a gate with output $y$, $y \in \boldsymbol{V}$, and inputs $w_1$, …, $w_j$ is denoted by $y = g_y(w_1, \ldots, w_j)$.

Each combinational circuit is represented by a directed acyclic graph $\boldsymbol{C} = (\boldsymbol{V}, \boldsymbol{E})$, referred to as the *circuit graph*, where $\boldsymbol{V}$ is the set of circuit nodes, and $\boldsymbol{E}$, the set of edges, corresponds to the gate input-output connections in the circuit. We further assume that each gate has bounded fanin, and hence $|\boldsymbol{E}| = O(|\boldsymbol{V}|) = O(N)$. Under this assumption, an algorithm with worst-case run time of $O(|\boldsymbol{E}| + |\boldsymbol{V}|)$ is said to run in time linear in the size of $\boldsymbol{V}$, because $O(|\boldsymbol{E}| + |\boldsymbol{V}|) = O(|\boldsymbol{V}|) = O(N)$. For each circuit node $y$, the following definitions apply:

1. $I(y)$ denotes the set of fanin nodes of $y$. For a primary input $y$, $I(y) = \varnothing$. $I^*(y)$ denotes the set of nodes in the transitive fanin of $y$.
2. $O(y)$ denotes the set of fanout nodes of $y$. For a primary output $y$, $O(y) = \varnothing$. $O^*(y)$ denotes

[3] Without loss of generality we assume that each primary output is fanout-free. For the current and following chapters this assumption suffices. More general circuit representations are described in Chapter IV.




(a) Circuit (b) Circuit graph

**Figure 2.2:** Example circuit and associated circuit graph

the set of nodes in the transitive fanout of *y.*

**Example 2.10.** An example of a well-formed combinational circuit and associated circuit graph are shown in Figure 2.2. □

Logic assignments to circuit nodes can be further characterized. A node $y$ is said to be *unjustified* whenever $(y = X) \vee (g_y(I(y) = X)$. $y$ is said to be *justified* whenever $(y \neq X) \wedge (I(y) = \varnothing \vee y = g_y(I(y)))$. Consequently, a justified node $y$ is assigned, and either its value is implied by the value of its fanin nodes or $y$ is a primary input. Predicate *Just*($y$) is defined to hold true if and only if $y$ is justified.

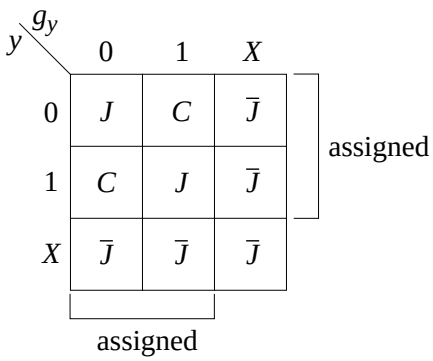
#### 2.3.2.2 CNF Representation

As with propositional formulas, algebraic expansions that correspond to combinational circuits can result in exponential size CNF formulas. Example 2.7 can be adapted to combinational circuits by considering a fanout-free tree of XNOR gates with $n$ primary inputs. Nevertheless, and as with propositional formulas, we can apply Tseitin's transformation to combinational circuits.

Each gate with output $y$, such that $y = g_y(w_1, \ldots, w_j) = g_y(I(y))$ in a combinational circuit is characterized by a *gate consistency function* $\xi_y$ defined over $y$ and $I(y)$,

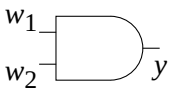
$$\xi_y(I(y), y) \equiv \xi_y(w_1, \ldots, w_j, y) = [\overline{g_y(I(y)) \oplus y}] \tag{2.9}$$

$\xi_y(I(y), y)$ evaluates to $X$ if either $y = X$ or $g_y(I(y)) = X$; otherwise, it evaluates to 1 whenever the




- $\bar{J} = (y = X) + (g_y = X)$
- $J = (y \neq \mathrm{X}) \cdot (g_y = y)$
- $C = (y \neq \mathrm{X}) \cdot (g_y \neq \mathrm{X}) \cdot (g_y \neq y)$

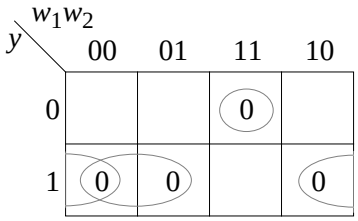
**Figure 2.3:** Value relation between $y$ and $g_y$




(a) Example gate

| $w_1$ | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|
| $w_2$ | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| $y$ | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| $\xi_y$ | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |

(b) Consistency function truth table




(c) POS simplification

**Figure 2.4:** Example of the consistency function of a gate

gate input values are consistent with the gate output value (i.e. $y = g_y(I(y))$), and evaluates to 0 whenever these values are not consistent (i.e. $y \neq g_y(I(y))$). For each simple gate consistency function $\xi_y(I(y), y)$, $\varphi_y$ denotes the CNF formula obtained by product of sums (POS) simplification of the truth table of $\xi_y(I(y), y)$. The different logical relations between each gate output $y$ and $g_y(I(y))$ are shown in Figure 2.3. Whenever the values of $y$ and $g_y$ are specified, the gate is either justified ($J$) or a conflict ($C$) is defined; otherwise the gate is unjustified ($\bar{J}$).

**Example 2.11.** For a two-input AND gate $y = w_1 \cdot w_2$, shown in Figure 2.4, the CNF formula can be obtained from product of sums simplification of $\xi_y$. The resulting expression is given by,

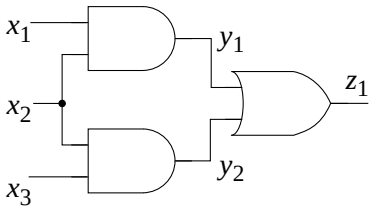
$$\varphi_y = (w_1 + \neg y) \cdot (w_2 + \neg y) \cdot (\neg w_1 + \neg w_2 + y) \tag{2.10}$$

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

**Table 2.1:** CNF formulas of simple gates

The construction of this expression immediately suggests how to extend it to a larger number of gate inputs, without explicitly building the truth table. The same procedure can be used to construct the CNF formulas for other simple gates. Finally, note that the derived CNF formula is isomorphic to Tseitin's transformation in the case of the $\wedge$ operator. □

Gate consistency functions and associated CNF formulas can be easily derived for the remaining simple gates, as well as for more complex gates for which it is computationally feasible to construct and simplify the truth table. For simple gates, the approach used to derive (2.10) can be extended to a larger number of inputs, as shown in Table 2.1. As a result, for simple gates, the size of $\varphi_y$ is linearly related to the number of gate inputs. For AND, NAND, OR and NOR gates with $j$ inputs, the CNF formula requires $j$+1 clauses. For NOT and BUFFER, two clauses define the consistency function. XOR and XNOR gates are represented in terms of the simple gates. For an XOR gate we have the following:




$$\varphi = (x_1 + \neg y_1) \cdot (x_2 + \neg y_1) \cdot (\neg x_1 + \neg x_2 + y_1) \cdot (x_2 + \neg y_2) \cdot (x_3 + \neg y_2) \cdot (\neg x_2 + \neg x_3 + y_2) \cdot (\neg y_1 + z_1) \cdot (\neg y_2 + z_1) \cdot (y_1 + y_2 + \neg z_1)$$

$$\boldsymbol{V} = \{ x_1, x_2, x_3, y_1, y_2, z_1 \}$$

**Figure 2.5:** Consistency function for an example circuit

$$\begin{aligned} y &= \mathrm{XOR}(w_1, w_2) = w_1 \oplus w_2 = \neg w_1 \cdot w_2 + w_1 \cdot \neg w_2 \\ y &= \mathrm{XOR}(w_1, \ldots, w_j) = (\ldots(w_1 \oplus w_2) \oplus \ldots) \oplus w_j \end{aligned} \tag{2.11}$$

and for an XNOR gate it is only necessary to consider the negation of (2.11). The CNF formula of these gates and all other complex gates can be constructed as the CNF formula of a subcircuit. As with propositional formulas, set $\boldsymbol{V}$ must contain any auxiliary variables that are used for representing the internal nodes of a subcircuit.

Given a combinational circuit, described by a circuit graph $\boldsymbol{C} = (\boldsymbol{V}, \boldsymbol{E})$, the CNF formula for the circuit is defined to be the conjunction of the CNF formulas of each of its gates:

$$\varphi = \prod_{y \in \boldsymbol{V}} \varphi_y \tag{2.12}$$

where $\varphi_y$ is the CNF formula associated with each gate output node $y$ in the circuit. (Note that $\varphi$ can also be viewed as the set union of the CNF formulas of each circuit gate.) Consequently, $\varphi$ defines a consistency function $\xi$ for the circuit such that for any assignment $\boldsymbol{A}$, $\xi|_{\boldsymbol{A}} = \varphi|_{\boldsymbol{A}}$.

**Example 2.12.** An example circuit and corresponding CNF formula are shown in Figure 2.5, where for each gate, the CNF formula is adapted from Table 2.1. □

The number of clauses in $\varphi$ is linearly related to the number of circuit nodes. For each simple gate with $j$ inputs, $j$+1 clauses are created. For a 2-input XOR (or XNOR) gate and from (2.11), the CNF formula of a subcircuit representing the gate requires 4 auxiliary nodes (one for each gate in the expansion of the XOR/XNOR gate) for a total of 11 clauses. The CNF formula of a $j$-input XOR/XNOR gate requires $4 \times (j - 1)$ auxiliary variables and $11 \times (j - 1)$ clauses. Since

bounded fanin is assumed, each gate contributes a constant number of clauses, and thus $|\varphi| = O(N)$. Moreover, under the bounded fanin assumption, in the worst-case there are $O(N)$ literals contained in the clauses of $\varphi$.

In some cases the CNF formula for a circuit can contain at most three literals per clause and the associated satisfiability problem is then referred to as 3-SAT. In such a situation, each simple gate with $k$ inputs is replaced by a subcircuit of $k - 1$ two-input gates of the same type. The resulting CNF formula for each gate contains at most three literals per clause, and so does the CNF formula of the circuit. This same solution holds for circuits containing complex gates, where each gate expansion must then be in terms of two-input simple gates.

It is interesting to note that the above construction of the CNF formula of a circuit is based on Tseitin's transformation for propositional formulas, but where the creation of auxiliary variables is unnecessary since we have access to each gate output. This transformation was rediscovered in recent years by T. Larrabee in [106], in an application of SAT algorithms to test pattern generation in combinational circuits. In [24] Chakradhar et al. proposed the representation of the *false function* of a circuit, which corresponds to the complement of the consistency function, also in the context of test pattern generation. Independently, in [35, pp. 940-945], the transformation of a combinational circuit into a product of clauses is also described, with the objective of proving polynomial-time reducibility of the circuit satisfiability problem into SAT.

## 2.4 Implications

Besides identifying conflicts and consistent assignments, CNF formulas (henceforth referred to as clause databases) provide one possible formal framework for the definition of *logical implications*. For example, in the clause database of the AND gate of Figure 2.4, suppose that $y = 1$; then, for any consistent assignment, $w_1$ must assume value 1 due to $(w_1 + \neg y)$ and $w_2$ must assume value 1 due to $(w_2 + \neg y)$, since otherwise the consistency function of the AND gate would evaluate to 0 and a conflict would be identified. Hence, we say that $y = 1$ implies the assignments $w_1 \leftarrow 1$ and $w_2 \leftarrow 1$. In general, given a unit clause $(l_1 + \ldots + l_k)$ of $\varphi$ with unique free literal $l_i$, consistency requires $l_i \leftarrow 1$, since this represents the only possibility for the clause to be satisfied. If $x$ is the variable associated with literal $l_i$, and $l_i$ is a positive literal of a variable $x$ (i.e. $l_i = x$), then

the value of $x$ must be set to 1; otherwise if $l_i$ is a negative literal of a variable $x$ (i.e. $l_i = \neg x$), then the value of $x$ must be set to 0. The assignment of $x$, caused by the requirement to satisfy a clause of the clause database, is referred to as a *logical implication* of $x$, and we say that the assignment of $x$ is *implied*.

By formulating implications as the required assignments to satisfy unit clauses, we relate the concept of logical implication (commonly used in computer-aided analysis of combinational circuits [1, p. 187]) to the *unit clause rule* (or *one-literal clause rule*) of the procedure for solving propositional satisfiability proposed by M. Davis and H. Putnam in 1960 [38]. The unit clause rule states that if there exists only one literal assignment that can satisfy a clause, then that assignment must be made[4]. The iterated application of the unit clause rule to a CNF formula is often referred to as *propositional* (or *Boolean*) *constraint propagation* [46, 115, 116, 178]. In view of the previous discussion, the derivation of implications in a combinational switching circuit will be referred to as Boolean constraint propagation (BCP) throughout this dissertation.

The pseudo-code description of BCP is shown in Figure 2.6. An original assignment set $\boldsymbol{A}_i$ is assumed, which may imply other assignments due to the unit clause rule. A variable, *status*, identifies the existence of a conflict. The procedure returns $\boldsymbol{A}_f$ as the resulting assignment set, being invoked as $\boldsymbol{A}_f \leftarrow$ BCP($\boldsymbol{A}_i$, *status*).

**Theorem 2.1.** With $\boldsymbol{A}_i$ and $\boldsymbol{A}_f$ defined above, the following holds true: (1) $\boldsymbol{A}_i \subseteq \boldsymbol{A}_f$; and (2) each pair $(x, v) \in \boldsymbol{A}_f$ identifies a necessary assignment for the clause database to be satisfiable given $\boldsymbol{A}_i$.

**Proof:** The first claim must hold, since BCP() only identifies assignments for unassigned nodes. For proving the second claim, we assume that there exists a consistent assignment set $\boldsymbol{A}$, such that $\boldsymbol{A}_i \subseteq \boldsymbol{A}$, and show that then we must have $\boldsymbol{A}_i \subseteq \boldsymbol{A}_f \subseteq \boldsymbol{A}$. We use induction on the number of assigned nodes.

*Basis step* ($k = 1$). The first implied assignment results from a unit clause where all literals but one are assigned due to $\boldsymbol{A}_i$. Hence, the assignment is necessary for $\boldsymbol{A}_i$ to be included in a consistent assignment $\boldsymbol{A}$.

---

4. In the original formulation [38], the application of the unit clause rule is defined in terms of algebraic operations on the clause database, but the final result is equivalent to assigning a literal to 1 (see for example [12, 64, 178] for a similar conclusion).

```
// Input arguments:    The initial assignment set A_i
// Output arguments:   status ∈ { SUCCESS, CONFLICT }
// Return values:      The final assignment set A_f
//
BCP (A_i, &status)
{
   status = SUCCESS;
   A_f ← A_i;                              // Initialize final assignment set
   commit assignment A_f;  // Set initial partial variable assignment
   while (clauses unsatisfied or unit clauses in φ) {
      if (exists unsatisfied clause ω) {
         status = CONFLICT;
         return A_f;
      }
      if (exists unit clause ω with free literal l = x^i) {
         x ← 1 ⊕ i ≡ ¬i ;                     // Assignment: l is set to 1
         A_f ← A_f ∪ { (x, ¬i) };             // Update final assignment set
      }
   }
   return A_f;
}
```

**Figure 2.6:** Boolean constraint propagation

*Induction Hypothesis* ($k = m$). Assume that the first $m$ implied assignments are necessary for $\boldsymbol{A}_i \subseteq \boldsymbol{A}$.

*Induction step* ($k = m + 1$). The $(m + 1)^{\text{th}}$ assignment is implied due to a unit clause where all literals but one are assigned due to either $\boldsymbol{A}_i$ or one or more of the first $m$ implied assignments. Since by hypothesis all these $m$ assignments are necessary for $\boldsymbol{A}_i \subseteq \boldsymbol{A}$, then the $(m + 1)^{\text{th}}$ assignment is also necessary.

We can thus conclude that all implied assignments are necessary for $\boldsymbol{A}_i \subseteq \boldsymbol{A}$, and so we must have $\boldsymbol{A}_f \subseteq \boldsymbol{A}$. ■

Let the assignment of a variable $x$ be implied due to a clause $\omega = (l_1 + \ldots + l_k)$. In such a situation, the *antecedent assignment* of $x$, referred to as $\boldsymbol{A}(x)$, is defined as the set of assignments of variables other than $x$ with literals in $\omega$:

$$\boldsymbol{A}(x) = \{(y, v(y)) | y^i \in \omega \wedge y \neq x\} \tag{2.13}$$

Given the definition of antecedent assignment, the *antecedent set* of the assignment of $x$, referred to as $\alpha(x)$, is defined as the set of variables, other than $x$, that are associated with the literals of $\omega$:

$$\alpha(x) = \{y | (y, v_y) \in \boldsymbol{A}(x)\} \tag{2.14}$$

where each variable in $\alpha(x)$ is referred to as an *antecedent* of $x$. Intuitively, the antecedent set of $x$ denotes a set of nodes whose logic values are directly responsible for implying the assignment of $x$.

**Example 2.13.** Consider a clause $\omega = (w + \neg y + \neg x)$ associated with a clause database, and let $w = 0$ and $y = 1$. With these assignments $\omega$ is a unit clause. This implies $x \leftarrow 0$, the antecedent assignment of $x$ is given by $\boldsymbol{A}(x) = \{ (w, 0), (y, 1) \}$ and the antecedent set of $x$ is defined by $\alpha(x) = \{ w, y \}$. □

Given the definitions of implication of variable assignment, antecedent assignment and antecedent set we have the following result:

**Theorem 2.2.** Let the logic value of $x$ be $v_x \in \{0, 1\}$. Let $\alpha(x)$ be the antecedent set of $x$ and let $\boldsymbol{A}(x)$ be the antecedent assignment of $x$. In such a situation, for any assignment set $\boldsymbol{A}$ such that $\boldsymbol{A}(x) \subseteq \boldsymbol{A}$, either the partial variable assignment $\boldsymbol{A}$ implies $x \leftarrow v_x$ or a conflict is identified.

**Proof:** Immediate from the definitions of implication of variable assignment, antecedent assignment and antecedent set. Consider a partial variable assignment $\boldsymbol{A}$, with $\boldsymbol{A} \supseteq \boldsymbol{A}(x)$. By hypothesis, the antecedent assignment of $x$ is $\boldsymbol{A}(x)$, and thus there exists a clause $\omega$ in $\varphi$ for which the assignments denoted by in $\boldsymbol{A}(x)$ imply $x \leftarrow v_x$. If, on the other hand, some other clause in $\varphi$ implies $x \leftarrow \overline{v_x}$, then $\omega$ becomes unsatisfied because of the value assigned to $x$ and of the assignments identified by $\boldsymbol{A}(x)$. Hence, a conflict is identified. ■

As mentioned earlier, clause databases do not necessarily contain all implicates of the associated consistency function. In general, it is not practical for a clause database to contain all

implicates, since that can be exponential in the number of variables. Consequently, the structure of most clause databases does not reveal *all* logical relations among their variables. In some situations, however, it may be of interest to augment the initial clause database with other implicates of the consistency function, to help identify some of these logical relations.

**Example 2.14.** Consider the example circuit of Figure 2.5 on page 36. If $z_1$ assumes value 1, then $x_2$ cannot assume value 0, and hence it must assume value 1. Assuming that $z_1$ is the only assigned node, the requirement that $x_2$ be assigned value 1 is not identified by Boolean constraint propagation over the clauses of the given clause database. The assignment $z_1 \leftarrow 1$ satisfies clauses $(\neg y_1 + z_1)$ and $(\neg y_2 + z_1)$, but implies no additional assignments. We note, however, that for any complete node assignment, if $z_1$ assumes value 1, then $x_2$ must also assume value 1 for the consistency function to evaluate to 1. As a result, the set of clauses describing the consistency function can be augmented with clause $(\neg z_1 + x_2)$. □

As the example illustrates, augmenting the set of implicates of the consistency function facilitates the identification of implications and of conflicts given certain partial variable assignments. Because BCP does not identify all logical consequences of an assignment, it is said to be *logically incomplete* [116]. Conversely, a procedure for identifying implications is said to be *logically complete* if it identifies all logical consequences of an assignment. In general, it is not feasible to expect an implication procedure to be logically complete, since any known algorithm for derivation of all logical consequences would require an exponential amount of work in the worst-case [116]. Other procedures, more complex than BCP, can be devised, examples of which are described in the following chapter and which identify no fewer implications than BCP. Even though BCP is logically incomplete in most clause databases, we show below that it is logically complete if the clause database is defined by all prime implicates of $\xi$ (i.e. $\varphi^P$). In general, we say that a node assignment is *derivable* if the considered implication procedure can imply that assignment through some sequence of implied assignments over a fixed clause database.

In the development of SAT algorithms, we will consider identifying implicates of the consistency function and adding them to the clause database. As mentioned above, these implicates help BCP in identifying more implications. In addition, these implicates are of key importance in

implementing several features of search-based SAT algorithms.

## 2.5 Algebraic Background

In this section we review a few concepts commonly used in Boolean algebra [18] and, with different names, in mechanical theorem proving [28, 110], and highlight some of their applications. These concepts are used throughout the description of search-based satisfiability algorithms.

### 2.5.1 Consensus and Ground Resolvent

Consider clauses $\omega_1$, $\omega'_1$, $\omega_2$, $\omega'_2$, such that there exists a literal $l_x$ associated with a variable $x$ (i.e. either $l_x = x$ or $l_x = \neg x$) and,

$$[\omega_1 = (\omega'_1 + l_x)] \wedge [\omega_2 = (\omega'_2 + \neg l_x)] \tag{2.15}$$

In this situation, the *consensus* [138][5] of the two clauses $\omega_1$ and $\omega_2$, with respect to variable $x$ is given by:

$$c(\omega_1, \omega_2, x) = \omega'_1 + \omega'_2 \tag{2.16}$$

In the mechanical theorem proving literature [28, 110, 140], the consensus of two clauses is commonly referred to as the *ground resolvent* (or *ground resolution operator*) of the two clauses. The ground resolution operator can be generalized over formulas of First Order Logic, being then referred to as the *resolution operator.* Consensus is commonly used to derive prime implicates (or implicants) of Boolean functions, whereas resolution and its variations constitute a fundamental

[5.] Although the name consensus is apparently due to Quine [138], the operation finds its roots in Mathematical Logic, where it is sometimes called *hypothetical syllogism* (see for example [94, p. 60]): $[(A \rightarrow B) \wedge (B \rightarrow C)] \Rightarrow [(A \rightarrow C)]$ , that indicates that if a propositional symbol $A$ implies $B$ and $B$ implies $C$ then we can conclude that $A$ implies $C$. An equivalent form for this expression is $[(\neg A \vee B) \wedge (\neg B \vee C)] \Rightarrow [(\neg A \vee C)]$ , which illustrates the relationship with consensus. According to Brown [18], the consensus operation had previously been used by Blake in [13], for Boolean function simplification, under the name *syllogistic result*. Kneale and Kneale [95, pp. 105-110] indicate that hypothetical syllogisms were most likely discovered by a Greek Philosopher, Theophrastus, a pupil of Aristotle, around 300 B.C., as a direct consequence of Aristotle's work on Logic.

component of theorem proving algorithms.

From (2.16) we can also conclude that $\omega_1$ and $\omega_2$ logically imply $c(\omega_1, \omega_2, x)$, i.e. if $\omega_1$ and $\omega_2$ are true, then $c(\omega_1, \omega_2, x)$ is also true, and if $c(\omega_1, \omega_2, x)$ is false, then either $\omega_1$ is false or $\omega_2$ is false.

We should note that the unit clause rule introduced by Davis and Putnam [38] (see Section 2.4) is a special case of consensus/resolution, being also referred to as *unit resolution* [10, 64]. Without loss of generality, let $[\omega_1 = (\omega'_1 + l_x)] \wedge [\omega_2 = (\neg l_x)]$ , with $l_x$ defined above. In such a situation, (2.16) is given by $c(\omega_1, \omega_2, x) = \omega'_1$, which is to say that if $\omega_1$ and $\omega_2$ are satisfied, then $\omega'_1$ must be satisfied. Applying consensus between every clause containing literal $l_x$ and $\omega_2$ is therefore equivalent to setting literal $l_x$ to 0 in every clause containing $l_x$, thus satisfying $\{ \neg l_x \}$.

### 2.5.2 Generation of Prime Implicates

Let us assume a Boolean function $\xi$ described by a CNF formula $\varphi$. As mentioned earlier, each clause of $\varphi$ identifies an implicate of $\xi$. Following the work of Quine [137, 138, 139], the iterated application of consensus operations to $\varphi$ yields the set of prime implicates of $\xi$ [138]. Subsumption operations (see Section 2.2.2) ensure that non-prime implicates are removed from the clause database. Given a consistency function $\xi$ defined over $N$ variables and represented with $|\varphi|$ implicates, then from [25] an upper bound on the number of prime implicates is $O(min(3^N, 2^{|\varphi|}))$, and hence an upper bound on the size of final clause database is $O(N \cdot min(3^N, 2^{|\varphi|}))$. We further assume the improvement proposed by Tison [168] in which the variables are ordered, and consensus operations are successively applied with respect to each variable. The algorithm for the generation of prime implicates given $\varphi$ is shown in Figure 2.7. An order of the variables is assumed. The algorithm consists of a sequence of subsumption and consensus operations. The consensus operation is described in Figure 2.8. For each clause $\omega$ containing literal $x$, the consensus of $\omega$ with respect to every clause containing literal $\neg x$ is computed. The definition of sets $C_0(x)$ and $C_1(x)$ implicitly capture the operation of consensus. The process is iterated for all clauses $\omega$ containing literal $x$. The temporary formula $\varphi(x)$ contains all the resulting consensus clauses.

```
// Input arguments:    Clause database φ
// Output arguments:   None
// Return values:      Clause database φ^P
//
Generate_Prime_Implicates(φ)
{
   Order variables to be resolved;
   φ ← Subsume(φ);                          // remove subsumed clauses
   while (x is next variable in order to be resolved) {
      φ ← Consensus(φ, x);                  // consensus over x
      φ ← Subsume(φ);
   }
   return φ;                                // return φ as φ^P
}
```

**Figure 2.7:** Algorithm for the generation of prime implicates

```
// Input arguments:    Clause database φ, consensus variable x
// Output arguments:   None
// Return values:      Consensus clause database φ ∪ φ(x)
//
Consensus(φ, x)
{
    C_0(x) = {ω − {x^0} | ω ∈ φ ∧ x^0 ∈ ω};      // Remove x from clauses
    C_1(x) = {ω − {x^1} | ω ∈ φ ∧ x^1 ∈ ω};      // Remove ¬x from clauses
    φ(x) = {ω_0 ∪ ω_1 | ω_0 ∈ C_0(x), ω_1 ∈ C_1(x)};   // Pairwise consensus
   return φ ∪ φ(x);
}
```

**Figure 2.8:** Consensus operation with respect to variable $x$

A simple implementation of the subsumption procedure (in Figure 2.7) is to compare each clause $\omega_1$ with each other clause $\omega_2$ of $\varphi$, and test whether $\omega_1$ subsumes or is subsumed by clause $\omega_2$[6]. The generation of the prime implicates of a given function $\xi$ (associated with a clause database $\varphi$) computes a new clause database $\varphi^P$.

In order to derive time and space complexity bounds we note that at any stage of procedure `Generate_Prime_Implicates()` there can never be more than $3^N$ clauses, because of

[6] A more efficient subsumption procedure can be found in [47], which is based on representing clauses in a *trie* data structure [96, pp. 481-499], to reduce the average time to decide subsumption relations and to remove subsumed clauses.

subsumption being applied. Consequently, at every stage the number of clauses is $O(3^N)$, each of which is size $O(N)$. Subsumption operations require comparing each clause with each other clause, and hence requires in the worst-case $O(N \cdot (3^N)^2)$ time, given that each comparison takes in the worst-case $O(N)$ time. $O(N \cdot (3^N)^2)$ is also an upper bound on time required by the consensus operation, because, in the worst case, consensus may be required between $O(3^N)$ clauses and another $O(3^N)$ clauses, and consensus between two clauses requires in the worst-case $O(N)$ time. Since we have to iterate over $N$ variables, then an upper bound on run time of the algorithm of Figure 2.7 is:

$$O(N^2 \cdot 3^{2N}) \tag{2.17}$$

It is important to note that procedure `Generate_Prime_Implicates()` converts a clause database $\varphi$ into a canonical form of the associated function $\xi$, because the set of all prime implicates of a Boolean function is unique. Similarly, the disjunction of all prime implicants is also a canonical form and is referred to as the *Blake Canonical Form* [18, pp. 71-86]. Furthermore, we note that there are more recent and efficient algorithms for computing the prime implicants/ implicates of a Boolean function (see for example [36, 121, 164]). However, for the purposes of the present dissertation, the basic algorithms described above suffice.

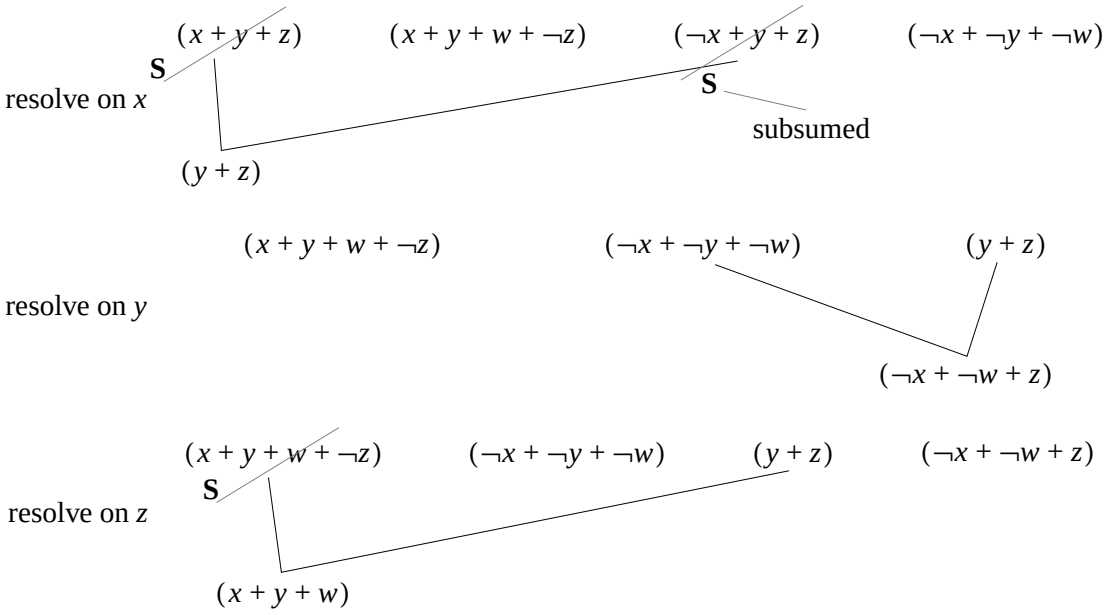
**Example 2.15.** An example of the application of the algorithm for prime implicate generation is shown in Figure 2.9, where the graphical notation proposed by Tison [168] is used. The objective is to compute the set of prime implicates of the switching function:

$$f(x, y, z, w) = (x + y + z) \cdot (x + y + w + \neg z) \cdot (\neg x + y + z) \cdot (\neg x + \neg y + \neg w)$$

At each iteration only the non-tautologous resulting clauses are represented. For example, the consensus of $(x + y + z)$ with $(\neg x + \neg y + \neg w)$, with respect to variable $x$, is a tautology (i.e. the resulting clause is 1) and so it is not represented. For this example, we note that, after resolving on $x$, $y$ and $z$, resolving on $w$ introduces no new prime implicates. □

### 2.5.3 Algebraic Solutions for SAT

The same principle that is used to generate the prime implicates of Boolean functions is




$$f(x, y, z, w) = (y + z) \cdot (\neg x + \neg y + \neg w) \cdot (x + y + w) \cdot (\neg x + \neg w + z)$$

**Figure 2.9:** Example of identification of prime implicates

used in theorem proving while establishing the validity of formulas of first-order logic [28, 110, 140]. For propositional CNF formulas, this process is referred to as *ground resolution*.

Different forms of resolution have been proposed [28, 110]. *Saturation resolution* [110, p. 66] for the propositional calculus is equivalent to Tison's algorithm for deriving prime implicates [168], but without subsumption operations. Consequently, the algorithm of Figure 2.7 can be used to test the satisfiability of a CNF formula. Let $\varphi^P$ be the clause database in which each clause is a prime implicate of $\xi$. Suppose that clause $\omega = \varnothing$ is derived, i.e. $\omega$ is identically false. Then $\omega$ is the only clause in $\varphi^P$, since it vacuously subsumes all other clauses, and $\xi$ is identically 0. Otherwise, $\xi$ is shown to be satisfiable, and we only need to provide a mechanism to recreate an assignment after the resolution steps are completed. As we show next, such a mechanism runs in linear time in the size of $\varphi^P$, which may nevertheless be exponential in the number of variables.

Suppose that the first assignment $y = v_y$ is chosen. Since $\varphi^P$ contains all the prime implicates of $\xi$, then either $y = v_y$ sets $\xi$ to 0, or there is at least one combination of values of the remaining variables that can satisfy $\xi$; otherwise $(y^{v_y})$ would be a clause of $\varphi^P$. The process of electing assignments is iterated for all variables, and at each step either the value $v_w$ of a variable $w$ is accepted or it must be complemented. In any case, it is never necessary to reconsider both elected

assignments for any variable *w*. To prove this fact, let us suppose that both elected assignments to a variable *w* had to be reconsidered. Then there would be an inconsistent assignment set $\boldsymbol{A}_1$ which would be discovered only after specifying two assignment sets $\boldsymbol{A}_2$ and $\boldsymbol{A}_3$, with $\boldsymbol{A}_1 \subseteq \boldsymbol{A}_2$, $\boldsymbol{A}_1 \subseteq \boldsymbol{A}_3$ and $\boldsymbol{A}_2 = [\boldsymbol{A}_3 - \{ w^i \}] \cup \{ w^{\neg i} \}$. But then the assignments included in $\boldsymbol{A}_1$ would identify an implicate of the consistency function not subsumed by all prime implicates of $\xi$; a contradiction.

After processing all variables, a satisfying assignment is identified. We further note that Boolean constraint propagation (BCP) can complement the above procedure by identifying additional assignments, thus reducing the number of conflicts. Since BCP only identifies assignments that are necessary for the identification of a solution (from Theorem 2.1), then we are again guaranteed that the proposed procedure never needs to reconsider both assignments to any variable. Furthermore, we have the following result:

**Theorem 2.3.** Given a clause database $\varphi^P$ that identifies the set of all prime implicates of a switching function $\xi$, then BCP (as defined in Figure 2.6) is logically complete, i.e. given an elective decision assignment, BCP identifies all possible logical consequences of such assignment.

**Proof:** The proof revisits and formalizes the ideas described in the above paragraph, but applied to BCP. Let $x \leftarrow v_x$ be an elective assignment and let the assignment set $\boldsymbol{A}$ denote all logical consequences identified by BCP as a result of the elective assignment. Suppose the existence of a logical consequence $w \leftarrow v_w$, which BCP does not identify (i.e. $(w, v_w) \notin \boldsymbol{A}$). In such a situation, $(w, \overline{v_w})$ would be an invalid assignment, and so we could construct the implicate,

$$\omega = w^{\overline{v_w}} + \sum_{(s, v_s) \in \boldsymbol{A}} s^{v_s}$$

which would then be a non-subsumed new implicate of $\xi$. Hence, a contradiction. ■

A different formulation and proof of the above theorem can be found in [46] in the context of Truth Maintenance Systems. We further note that even though BCP is logically complete for a clause database composed of prime implicates, it will not necessarily identify the assignment of all variables. As a result, it is necessary to elect some node assignments when applying the procedure described above.

```
// Input arguments:    Clause database φ, consensus variable x
// Output arguments:   None
// Return values:      Consensus clause database φ ∪ φ(x)
//
Consensus(φ, x)
{
    C_0(x) = {ω − {x^0} | ω ∈ φ ∧ x^0 ∈ ω};          // Remove x from clauses
    C_1(x) = {ω − {x^1} | ω ∈ φ ∧ x^1 ∈ ω};          // Remove ¬x from clauses
    φ(x) = {ω_0 ∪ ω_1 | ω_0 ∈ C_0(x), ω_1 ∈ C_1(x)};   // Pairwise consensus
   // Delete all clauses containing x
     φ ← φ − [{ω ∪ {x^0} | ω ∈ C_0(x)} ∪ {ω ∪ {x^1} | ω ∈ C_1(x)}] ;
   return φ ∪ φ(x);
}
```

**Figure 2.10:** The Davis-Putnam resolution procedure

**Example 2.16.** Let us consider the example function *f*(*x*, *y*, *z*, *w*) shown in Figure 2.9. The decision assignment $z = 0$ implies the assignment $y = 1$, but no other node assignments are implied. This fact just indicates that, to satisfy *f*, we still have non-unique options with respect to the values of the other variables. □

It is interesting to note that one of the first algorithms for CNF satisfiability, proposed by Davis and Putnam in [38], is based on a slightly modified form of saturation resolution. With respect to procedure `Generate_Prime_Implicates`() given in Figure 2.7, the consensus procedure to be invoked is now defined in Figure 2.10. (Observe that in [38] the consensus operation is implemented by two different phases of the algorithm: the rule for eliminating one variable and the procedure for reconstructing a new CNF. The operation of these two steps is equivalent to the resolution procedure of Figure 2.10.) The Davis-Putnam resolution procedure ensures that after resolving with respect to a variable *x*, all clauses containing literals on *x* are deleted from the clause database, since these clauses are irrelevant for the goal of proving satisfiability. Let $\varphi_{k-1}$ be the current clause database and let $\varphi_k$ be the clause database that results from applying the Davis-Putnam resolution procedure with respect to a variable *x*, $\varphi_k \leftarrow$ `Consensus`($\varphi_{k-1}$, *x*).

**Theorem 2.4.** Given the definitions of $\varphi_{k-1}$ and $\varphi_k$, $\varphi_{k-1}$ is satisfiable if and only if $\varphi_k$ is satisfiable.

**Proof:** Suppose an assignment set $\boldsymbol{A}$ such that the value $\varphi_{k-1}$ is 1. Now suppose, without loss of generality, that $x = 0$. Then every clause in $C_0(x)$ must be satisfied by hypothesis. The Davis-Putnam procedure replaces the set of clauses containing a literal in $x$ by $\varphi(x)$ (see Figure 2.10). Now all clauses in $\varphi(x)$ are satisfied, since each resulting clause contains a clause in $C_0(x)$, which is satisfied. Hence, $\varphi_k$ is satisfied. The same reasoning applies for $x = 1$.

Conversely, suppose an assignment set $\boldsymbol{A}$ (not containing $x$) such that the value of $\varphi_k$ is 1. We show that the value of $x$ can be set so that $\varphi_{k-1}$ also evaluates to 1. Suppose $\omega_0 \in C_0(x)$ such that $\omega_0$ is unsatisfied under $\boldsymbol{A}$. By definition, $\varphi(x)$ contains the set $\{\omega_0 \cup \omega_1 | \omega_1 \in C_1(x)\}$. Hence, because $\omega_0$ is unsatisfied, all clauses in $C_1(x)$ must be satisfied, since $\varphi(x)$ valuates to 1. In such a situation, just set $x = 1$ and $\varphi_{k-1}$ is also satisfied. A similar reasoning applies for $\omega_1 \in C_1(x)$ such that $\omega_1$ is unsatisfied under $\boldsymbol{A}$. Note that it is not possible to have clauses $\omega_0$ and $\omega_1$, $\omega_0 \in C_0(x)$ and $\omega_1 \in C_1(x)$, such that both are unsatisfied under $\boldsymbol{A}$, because, by definition of $\varphi(x)$, $\varphi_k$ would not be satisfied; a contradiction.

By applying this analysis to all resolution steps we are ensured that the clause database that results from iterated application of the Davis-Putnam resolution procedure is satisfiable if and only if the original clause database is satisfiable. ■

The above proof is based on the proof given in [40, theorem 6.2, p. 248], thus proving the resolution-based Davis and Putnam procedure to be sound and complete. Interestingly, another proof of correctness and completeness of an inference rule equivalent to this form of resolution was established by T. Skolem in 1928 [158]. Furthermore, the application of this form of consensus is also implicit in Boole's work [14, see Chapter VII on elimination].

We note that the size of the clause database that results from applying the Davis-Putnam resolution procedure is strictly smaller than for the previous procedure (i.e. saturation resolution). It can also be concluded that the Davis-Putnam resolution procedure cannot be used to identify all the prime implicates of a Boolean function, since some sets of non-subsumed clauses are discarded at each step of the procedure. The original satisfiability algorithm proposed by Davis and Putnam also uses a few additional rules to simplify the CNF formula, particularly the *unit clause rule* (discussed in Section 2.4) and the *pure literal rule*. These rules are analyzed in more detail in the next section. We further note that the search-based SAT algorithm commonly referred to as the

Davis-Putnam procedure is actually described in [39, 110], and is based on the plain backtracking search procedure (there the backtracking operation being referred to as the *splitting rule*).

The description of the resolution-based Davis-Putnam procedure does not provide mechanisms to construct a satisfying assignment given that a CNF formula is proved to be satisfiable. In order to identify the satisfying assignment for the original CNF formula, we study formula $\varphi_k$ that results from resolving formula $\varphi_{k-1}$ with respect to a variable $x$, $\varphi_k \leftarrow \texttt{Consensus}(\varphi_{k-1}, x)$. Suppose $\varphi_k$ is known to be satisfiable. Then, from our analysis above, either all clauses of $C_0(x)$ evaluate to 1, or all clauses in $C_1(x)$ assume value 1. Hence, we choose $x$ so that all clauses in $\{\omega \cup \{x^0\} \mid \omega \in C_0(x)\} \cup \{\omega \cup \{x^1\} \mid \omega \in C_1(x)\}$ are satisfied. The process is repeated for all $k$ in decreasing order, and hence if the problem is satisfiable, then a satisfying assignment is identified.

Saturation resolution and the resolution-based Davis-Putnam procedure incur significant computational overhead in both time and space. There are several variations and improvements to saturation resolution that can potentially be applied to SAT, even though such improvements are typically proposed for First-Order Logic (see for example [28, 110], and [131, 163] for more recent results). Currently, practical and complete algorithmic approaches to solving SAT are in the vast majority of cases variations of the plain backtracking search procedure.

As a final note, we study one possible extension of consensus for solving instances of SAT, which builds on a specialization of the generalized definition of consensus proposed by Tison [168]. Let us assume a set of $M$ clauses $\mu_i$, $1 \leq i \leq M$, defined over $m$ variables $\{ x_1, \ldots, x_m \}$ such that:

1. $\prod_i \mu_i = 0$.

2. For some $\mu_l$, $\prod_{i \neq l} \mu_i \neq 0$.

Assume a set of $J$ clauses $\gamma_j$ and a set of $K$ clauses $\upsilon_k$ defined over other variables not including the variables $\{ x_1, \ldots, x_m \}$. Further, let the clause database be given by:

$$\varphi = \prod_k \upsilon_k \cdot \prod_j (\gamma_j + \mu_l) \cdot \prod_{i \neq l} \mu_i \tag{2.18}$$

where $\mu_l$, $1 \leq l \leq M$, is one of the clauses defined over the variables $\{ x_1, \ldots, x_m \}$. In such a situation, $\xi$ associated with $\varphi$ is satisfiable if and only if the following clause database is satisfiable:

$$\varphi = \prod_k \upsilon_k \cdot \prod_j \gamma_j \tag{2.19}$$

In order to justify the proposed transformation, we note conditions 1 and 2 above. An assignment to the variables in $\{ x_1, \ldots, x_m \}$, that satisfies all clauses other than $\mu_l$, must set $\mu_l$ to 0. On the other hand, any assignment that satisfies $\mu_l$ must set some other clause $\mu_i$ to 0, and so the clause database becomes inconsistent. We can thus conclude that only assignments to the variables in $\{ x_1, \ldots, x_m \}$ that set $\mu_l$ to 0 can be part of a consistent assignment for $\varphi$. Hence, (2.19) follows. The transformation of (2.18) into (2.19) is a generalization of the unit clause rule (see page 38), and thus it is also logically incomplete. (Note that the case $m = 1$ corresponds to the unit clause rule.) The above generalization can potentially lead to more simplifications than the unit clause rule.

**Example 2.17.** Let us consider the following clause database:

$$(y + \neg w + \neg s) \cdot (y + w + s + x + \neg z) \cdot (\neg y + s + x + \neg z) \cdot (x + z) \cdot (\neg x + \neg z) \cdot (\neg x + z)$$

Even though the unit clause rule cannot be applied, its generalization can. Let the set of variables be $\{x, z\}$ then, using the transformation of (2.18) into (2.19), the resulting clause database becomes $(y + \neg w + \neg s) \cdot (y + w + s) \cdot (\neg y + s)$ . □

The generalization of the unit clause rule is useful in situations where the set of variables $\{x_1, \ldots, x_m\}$ and associated clauses can be easily identified. Even though this rule is theoretically appealing, given its apparent simplification ability, its practical use is questionable. For example, the identification of resolution sets of variables $\{x_1, \ldots, x_m\}$ requires considering all subsets of variables of size less than or equal to a threshold value *m*, that must satisfy the conditions required

for transforming (2.18) into (2.19). Testing such conditions is in the worst-case exponential in $m$. In addition, and for clause databases derived from combinational circuits, the sparse nature of the generated clause databases suggests that the extension to the unit clause rule might be seldom applied.

### 2.5.4 Search-Based Davis-Putnam Procedure

In this section we review the search-based formulation of the Davis-Putnam procedure for solving SAT that is described in [39, 110]. In particular, we follow the description of [39]. This procedure (referred to as DP-SAT) forms the basis of a large number of other search algorithms for SAT and is composed of the following main steps:

1. *Unit clause rule*: If there is a unit clause $\{x^i\}$, then remove literal $\{x^{\neg i}\}$ from any clause containing it. This can be viewed as assigning value $\neg i$ to variable $x$, and consequently, setting to 0 any occurrence of literal $\{x^{\neg i}\}$.
2. *Pure literal rule*[7]: If a variable $x$ is monoform, i.e. if the clause database just contains literals on variable $x$ of the form $\{x^i\}$, then remove from the clause database all clauses containing $\{x^i\}$. This can be viewed as the ability to disregard a set of clauses containing the same literal which does not appear complemented. In this situation, the literal will never need to be complemented, and so it can assume value $\neg i$.
3. *Splitting rule*: If steps 1 and 2 cannot be applied, choose a variable and split the clause database into two. The first one with all clauses containing $\{x^0\}$ removed, and $\{x^1\}$ removed from all clauses. The second one with all clauses containing $\{x^1\}$ removed, and $\{x^0\}$ removed from all clauses. Each resulting clause database is processed separately. (Note that the splitting rule is just a different organization of the plain backtracking algorithm. Furthermore, in [39] generated clause databases are maintained in a FILO queue, that corresponds to backtracking.)
4. Repeat steps 1 through 3 while queue of clause databases is not empty. Discard clause data-

[7]. The pure literal rule was so named by Davis and Putnam [38] in 1960, even though the same idea was first described in [56] in 1959, in a search-based algorithm for proving the satisfiability of boolean formulas, and whose only inference rule was the pure literal rule. This algorithm did not convert a boolean formula into normal form; the rule was described for formulas with connectives $\vee$ and $\wedge$, and $\neg$ only associated with literals.

bases where the empty clause is derived (i.e. unsatisfiable clause).

5. If the queue becomes empty, then the formula is unsatisfiable. Otherwise, if the empty formula is derived (i.e. $\varphi = \varnothing$), then a solution exists.

From the above description the following facts are clear. DP-SAT derives no fewer implications than BCP, and it may derive more implications. This fact results from the pure-literal rule, that can be used to remove some variables that BCP does not. Due to its simplicity, the worst-case and average-case complexity of DP-SAT, and of some of its simplifications and improvements, have been extensively studied (see for example [59, 73, 125]). As a side remark, note that after applying BCP, the pure literal rule may identify a few more implications. However, no more unit clauses will be created until another decision assignment is made, and so BCP needs *not* be invoked again. A straightforward corollary is that the pure literal rule will *not* identify a conflict that BCP fails to identify (provided that BCP has been invoked prior to applying the pure literal rule).

## 2.6 Queries on Clause Databases

Let us assume a clause database $\varphi_o$. The most general form of instance of the satisfiability problem is to specify a (possibly empty) set of variable assignment objectives, and evaluate whether a satisfying assignment can be identified. For this purpose, the set of objectives is represented by a CNF formula $\varphi_q$, which we refer to as a *query*. The objective of SAT algorithms is to identify a satisfying assignment to each query.

The operation of a SAT algorithm on an original clause database $\varphi_o$, given a query $\varphi_q$, is defined by the following computation:

$$(\varphi_r, \boldsymbol{A}_r) \leftarrow \mathrm{SAT}(\varphi_o \cup \varphi_q, \varnothing, \mathit{status}) - (\varphi_q, \varnothing) \tag{2.20}$$

which assumes the interface to SAT algorithms defined in Figure 2.1. The SAT algorithm identifies whether the original clause database $\varphi_o$, appended with a query $\varphi_q$, is satisfiable. The outcome is made available through *status*. The resulting clause database $\varphi_r$ is given by the (modified) clause database returned by the SAT algorithm, but with the original query removed. The computation is said to be *valid* if and only if $\varphi_r\big|_{\boldsymbol{A}} = \varphi_o\big|_{\boldsymbol{A}}$ for every assignment set $\boldsymbol{A}$.

**Example 2.18.** For the example circuit of Figure 2.5 on page 36, an example of a set of objectives is $x_3 = 1$ and $z_1 = 1$. Consequently, $\varphi_q = (x_3) \cdot (z_1)$. Another query would be to specify no objectives, thus testing whether the clause database is itself satisfiable. Hence, $\varphi_q = \varnothing$. □

In the description of SAT algorithms in the following chapters we restrict queries to denote cubes in the $N$-dimensional Boolean space. Consequently, a query is represented by a clause database where each clause has *exactly* one literal. Such a query can be viewed as a restriction of the consistency function $\xi$ (associated with $\varphi_o$) to the cube specified by $\varphi_q$, and is denoted by $\xi_q$.

The proposed search algorithms for SAT can identify and add clauses to the initial clause database $\varphi_o \cup \varphi_q$, which are implicates of $\xi_q$. Since $\xi_q$ is a restriction of $\xi$ to the cube specified by $\varphi_q$, these clauses are also implicates of $\xi$. Hence, these clauses can be added to the original clause database $\varphi_o$, independently of the query. We can thus conclude that implicates of $\xi_q$ identified by execution of a SAT algorithm can be used for solving other queries, as long as each query defines a cube in the $N$-dimensional Boolean space.

In the following analysis of SAT algorithms, a query is always implicitly assumed. The usefulness of allowing the SAT algorithm to modify the clause database, even in the presence of a query, will become apparent as we study search-based SAT algorithms.

## 2.7 Summary

In this chapter we introduced the mathematical framework that serves, in the remaining chapters, to formalize the description and analysis of search-based SAT algorithms. The definitions introduced can be categorized as follows:

- Definition of variables, literals, clauses and CNF formulas, with the objective of defining a unified representation for instances of SAT.
- Definition of the consistency function associated with a CNF formula, and motivation for CNF formulas to be referred to as clause databases.
- Representation of instances of SAT as CNF formulas. In particular, we described how to create the CNF formula for a propositional formula and for a combinational circuit.
- Formalization of implications in CNF formulas. Definition of Boolean constraint propaga-

tion. The concepts of antecedent assignments and antecedent set were also introduced, which play a key role in the description of SAT algorithms in the remainder of the dissertation.

- Review of the concepts of consensus and resolution. Description of simple procedures for generating prime implicates. Study of how these concepts can be applied to solving SAT algebraically. Study of the search-based Davis-Putnam SAT algorithm.
- Definition of queries on clause databases, which define how distinct sets of objectives can be examined on the same clause database.

# CHAPTER III

# SEARCH ALGORITHMS FOR SATISFIABILITY

## 3.1 Introduction

In this chapter we undertake the study of search algorithms for the satisfiability problem (SAT). As mentioned in Chapter I, a search algorithm entails a decision procedure that implicitly enumerates a given search space. For SAT, and under the definitions of the previous chapter, the search space is given by $\{0,1\}^N$. Our approach for solving SAT is to augment the plain backtracking search algorithm (see Chapter I on page 6) with several *engines* for inferring facts, in order to reduce the amount of search. These engines can be categorized as follows:

1. *Selection engine*, which decides the sequence of assignments to guide the search process. The degree by which the selection engine reduces the amount of search is referred to as the *selection ability.*
2. *Deduction engine*, which infers necessary assignments as a result of other assignments, that result from decisions or from implications. Deduction engines are characterized by their *deduction ability*, that quantifies the capability of the deduction engine to identify logical consequences.
3. *Diagnosis engine*, which infers the causes of conflicts, and can generate adequate information to prevent the same conflicts from occurring later during the search. Diagnosis engines are characterized by their *diagnosis ability*, that measures the capability of the diagnosis engine to identify the causes of conflicts.

Deduction engines implement a form of *forward reasoning* as a result of decision assign-

ments. In contrast, diagnosis engines implement a form of *backward reasoning* as a result of identified conflicts.

The above engines are applied during the search for a solution to a query. In addition, other engines can be applied before the search or after a solution is identified:

1. *Preprocessing engines* identify stronger implicates of the clause database, that can be used to increase the deduction ability of the deduction engines. *Preprocessing ability* quantifies how effectively the preprocessing engine computes implicates of the clause database.
2. *Postprocessing engines* remove redundancies from computed solutions to queries, and can cache signatures of computed solutions. These cached signatures can be used during the search for subsequent queries to reduce the amount of search.

Selection engines are largely heuristic, in the sense that the sequence of assignments chosen is *not* guaranteed to reduce the amount of search. In contrast, deduction, diagnosis and preprocessing engines infer facts that are *guaranteed* not to increase the amount of search. The information cached by postprocessing engines is also guaranteed to reduce the amount of search, but only for subsequent queries.

### 3.1.1 Chapter Objectives

The first objective of this chapter is to describe GRASP, a search algorithm for SAT, that can be customized with different engines. As a result, we are able to provide, with the same algorithmic framework, a wide range of search pruning ability as a function of the computational effort to conduct the search. The second objective is to detail each engine, as well as related simplifications and improvements. In particular, we propose to describe families of engines for identification of implications, diagnosis of conflicts and preprocessing of clause databases. Moreover, we describe engines for postprocessing and for making decisions.

With respect to other algorithms for SAT, the most significant contribution of GRASP is its ability to diagnose the causes of conflicts. Three different methods, referred to as *pruning methods*, are proposed for diagnosing conflicts:

1. *Conflict-Directed Backtracking* (CDB) is a form of non-chronological backtracking based on conflict diagnosis that, under some conditions, allows the search process to backtrack over

several decision assignments that can be shown not to be relevant for the identification of a solution.

2. *Conflict-Based Equivalence* (CBE) identifies sufficient conditions for two different stages in the search process to lead to the same conflicts. Conflict-based equivalence conditions permit early backtracking instead of requiring conflicts to be explicitly identified.
3. *Failure-Driven Assertions* (FDAs) denote required assignments to variables that are identified as a result of a conflict. Different forms of FDAs can be defined, some of which exploit the structure of the conflicts in order to derive stronger assertions.

### 3.1.2 Chapter Outline

Section 3.2 introduces all the structures required for the description of GRASP. The top-level organization of GRASP is introduced in Section 3.3, followed by a definition of the main conflict analysis methods in Section 3.4. The next step is to describe the different engines that customize GRASP. We start with the deduction engines, followed by the diagnosis engines. A particular emphasis is given to diagnosis engines, since they have seldom been applied to SAT algorithms.

Preprocessing engines are described in Section 3.7. These engines resemble advanced deduction engines and are used to complete the structure of the clause database by computing additional implicates of the consistency function.

We then analyze the postprocessing engine, which is specific to combinational circuits. This entails removing redundancies from solutions, i.e. decision assignments that are provably irrelevant for satisfying the query, and caching signatures of each solution, in order to simplify the search for solutions to subsequent queries.

An orthogonal issue is how to guide the search process by appropriately making decision assignments. Decision making procedures are studied in Section 3.9. We also describe methods to reduce the number of decision variables whenever clause databases represent combinational circuits.

Section 3.10 concludes the chapter by highlighting the major contributions of GRASP and motivating the application of the ideas in GRASP to other domains.

Although most examples in this chapter are based on combinational circuits and clause databases are usually assumed to be derived from circuits, the techniques described, unless otherwise noted, are applicable to *any* clause database. Furthermore, all formal results in this chapter are stated without proof; proofs are provided in Appendix A.

## 3.2 Structures for Search

As described in Chapter I, a backtracking search algorithm implements a *search process* that creates a *decision tree* and implicitly traverses the search space. Each node in the decision tree specifies an elective assignment to a chosen variable, referred to as the *decision assignment*. In the case of a combinational circuit, decision assignments can be restricted to the primary inputs[1], since assigning all primary inputs guarantees that all circuit nodes become assigned.

A *decision level* is associated with each decision assignment to denote its depth in the decision tree; the first decision assignment at the root of the tree is at decision level 1, and the objectives are specified at decision level 0. A decision level $\delta(x)$ is associated with every assigned variable $x$, and denotes the decision level at which the assignment of $x$ is decided or implied. At any stage of the search process, the entries in the decision tree define the *active* decision assignments.

Every time a decision assignment is made it triggers other assignments that define an *implication sequence*. The node that triggers each implication sequence is referred to as the *trigger node*[2]. During the search process the implication sequences resulting from active decision assignments are represented by an *implication graph*, $\boldsymbol{I_C}$, that is defined as follows:

1. The current assignment ($x = v_x$) defines a vertex in the implication graph.
2. The incoming edges to each vertex ($x = v_x$) in the implication graph correspond to the variable assignments identified by $\boldsymbol{A}(x)$ (given by (2.13) on page 40).
3. In the presence of a conflict, a *conflict node,* $\kappa$, is added to the implication graph such that its incoming edges are the node assignments that force a clause $\omega$ of $\varphi$ to be unsatisfied. Conse-

[1.] Actually, in some cases decisions may be made with respect to other nodes, as will be discussed in Section 3.9.

[2.] Besides decision assignments, other assignments that trigger implication sequences do exist and are described later in this chapter. Moreover, implication sequences may be triggered by sets of assignments.

quently, the *antecedent assignment* of $\kappa$, $\boldsymbol{A}(\kappa)$, is defined as the set of variable assignments associated with the unsatisfied clause $\omega$ that is identified as causing the conflict:

$$\boldsymbol{A}(\kappa) = \{(y, \nu(y)) | y \in \omega \vee \neg y \in \omega\} \tag{3.1}$$

Accordingly, the *antecedent set*, $\alpha(\kappa)$, is defined for each conflict due to clause $\omega$:

$$\alpha(\kappa) = \{y | (y, \nu(y)) \in \boldsymbol{A}(\kappa)\} \tag{3.2}$$

Each assigned node $x$ is also characterized by an *implication level*, $\iota(x)$, that denotes the length of the longest path in the implication graph from the trigger node to $x$. Implication levels provide a partial order on the implications, and are central for implementing conflict diagnosis.

With the exception of the trigger node, the decision and implication levels of each assigned node $x$ are defined according to:

$$\begin{aligned} \delta(x) &= max\{\delta(y) | y \in \alpha(x)\} \\ \iota(x) &= 1 + max\{\iota(y) | y \in \alpha(x) \wedge \delta(y) = \delta(x)\} \end{aligned} \tag{3.3}$$

For the trigger node, the decision level is defined as the current decision level of the search process, and the implication level is 0.

When referring to an assigned node $x$, the notation $x = v \ @ \ d \ / \ i$ is used to denote that the value of $x$ is $v$, $x$ is assigned at decision level $d$ and implication level $i$. Whenever the implication level is not significant, the notation $x = v \ @ \ d$ is used instead. Finally, if only the value of $x$ is relevant then the notation $x = v$ is used.

For a combinational circuit, $JF(c)$ denotes the set of unjustified assigned nodes in the circuit at a decision level $c$:

$$JF(c) = \{y \in \boldsymbol{V} | \nu(y) \neq X \wedge \neg Just(y)\} \tag{3.4}$$

$JF(c)$ is commonly referred to as the *j-frontier* in test pattern generation [1, pp. 192-193].

**Example 3.1.** Examples of a decision tree and implication graph are shown in Figure 3.1. It is assumed that a conflict is detected after the decision assignment $x_3 = 1$ is made. As illustrated in

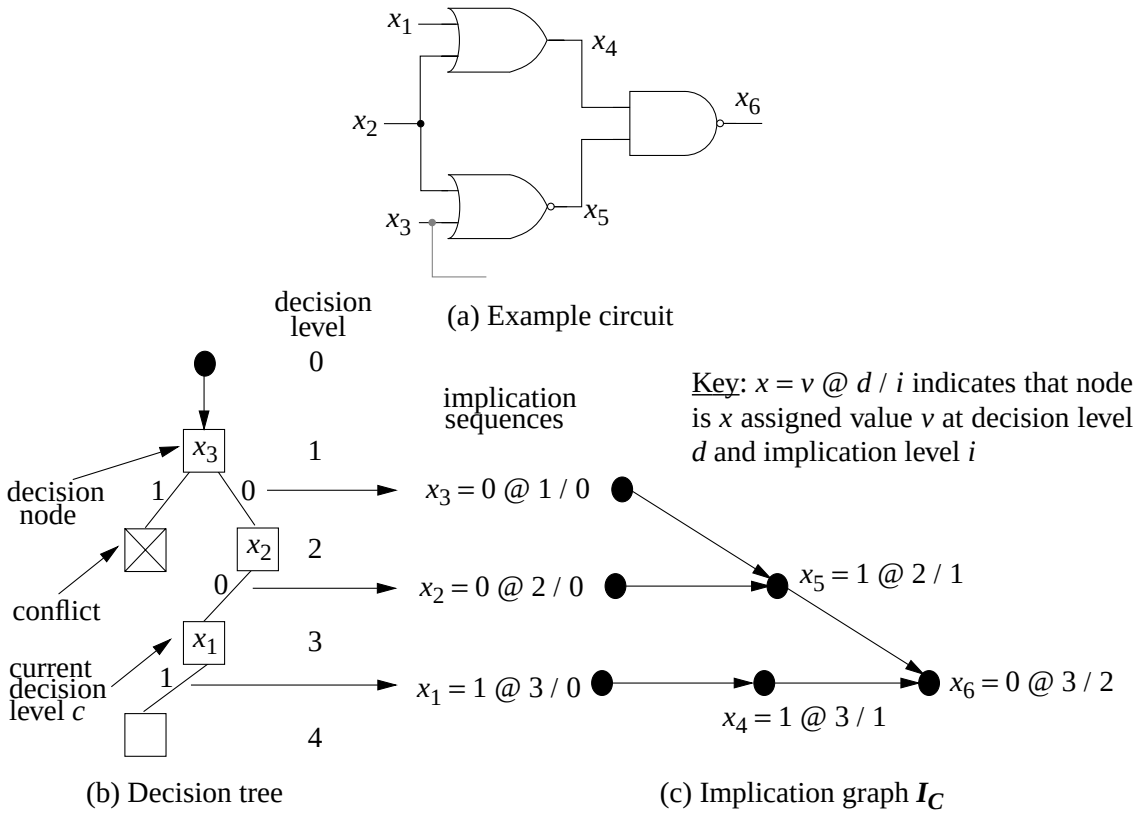



**Figure 3.1:** Examples of decision tree and implication graph

Figure 3.1-c, and according to the definition of implication graph, the incoming edges to each vertex $x = v_x$ in the implication graph are associated with the assignments in the antecedent assignment of $x$. □

## 3.3 Backtracking Search Algorithm

The top-level description of GRASP is shown in Figure 3.2. It assumes a clause database $\varphi$ and an assignment set $\boldsymbol{A}$ as global variables. Initially, the clause database $\varphi$ contains the original clause database $\varphi_o$ augmented with a query $\varphi_q$. Note, however, that when referring to the consistency function $\xi$, the original clause database $\varphi_o$ is assumed. The main purpose of GRASP is to invoke procedure `Search`(), which implements the search process. Two other procedures can be invoked; `Preprocess`() and `Postprocess`():

- `Preprocess`() implements the preprocessing engine. It can complete the clause database with additional implicates of the consistency function and may imply necessary assignments.

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

// Input argument:        Current decision level c
// Output argument:       Backtracking decision level β_L
// Return value:          CONFLICT or SUCCESS
//
Search (c, &β_L)
{
   if (Select (VAR+VAL) == SUCCESS)              // Make decision
      return SUCCESS;
   while (TRUE) {
      if (Deduce() != CONFLICT) {                // Imply assignments
         if (Search (c + 1, β_L) == SUCCESS) return SUCCESS;
         else if (β_L != c) { Erase(); return CONFLICT; }
      }                                          // Diagnose conflict
      if (Diagnose (c, β_L) == CONFLICT) { Erase(); return CONFLICT; }
      Erase();
      Select (VAL);                 // Modify decision assignment
   }
}
```

**Figure 3.2:** Description of GRASP

The operation of `Preprocess`() is characterized by the operations $\varphi^i \rightarrow \varphi^f$ and $\boldsymbol{A}^i \rightarrow \boldsymbol{A}^f$, $\boldsymbol{A}^i \subseteq \boldsymbol{A}^f$. In general, the preprocessing engine can be applied to either $\varphi$ or $\varphi_0$.

- `Postprocess`() implements the postprocessing engine. This engine is solely developed for combinational circuits. It can remove redundancies from computed solutions and can cache

solutions for subsequent use. The operation of redundancy removal is characterized by $A^i \rightarrow A^f, A^i \supseteq A^f$. The clause database is not changed by the postprocessing engine.

Depending on the configuration of GRASP, the actual implementation of the two procedures above can realize no functionality.

The interface to the SAT algorithm is defined by *outcome* ← GRASP() . The pseudo code for procedure `Search`() is shown in Figure 3.2. At each decision level $c$ in the decision tree a decision variable and associated logic value are chosen with procedure `Select`(VAR+VAL), provided that a decision assignment can still be made; otherwise a consistent complete node assignment has been identified and the search process can terminate. If a decision assignment is chosen, logical implications are derived with `Deduce`(). If no conflict is identified, the search algorithm is recursively invoked at decision level $c + 1$. Afterwards, if a solution has been identified, a termination indication is passed on to the previous decision levels. Otherwise, the backtracking decision level is tested, and if different from the current decision level, further processing at the current decision level is skipped. In the presence of a conflict, the last implication sequence is analyzed with `Diagnose`(), and used to decide whether backtracking is needed and where to backtrack to. Finally, procedure `Select`(VAL) allows defining the next decision assignment with respect to the current decision variable, provided another decision assignment can be made on that variable. As we will see below, `Select`(VAL) may be defined to implement no functionality, since conflict diagnosis may create conditions for implying the assignment that would correspond to the second branch at each decision level.

The search process creates a sequence of partial variable assignments $A_0$, $A_1$, …, $A_c$, where $c$ denotes the current decision level, and guarantees that $A_0 \subseteq A_1 \subseteq \ldots \subseteq A_c$, since otherwise completeness of the search algorithm would not be guaranteed. At each decision level $c$, the operation of each engine can cause modifications to either the clause database $\varphi_c$ or the current partial variable assignment $A_c$. The operation of each engine is defined as follows:

- `Select`(VAR+VAL) implements the selection engine. It selects a variable and a logic value to assign to that variable. The decision variable is kept until the search process backtracks below the current decision level. `Select`(VAR+VAL) causes no modifications to the clause database and only adds the decision assignment to the current partial variable assignment

$$(\boldsymbol{A}_c^i = \boldsymbol{A}_{c-1}) \rightarrow [\boldsymbol{A}_c^f = \boldsymbol{A}_{c-1} \cup \{(x, v_x)\}] \ .$$

- `Deduce`() implements the deduction engine, which creates implication sequences caused by trigger assignments. The operation of the deduction engine is characterized by $\varphi_c^i \rightarrow \varphi_c^f$ and by $\boldsymbol{A}_c^i \rightarrow \boldsymbol{A}_c^f$, $\boldsymbol{A}_c^i \subseteq \boldsymbol{A}_c^f$.
- `Diagnose`() implements the diagnosis engine, which identifies the causes of conflicts. The operation of the deduction engine is characterized by $\varphi_c^i \rightarrow \varphi_c^f$ and $\boldsymbol{A}_c^i = \boldsymbol{A}_c^f$.
- `Erase`() clears the implication sequence defined at the current decision level. It causes no modifications to the clause database, and the resulting partial variable assignment becomes $\boldsymbol{A}_c^i \rightarrow (\boldsymbol{A}_c^f = \boldsymbol{A}_{c-1})$.
- `Select`(VAL) selects another value for the variable chosen at the current decision level as the decision variable. It causes no modifications to the clause database and the resulting partial variable assignment becomes $(\boldsymbol{A}_c^i = \boldsymbol{A}_{c-1}) \rightarrow [\boldsymbol{A}_c^f = \boldsymbol{A}_{c-1} \cup \{(x, \overline{v_x})\}]$, assuming the original decision assignment to be $x = v_x$.

The actual operations on the clause database and on the partial variable assignments depend on how each engine is implemented. GRASP, as described in Figure 3.2, can be configured to realize different SAT algorithms, each of which is characterized by how the clause database and the partial variable assignments evolve with the search process.

The purpose of the following sections is to detail each engine invoked by `GRASP`() and by `Search`() and study different tradeoffs between pruning ability and computational overhead requirements. For the deduction and diagnosis engines several implementations are possible, and special emphasis is given to the *basic* engines, which can be implemented in linear time in the size of the clause database. Given the definition of the basic implementations of `Deduce`() and `Diagnose`() in the following sections, the following holds:

**Theorem 3.1.** The search algorithm for solving SAT, described in Figure 3.2, customized with the basic implementations of `Select`(VAR+VAL), `Deduce`() and `Diagnose`() is sound and complete.

The proof of the above theorem, as well some complexity bounds for the search algorithm, can be found in Appendix A. The proof of the theorem hinges on the fact that each clause that can

be added to the clause database is indeed an implicate of the consistency function, and that partial assignment sets associated with a sequence of decision levels satisfy a containment relation, i.e. $\boldsymbol{A}_0 \subseteq \boldsymbol{A}_1 \subseteq \ldots \subseteq \boldsymbol{A}_c$.

## 3.4 Conflict Analysis

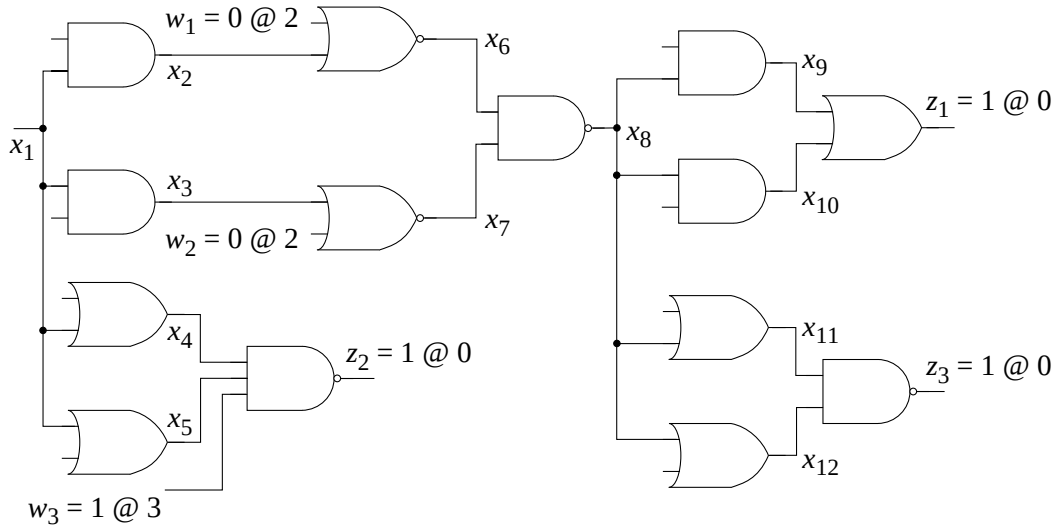
The analysis of the causes of conflicts is central to the implementation of the different engines. In this section we describe the basic mechanism for identifying sufficient conditions for a conflict to occur. These conditions are used with different purposes, the most relevant being the definition of implicates of the consistency function.

**Example 3.2.** The derivation of an implication sequence, based on Boolean constraint propagation (BCP), that yields a conflict is shown in Figure 3.3. The decision assignment $x_1 = 0$ at decision level 5 implies the assignments of $x_2$, $x_3$, $x_6$, $x_7$, $x_8$, $x_9$ and $x_{10}$, which result in a conflict involving the assignments of $x_9$, $x_{10}$ and $z_1$. The portion of the implication graph that is relevant for this implication sequence is shown in Figure 3.3-b. Only the implication levels of the nodes assigned at decision level 5 are shown. In terms of the clause database, the gate with output $z_1$ (an OR gate) is characterized by the following consistency function (adapted from Table 2.1 on page 35):

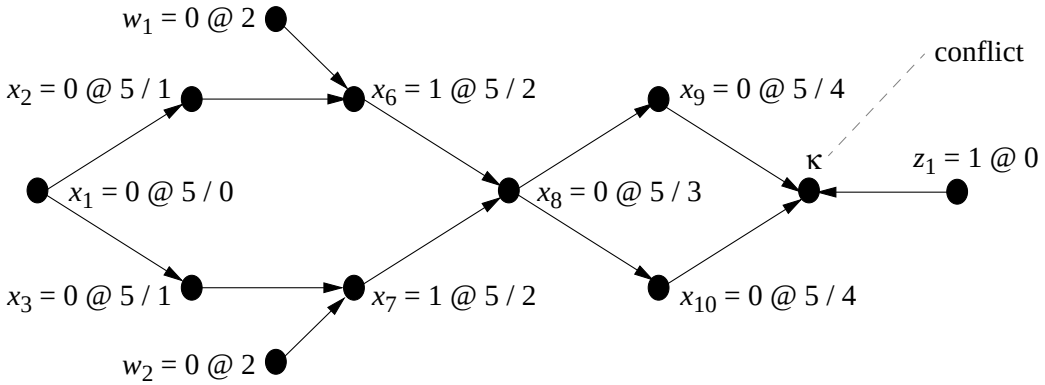
$$\varphi_{z_1} = (\neg x_9 + z_1) \cdot (\neg x_{10} + z_1) \cdot (x_9 + x_{10} + \neg z_1)$$

The implication sequence causes clause $(x_9 + x_{10} + \neg z_1)$ to become unsatisfied since $x_9 = x_{10} = 0$ and $z_1 = 1$. By definition, the antecedent assignment of $\kappa$ is given by $\boldsymbol{A}(\kappa) = \{ (x_9, 0), (x_{10}, 0), (z_1, 1) \}$, which explains the edges connected to $\kappa$ in Figure 3.3-b. (In the following, the derivation of implications in terms of the clause database is omitted and implications are described in terms of propagation of logic values in a circuit. However, in any situation we could readily create the clause database for the consistency function of the circuit, and justify the assignment of any circuit node as the requirement to satisfy a unit clause of the clause database.) □

In the remainder of this section we describe methods for identifying the causes of conflicts. In all situations, the purpose of conflict analysis is to identify implicates of the clause database that describe the causes of identified conflicts. Two methods are described; the first is the




(a) Example circuit




(b) Partial implication graph

**Figure 3.3:** Example circuit and partial implication graph

standard formulation of conflict analysis, whereas the second exploits the *structure* of implication sequences.

### 3.4.1 Conflicting Clauses

Conflict analysis is based on traversing the portion of the implication graph associated with the current decision level and establishing a set of conditions that are responsible for the conflict. Suppose that the current decision level is $c$ and that the current decision assignment leads to a conflict. In terms of the implication graph this is represented by an implication sequence of nodes assigned at decision level $c$ that terminates at a conflict node $\kappa$. For each node $x$ assigned at deci-

sion level $c$, and for the conflict node $\kappa$, the antecedent assignment is divided into two disjoint sets, of assignments $\Lambda(x)$ at lower decision levels, and assignments $\Sigma(x)$ at the current decision level $c$:

$$\begin{aligned} \Lambda(x) &= \{(y, \nu(y)) \in \boldsymbol{A}(x) | \delta(y) < \delta(x)\} \\ \Sigma(x) &= \{(y, \nu(y)) \in \boldsymbol{A}(x) | \delta(y) = \delta(x)\} \end{aligned} \tag{3.5}$$

**Example 3.3.** For the implication graph of Figure 3.3-b, the antecedent assignment of $x_6$ is partitioned into $\Lambda(x_6) = \{ (w_1, 0) \}$ and $\Sigma(x_6) = \{ (x_2, 0) \}$, since $\delta(w_1) = 2$ and $\delta(x_2) = 5$. □

The partition of $\boldsymbol{A}(x)$, for each variable $x$ assigned at decision level $c$, is used to identify the causes of a conflict and to represent those causes by a *conflicting assignment set* ($\boldsymbol{A}_{CS}$), which is defined as follows:

$$\boldsymbol{A}_{CS} = causesof(\kappa) \tag{3.6}$$

where $causesof : \boldsymbol{V} \cup \{\kappa\} \rightarrow 2^{\boldsymbol{V} \times \{0, 1\}}$ and,

$$causesof(x) = \begin{cases} (x, \nu(x)), & \text{if } \boldsymbol{A}(x) = \varnothing \\ \Lambda(x) \cup \left[ \bigcup_{(y, \nu(y)) \in \Sigma(x)} causesof(y) \right], & \text{otherwise} \end{cases} \tag{3.7}$$

*causesof*($x$) identifies the node assignments that create an implication sequence leading to the assignment of $x$. In particular, *causesof*($\kappa$) identifies a set of node assignments which represent a sufficient condition for a conflict to occur. Note that $\boldsymbol{A}_{CS} \subseteq \boldsymbol{A}_{c-1} \cup \{(x, \nu(x))\}$, where $x$ is the trigger node, and so the causes of the conflict are contained in the partial node assignment before the last implication sequence. This is clear from the definition of conflicting assignment set in (3.6) and (3.7), which can only contain assigned nodes, all of which must be included in $\boldsymbol{A}_{c-1} \cup \{ (x, \nu(x)) \}$.

A conflicting assignment set can also be viewed as a conjunction of node assignments that are identified as a sufficient condition for a conflict to be identified. The *negation* of this conjunction of node assignments provides a *conflicting clause* that represents an *implicate* of the consis-

tency function of the circuit[3]. The clause to be created from the conflicting assignment set is defined by:

$$\omega = \{ x^{\nu(x)} \big| (x, \nu(x)) \in \boldsymbol{A}_{CS} \} \tag{3.8}$$

which guarantees that the value of ω is 0 if and only if all node assignments specified the conflicting assignment set occur simultaneously. ω is added to the clause database, which is to say that if each assignment in a conflicting assignment set is satisfied, then the same conflict is identified without the same implication sequence being recreated.

The definition of conflicting clause also implies that at most one literal of ω is assigned at the current decision level. In such a situation, if the last implication sequence is erased, then ω becomes a unit clause and thus it implies the assignment that will trigger another implication sequence at decision level *c*. In contrast with most search-based SAT algorithms, which exhaust the possible node assignments for each decision without establishing relations among the decision branches, the procedure we propose generates a second branch at decision level *c* that results from an implication and *not* from a decision. Such an implication is referred to as an *assertion*, and denotes a node that is assigned at decision level *c* even though all nodes in its antecedent set are assigned at decision levels less than *c*. Any implication that results from a conflicting clause is referred to as a *failure-driven assertion* (FDA). Note that since FDAs are derived and create the second branch at decision level *c*, procedure `Select`(VAL) in GRASP realizes no functionality.

Suppose that ω is created due to a conflict, associated with assigning a decision node *x*. Then ω adds additional information to the clause database. Without ω in the clause database the assignment of *x* creates an implication sequence that leads to the same conflict; with ω in the clause database the assignment of *x* to the complemented value would be implied without the decision being made, because it would be the only free literal in ω. In this situation, the same conflict would either not be identified or be identified without a decision assignment being made.

**Example 3.4.** To illustrate the derivation of conflicting assignments and associated clauses, con-

[3] Conditions similar to implicates of ξ are referred to as *nogoods* in truth maintenance systems [60, 116, 161] and in some algorithms for constraint satisfaction problems [143]. Nevertheless, the basic mechanism for creation of conflicting clauses differs, since conflicting clauses are not necessarily expressed in terms of decision variables, whereas nogoods are.

sider the example of Figure 3.3-a. Given the conflict of Figure 3.3-b, the following conflicting assignment set is derived using (3.6):

$$\boldsymbol{A}_{CS} = \{(x_1, 0), (w_1, 0), (w_2, 0), (z_1, 1)\}$$

because $\Lambda(\kappa) = \{ (z_1, 1) \}$, $\Lambda(x_6) = \{ (w_1, 0) \}$, $\Lambda(x_7) = \{ (w_2, 0) \}$ and $\boldsymbol{A}(x_1) = \varnothing$. From (3.8), the following conflicting clause is created:

$$\omega = (x_1 + w_1 + w_2 + \neg z_1) \tag{3.9}$$

which states that the assignment $x_1 = w_1 = w_2 = 0$ and $z_1 = 1$ cause a conflict, as the partial implication graph of Figure 3.3-b shows. This clause is now added to the clause database, thus providing stronger conflicting conditions in the presence of partial node assignments. After erasing the conflicting implication sequence, $\omega$ becomes a unit clause with $x_1$ as its free literal. This implies $x_1 \leftarrow 1$ and $x_1$ is said to be asserted. As a result, the second branch at decision level 5 is no longer elected, but forced by $\omega$, which produces a failure-driven assertion and implies the assignment of $x_1$ to 1. (The second branch can only be specified as an assertion because each node can only assume two logic values. In more general search problems (e.g. constraint satisfaction problems [169]), this form of assertion is not derivable because the domain of each variable can have more than two possible values.) □

### 3.4.2 Unique Implication Points

An implication sequence at a given decision level $c$ defines a subgraph contained in the implication graph $\boldsymbol{I}_{\boldsymbol{C}}$. Assuming a conflict is detected, let $U = \{ (u_1, \nu(u_1)) , \ldots, (u_k, \nu(u_k)) \}$ denote the set of *dominators* [166] of $\kappa$, with respect to the decision assignment or set of asserted assignments in the implication graph at decision level $c$, that trigger the conflicting implication sequence. Each $(u_i, \nu(u_i))$ is referred to as a *unique implication point* (UIP), and can be viewed as triggering an implication sequence at decision level $c$ that leads to the same conflict.

**Theorem 3.2.** Let a conflict be identified at decision level $c$, and let $U = \{ (u_1, \nu(u_1)) , \ldots, (u_k, \nu(u_k)) \}$ denote the set of UIPs. Then the isolated assignment of each UIP is a sufficient condi-

tion for causing the same conflict.

**Example 3.5.** To illustrate the application of UIPs, let us consider again the implication sequence of Figure 3.3-b (see page 66). The set of dominators of $\kappa$ with respect to $x_1$ is $\{ (x_1, 0), (x_8, 0) \}$. The assignment $x_8 = 0$ is a sufficient condition to trigger an implication sequence leading to the same conflict. Hence, clause $\omega_1 = (x_8 + \neg z_1)$ identifies an implicate of the consistency function. On the other hand the node assignments on $x_1$, $w_1$ and $w_2$ imply the assignment of $x_8$. Hence, we can create another clause, $\omega_2 = (x_1 + w_1 + w_2 + \neg x_8)$, which states that the assignments $x_1 = w_1 = w_2 = 0$ imply $x_8 \leftarrow 0$. This fact is clear from the circuit structure and derivable from the clause database with BCP. However, the same clause also states that the assignments $x_8 = 1$ and $w_1 = w_2 = 0$ imply $x_1 \leftarrow 0$, which is no longer derivable from the clause database with BCP. We can thus conclude that $\omega_1$ and $\omega_2$ represent two implicates of the consistency function that can potentially provide additional implications in the presence of partial node assignments. □

In the following we assume an implication sequence leading to a conflict and a set of UIPs $U = \{ (u_1, \nu(u_1)) , \ldots, (u_k, \nu(u_k)) \}$. Let $(u, \nu(u)) \in U$ and let $(x, \nu(x))$ be an assignment that is part of the implication sequence triggered by $(u, \nu(u))$. Then, the causes of the assignment $(x, \nu(x))$ given $(u, \nu(u))$ are defined as follows:

$$causesof(x, u) = \begin{cases} (u, \nu(u)), & \text{if } (x = u) \\ \Lambda(x) \cup \left[ \bigcup_{(y, \nu(y)) \in \Sigma(x)} causesof(y, u) \right], & \text{otherwise} \end{cases} \tag{3.10}$$

Hence, *causesof*$(x, u)$ identifies a set of assignments that imply the assignment $(x, \nu(x))$, restricted to assignment $(u, \nu(u))$ as the trigger of the implication sequence. Given the definition of *causesof*$(x, u)$, the following conflicting clauses are created and added to the clause database:

$$
\begin{aligned}
\omega_1 &= \{x^{\nu(x)} \,|\, (x, \nu(x)) \in causesof(\kappa, u_k)\} \\
\omega_2 &= \{x^{\nu(x)} \,|\, (x, \nu(x)) \in causesof(u_k, u_{k-1})\} \cup \left\{u_k^{\overline{\nu(u_k)}}\right\} \\
&\cdots \\
\omega_k &= \{x^{\nu(x)} \,|\, (x, \nu(x)) \in causesof(u_2, u_1)\} \cup \left\{u_2^{\overline{\nu(u_2)}}\right\}
\end{aligned}
\tag{3.11}
$$

$\omega_1$ states that the assignments in *causesof*($\kappa$, $u_k$) are a sufficient condition for the same conflict to be identified. $\omega_2$ states that the assignments in *causesof*($u_k$, $u_{k-1}$) cannot imply the assignment of $u_k$ to a logic value other than $\nu(u_k)$. The structure of the implication sequence is used to associate the causes of the conflict with each UIP. Hence, the conflicting clauses provided by (3.11) are necessarily stronger than the ones provided by (3.8). Note that clauses $\omega_2$ through $\omega_k$ establish sufficient conditions for the identification of conflicts that were not actually identified. The motivation for these conditions is that they provide implications that otherwise would not be derivable by the deduction engine.

**Example 3.6.** For the previous example, conflicting clause $\omega_2 = (x_1 + w_1 + w_2 + \neg x_8)$ is associated with a conflict that was not identified. However, if $w_1 = w_2 = 0$ and $x_8 = 1$, then it is clear from Figure 3.3-a that $x_1$ cannot assume value 0 and thus it must assume value 1. Furthermore, this implication is not derivable from the original clause database with BCP, but adding $\omega_2$ to the clause database makes such implication explicit. □

In some cases we may have two UIPs, $u_i$ and $u_{i+1}$, such that there is only one implication path between ($u_i$, $\nu(u_i)$) and ($u_{i+1}$, $\nu(u_{i+1})$). In this situation, either assignment ($u_i$, $\nu(u_i)$) or ($u_{i+1}$, $\nu(u_{i+1})$) implies the other assignment, and so all implications identified by the conflicting clause created with (3.11) for ($u_i$, $\nu(u_i)$) and ($u_{i+1}$, $\nu(u_{i+1})$) are already derivable with the current clause database. Consequently, each pair of UIPs connected by only one implication path does *not* contribute with a conflicting clause to the clause database. Creating this conflicting clause is avoided by removing ($u_i$, $\nu(u_i)$) from set $\boldsymbol{U}$.

There can be situations where several asserted assignments, $W = \{ (w_1, \nu(w_1)) , \ldots, (w_m, \nu(w_m)) \}$, create an implication sequence that leads to a conflict. In such situations, ($u_1$, $\nu(u_1)$)

does not correspond to the trigger node, because all assignments in $W$ are actually responsible for triggering the implication sequence. As a result, (3.11) can be completed with an additional conflicting clause $\omega_{k+1}$. The derivation of $\omega_{k+1}$ requires extending (3.10) to the following form:

$$causesof(x, W) = \begin{cases} (x, \nu(x)), & \text{if } ((x, \nu(x)) \in W) \\ \Lambda(x) \cup \left[ \bigcup_{(y, \nu(y)) \in \Sigma(x)} causesof(y, W) \right], & \text{otherwise} \end{cases} \tag{3.12}$$

and consequently $\omega_{k+1}$ is given by:

$$\omega_{k+1} = \{ x^{\nu(x)} \,|\, (x, \nu(x)) \in causesof(u_1, W) \} \cup \left\{ u_1^{\overline{\nu(u_1)}} \right\} \tag{3.13}$$

that is analogous to $\omega_2$ through $\omega_k$ in (3.11) but where the assignment of $u_1$ is triggered by a set of node assertions.

**Example 3.7.** Even though conflict analysis has been exemplified exclusively with circuit examples, it also finds application in more general clause databases. Moreover, the structure revealed by UIPs can also be found in those general clause databases. An example of such a clause database is shown in Figure 3.4. The decision assignment $x_1 \leftarrow 1$ triggers an implication sequence that yields a conflict. Two UIPs are identified that are associated $x_1$ and $x_4$. Hence, two conflicting clauses are created and added to the original clause database. □

### 3.4.3 Maintenance of the Clause Database

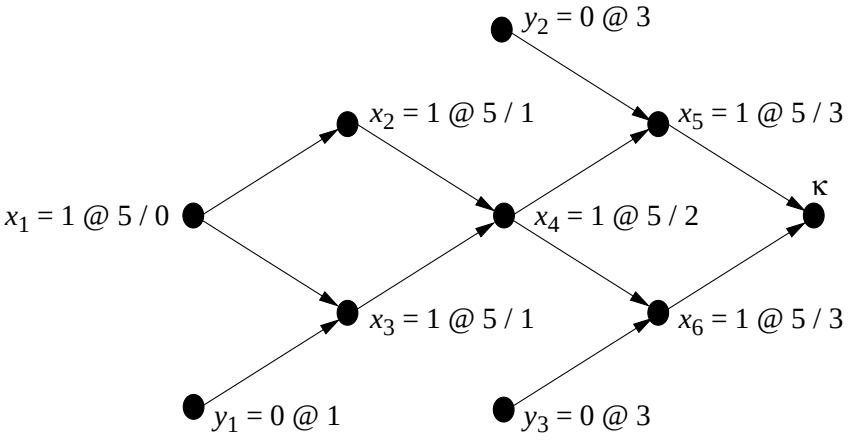
As shown above, conflict analysis involves creating conflicting clauses which are then added to the clause database. These clauses are used with two distinct purposes:

1. They imply additional assignments, which we refer to as *failure-driven assertions*. For each conflicting clause that is also a unit clause at a given stage of the search process, an implication is derived, which attempts to prevent a known conflict from occurring and which could not be prevented by the sole application of the deduction engine.

$$\varphi = (\neg x_1 + x_2) \cdot (\neg x_1 + x_3 + y_1) \cdot (\neg x_2 + \neg x_3 + x_4) \cdot$$
$$(\neg x_4 + x_5 + y_2) \cdot (\neg x_4 + x_6 + y_3) \cdot (\neg x_5 + \neg x_6) \cdot$$

- Current assignments: $y_1 = 0$ @ 1, $y_2 = 0$ @ 3, $y_3 = 0$ @ 3
- Decision assignment: $x_1 \leftarrow 1$

(a) Initial conditions




(b) Implication graph

- Conflicting clauses: $\omega_1 = (y_2 + y_3 + \neg x_4)$ and $\omega_2 = (y_1 + \neg x_1 + x_4)$

(c) Computed conflicting clauses

**Figure 3.4:** CNF-based example

2. Whenever one of these clauses becomes unsatisfied, a conflict is identified, which corresponds to conflicting conditions previously inferred during the search process. As a result, the current partial node assignment, created by the search process, is said to lead to a conflict, because it shares with some other partial node assignment the same assignments that were identified as responsible for yielding a conflict (or set of conflicts). This form of early identification of conflicting conditions is referred to as *conflict-based equivalence*.

As the search process evolves, the number of added conflicting clauses may become significantly large. In addition, some of these clauses can either be removed from the clause database, as is the case with subsumed clauses, or be simplified by combining pairs of clauses. In general, subsumption operations are computationally expensive. From Section 2.5.2 (see page 44), the cost of a subsumption operation is $O(N \cdot |\varphi|^2)$, where $|\varphi|$ is the number of clauses in $\varphi$. However, if we

allow subsumption operations only with respect to each added conflicting clause, then computational cost is reduced to $O(N \cdot |\varphi|)$.

Furthermore, in some situations pairs of clauses can be combined and simplified. If two clauses, $\omega_1 = \{l_{1,1}, \ldots, l_{1,j}\}$ and $\omega_2 = \{l_{2,1}, \ldots, l_{2,j}\}$ of the clause database contain the same number of literals and differ only in two literals, $l_{1,i}$ and $l_{2,i}$, such that $l_{1,i} = \neg l_{2,i}$, then a new clause $\omega_3 = \omega_1 - \{l_{1,i}\} = \omega_2 - \{l_{2,i}\}$ can replace $\omega_1$ and $\omega_2$ in the clause database. In this situation we say that $\omega_3$ results from *merging* $\omega_1$ and $\omega_2$.

**Theorem 3.3.** With the definitions of $\omega_1$, $\omega_2$ and $\omega_3$ given above, $\omega_1 \cdot \omega_2 \leftrightarrow \omega_3$. Clause $\omega_3$ is an implicate of the consistency function $\xi$. Moreover, $\omega_1$ and $\omega_2$ can be removed from the clause database if $\omega_3$ is added to the clause database.

**Example 3.8.** Let $\omega_1 = (x + y + \neg z)$ and $\omega_2 = (x + \neg y + \neg z)$. Then the two clauses can be merged and replaced by a new clause $\omega_3 = (x + \neg z)$. □

Merging operations can be applied whenever a new conflicting clause is added to the clause database. The computational cost is the same as that of the restricted subsumption operation, i.e. $O(N \cdot |\varphi|)$.

Despite the above restrictions, subsumption and merging operations still incur in significant computational overhead. Consequently, and by default, these operations are not part of the diagnosis engine described below in Section 3.6. Nevertheless, they can be optionally applied.

## 3.5 Deduction Engines

The purpose of this section is to describe deduction engines. The simplest deduction engine, referred to as the *basic deduction engine*, implements Boolean constraint propagation but it also updates the structures associated with the search process. Other deduction engines are described. For example, in conflict diagnosis it is often useful to analyze multiple conflicts, and thus we describe a simple extension of BCP that identifies multiple conflicts. Other, more complex deduction engines, can be devised, which are based on the identification of conflicts and creation of implicates of the clause database.

```
// Global variables:   Implication graph I_C
// No input or output arguments
// Return value:       CONFLICT or SUCCESS
//
Deduce()
{
   while (clauses unsatisfied or unit clauses in φ) {
      if (exists unsatisfied clause ω) {
         define new conflict node κ;
         define α(κ) as the elements of ω;
         return CONFLICT;
      }
      if (exists unit clause ω with free literal l = x^i ) {
         define α(x) as the set of the elements of ω other than x;
         compute δ(x) = c and ι(x);                    // Using (3.3)
         x ← 1 ⊕ i ≡ ¬i ;                              // i.e. l is set to 1
      }
   }
   return SUCCESS;
}
```

**Figure 3.5:** Description of the basic deduction engine

### 3.5.1 Basic Deduction Engine

The basic deduction engine is shown in Figure 3.5. It implements Boolean constraint propagation, as is described in Figure 2.6 (see page 39), but modified to maintain additional information required by the search process. For each assigned node an antecedent set is identified, which then implicitly defines the antecedent assignment. Furthermore, in the presence of a conflict, a conflict node $\kappa$ is defined. The basic deduction engine causes no modifications to the clause database.

Let `Deduce()` compute the assignment set $\boldsymbol{A}_c^f = \boldsymbol{A}_c$. Then, from Theorem 2.1 (see page 38), and with $\boldsymbol{A}_c^i \equiv \boldsymbol{A}_{c-1} \cup \{(x, v_x)\}$, we have $\boldsymbol{A}_{c-1} \subseteq \boldsymbol{A}_c$. Furthermore, noting the definition of procedure `Erase()` (see page 64) that is applied to every identified conflict, and iterated application of Theorem 2.1 yields:

**Theorem 3.4.** Assume a sequence of active decisions and let $c$ be the current decision level. Then

$\boldsymbol{A}_0 \subseteq \boldsymbol{A}_1 \subseteq \ldots \subseteq \boldsymbol{A}_c$.

**Implementation**

A direct implementation of the proposed deduction engine can be used. The most relevant implementation detail is that every time a variable $x$ is assigned, only the clauses with literals on $x$ need be examined for implying additional assignments. Consequently, we can immediately guarantee a run time that is linear in the size of the clause database. Furthermore, by maintaining a counter of unassigned literals, a clause just needs to be examined when it is known to be unit; this reduces the overhead of clause manipulation.

For combinational circuits, and given the assumption of gates with bounded fanin, the identification of unit clauses or unsatisfied clauses requires constant time for each assigned node. Since the number of assigned nodes is in the worst case $O(N)$, the worst-case run time of the deduction engine at each decision level is $O(N)$, which is also the amortized[4] worst-case run time over all decision levels (assuming that no conflicts are identified). If the number of clauses in the clause database is allowed to grow (due to information derived by the search process), then this bound on the run time of the deduction engine no longer holds. Furthermore, the bound on the amortized worst-case time does not hold if conflicts are detected.

The order in which clauses are ordered for implying assignments is relevant, and two simple ordering mechanisms can be envisioned. In the first one, unit clauses are kept in a FIFO (first-in-first-out) queue, which defines the processing order and causes implications to evolve in a *breadth-first* manner. The second mechanism keeps the unit clauses in a FILO (first-in-last-out) queue that causes implications to evolve in a *depth-first* manner. The practical implementation of `Deduce()` uses breadth-first implications, mainly because they ensure the shortest implication paths from the trigger node to a conflict. Although choosing breadth-first implications is heuristic, its justification is that shorter implication sequences are *likely* to facilitate the operation of the diagnosis engine.

### 3.5.2 Deduction Engine with Multiple Conflicts

An implication sequence that yields one conflict can in some cases yield other conflicts.

4. A definition of amortized complexity can be found in [35, pp. 356-377].

```
// Global variables:   Implication graph I_C
// Return value:       CONFLICT or SUCCESS
//
Deduce_MC()
{
   status = SUCCESS; j = 1;
   while (exists unit clause) {
      if (exists j^th unsatisfied clause ω) {
         define new conflict node κ[j];                    // Define κ_j
         define α(κ[j]) as the elements of ω;
         status = CONFLICT; j++;
      }
      if (exists unit clause ω with free literal l = x^i ) {
         define α(x) as the set of the elements of ω other than x;
         compute δ(x) = c and ι(x);                        // Using (3.3)
         x ← ¬i;                                  // i.e. l is set to 1
      }
   }
   return status;
}
```

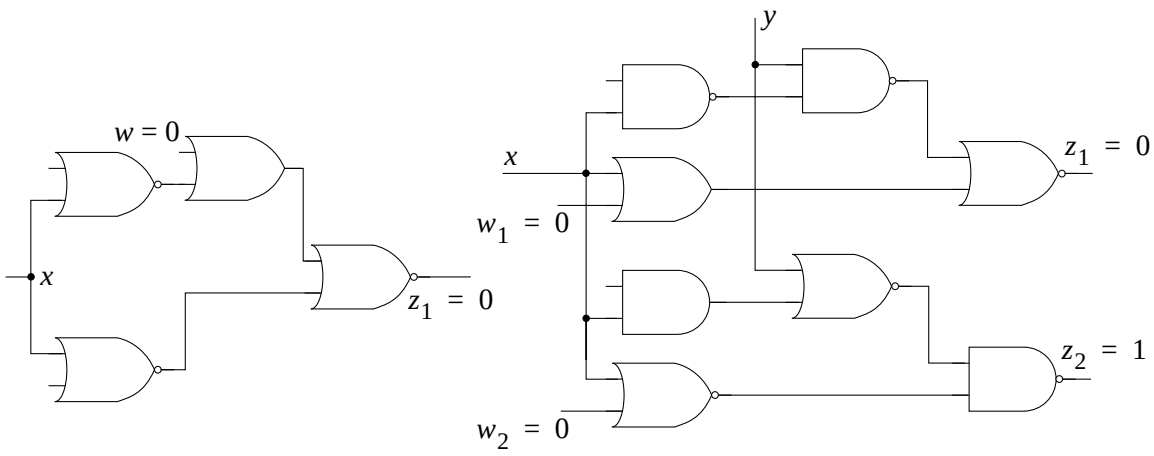
**Figure 3.6:** Description of the deduction engine for detecting multiple conflicts

For conflict diagnosis it is often useful to be able to choose, among different conflicts, the one that leads to more pruning of the search. The deduction engine of Figure 3.5 can be readily modified to identify multiple conflicts. Basically, we allow for any number of conflict nodes κ to be added to the implication graph. These conflict nodes can then be used by the diagnosis engine for conflict analysis.

A deduction engine that handles multiple conflicts is shown in Figure 3.6 and is referred to as `Deduce_MC()`. It basically implements BCP, but relaxation on unit clauses can continue despite unsatisfied clauses being detected. Each unsatisfied clause is recorded and associated with a different conflict node.

### 3.5.3 Advanced Deduction Engines

The basic deduction engine does not identify all logical consequences of a decision assignment. This results from Boolean constraint propagation (BCP) being logically incomplete.




(a) The assignment $x = 1$ is inconsistent (b) The assignment $x = 1$ is also inconsistent

**Figure 3.7:** Boolean constraint propagation is logically incomplete

For some instances of SAT, deduction engines that can identify more logical implications than BCP can be particularly useful. The purpose of this section is to describe a hierarchy of such deduction engines.

**Example 3.9.** Two examples illustrating the drawbacks of BCP are shown in Figure 3.7. For the example circuit of Figure 3.7-a, $z_1 = 0$ does not directly imply other assignments. However, the only consistent value for $x$ is 0, which is not identified by BCP. For example, if we test the assignment $x = 1$ and derive implications, then a conflict with $z_1$ is identified. The derived conflicting assignment set yields the conflicting clause $(\neg x + w + z)$, which implies $x \leftarrow 0$ and identifies the causes of such assignment (i.e. antecedents of $x$ are $w$ and $z$).

A more complex example is shown in Figure 3.7-b. The assignment $x = 0$ is not consistent. However, this fact only becomes apparent after considering both logic value assignments to node $y$. The assignments $x = 0$ and $y = 0$ trigger an implication sequence that yields a conflict identified by conflicting clause $\omega_{0,0} = (x + y + w_2 + \neg z_2)$, whereas $x = 0$ and $y = 1$ cause a conflict identified by clause $\omega_{0,1} = (x + \neg y + w_1 + z_1)$. Both clauses denote implicates of the consistency function. If the clause database is satisfiable for some complete variable assignment, then both clauses will be satisfied, *and* the consensus $c(\omega_{0,0}, \omega_{0,1}, y)$ will also be satisfied. Conversely, whenever $c(\omega_{0,0}, \omega_{0,1}, y)$ evaluates to 0, then either $\omega_{0,0}$ or $\omega_{0,1}$ evaluates to 0. The resulting clause $\omega_0 = c(\omega_{0,0}, \omega_{0,1}, y) = (x + w_1 + w_2 + z_1 + \neg z_2)$ is also an implicate of the consistency function $\xi$ and

can be added to the clause database. Furthermore, $\omega_0$ implies the assignment $x = 1$ because all other literals in the clause are assigned value 0. The derived implicate $\omega_0$ is minimal with respect to the set of variables that is used in its derivation, i.e. $\omega_0$ cannot be further simplified just by independent analysis of *x* and *y*. We observe that this implicate is not necessarily a prime implicate of $\xi$, because it can be combined with implicates derived from other sets of variables. The two examples of Figure 3.7 also suggest a generic procedure for implementing arbitrary complex deduction engines, which we now detail. □

We propose to use Boolean constraint propagation as the basic building block of a hierarchy of deduction engines, each with increasing deduction ability at the cost of added complexity in processing each decision assignment. For a given number *k*, `Deduce_`*k*() ensures that at each decision level *all* combinations of assignments of *all* subsets of *k* unassigned variables are tested for consistency using `Deduce`(). The algorithm for prime implicate generation (from Figure 2.7 on page 44) is then used to simplify the conditions associated with any identified conflicts. This may result in nodes being asserted, which in turn is used to imply further assignments. The need to use the algorithm for prime implicate generation was previously motivated with the example circuit of Figure 3.7-b. If `Deduce_`*k*() does not terminate in a conflict, then there is at least one combination of assignments, for each subset of *k* nodes, which does not cause a conflict. For this reason, we refer to operation of `Deduce_`*k*() as *k-consistency*[5].

The generic implementation of `Deduce_`*k*() is described in Figure 3.8, and it invokes procedure `Deduce`() given in Figure 3.5. As suggested above, for every set $\gamma$ of *k* unassigned nodes, each possible logic value assignment is applied to the variables. For each assignment, Boolean constraint propagation is used to imply assignments and identify any conflicts. Whenever a conflict is identified, a new conflicting clause is added to a dedicated clause database $\varphi_\gamma$. After all logic value assignments to the nodes in $\gamma$ are processed, the prime implicates of $\varphi_\gamma$ are computed, and a temporary clause database $\varphi_k$ for all sets $\gamma$ is updated. (Note that while computing the prime implicates of $\varphi_\gamma$, consensus operations can only be done with respect to the *k* variables in $\gamma$, since the other variables in the conflicting assignment set assume fixed values, which will not be modified

[5] We note that this definition is markedly different from *k*-consistency in constraint satisfaction problems (see for example [169, pp. 55-57]). In the present and remaining chapters, and unless otherwise stated, *k*-consistency refers to the application of `Deduce_`*k*() as described in this section.

```
// Global variables:   Implication graph I_C
//                     Clause database φ
// Return value:       CONFLICT or SUCCESS
//
Deduce_k()
{
   if (Deduce() == CONFLICT) return CONFLICT;
   status = SUCCESS; φ_k ← ∅;
   Let Γ be the set of all sets of k unassigned nodes;
   for (each set γ ∈ Γ and while status != CONFLICT) {
      status = CONFLICT; φ_γ ← ∅;
      for (each distinct logic value assignment to the nodes in γ) {
         if (Deduce() == CONFLICT) {
            ω = Create_Conflicting_Clause();    // With (3.6), (3.8)
            φ_γ ← φ_γ ∪ { ω };
         } else status = SUCCESS;
         Erase_Last_Assignments();
      }
      Generate_Prime_Implicates (φ_γ);      // See Figure 2.7 on page 44
      φ_k ← φ_k ∪ φ_γ;
   }
   if (SIMPLIFY_φ_k)
      Generate_Prime_Implicates (φ_k);   // It is optionally invoked
   φ ← φ ∪ φ_k;
   return Deduce();          // Create κ or derive more assignments
}
```

**Figure 3.8:** Advanced deduction engine — without relaxation

between assignments to the variables in $\gamma$.) After all sets $\gamma$ are processed, the clause database $\varphi$ is updated with $\varphi_k$. Observe that updating the clause database is deliberately done *after* all sets $\gamma$ are processed. This solution ensures that the final composition of the clause database is *independent* of the order in which the sets $\gamma$ are processed.

Before adding $\varphi_\gamma$ to $\varphi_k$, the prime implicates of the Boolean function associated with $\varphi_\gamma$ are derived. This entails consensus operations among all variables in $\gamma$. As a result, we can claim that the implicates that are derived are *minimal* in the variables of $\gamma$. This result follows immediately from the implementation of `Generate_Prime_Implicates`(). We note, however, that the implicates due to a given $\gamma$ may be further simplified when combined with the implicates of

```
// Global variables:   Implication graph I_C
//                     Clause database φ
// Return value:       CONFLICT or SUCCESS
//
Deduce_k,R()
{
   do {
      let φ_i denote current φ;
      status = Deduce_k();                          // From Figure 3.8
   } while (φ != φ_i and status != CONFLICT);
   return status;
}
```

**Figure 3.9:** Advanced deduction engine — with relaxation

other subsets. In addition, even though `Generate_Prime_Implicates`() is applied to $\varphi_\gamma$, the resulting clause database does not necessarily contain prime implicates of $\xi$. In general, this should not be the case, since only a subset of the variables associated with $\xi$ is considered. Procedure `Generate_Prime_Implicates`() can be optionally invoked on $\varphi_k$ if the flag SIMPLIFY_$\varphi_k$ is set.

Deduction engines can be further improved. A simple enhancement is to continue invoking `Deduce_`*k*() while more conflicting clauses are added to the clause database. This procedure is referred to as *k*-consistency *with relaxation*, referred to as `Deduce_`*k*,*R*(). Clearly, relaxation introduces some computational overhead, but may also contribute to reducing the search. The procedure for *k*-consistency with relaxation is shown in Figure 3.9. The main loop is iterated while the clause database is modified by `Deduce_`*k*(). The motivation is that if more implications are identified, then they may contribute to identify additional implications or yield a conflict. It is worth noting that `Deduce_`*k*,*R*() identifies no fewer implications than `Deduce_`*k*() if both procedures start from the same partial node assignment.

**Example 3.10.** The application of `Deduce_`*k*() is illustrated with the example circuit of Figure 3.7-b, assuming $k = 2$. As illustrated before, the assignments $x = 0$ and $y = 0$ cause the derivation of conflicting clause $\omega_{0,0} = (x + y + w_2 + \neg z_2)$, whereas $x = 0$ and $y = 1$ cause the derivation of conflicting clause $\omega_{0,1} = (x + \neg y + w_1 + z_1)$. The remaining assignments to $x$ and $y$ yield no conflicting clauses. Invoking procedure `Generate_Prime_Implicates`() then produces

$\omega_0 = (x + w_1 + w_2 + z_1 + \neg z_2)$. Other subsets of nodes of size two are processed and can also contribute with additional implicates. Eventually $\omega_0$ is added to the clause database. Given the assignments of the other literals, $\omega_0$ is a unit clause, and hence it implies the assignment $x = 1$, as intended. We note that other subsets of variables involving $x$ would lead to the same conflicting clause. In this situation, subsumption operations (on $\varphi_k$) ensure that no repeated clauses are added to the clause database.

The example circuit of Figure 3.3 (see page 66) can also be used to illustrate the application of `Deduce_`*k*(), assuming $k = 1$. For this example, after the set of objectives is specified, `Deduce_`1() identifies a conflict for both logic value assignments to $x_8$. Hence the query is proved to be unsatisfiable without any search. `Deduce_`1() also derives the implicate $(\neg z_1 + \neg z_3)$, which states that $z_1$ and $z_3$ cannot be simultaneously 1. □

Let us consider `Deduce_`*k*() as described in Figure 3.8 and `Deduce_`*k*,*R*() as described in Figure 3.9. Then the algorithmic complexity of these procedures is given by the following:

**Theorem 3.5.** Let $\varphi$ be a clause database. Let $|\boldsymbol{I_C}|$ be the current number of vertices of the implication graph, and let $N - |\boldsymbol{I_C}|$ identify the total number of unassigned nodes. Then, the worst-case run time of `Deduce_`*k*(), assuming that SIMPLIFY_$\varphi_k$ does not hold, is bounded by:

$$O\left(\binom{N - |\boldsymbol{I_C}|}{k} \cdot (\|\varphi\| \cdot 2^k + N \cdot k \cdot (3^k)^2)\right) \tag{3.14}$$

The worst-case run time of `Deduce_`*k*,*R*() is bounded by,

$$O\left(\binom{N - |\boldsymbol{I_C}|}{k} \cdot \left[\left[\|\varphi\| + N \cdot \binom{N - |\boldsymbol{I_C}|}{k} \cdot 3^k\right] \cdot 2^k + N \cdot k \cdot (3^k)^2\right] \cdot \left[\binom{N - |\boldsymbol{I_C}|}{k} \cdot 3^k\right]\right) \tag{3.15}$$

For both procedures the worst-case space is bounded by,

$$O\left(\|\varphi\| + N \cdot \binom{N - |\boldsymbol{I_C}|}{k} \cdot 3^k\right) \tag{3.16}$$

and the overall space growth if bounded by $O(N \cdot 3^N)$.

Deduce_*k*() with $k = 0$ corresponds to Deduce(). For $k = 1$ an upper bound on the worst-case run time complexity is $O(\|\varphi\| \cdot (N - |\boldsymbol{I}_C|))$. For $k = N - |\boldsymbol{I}_C|$ the worst-case run time complexity is $O(N \cdot (N - |\boldsymbol{I}_C|) \cdot (3^{N - |\boldsymbol{I}_C|})^2)$. Hence, $k = N - |\boldsymbol{I}_C|$ ensures that all possible combinations of node assignments are examined, and consequently that a query is solved *without search*. However, the complexity of this consistency procedure is prohibitive in most practical examples. 1-consistency leads to a worst-case quadratic time deduction engine (assuming $\|\varphi\| = O(N)$), which in some applications can be useful in reducing the amount of search, even if applied only at decision level 0 (see for example [37, 145, 162]).

With relaxation, $k = 1$ yields a worst-case running time of $O(\|\varphi\| \cdot (N - |\boldsymbol{I}_C|)^2)$. Assuming $\|\varphi\| = O(N)$, then relaxation transforms a quadratic time procedure, i.e. Deduce_1(), into a cubic time one, Deduce_1,*R*().

It is worth noting that *k*-consistency effectively reduces the worst-case size of the decision tree to be traversed. Consider a circuit with $|PI|$ primary inputs, and a *k*-consistency deduction engine. As a result, the largest depth of the search tree, before backtracking, is $|PI| - k$, because the last *k* unassigned primary inputs either are consistent, and a solution can be identified, or are inconsistent, and the search process is forced to backtrack. Consequently, the worst-case size of the decision tree becomes $2^{|PI| - k}$. This reduction on the size of the decision tree is compensated by the added overhead to process each decision. Furthermore, for combinational circuits, the largest *k* of interest is given by $k = |PI|$, since for any primary output objective all combinations of primary input assignments suffice to determine whether a solution can be identified.

Deduce_*k*() ensures that the clause database does not change until all conflicting clauses are derived and simplified. Nevertheless, the actual implementation can allow the clause database to be updated as different subsets are processed. This option provides additional implications without the overhead of full relaxation. The problem is that now the final composition of the clause database depends on the order in which the variables are processed. In addition, the added overhead may increase the run time to some degree.

Although we described a procedure that ensures a given level *k* of consistency for each *k*, we may inquire whether for a fixed *k* one can identify all possible logical consequences of a decision assignment on every clause database. Clearly, this should not be the case, since SAT is an *NP*-

complete problem, and any algorithmic procedure is expected in the worst case to require exponential time in the size of the problem. Accordingly, the following holds:

**Theorem 3.6.** For each deduction engine `Deduce_`*k*() with fixed *k*, it is always possible to construct a clause database for which the identification of all implications requires a deduction engine `Deduce_`*m*(), with $m > k$.

The advanced deduction engine described in Figure 3.8 does not attempt to identify UIPs. As shown before, UIPs contribute to identifying stronger implicates. If the advanced deduction engine identifies UIPs, then the average run time of `Generate_Prime_Implicates`() is reduced, since implicates of smaller size will be created prior to generating prime implicates. For reasonably large *k*, `Deduce_`*k*() based on UIP identification can thus prove useful.

**Other Approaches**

The description of `Deduce_`*k*() and `Deduce_`*k*,*R*() assumes a procedure for generation of prime implicates, which is used to simplify implicates of $\xi$. One can envision other algorithmic solutions *not* based on `Generate_Prime_Implicates`(). For example, one possible solution (for *k*-consistency) is to process all subsets of variables with size *j* ranging from *k* down to 1, in this order. For each *j*, all possible subsets and associated assignments are tested. Any conflicts result in conflicting clauses being added to the clause database. If any of these conflicting clauses can be simplified, then for some size *i*, less than *j*, this fact will become apparent and a new conflicting clause will be created. Eventually, derivable assertions are identified for $j = 1$. Our implementation of `Deduce_`*k*() is preferable for small *k*, because the overhead of procedure `Generate_Prime_Implicates`() is negligible and only one size *k* of subsets is processed.

**Perspective**

Some special cases of the family of deduction engines defined by `Deduce_`*k*() and `Deduce_`*k*,*R*() can be related to deduction procedures proposed by other authors, both in the context of satisfiability algorithms and in other areas. With respect to satisfiability algorithms, the deduction engines entailed by `Deduce_`*k*() and `Deduce_`*k*,*R*() comprise several distinct algorithms proposed by other authors, most notably in test pattern generation [24, 101, 145, 162],

which we review in Chapter V. In backtracking search algorithms there are techniques to predict the best variable to branch upon, which consider sets of assignments to unassigned variables in order to decide the most promising decision variable [11, 134, 178]. Such techniques are commonly referred to as (*multi-level*) *search rearrangement*. *k*-consistency algorithms are used in constraint satisfaction problems (CSPs) to preprocess a given problem prior to searching for a solution [61, 169]. These algorithms also examine combinations of assignments with the objective of reducing the number of acceptable domain values for each variable. Our proposed procedure is distinct in a few significant ways. First, Boolean constraint propagation is applied for every combination of assignments. This is not the case with *k*-consistency in CSPs. For SAT, BCP increases the likelihood of finding conflicts. Second, our procedure generates and manipulates conflicts, using procedures for generating prime implicates, so as to imply further assignments. This technique is apparently new, and for SAT algorithms it is particularly useful for levels of consistency greater than 1. Finally, the procedure can be readily used within the search framework of GRASP. This implies that a search algorithm can be based on *k*-consistency and implement all pruning methods associated with conflict diagnosis. At the time of this writing, the integration of *k*-consistency with procedures for diagnosing conflicts is still an open problem in CSPs [169, p. 152].

## 3.6 Diagnosis Engines

In this section we describe diagnosis engines. The prime objective of conflict diagnosis is to derive implicates of the consistency function, using the conflict analysis methods described in Section 3.4. These implicates allow implementing the different pruning methods, including conflict-directed backtracking, that requires computing the backtracking decision level whenever backtracking is required. Consequently, conflict diagnosis consists of identification of implicates and computation of backtracking decision levels.

Several diagnosis engines can be devised. We start by describing a basic formulation based on the conflict analysis methods described in Section 3.4. Next, we describe simple extensions that allow computing lower backtracking decision levels, by examining more than one conflict. Another concern is the computational overhead of adding a large number of implicates to the clause database. Accordingly, we propose variations of conflict diagnosis which guarantee bounds

on the growth of the clause database. Finally, we briefly describe extensions to conflict analysis, which are based on specifying conflicting assignments sets in terms of decision assignment sets and simplifying those sets.

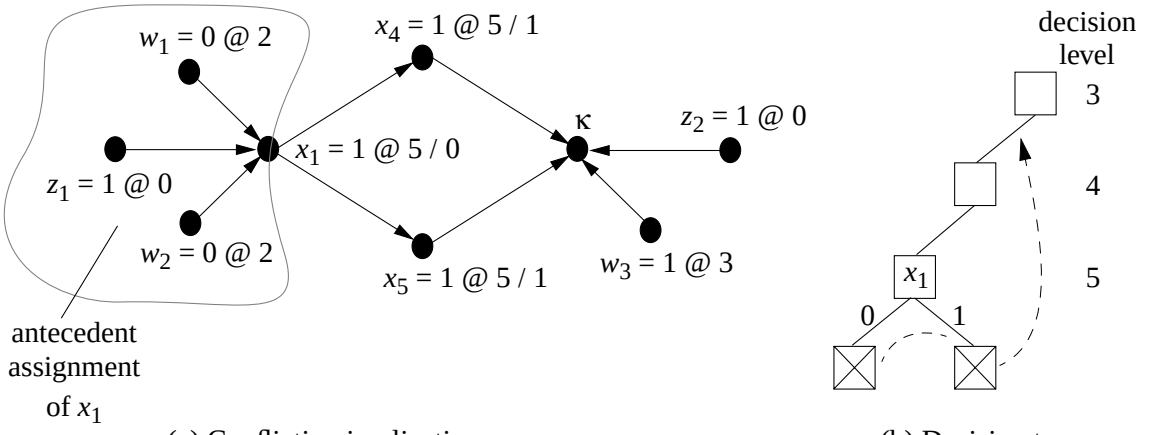
### 3.6.1 Basic Diagnosis Engine

After each conflict is detected, the basic conflict diagnosis engine creates a conflicting clause with the conflict analysis methods of Section 3.4. If the derived clause involves assignments at the current decision level, then failure-driven assertions are defined and the search process proceeds. A different situation occurs whenever all elements of a conflicting assignment set are assigned at decision levels *less* than the current decision level $c$. This situation can only take place when the current conflict results from diagnosing a previous conflict, after which a decision node had been asserted with an antecedent assignment composed of assignments implied at decision levels less than $c$. If all elements of a conflicting assignment set are assigned at decision levels less than $c$, then it is established that the conflicts found are only caused by those lower decision levels, and hence the search process can only find a solution if it backtracks *directly* to the source of the conflicts. The backtracking decision level is identified by the highest decision level of the assignments in the conflicting assignment set:

$$\beta_L = max\{\delta(x) | (x, \nu(x)) \in \boldsymbol{A}_{CS}\} \tag{3.17}$$

$\beta_L$ corresponds to the decision level that is returned as a reference argument by the diagnosis engine, which is invoked in the procedure of Figure 3.2. $\beta_L = c$ means that no backtracking is required, whereas the condition $\beta_L < c - 1$ corresponds to *non-chronological* backtracking. We note that by creating a clause $\omega$ specified by a conflicting assignment set using (3.8), and by backtracking to the decision level $\beta_L$ specified by (3.17), at this decision level a conflict is now defined by $\omega$. This *forced* conflict is used to analyze the implication sequence at decision level $\beta_L$, which can then be used to either assert the decision node or to decide a new backtracking decision level.

**Example 3.11.** For the example circuit of Figure 3.3-a (see page 66), and after diagnosing the conflict due to $x_1 = 0$ (shown in Figure 3.3-b), the assertion $x_1 = 1$ is obtained, with antecedent assignment $\{(w_1, 0), (w_2, 0), (z_1, 1)\}$ . The resulting implication sequence is shown in Figure 3.10-a



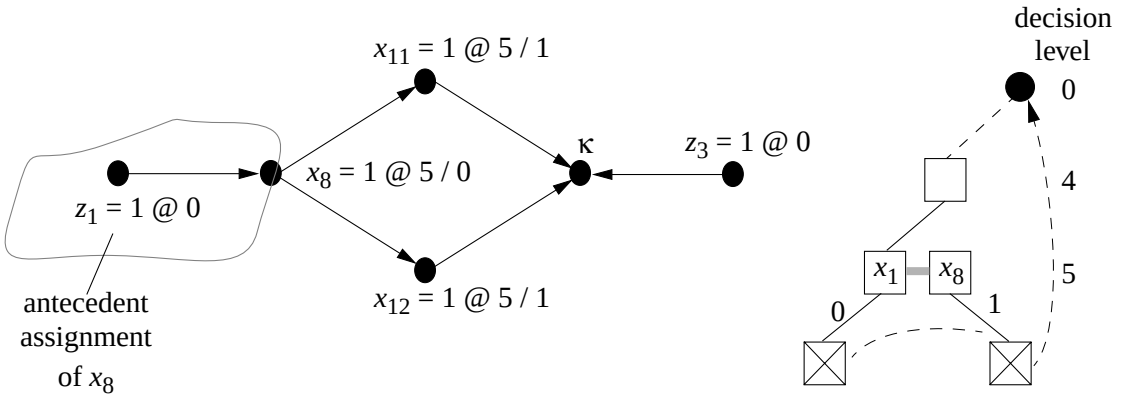

(a) Conflicting implication sequence (b) Decision tree

**Figure 3.10:** Implication sequence and backtracking due to assertion $x_1 = 1$

and results in a conflict with the assignment $z_2 = 1$. From (3.6) the following conflicting assignment set is obtained: $\boldsymbol{A}_{CS} = \{(w_3, 1), (z_2, 1), (w_1, 0), (w_2, 0), (z_1, 1)\}$ . In this situation, all elements of the conflicting assignment set are assigned at decision levels less than 5, which means that there exists a set of node assignments that is in conflict at decision levels less than the current decision level. Thus, the search process needs to backtrack, and from (3.17) the backtracking decision level is evaluated to be 3, due to the assignment of $w_3$ at decision level 3. Consequently, the implication sequences at decision levels 5 and 4 can be erased and the implication sequence at decision level 3 will now force a conflict that must be diagnosed. The result of non-chronological backtracking is illustrated in Figure 3.10-b.

In addition, the conflicting assignment set is used to derive another implicate of the consistency function: $\omega = (w_1 + w_2 + \neg w_3 + \neg z_1 + \neg z_2)$, which is added to the clause database, and which states that $w_1 = w_2 = 0$ and $w_3 = z_1 = z_2 = 1$ are not consistent assignments for the circuit of Figure 3.3-a. Although this fact is not derivable with the Boolean constraint propagation procedure (described in Figure 3.5), it is deduced by the search process. Whenever the same node assignments are specified, ω will cause a conflict, thus avoiding the need to repeat the work of the search process to derive the same conclusion again. Moreover, ω can be used to derive FDAs; if for example $w_1 = w_2 = 0$ and $z_1 = z_2 = 1$, then ω implies $w_3 = 0$, which would not be derivable, assuming BCP, without the added conflicting clause.

After backtracking to decision level 3, ω is unsatisfied and so it causes a conflict node to




(a) Implication sequence (b) Evolution of the search process

**Figure 3.11:** Implication sequence and backtracking due to assertion $x_8 = 1$

be added to the implication graph. From this conflict node the conditions for asserting the decision node are identified. The assertion on the decision node will represent the second branch at decision level 3. □

The non-chronological backtracking procedure described above relies on (1) traversing implication sequences in order to identify the causes of conflicts; and (2) representing the causes of conflicts as conflicting clauses. We refer to this form of non-chronological backtracking as *conflict-directed backtracking*.

Identification of unique implication points (UIPs) can be applied during conflict diagnosis, with the goal of deriving stronger implicates and potentially lower backtracking decision levels.

**Example 3.12.** For the implication sequence of Figure 3.3-b (see page 66), and using (3.11), the clauses derived are $\omega_1 = (x_8 + \neg z_1)$ and $\omega_2 = (x_1 + w_1 + w_2 + \neg x_8)$. After erasing the last implication sequence, $\omega_1$ implies the assignment $x_8 = 1$. (We should note that the second branch at decision level 5 results from an assertion on a node other than the decision node, i.e. $x_8$ instead of $x_1$.) The resulting implication sequence leads to a conflict that is shown in Figure 3.11-a. In this situation the conflicting assignment set is { ($z_1$, 1), ($z_3$, 1) } and the backtracking decision level is 0 from (3.17). Hence, the problem is proved unsatisfiable by backtracking directly to decision level 0. In addition, (3.11) is used to derive the clause $\omega_4 = (\neg x_8 + \neg z_3)$ which states that if $z_3 = 1$, then $x_8$ must be implied to 0. The evolution of the search process when unique implication points are taken into consideration is shown in Figure 3.11-b. For this example, UIPs allow deriving stronger

| without UIPs | with UIPs |
|---|---|
| $\omega_1 = (x_1 + w_1 + w_2 + \neg z_1)$ | $\omega_1 = (x_8 + \neg z_1)$ |
| $\omega_2 = (w_1 + w_2 + \neg w_3 + \neg z_1 + \neg z_2)$ | $\omega_2 = (x_1 + w_1 + w_2 + \neg x_8)$ |
| | $\omega_3 = (\neg x_8 + \neg z_3)$ |
| | $\omega_4 = (\neg z_1 + \neg z_3)$ |

**Table 3.1:** Comparison of conflicting clauses

backtracking conditions that prove unsatisfiability with two conflicts, whereas without UIPs the two conflicts found only allow backtracking to decision level 3 (see Figure 3.10-b). In Table 3.1, the conflicting clauses derived for the example of Figure 3.3-a, with and without the identification of UIPs, are shown. Note that $\omega_4 = (\neg z_1 + \neg z_3)$ is created with (3.8) from $\boldsymbol{A}_{CS}$. For this example, the use of UIPs permits the derivation of more conflicting clauses that are also stronger than the clauses derived without UIPs. □

As the above example suggests, the identification of unique implication points helps in identifying more implicates of the consistency function, which are stronger than the implicates derived without the identification of UIPs. A set of *k* UIPs partitions a conflicting assignment set into *k* assignment sets, each of which defines a conflicting clause. As a result, each derived conflicting clause necessarily contains no more literals than the clause associated with the original conflicting assignment set, and thus represents a stronger implicate of the consistency function. UIPs may also prove useful in identifying tighter conditions when deciding the backtracking decision level.

Even though the implicates of $\xi$ derived with UIPs are necessarily stronger, we note that the computed backtracking decision level $\beta_1$ may be greater than it would be without UIPs, $\beta_2$, because a different conflict may identified. However, we note that in this situation a sequence of conflicts will eventually force backtracking to the lower decision level (i.e. $\beta_2$), due to the fact that the clause database has been updated with conflicting clauses, that will imply assignments (from failure-driven assertions) and cause conflicts until the conflict forcing backtracking to $\beta_2$ is identified.

The pseudo-code for the diagnosis engine (invoked from the top-level search algorithm) is

```
// Global variables:   Implication graph I_C
//                     Clause database φ
// Input variable:     Current decision level c
// Output variable:    Backtracking decision level β_L
// Return value:       CONFLICT or SUCCESS
//
Diagnose (c, &β_L)
{
   ω = Create_Conflicting_Clause();       // Using (3.6) and (3.8)
   Update_Clause_Database (ω);            // Add clause to database
   if (REDUCE_DATABASE)                   // Subsume/merge clauses
      Subsume_Merge_Clauses (ω);
   β_L = Compute_Max_Level();             // Using(3.6) and (3.17)
   if (β_L != c) {
      define new conflict node κ;      // Set up new conflict node
      define α(κ) as the elements of ω;
      return CONFLICT;
   }
   return SUCCESS;
}
```

**Figure 3.12:** Description of the basic diagnosis engine

shown in Figure 3.12, and it illustrates the main features of basic conflict diagnosis in GRASP. The procedure basically implements the steps described in the previous sections to compute conflicting clauses, update the clause database, identify the need to backtrack and compute the backtracking decision level. Subsumption and merging operations can be optionally applied, provided REDUCE_DATABASE is set to true. Even though not shown, the above procedure can be easily modified for computing implicates with unique implication points. Because conflict diagnosis updates the clause database with conflicting clauses, the search algorithm is able to implement conflict-directed backtracking, failure-driven assertions and conflict-based equivalence.

**Implementation**

The fundamental aspect of conflict diagnosis is the definition of conflicting assignment sets. We note that the deduction engine generates all the antecedent assignment information required to create conflicting assignment sets. Consequently, a breadth-first traversal of the implication graph, from the conflict node, through nodes assigned at decision level *c*, and terminating at

the trigger node, is sufficient for constructing the conflicting assignment set. We further note that the overhead of this traversal is asymptotically *no* worse than the derivation of the implication sequence. Furthermore, overhead is analogous, since conflicting assignment set identification is solely based on tracing implication sequences.

UIPs are identified in the same traversal of the implication sequence that is used to build the conflicting assignment set, thus reducing overhead considerably. A levelized breadth-first traversal[6], on the *implication level* of each assigned node, is used to define the global conflicting assignment set, and to identify UIPs; dominators of the implication subgraph correspond to stages of the levelized traversal when the traversal width is set to 1. Thus, UIPs and associated conflicting clauses are identified in time linear in the size of the clause database.

### 3.6.2 Reducing the Backtracking Decision Level

The purpose of this section is to describe two methods that can be used to reduce the backtracking decision level computed with the basic diagnosis engine. Given that the backtracking decision level depends on which conflicts are diagnosed, the proposed methods diagnose sets of conflicts in a number of different ways.

#### 3.6.2.1 Iterated Conflicts

We can envision a simple extension to the identification of unique implication points. Whenever it is necessary to backtrack, a conflicting clause is created which accounts for all the elements in the conflicting assignment set. Nevertheless, assuming UIPs have been identified, we may be able to generate other conflicts that yield lower backtracking decision levels.

**Example 3.13.** The example circuit shown in Figure 3.13 illustrates how identification of conflicts can be iterated to reveal more aggressive backtracking decision levels. Let the current decision level be 5, and assume decision assignment $x_1 = 0$. The resulting conflict yields conflicting clause $\omega_1 = (x_2 + \neg z_1)$. Consequently, the second branch at decision level 5 corresponds to the asserted assignment $x_2 = 1$, which causes a conflict with $z_2$, yielding conflicting clause $\omega_2 = (x_3 + w_1 +$

[6] A levelized breadth first traversal visits nodes in breadth-first manner but using a chosen level order [155]. A width is defined which measures the number of nodes to be visited.

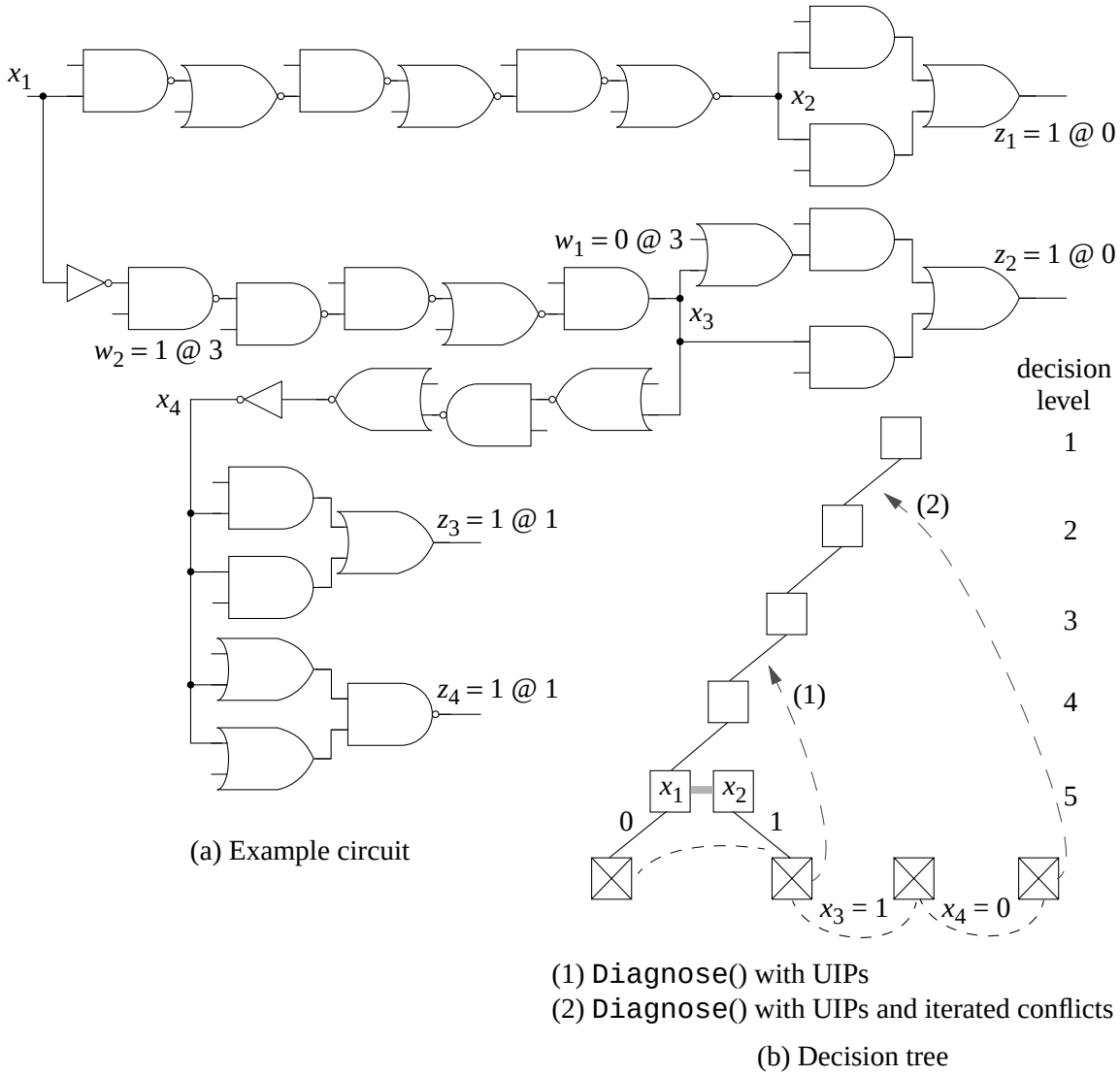



**Figure 3.13:** Example of iterated conflict identification

$\neg z_2$). The basic diagnosis engine would then create conflicting clause $\omega_3 = (\neg z_1 + \neg z_2 + w_1 + \neg w_2)$ and decide the backtracking decision level to be 3. However, for this particular example, other conflicts can be generated, which improve the backtracking decision level.

Let us assume that instead of creating $\omega_3$ and backtracking, the search process iterates assertions at decision level 5. Therefore, $\omega_2$ implies the assignment $x_3 = 1$, which causes a conflict with $z_4$ and yields conflicting clause $\omega_4 = (\neg x_4 + \neg z_4)$. (Observe that this conflict is detected before the conflict with $x_2$, assigned due to $\omega_1$, is identified. Further note that clause $(w_1 + \neg w_2 + \neg z_4 + \neg z_3)$ would be created if backtracking was decided.) Moreover, $\omega_4$ implies the assignment

$x_4 = 0$, which causes a conflict with $z_3$ and yields conflicting clause $\omega_5 = (x_4 + \neg z_3)$.

Note that the last conflict can be used to create the conflicting clause $\omega_6 = (\neg z_3 + \neg z_4)$, which sets the backtracking decision level at 1. Further note that if the search process backtracks to decision level 3, the conflicting clause implying the assignment of $x_3$ to 1 is no longer a unit clause. Hence, the pair of conflicts between $z_3$ and $z_4$ is not revealed and the search process does not necessarily backtrack further. □

As the above example suggests, after backtracking conditions are identified, conflicts are iterated while a different conflict is found and the computed backtracking decision level does not increase. Eventually, either a known conflict is revisited or the backtracking decision level increases, in which case no more conflicts are iterated. The backtracking decision level is then the minimum of the computed backtracking decision levels. Besides computing a lower backtracking decision level, iterated conflicts reveal additional implicates of the clause database that can contribute to pruning the search.

For combinational circuits, the iterated identification of conflicts can prove useful whenever the size of the j-frontier is large, since this facilitates finding distinct conflicts, and whenever conflicts result from long implication sequences, since this facilitates finding other conflicts before identifying known conflicts. However, the worst-case time complexity is quadratic in the size of the clause database, and the existence of large j-frontiers is application-dependent. As a result, the iterated identification of conflicts should only be optionally applied and reserved for those specific circuit structures that create large j-frontiers. In case iterated conflicts are applied, we can limit the number of iterated conflicts to a fixed value *m*, thus ensuring that conflict diagnosis is performed in time linear in the size of the clause database.

#### 3.6.2.2 Multiple Conflicts

As mentioned earlier in Section 3.5.2, implication sequence can yield multiple conflicts. In this section we show that manipulation of multiple conflicts can be used to compute lower backtracking decision levels. Let $\{\kappa_1, \kappa_2, \ldots, \kappa_m\}$ be the set of conflicts identified by a given implication sequence using `Deduce_MC`(), described in Figure 3.6 on page 77. Equation (3.6) is used to associate a conflicting assignment set $\boldsymbol{A}_{CS}^{i}$ with each conflict node $\kappa_i$. Implicates of the consis-

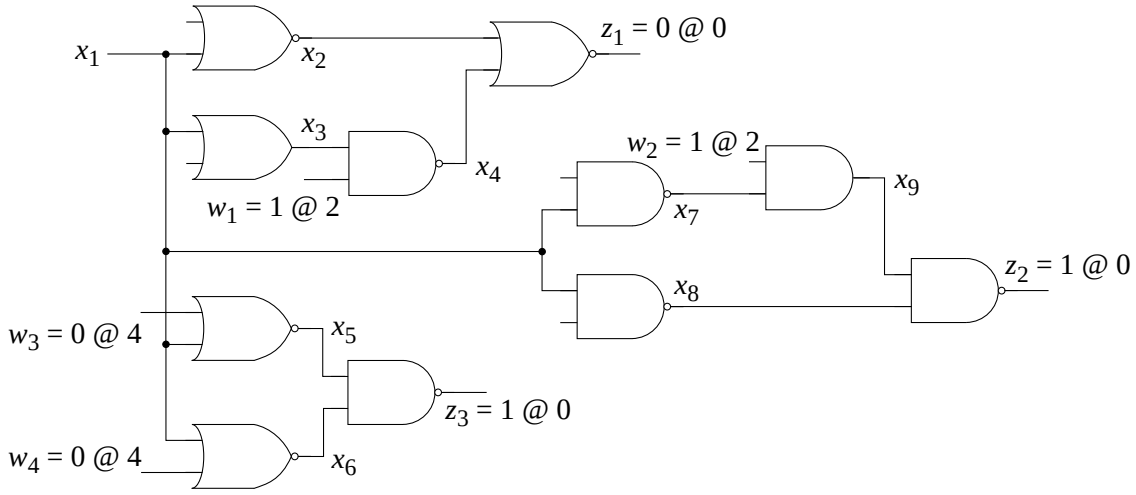
tency function are created with (3.8) or with (3.11), in which case a different set of UIPs is defined for each $\kappa_i$. By considering multiple conflicts, a larger number of conflicting clauses can be created and added to the clause database. The backtracking decision level is computed according to:

$$\beta_L = \min_{1 \le i \le m} [max\{\delta(x) | (x, \nu(x)) \in \boldsymbol{A}_{CS}^{i}\}] \tag{3.18}$$

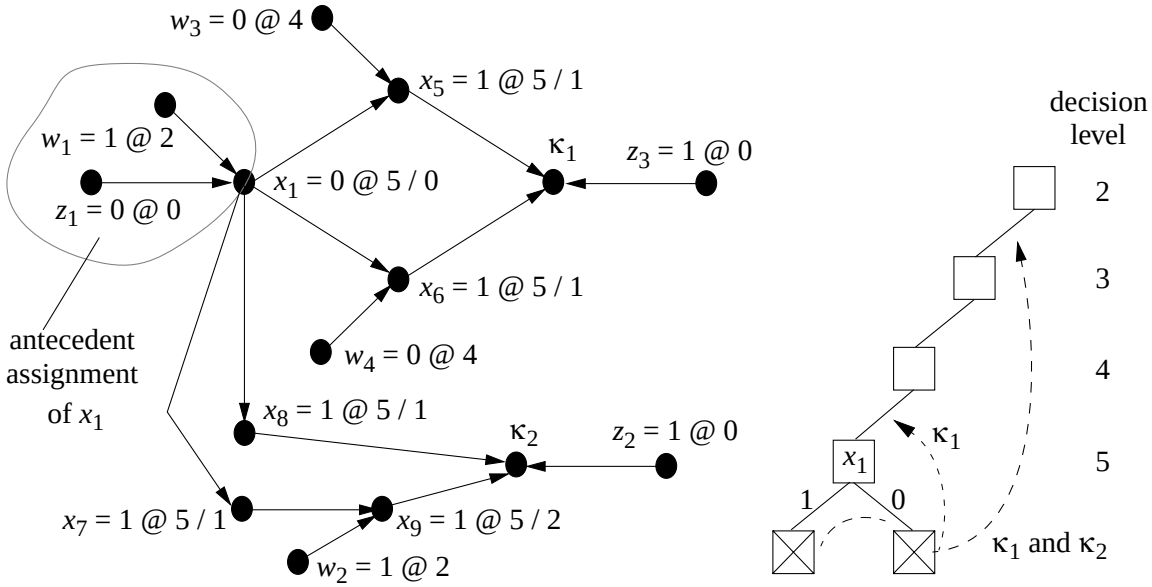
Hence, the existence of multiple conflicts is used to find a minimum backtracking decision level among the possible backtracking decision levels. From the results of Appendix A we can conclude that the conflicting clause $\omega_i$ for each $\kappa_i$ is a valid implicate of the consistency function $\xi$. Furthermore, each conflicting clause identifies an independent and sufficient set of assignments for a conflict to be detected; thus the backtracking decision level given by (3.18) is correct and completeness is guaranteed.

**Example 3.14.** Figure 3.14 illustrates the application of multiple conflicts for identifying more conflicting clauses and for finding lower backtracking decision levels. $x_1$ is assumed to be asserted due to a previous conflict with $z_1$, denoted by the conflicting clause $\omega_1 = (z_1 + \neg w_1 + \neg x_1)$. As shown in Figure 3.14-b, the resulting implication sequence leads to two conflicts, with $z_3$ and with $z_2$, that are represented by the conflict nodes $\kappa_1$ and $\kappa_2$, respectively. A conflicting assignment set is associated with each conflict node: $\boldsymbol{A}_{CS}^{1} = \{(z_1, 0), (w_1, 1), (w_3, 0), (w_4, 0), (z_3, 1)\}$ with $\kappa_1$ and $\boldsymbol{A}_{CS}^{2} = \{(z_1, 0), (w_1, 1), (w_2, 1), (z_2, 1)\}$ with $\kappa_2$, respectively. As a result, the two conflicting assignment sets are used to compute the backtracking decision level using (3.18), i.e. $\beta_L = min(4, 2) = 2$ due to $\kappa_2$. We note that without multiple conflicts the backtracking decision level would be 4, provided $\kappa_1$ was identified first (as would be the case with breadth-first implications). Moreover, $x_1$ is the only UIP of any of the conflicts, and thus the following conflicting clauses are created:

$$\omega_2 = (\neg z_2 + \neg w_2 + x_1)$$
$$\omega_3 = (\neg z_3 + w_3 + w_4 + x_1)$$
$$\omega_4 = (z_1 + \neg w_1 + w_3 + w_4 + \neg z_3)$$
$$\omega_5 = (z_1 + \neg w_1 + \neg w_2 + \neg z_2)$$




(a) Example circuit




(b) Implication graph (c) Decision tree

**Figure 3.14:** Application of multiple conflicts

where $\omega_2$ and $\omega_3$ are derived using (3.11), $\omega_4$ is associated with $\boldsymbol{A}_{CS}^{1}$ and $\omega_5$ with $\boldsymbol{A}_{CS}^{2}$. The effect of considering multiple conflicts is shown in Figure 3.14-c. □

The actual implementation of the procedure for diagnosing multiple conflicts defines its complexity. We start by analyzing a conflict diagnosis procedure that analyzes each conflict *separately*, by computing its UIPs and associated conflicting assignment set. After processing all conflicts, (3.18) is used to compute the backtracking decision level.

**Theorem 3.7.** Diagnosis of multiple conflicts, where each conflict is separately diagnosed and $\|\varphi\| = O(N)$, has a lower bound on the worst-case run time of $\Omega(N^2)$.

For most practical examples, the number of expected conflicts is usually small and so the overhead of diagnosing all conflicts should in general be smaller than the bound given above. Moreover, diagnosing multiple conflicts may contribute to significantly prune the search. A possible simplification for diagnosing multiple conflicts consists in relaxing the requirement to compute UIPs for each conflict, thus accepting the derivation of conflicting clauses only for the conflict that defines the backtracking decision level. In this situation, the worst-case time complexity for diagnosing multiple conflicts is still linear in $\|\varphi\|$. The procedure for diagnosing multiple conflicts is shown in Figure 3.15. The implication sequence is traversed, and the highest decision level $D_y$ that contributes to the assignment of each node $y$ is recorded. Afterwards, the conflict node $\kappa$ with the lowest recorded decision level is chosen. $\kappa$ is then used for diagnosing the conflict. The procedure given in Figure 3.15 computes UIPs for the chosen conflict node, and generates conflicting clauses accordingly. Note that a conflicting clause involving the conflicting assignment set must be created if backtracking is required, since it identifies the causes of conflicts at the backtracking decision level.

Although the procedure shown in Figure 3.15 is more efficient in the worst case, it sacrifices the derivation of some information that would otherwise be computed by separately diagnosing each conflict. Thus the number of identified conflicting clauses can be significantly smaller, and conflicts for which a conflict clause is not created may be found later at other stages of the search process. As with other tradeoffs of GRASP, the procedure that is best suited for diagnosing multiple conflicts depends on the structure of the application problems.

Identification of multiple conflicts can be further improved. We start by identifying a set of conflicting assignment sets that yield the same backtracking decision level $\beta_L$. Any of these conflicting assignment sets can be chosen to generate a conflict at decision level $\beta_L$. However, we can create multiple conflicts at decision level $\beta_L$ by using the identified conflicting assignment sets. As a result, if backtracking is required at decision level $\beta_L$, then multiple conflicts can be diagnosed in order to choose a lower backtracking decision level at decision level $\beta_L$.

```
// Global variables:   Implication graph I_C
//                     Clause database φ
// Input variable:     Current decision level c
// Output variable:    Backtracking decision level β_L
// Return value:       CONFLICT or SUCCESS
//
Diagnose_MC (c, &β_L)
{
   traverse implication subgraph at decision level c
      { compute highest decision level D_y implying each y ← v_y; }
   find conflict node κ with lowest decision level D_κ;
   set κ as chosen conflict node;
                                          // Diagnose conflict on κ
   U = Identify_UIPs();                   // From implication graph
   Ω_U = Create_Conflicting_Clauses (U);  // Using (3.11) and (3.13)
   Update_Clause_Database (Ω_U);        // Update database with set Ω
   β_L = Compute_Max_Level();             // Using (3.6) and (3.18)
   if (β_L != c) {
      ω = Create_Conflicting_Clause();    // Using (3.6) and (3.8)
      Update_Clause_Database (ω);
      define new conflict node κ;      // Set up new conflict node
      define α(κ) as the elements of ω;
      return CONFLICT;
   }
   return SUCCESS;
}
```

**Figure 3.15:** Linear-time diagnosis engine with identification of multiple conflicts

### 3.6.3 Implementation Tradeoffs

The proposed basic diagnosis engine and its variations add one or more conflicting clauses to the clause database after diagnosing each conflict. For a large number of backtracks, the size of the clause database grows accordingly, and this can introduce significant computational overhead for processing subsequent queries. The purpose of this section is to describe other diagnosis engines which guarantee that the size of the clause database does not grow exponentially in the number of variables. We start by describing a diagnosis engine that guarantees a *constant size* clause database, by trading off some diagnosis ability and by not implementing conflict-based

equivalence. Afterwards, we describe a family of diagnosis engines that guarantee a polynomial size growth of the clause database in the number of variables.

#### 3.6.3.1 Constant Size Clause Database

In this section we discuss one alternative diagnosis engine that targets reducing the overhead associated with maintaining the clause database during the search process. The main purpose of this engine is to implement some of the pruning methods described in previous sections, while guaranteeing that the size of the clause database remains constant throughout the search process. The main differences of the new procedure are as follows:

1. A conflicting assignment set $\boldsymbol{A}_{CS}[i]$ (referred to as *level conflicting assignment set*) is associated with each decision level $i$.
2. A failure-driven assertion (FDA) is now defined as a 3-tuple $\langle x, v_x, d_x \rangle$ that indicates a node $x$ whose value cannot be other than $v_x$ at decision levels greater than $d_x$ (i.e. that include the global assignment set $\boldsymbol{A}_{d_x}$).
3. The antecedent assignment of an assertion $\langle x, v_x, d_x \rangle$ is defined as follows[7]:

$$\boldsymbol{A}(x) = \bigcup_{i=1}^{d_x} \boldsymbol{A}_{CS}[i] \tag{3.19}$$

Every time a conflict is detected, a temporary conflicting assignment set $\boldsymbol{A}_{CS}$ is computed (with (3.6) on page 67). This conflicting assignment set is then used to update the level conflicting assignment sets as follows:

$$\boldsymbol{A}_{CS}[i] \leftarrow \boldsymbol{A}_{CS}[i] \cup \{(x, v_x) \in \boldsymbol{A}_{CS} \,|\, \delta(x) = i\} \tag{3.20}$$

An assertion is created for every UIP $(u, \nu(u))$ of the implication sequence leading to a conflict. Its value is $\overline{\nu(u)}$ and the assertion decision level is defined as the highest decision level that is identified as contributing to the conflict, where $(u, \nu(u))$ is assumed to trigger the conflicting implication sequence. The assertion decision level can be computed through *causesof*$(x, u)$ defined in

[7] In the actual implementation, a predicate *asserted*$(x)$ indicates whether $x$ is asserted. The sole purpose of introducing (3.19) is to allow the conflict analysis equations of Section 3.4 and of Section 3.6 to be used.

```
// Global variables:   Implication graph I_C
// Input variable:     Current decision level c
// Output variable:    Backtracking decision level β_L
// Return value:       CONFLICT or SUCCESS
//
Diagnose_C (c, &β_L)
{
   Update_Level_A_CS();                          // Using (3.6) and (3.20)
   U = Identify_UIPs();                        // From implication graph
   Create_FDAs (U);                          // Failure-driven assertions
   β_L = Compute_Max_Level();                              // Using (3.21)
   clear A_CS[c];
   if (β_L != c) {
      create new κ with incoming edges from all nodes in ⋃_{0≤i≤β_L} CS[i];
      return CONFLICT;
   }
   return SUCCESS;
}
```

**Figure 3.16:** Pseudo-code for simplified diagnosis engine

Section 3.4 (see page 70). As with the basic diagnosis engine, backtracking is required whenever the node triggering the conflicting implication sequence is already asserted. In this situation, the backtracking decision level $\beta_L$ is defined as follows:

$$\beta_L = max\{i \mid 0 \leq i \leq |PI| \wedge \boldsymbol{A}_{CS}[i] \neq \emptyset\} \tag{3.21}$$

Note that $\beta_L$ is always well-defined since the causes of any conflict must be assigned at some decision levels. At decision level $\beta_L$ (if $\beta_L \neq 0$) a conflict node is defined, which involves all the nodes in $CS[i]$, for all $i$ less than or equal to $\beta_L$. This forced conflict is diagnosed, and either the decision node at decision level $\beta_L$ is asserted or a new backtracking decision level is computed (if the node triggering the implication sequence at decision level $\beta_L$ was already asserted).

The pseudo-code for the diagnosis engine described above is shown in Figure 3.16 (which is referred to as `Diagnose_C()`[8]). With respect to `Diagnose()`, the most relevant differences are:

[8] *C* indicates that the diagnosis engine guarantees a *constant* size clause database.

1. The size of the clause database φ remains constant, since no conflicting clauses are explicitly identified and added to φ.
2. Antecedent assignments of assertions are implicitly maintained with the conflicting assignment sets $\boldsymbol{A}_{CS}[i]$. Hence, the overhead of manipulating large antecedent sets is eliminated.
3. Conflict-based equivalence is no longer implemented. This would require updating φ with clauses derived from conflicts, which is exactly what `Diagnose_C`() avoids.
4. Each level conflicting assignment set $\boldsymbol{A}_{CS}[i]$ is updated after a temporary conflicting assignment set is computed.
5. Whenever backtracking is required, a conflict node κ is defined. The incoming edges to κ are defined as all the elements of $\boldsymbol{A}_{CS}[i]$, where $i$ ranges from 0 to $\beta_L$.
6. $\boldsymbol{A}_{CS}[c]$ must be cleared after diagnosing each conflict at decision level $c$.
7. Procedure `Create_FDAs`() defines each assertion as a 3-tuple $\langle x, v_x, d_x \rangle$. As a result, procedure `Erase`() (see Figure 3.2 on page 62) must ensure that assertions at each decision level $d$ are cleared as a consequence of the search process backtracking to $d$.

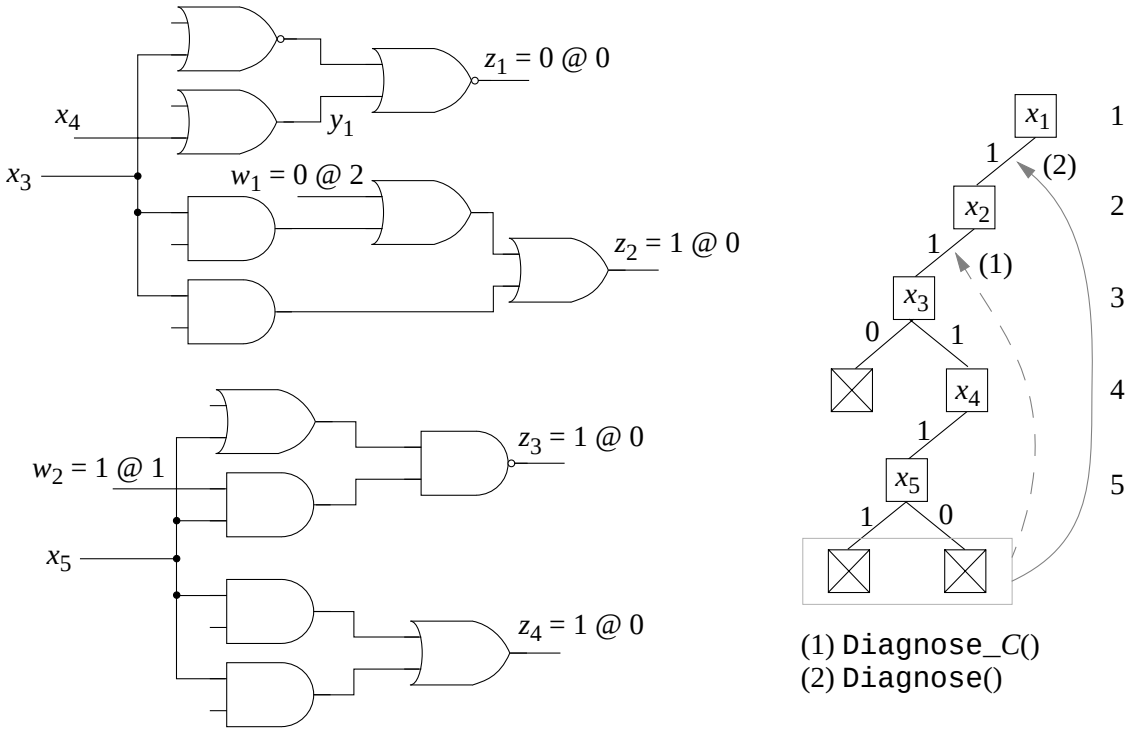
The proposed diagnosis engine also identifies FDAs due to unique implication points. In addition, the diagnosis engine could handle iterated conflicts and multiple conflicts. Note, however, that implementation of iterated conflicts would be irrelevant, since all identified conflicts update the level conflicting assignments sets, which would prevent finding lower backtracking decision levels.

**Theorem 3.8.** The search algorithm for solving SAT, described in Figure 3.2 (see page 62), customized with `Select`(VAR+VAL), `Deduce`() and `Diagnose_C`(), is sound and complete.

The proof of the above theorem hinges on the fact that, after each conflict, the union of the level conflicting assignment sets is an implicate of the consistency function.

Although the diagnosis engine proposed in this section is simpler to implement and ensures a constant size clause database, it has a few drawbacks. First, conflict-based equivalence is no longer implemented. Second, the computation of the backtracking decision level is not pruned as much as the one computed by `Diagnose`().

**Example 3.15.** The difference in computed backtracking decision levels is illustrated with the




(a) Example circuit

(b) Backtracking decision levels

**Figure 3.17:** Difference in computed backtracking decision level

example of Figure 3.17. Assume that the decision assignments on $x_1$ and $x_2$ imply the assignments on $w_2$ and $w_1$, respectively. Further, let us consider first the application of `Diagnose_C`(). At decision level 3, assume decision $x_3 \leftarrow 0$, which causes a conflict with $z_2$. Hence, $\boldsymbol{A}_{CS}[2]$ is updated with ($w_1$, 0) and $\boldsymbol{A}_{CS}[0]$ is updated with ($z_2$, 1). Conflict diagnosis causes $x_3$ to be asserted to 1 at decision level 2. The next decision, $x_4 \leftarrow 1$, satisfies the objective on $z_1$ (note that it actually satisfies $y_1 = 1$@ 3 due to $x_3$). The search process then tries to satisfy the objective on $z_4$. Let the next decision assignment be $x_5 \leftarrow 1$. This assignment leads to a conflict with $z_3$, such that $\boldsymbol{A}_{CS}[1]$ is updated with ($w_2$, 1), $\boldsymbol{A}_{CS}[0]$ is updated with $z_3$, and $x_5$ is asserted to 0. (Note that $\boldsymbol{A}_{CS}[5]$ is updated with ($x_5$, 1), but is cleared after diagnosing the conflict.) The resulting implication sequence causes a conflict with $z_4$. Thus, $\boldsymbol{A}_{CS}[0]$ is updated with ($z_4$, 1). Since $x_5$ is asserted, it is necessary to backtrack. The highest $i$ such that $\boldsymbol{A}_{CS}[i]$ is non-empty is 2. Hence the search process backtracks to decision level 2, as shown in Figure 3.17-b. On the other hand, `Diagnose`() would compute 1 as the backtracking decision level, because the two conflicts associated with $x_5$ do not depend on assignments at decision level 2. □

In order to reduce the overhead of manipulating conflicting clauses and antecedent assignments of assertions, the search process maintains a level conflicting assignment set $\boldsymbol{A}_{CS}[i]$ for each decision level *i*, each of which represents dependencies with respect to that decision level. Whenever backtracking is required, this global dependency information is considered, and thus unrelated conflicts are now related by considering the union of every $\boldsymbol{A}_{CS}[i]$. Consequently, for a given conflicting condition of a search process, we can conclude that the backtracking decision level computed with `Diagnose_`*C*() is always no less than the one computed with `Diagnose`().

**Implementation**

Identification of conflicts sets in `Diagnose_`*C*() is implemented as in `Diagnose`(). However, the level conflicting assignment sets are updated directly with references to the traced variables, thus guaranteeing a total size of the level conflicting assignment sets of *O*(*N*). Consequently, all decision level processing is implemented in time linear in the size of the *initial* clause database. For combinational circuits with bounded fanin, this implies that processing each decision level (either implications or conflict diagnosis) is accomplished in *O*(*N*) time. Further note that all asserted nodes have antecedent sets implicitly defined by the level conflicting assignment sets. Hence, the definition of failure-driven assertions does not significantly increase the computational overhead of processing each decision level.

#### 3.6.3.2 Polynomially Bounded Clause Database

We now describe diagnosis engines that represent possible compromises between `Diagnose`() and `Diagnose_`*C*(), by allowing restricted forms of conflict-based equivalence. Instead of not adding conflicting clauses to the clause database, as in `Diagnose_`*C*(), we allow clauses of size no larger than *m* to be added, while conflicts due to larger clauses are used to update the level conflicting assignment sets $\boldsymbol{A}_{CS}[i]$. Each of these diagnosis engines is referred to as `Diagnose_`*Pm*()[9]. A direct consequence of this approach is that the size of the clause database can only grow polynomial in *N*, even if an exponential number of backtracks is assumed. Whenever it is necessary to backtrack, if the causes of the conflict can be solely attributed to conflicting

[9] *Pm* indicates that the diagnosis engine can cause a worst-case growth of the clause database that is polynomial in *N* due to the conflicting clause size constraint *m*.

clauses, then these clauses define the backtracking decision level. Otherwise, the level conflicting assignment sets are consulted, by (3.21), for computing the backtracking decision level. `Diagnose_`*Pm*() defines a hierarchy of diagnosis engines that guarantee a polynomial size increase of the clause database, and that implement restricted forms of conflict-based equivalence.

It is worth noting that the identification of UIPs extends the usefulness of `Diagnose_`*Pm*(), since UIPs reduce the size of conflicting clauses. The identification of UIPs does not change the worst-case time complexity of `Diagnose_`*Pm*(), and it increases the likelihood of creating clauses over updating the level conflicting assignment sets.

A related diagnosis engine consists of limiting the total number of added conflicting clauses, without regard to the size of each clause. As with `Diagnose_`*Pm*(), the level conflicting assignment sets are required to ensure that all dependencies are properly accounted for. This approach guarantees a constant size increase of the clause database. Another variation is to consider `Diagnose_`*Pm*() but where the total number of added conflicting clauses is bounded. This diagnosis engine ensures a constant increase in the size of the clause database, and such that each added clause has at most *m* literals.

Note that the most significant advantage of `Diagnose_`*Pm*() over `Diagnose`() is that the space requirements are bounded by a polynomial in *N*. Hence, the time required to process a given decision level never becomes exponential in *N*.

These different diagnosis engines provide different tradeoffs between computational overhead at each decision level and the amount of search. Depending of the end application, each procedure may represent the best solution. For example, in the course of our work and in the context of test pattern generation, a conflict diagnosis procedure similar to `Diagnose_`*C*() was described in [155]. Experimental data suggests that the diagnosis ability of `Diagnose_`*C*() may be a balanced solution for most practical test pattern generation problems.

### 3.6.4 Advanced Diagnosis Engines

The basic diagnosis engine, described in Section 3.6.1, is not guaranteed to identify conflicting assignment sets of minimum size. The purpose of this section is to study techniques to remove redundancies from conflicting assignment sets.

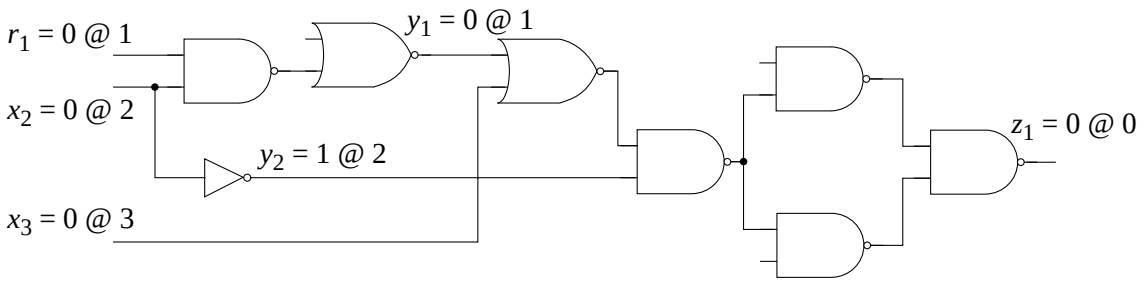



**Figure 3.18:** Over-specified conflicting assignment set

**Example 3.16.** An example where a conflicting assignment set contains redundant information is shown in Figure 3.18. The decision assignment $x_3 = 0$ creates an implication sequence that leads to a conflict. The conflicting assignment set that is identified (with (3.6) on page 67) is $\boldsymbol{A}_{CS} = \{ (x_3, 0), (y_1, 0), (y_2, 1), (z_1, 0) \}$. We now express the conflicting assignment set in terms of the decision assignments and assignments at decision level 0 (i.e. objective assignments), thus obtaining $\boldsymbol{A}_I = \{(r_1, 0), (x_2, 0), (x_3, 0), (z_1, 0)\}$ . Now suppose the independent application of the assignment set $\boldsymbol{A} = \{(x_2, 0), (x_3, 0), (z_1, 0)\}$ . In such a situation, the same conflict is detected. Consequently, we can conclude that the assignment $(r_1, 0)$ is redundant since it does not represent a necessary condition for identifying the same conflict. The derivation of this fact required considering one subset of the original conflicting assignment set containing one element less. The reduced conflicting assignment set can be used to reduce the number of backtracks. For the above example, backtracking to the first decision level will no longer be required (assuming that no other dependencies on $r_1$ are identified). □

In order to relate the size of conflicting assignment sets with the number of backtracks, we minimize the size of conflicting assignment set defined in terms of decision variables and objective assignments. As a result, if a (decision) variable assignment can be removed from a conflicting assignment set, then the dependency of a conflict with respect to a decision level is eliminated. When backtracking is required, removed decision variables will not be considered as target backtracking points. Consequently, by minimizing the size of the conflicting assignment set as proposed, we are guaranteed to require no more backtracks, and we increase the likelihood of reducing, in some situations, the total number of backtracks. As the above example suggests, it is necessary to represent conflicting assignment sets in terms of decision variables and objective

assignments. Otherwise, we might reduce the size of a conflicting assignment set, but create conditions for increasing the number of backtracks by introducing dependencies on other decision levels not contained in the original conflicting assignment set.

Procedure `Simplify_Conflict_Set_`*j*(), for minimizing conflicting assignment sets, is divided into three distinct phases:

1. Specify the conflicting assignment set in terms of its decision variables and objective assignments, i.e. create $\boldsymbol{A}_I$. Erase all assignments.
2. For all $i$, $1 \leq i \leq j$, and for each subset $\boldsymbol{A}$ of $\boldsymbol{A}_I$, of size $\left|\boldsymbol{A}_I\right| - i$ and composed only of decision assignments, use `BCP`() (described in Figure 2.6 on page 39) to test whether implications derived from $\boldsymbol{A}$ yield a conflict. If $\boldsymbol{A}$ yields a conflict, then record the associated conflicting assignment set (referred to the primary inputs and objective assignments).
3. Pick one of the recorded conflicting assignment sets with the smallest size. (For example, one acceptable heuristic is to pick the one involving the smallest decision levels.)

This procedure ensures that we identify the smallest set of node assignments of size ranging from $\left|\boldsymbol{A}_I\right| - j$ to $\left|\boldsymbol{A}_I\right|$, included in $\boldsymbol{A}_I$, that also causes a conflict. The complexity of the procedure for any given $j$ is:

$$O\left(\|\varphi\| \cdot \sum_{i=1}^{j}\binom{\left|\boldsymbol{A}_I\right|}{\left|\boldsymbol{A}_I\right| - i}\right) \tag{3.22}$$

where the term $\|\varphi\|$ denotes the overhead of executing `BCP`(). The contribution of phase 1 is $\|\varphi\|$, it is negligible and it is not considered. Diagnosis engines based on the above procedure, referred to as `Diagnose_`*j*(), can be readily implemented by appropriate modifications to procedure `Diagnose`().

The idea of simplifying dependency sets has been extensively studied in constraint satisfaction problems [19, 41, 42, 143, 169], truth maintenance systems [43, 44, 54] and logic programming/automated deduction [20, 21, 130]. In satisfiability algorithms for combinational circuits, redundancies on conflicting assignment sets require different forms of reconvergent fanout that in practice are difficult to find. For example, that is the case for the example we used in Figure 3.18. Therefore, we conjecture that minimizing conflicting assignment sets in clause databases associ-

ated with combinational switching circuits may not provide significant search pruning.

## 3.7 Preprocessing the Clause Database

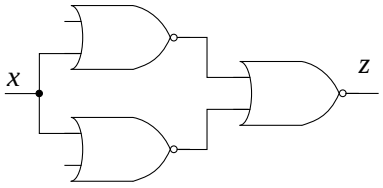
Preprocessing a clause database entails the identification of implicates of the consistency function, prior to searching for a solution to a query. In the SAT algorithm of Figure 3.2 (see page 62), preprocessing the clause database is performed by the preprocessing engine `Preprocess`(). The effort one is willing to spend in preprocessing a clause database defines the preprocessing ability, and so a family of preprocessing engines `Preprocess_`*m*() can be defined.

Note that each deduction engine `Deduce_`*k*() can be used for preprocessing purposes. Nevertheless, each of these deduction engines can be modified to identify more implicates. While each `Deduce_`*k*() is based on diagnosing conflicts for creating implicates, `Preprocess_`*m*() also examines the structure of implication sequences with the goal of deriving additional implicates. As we show in the sequel, deriving implicates from the structure of implication sequences can introduce a large number of redundant implicates. While the overhead of detecting and removing these redundancies can be acceptable from a preprocessing perspective, it can be prohibitive for a deduction engine. The objective of this section is to illustrate how deduction engines can be modified for preprocessing purposes.

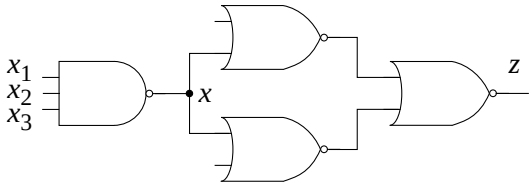
For example, let us suppose the assignment $x \leftarrow v_x$, that implies the assignment $y \leftarrow v_y$, with $|\alpha(y)| > 1$ and *causesof*($y$) = { ($x$, $v_x$) }. This also implies that, in terms of the implication graph, more than one path is involved in implying the assignment of *y*. On the other hand, the assignment $y \leftarrow \overline{v_y}$ does not necessarily imply $x \leftarrow \overline{v_x}$, even though this latter assignment is necessary for a consistent assignment. Consequently, $\{x^{v_x}, y^{\overline{v_y}}\}$ is an implicate of the consistency function and may identify implications that otherwise might not be derivable.

**Example 3.17.** Let us consider the example circuit of Figure 3.19-a. $x \leftarrow 1$ implies $z \leftarrow 1$. Hence, we can derive the implicate $\omega = (z + \neg x)$. Without this implicate $z \leftarrow 0$ would not imply $x \leftarrow 0$. $\omega$ ensures that this assignment is implied. □

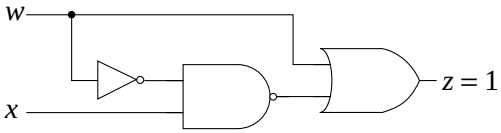
In a more general situation, consider the assignment set { ($x_1$, $v_1$), …, ($x_m$, $v_m$) }, that implies the assignment $y \leftarrow v_y$. Let us further assume that *causesof*($y$) = { ($x_1$, $v_1$), …, ($x_m$, $v_m$) }.




(a) Applying `Preprocess_1()`




(c) Redundancy in preprocessing




(b) Effect of assignments on `Preprocess_1()`

**Figure 3.19:** Example of preprocessing

In such a situation, $\left\{x_1^{v_1}, \ldots, x_m^{v_m}, y^{\overline{v_y}}\right\}$ denotes an implicate of the consistency function. Note that this implicate is only relevant if it identifies otherwise non-derivable implications. Suppose now that there are other independent assignments, i.e. *y* is assigned due to the assignment set and due to other assignments already specified for the circuit. Then, *causesof*(*y*), from (3.7) on page 67, can be used to identify which nodes are actually responsible for implying the assignment of *y*. In such a situation, the implicate to be created is given by:

$$\omega = \left[\bigcup_{w \in causesof(y)} \{w^{v(w)}\}\right] \bigcup \{y^{\overline{v(y)}}\} \tag{3.23}$$

which denotes the general form for the generation of implicates by preprocessing. The objective of `Preprocess_`*m*() is to derive and simplify all implicates of the above form, for all assignments of all subsets of size *m* of unassigned variables. The procedure for preprocessing a clause database is given in Figure 3.20 and it follows the implementation of `Deduce_`*k*(). However, besides analyzing conflicts, any node assignment that is implied due to multiple implication paths causes a new implicate to be added to the clause database. For each subset of nodes of size *m*, all possible logic value assignments are tested. For each assignment, implications are derived. If a conflict is detected, then the procedure generates a conflicting clause as `Deduce_`*k*() does. Otherwise, the structure of the implication sequence is examined, and conflicting clauses are generated with respect to each assigned node with an antecedent set of size larger than 1. (As mentioned earlier,

```
// Global variables:   Implication graph I_C
//                     Clause database φ
// Return value:       CONFLICT or SUCCESS
//
Preprocess_m()
{
   if (!PREPROCESS_QUERY) return SUCCESS;      // No preprocessing
   if (Deduce() == CONFLICT) return CONFLICT;
   status = SUCCESS; φ_m ← ∅;
   Let Γ be the set of all sets of m unassigned nodes;
   for (each set γ ∈ Γ and while status != CONFLICT) {
      status = CONFLICT; φ_γ ← ∅;
      for (each distinct logic value assignment to the nodes in γ) {
         if (Deduce() == CONFLICT) {
            ω = Create_Conflicting_Clause();    // Using (3.6), (3.8)
            φ_γ ← φ_γ ∪ { ω };
         } else {                                // No conflict detected
            for (each assigned node y with |α(y)| > 1 ) {
               compute causesof(y);             // Using (3.7) on page 67
               create conflicting clause ω;     // Using (3.23)
               φ_γ ← φ_γ ∪ { ω };
            }
            status = SUCCESS;
         }
         Erase_Last_Assignments();
      }
      Generate_Prime_Implicates (φ_γ);      // See Figure 2.7 on page 44
      φ_m ← φ_m ∪ φ_γ;
   }
   if (SIMPLIFY_φ_m) Generate_Prime_Implicates (φ_m);
   φ ← φ ∪ φ_m;
   return Deduce();
}
```

**Figure 3.20:** Description of the preprocessing engine

an antecedent set of size larger than 1 covers all cases of node assignments implied due to multiple implication paths.) From the above discussion, we can conclude that preprocessing can introduce some superfluous conflicting clauses, that are removed by `Generate_Prime_Implicates()`.

`Preprocess_`*m*() terminates in a conflict if it can establish that the clause database is not satisfiable.

Although not included in the procedure shown in Figure 3.20, UIPs can be identified either if diagnosing a conflict, or for each assigned node due to multiple paths. UIPs reduce the size of implicates, and as described below, can be applied in removing some forms of preprocessing redundancy.

Further note that even though `Preprocess_`*m*() apparently derives more conflicting clauses than `Deduce_`*k*(), the size of the clause databases computed by `Deduce_`*k*() and `Preprocess_`*m*() are bounded by the final size of $\varphi$, i.e. the size of the prime implicate representation $\varphi^P$. Hence, for $k = m$, `Preprocess_`*m*() *tends* to accelerate the derivation of prime implicates. (This situation will not hold whenever `Preprocess_`*m*() just derives redundant implicates.)

`Preprocess_`*m*() is defined without any form of relaxation. As with deduction engines, we can define preprocessing *with relaxation*, `Preprocess_`*m*,*R*(). The procedure of Figure 3.9 (see page 81) can be straightforwardly adapted to implement `Preprocess_`*m*,*R*().

**Example 3.18.** A few more details of the operation of `Preprocess_`1() are described with the example circuit of Figure 3.19-b. Suppose that the assignment $z = 1$ is given, and assume `Preprocess_`1() is invoked. For $\gamma = \{ w \}$, and for the assignment $w = 0$, then $x \leftarrow 0$. Hence, from (3.7) the causes for assigning $x$ are defined by $\{ (w, 0), (z, 1) \}$. Consequently, the implicate $\omega = (w + \neg z + \neg x)$ is created. Note that the implicate holds independently of the assignment to $z$, i.e. it identifies a logical relation of the original clause database. Suppose now that for a different query, $x \leftarrow 1$ and $z \leftarrow 1$. Then $\omega$ implies $w \leftarrow 1$, which would not be derived with `Deduce`() alone. □

The actual of implementation of `Preprocess_`*m*() may allow implicates to be added to the clause database as they are identified. This solution increases the number of derived implicates, but makes the final result dependent upon the order in which the subsets $\gamma$ are processed. For example, in the course of our work [155], and as a preprocessing step, we implemented `Preprocess_`1() but allowing the clause database to be updated as new implicates were derived.

For combinational circuits, the best order of the variables consisted in starting from the primary inputs and then proceed in level order to the primary outputs. This fact can be justified by noting that by adding additional implicates, implication sequences are more likely to reach farther backwards than without dynamically adding implicates. Hence more implicates can then be created.

Despite the large body of research work on preprocessing techniques, particularly in test pattern generation algorithms, the fact that preprocessing may introduce some redundant information has been overlooked in the past.

**Example 3.19.** Consider the example circuit of Figure 3.19-c, and assume that the circuit is preprocessed with `Preprocess_1()` without identifying UIPs (a related procedure is commonly referred to as *static learning* in several algorithms for test pattern generation [70, 71, 102, 144, 145, 155, 162, 167, 174]). Preprocessing the circuit with `Preprocess_1()` yields the following implicates (e.g. $x \leftarrow 1$ implies $z \leftarrow 1$, hence add clause $(z + \neg x)$ to the clause database):

$$(z + x_1) \cdot (z + x_2) \cdot (z + x_3) \cdot (z + \neg x) \tag{3.24}$$

However, it is immediate that the first three implicates provide no additional implications than the implications provided by the fourth implicate and by the original clause database. In fact, assume $z = 0$; then the fourth implicate implies $x \leftarrow 0$, which then implies the assignments $x_1 \leftarrow 1$, $x_2 \leftarrow 1$, and $x_3 \leftarrow 1$, due to the NAND gate. Note that these assignments would otherwise be implied by the first three implicates of (3.24). Even though these implicates are not subsumed by other implicates in the clause database, they can be considered redundant. □

The sole effect of these redundant implicates is to add computational overhead. Hence, preprocessing ought to avoid introducing redundant implicates. In the case of `Preprocess_1()`, the derivation of implicates based on UIPs can be used to prevent some of these redundant implicates.

**Example 3.20.** For the example circuit of Figure 3.19-c, assume that `Preprocess_1()` identifies UIPs. Let $x_1 = 0$ be the first assignment. It then implies $x \leftarrow 1$, which in turn implies $z \leftarrow 1$. Node $x$ denotes a UIP, and so the clause added is $(\neg x + z)$. No other implicates are added because there is only one implication path from $x_1$ to $x$. Next, consider the assignment $x_2 = 0$ (or $x_3 = 0$). Again,

$x \leftarrow 1$ is implied and (from $(\neg x + z)$ due to breadth-first implications) $z \leftarrow 1$ is also implied. However, given the last implication sequence, no more implicates are derived. Finally, the assignment $x \leftarrow 1$ implies $z \leftarrow 1$ (also due to $(\neg x + z)$), but no more implicates are derived, since only one implication path connects $x$ to $z$. Consequently, identification of UIPs prevents `Preprocess_1()` from creating redundant implicates. □

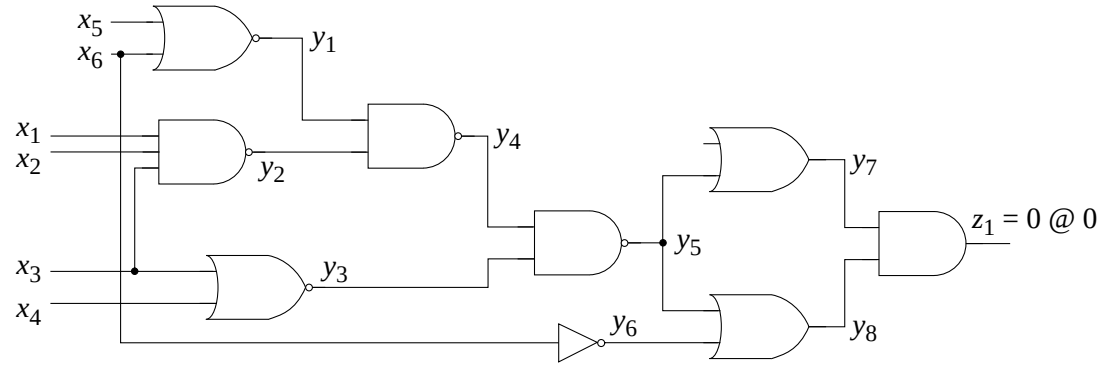
As mentioned earlier in Chapter I, preprocessing methods are ubiquitous in algorithms for constraint satisfaction problems [61, 99, 111, 126, 127, 169]. In test pattern generation restricted forms of preprocessing techniques have been proposed in recent years as a possible solution to reduce the complexity of search during the test generation phase [37, 71, 102, 105, 144, 145, 155, 162, 167, 174], as will be reviewed in Chapter V.

The hierarchy of preprocessing algorithms, described in this section, illustrates how any degree of consistency can be attained prior to computing solutions to queries. The algorithms are admittedly quite inefficient for large $m$, and more efficient procedures ought to be devised for those cases. The advantages of preprocessing are dependent on the application. For example, in test pattern generation, it is now commonly accepted that simplified forms of `Preprocess_1()` have advantages over no preprocessing [37, 71, 102, 106, 144, 155, 162, 167, 174]. In Chapter VII, we provide experimental results that show that this may not always be the case.

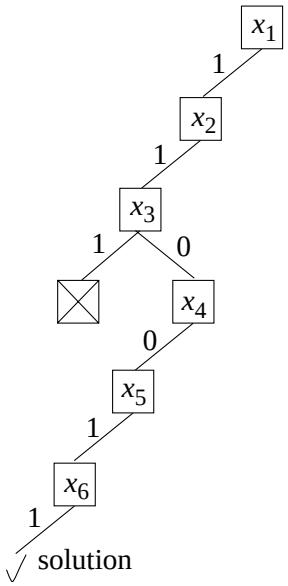
## 3.8 Postprocessing Engine

Solutions computed by the SAT algorithm can have redundancies. This means that some decision assignments are irrelevant for satisfying the specified objectives and can be discarded. For some applications, solutions of a smaller size may be particularly useful. In addition, the search for different queries to the clause database may have portions of the decision tree that are isomorphic. In this situation, cached information of the solution to each query can be used to simplify the search for subsequent queries.

This section studies techniques for removing redundant decision assignments from solutions and for caching information regarding identified solutions to queries. The techniques proposed apply exclusively to clause databases derived from combinational circuits, and specifically assume a circuit structure.




(a) Example circuit




(b) Decision tree

- Computed solution:

$$\boldsymbol{A} = \{(x_1, 1), (x_2, 1), (x_3, 0), (x_4, 0), (x_5, 1), (x_6, 1)\}$$

- Another valid solution:

$$\boldsymbol{A} = \{(x_3, 0), (x_4, 0), (x_5, 1), (x_6, 1)\}$$

(c) Valid solutions to the satisfiability problem

**Figure 3.21:** Over-specification of satisfying assignment

### 3.8.1 Removing Redundancies from Solutions

The solution computed for a given query can contain some redundancies because some decisions may be irrelevant to satisfying the original objectives.

**Example 3.21.** Consider the example circuit of Figure 3.21-a. where the order of decisions of Figure 3.21-b is assumed. GRASP, configured with `Deduce`() and `Diagnose`() would compute the following assignment set:

$$\boldsymbol{A} = \{(x_1, 1), (x_2, 1), (x_3, 0), (x_4, 0), (x_5, 1), (x_6, 1)\}$$

However, it is clear from the circuit that the assignments $x_1 = 1$ and $x_2 = 1$ are not relevant for sat-

isfying the original objective, and hence these assignments are said to be *redundant*. Accordingly, $\boldsymbol{A} = \{(x_3, 0), (x_4, 0), (x_5, 1), (x_6, 1)\}$ also constitutes a valid satisfying assignment. This latter assignment set is preferred in some applications of satisfiability algorithms. Moreover, we observe that the assignment set $\boldsymbol{A}$ can be further simplified, as is shown below. □

Even though the search algorithm does not provide direct mechanisms for simplifying satisfying assignments, in this section we describe simple techniques for removing redundancies from solutions.

The following analysis assumes that a solution to a query has been found, and that our goal is to identify which decisions effectively contribute for satisfying the query. Let us consider an assignment set $\boldsymbol{A_S}$ with respect to the primary inputs,

$$\boldsymbol{A_S} = \{(x, \nu(x)) \,|\, x \in PI \wedge \nu(x) \neq X\} \tag{3.25}$$

that satisfies a set of goals at the primary outputs. The resulting complete node assignment is used to create the *node justification graph* ($J_G$), which describes how the primary output node objectives are justified, by recursively identifying how each assigned circuit node is justified by the assignments to its fanin nodes.

The construction of $J_G$ requires a few preliminary definitions. For each assigned node $y$, with $y = v_y$, let $M(y)$ denote the set of fanin nodes that justify $y$ under the following conditions:

1. $M(y)$ is a minimal subset of fanin nodes of $y$ such that the assignments to the nodes in $M(y)$ justify the value of $y$.
2. $M(y)$ has the least highest decision level over all possible sets $M(y)$. If more than one candidate $M(y)$ has the least highest decision level, choose the $M(y)$ with the lowest highest implication level for the nodes assigned at the highest decision level in each $M(y)$.

Note that for simple gates either $M(y)$ contains all gate inputs or contains only one gate input, depending on whether the gate respectively assumes a *non-controlled* or a *controlled* value. (A definition of controlling/non-controlling values can be found in [1, p. 59].)

**Example 3.22.** For example, let $y = 0$ @ 0 be the output of an AND gate, $y = AND(x, w, u)$, such that $x = 0$ @ 1 / 3, $w = 0$ @ 1 / 5 and $u = 0$ @ 2 / 1. The value of $y$ is justified by any of its inputs,

and so any of these nodes can potentially define $M(y)$. Node $u$ is not considered because it is assigned at a decision level higher than that of either $x$ or $w$. Consequently, $M(y) = \{ x \}$, because both $x$ and $w$ are assigned at the same decision level and $x$ has the lowest implication level. □

Using the above definitions, the node justification graph $J_G = (V_J, E_J)$ is created as follows:

1. Every primary output objective $z = v_z$ corresponds to a vertex $\eta(z)$ in $V_J$.
2. For each vertex $\eta(y)$ in $V_J$, denoting the assignment $y = v_y$ and such that $\eta(y)$ has no incoming edges and $y$ is not a primary input, identify $M(y)$. For each node $w$ in $M(y)$, add $\eta(w)$ to $V_J$ and let $(\eta(w), \eta(y)) \in E_J$.

From the definition of $J_G$, it is clear that there may be assigned nodes not in $J_G$. In some situations, as illustrated by the example of Figure 3.21, every assigned node at a given decision level is not in $J_G$. This then signifies that such decision assignments are irrelevant for satisfying the original objectives. We further note that in some situations the node justification graph corresponds to a subgraph of the implication graph, but in general this is not the case. Justifications are gate input-output relations, whereas the implication graph denotes how implication sequences evolve, which are not necessarily based on gate input-output relations.

The set of primary input assignments to be considered is defined as follows:

$$\boldsymbol{A_S}' = \{(x, \nu(x)) | x \in PI \wedge \eta(x) \in V_J\} \tag{3.26}$$

where, $\boldsymbol{A_S}' \subseteq \boldsymbol{A_S}$.

**Example 3.23.** The node justification graph for the example circuit of Figure 3.21 is shown in Figure 3.22. The construction of $J_G$ reveals that the decisions on $x_1$ and $x_2$ are redundant. Furthermore, from (3.26) the reduced assignment set becomes $\boldsymbol{A_S}' = \{ (x_3, 0), (x_4, 0), (x_5, 1), (x_6, 1) \}$.

It is important to note that the assignment set defined by (3.26) is *not* minimal in the number of decision assignments. This fact results from the order in which decision assignments are made, which may require considering a decision that otherwise could be made redundant. Consider again the example circuit of Figure 3.21-a and let $\boldsymbol{A_S}'' = \{(x_3, 0), (x_4, 0), (x_6, 1)\}$ be an assignment set. By inspection, we can conclude that $\boldsymbol{A_S}''$ is also a solution to the original query,

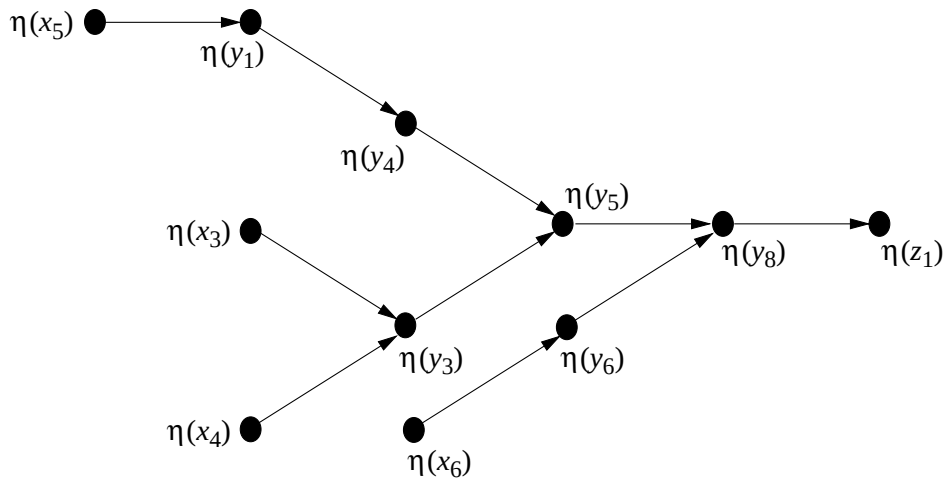



**Figure 3.22:** Node justification graph ($J_G$) for the example circuit of Figure 3.21

because $x_6 \leftarrow 1$ implies the assignments that were otherwise implied by the assignment of $x_5$. □

As the above example suggests, a solution assignment set can be further simplified by separately considering some of its subsets. For each such combination, logical implications are derived, and one tests whether all the objectives are satisfied. If so, then the subset of the original assignment set is indeed a solution to original set of objectives. The size of the subsets depends on the amount of effort one is willing to spend reducing the size of the solution assignment set. In general, we define redundancy removal of order $k$ to signify that for an original solution assignment set of size $|\boldsymbol{A_S}|$, all subsets of size $|\boldsymbol{A_S}| - k$ or larger are analyzed. A straightforward implementation of these ideas is given in Figure 3.23. Basically, for each subset of variables of $\boldsymbol{A_S}$ of size greater than or equal to $|\boldsymbol{A_S}| - k$, a tentative solution for the objectives is checked. If one of these subsets is indeed a solution, then it represents one possible reduced assignment set for the original query. The procedure starts from the smallest subset of variables and proceeds to the largest subsets (i.e. of size $|\boldsymbol{A_S}| - 1$). The outer loop can be removed if one is only interested in solutions of size $|\boldsymbol{A_S}| - k$. For each $j$, the number of subsets to be analyzed is:

$$\binom{|\boldsymbol{A_S}|}{|\boldsymbol{A_S}| - j} = \binom{|\boldsymbol{A_S}|}{j}$$

Since testing whether a given assignment set is a solution requires $O(\|\varphi\|)$ time, then an upper

```
Remove_Solution_Redundancies_k(A_S)
{
   create_J_G();
   for (j = k down to 1) {
      let Γ_j be the set of all subsets of A_S with size |A_S| − j;
      for each (subset γ of Γ_j) {
         clear all assignments except objectives;
         if (γ satisfies the objectives) return γ;
      }
   }
   return A_S;
}
```

**Figure 3.23:** Pseudo-code for removing redundancies from solutions

bound on the run time of the procedure of Figure 3.23 is given by:

$$O\left(\|\varphi\| \cdot \sum_{j=1}^{k} \binom{|A_S|}{j}\right) \tag{3.27}$$

that basically limits the applicability of the procedure to small $k$. As a final remark, we emphasize that for any $k$ it is not possible to guarantee a solution of minimum size. Such minimum size solution may only be defined with decision assignments not even involved in the computed solution. For $k = |A_S|$, we can guarantee that the solution of minimum size is computed, *given* the original solution.

The significance of removing redundancies from solutions to queries depends on the end application. In Chapter V, we describe the extension of these ideas to test pattern generation. For circuits where a large number of decisions are not relevant for the identification of a solution, removing redundancies from a satisfying assignment has several advantages, most noteworthy, testing time and test size.

### 3.8.2 Caching Solutions

In applications where a large number of queries is to be posed to the clause database, it is often useful to record previously identified solutions so that similarities between distinct queries can be used to reduce the search effort. For example, this is the case in test pattern generation,

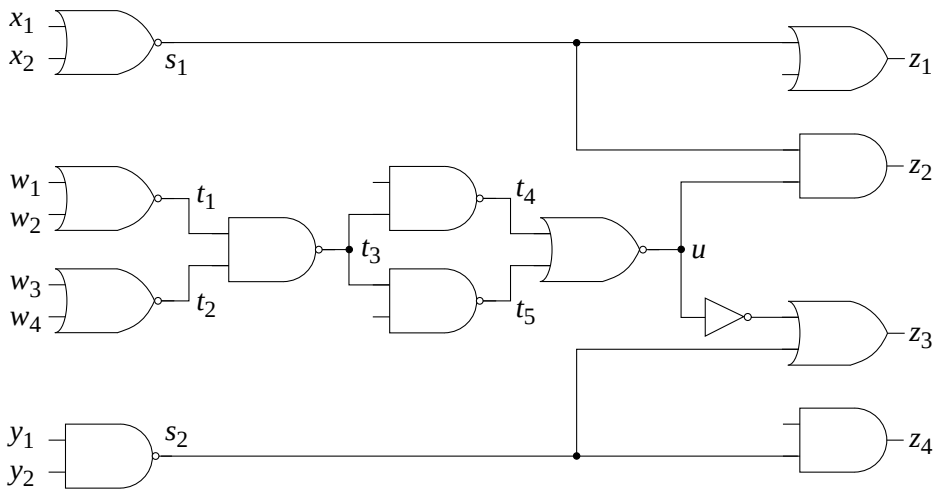



**Figure 3.24:** Example of caching solutions

where a large number of faults must be detected and detecting each fault can be viewed as an instance of SAT. In situations where the search effort to identify the solution to a query is significant, it can be useful to identify ways to encode that solution in order not to repeat the same search effort again for subsequent queries. Such encoding can be viewed as *caching* the solution to the query in order to use it again afterwards. In this section we analyze one possible procedure for caching solutions to queries on clause databases.

**Example 3.24.** An example illustrating how cached solutions can be defined and used is shown in Figure 3.24. The first set of objectives is assumed to be $z_1 = 1$ and $z_2 = 0$. Let us assume that the ordered sequence of decision assignments is $x_1 = 0$, $x_2 = 0$, $w_1 = 0$, $w_2 = 0$, $w_3 = 0$, $w_4 = 0$. We observe that after the second decision assignment (i.e. $x_2 = 0$), $s_1 \leftarrow 1$. Due to this assignment, and because $z_2 = 0$, $u \leftarrow 0$ is implied. As a direct consequence, the j-frontier[10] after decision level 2 consists only of node $u$. Moreover, the above set of decision assignments denotes a solution to the original set of objectives.

The second set of objectives is given by $z_3 = 1$ and $z_4 = 0$. Let us assume that the ordered sequence of decisions is now $y_1 = 1$, $y_2 = 1$, $w_1 = 0$, $w_2 = 0$, $w_3 = 0$, $w_4 = 0$. However, for this second query, after the second decision (i.e. $y_2 = 0$) we have $s_2 \leftarrow 0$, which then implies (with $z_3 = 1$) $u \leftarrow 0$. Hence the j-frontier becomes the same as in the previous case. Furthermore, with respect

[10] Since combinational circuits are explicitly assumed, j-frontiers are well-defined. A definition of j-frontier is given by (3.4) on page 60.

to the first query, all decision nodes involved in decisions at decision levels greater than 2 are still unassigned. The same holds true for all other nodes that were assigned at decision levels greater than 2. Consequently, we can use the information of the previous solution to *immediately* identify a solution to the current set of objectives. The solution to the current query uses the decision assignments already made (i.e. $y_1 = 1$ and $y_2 = 1$) and extracts from the previous solution the set of decision assignments required to satisfy $u = 0$ (i.e. $w_1 = 0$, $w_2 = 0$, $w_3 = 0$, $w_4 = 0$). For this second query, the search process identifies a solution after two decisions. Further note that the two computed solutions are actually distinct. Caching solutions allows extracting parts of a cached solution to complete the solution currently being computed. □

The procedure we propose identifies a set of node assignments (or lack of such assignments) which guarantee that a solution to a new query can be constructed if the assignments of the new query are adequately related with the assignments of some previously cached solution.

Let us assume that the solution to a query has been computed, and that the node justification graph has been created. Given the definition of $J_G$, the decision level at which each node $y$ is assigned, with $\eta(y) \in V_J$, provides a partition of $V_J$. Let $K$ be the depth of the decision tree, and let $\boldsymbol{P}_0, \boldsymbol{P}_1, \ldots, \boldsymbol{P}_K$ denote each set in the partition of $V_J$, such that $\eta(y) \in \boldsymbol{P}_j$ if and only if $\delta(y) = j$. Note that from the discussion in the previous section, some sets $\boldsymbol{P}_j$ may be empty.

Define a predicate $J(w, y)$ to hold true if and only if in $J_G$ either $w$ contributes to justifying $y$ (i.e. $w \in M(y)$) or, conversely, $y$ contributes to justifying $w$ (i.e. $y \in M(w)$). Hence, $J(w, y)$ holds true if and only if either $(\eta(w), \eta(y)) \in E_J$ or $(\eta(y), \eta(w)) \in E_J$. Furthermore, we define a *level cut* $\boldsymbol{T}_j$, for decision level $j$, as follows:

$$\begin{aligned}\boldsymbol{T}_j = \; & \{p \in V_J \,|\, p \in \boldsymbol{P}_j \wedge [(p, q) \in E_J \vee (q, p) \in E_J] \wedge q \in \boldsymbol{P}_l \wedge l > j\} \cup \\ & \{p \in V_J \,|\, p \in \boldsymbol{P}_i \wedge i < j \wedge [(p, q) \in E_J \vee (q, p) \in E_J] \wedge q \in \boldsymbol{P}_l \wedge l > j\}\end{aligned} \tag{3.28}$$

$\boldsymbol{T}_j$ contains the vertices $p$ of $V_J$ such that either:

1. $p \equiv \eta(w)$, $w$ is assigned at decision level $j$, and there exists $y$ such that $J(w, y)$ holds, with $q \equiv \eta(y)$ and $\delta(y) > j$.
2. $p \equiv \eta(w)$, $w$ is assigned at decision level $i$, with $i < j$, and there exists $y$ such that $J(w, y)$ holds, with $q \equiv \eta(y)$ and $\delta(y) > j$.

Consequently, level cut $\boldsymbol{T}_j$ contains *all* vertices associated with nodes assigned at decision levels less than or equal to $j$ such that these nodes contribute to justify or are justified by nodes assigned at decision levels higher than $j$.

**Example 3.25.** Consider the example circuit of Figure 3.24. For the first query, at decision level 4 and after deriving implications (due to $w_2 = 0$), $\boldsymbol{T}_4$ is defined by,

$$\boldsymbol{T}_4 = \{\eta(t_1)\} \cup \{\eta(u)\} \tag{3.29}$$

$\eta(u)$ belongs to the second subset because $u$ is assigned at decision level 2, and there are nodes assigned at decision levels higher than 4 that contribute to justify $u$. □

Each level cut $\boldsymbol{T}_j$ uncouples assignments at decision levels greater than $j$ from assignments at decision levels less than or equal to $j$. In particular, after defining $\boldsymbol{T}_j$, any node $y$ assigned at a decision level greater than $j$, can only contribute to justify or be justified by nodes $w$ assigned at decision levels greater than $j$ or such that $\eta(w)$ is in $\boldsymbol{T}_j$.

**Lemma 3.1.** Assume a solution to a query identified by an assignment set $\boldsymbol{A}$, and let the associated node justification graph $J_G$ be defined. For each decision level $j$, define $\boldsymbol{T}_j$ with (3.28). In such a situation, for any node $y$, such that $J(y, x)$ holds for $x$ assigned at a decision level greater than $j$, either $y$ is also assigned at a decision level greater than $j$ or is such that $\eta(y) \in \boldsymbol{T}_j$.

Let the assignment set $\boldsymbol{A}$ be a solution of size $K$ to a query, and let $\boldsymbol{T}_j$ be a level cut associated with $\boldsymbol{A}$. Suppose now a new query, such that at decision level $c$ with current assignment set $\boldsymbol{A}_c$, the assignments implied by $\boldsymbol{T}_j$ are matched, and any assignment (under $\boldsymbol{A}$) at a decision level greater than $j$ is not contradicted. Then a solution to the present query is to append to $\boldsymbol{A}_c$ the result of the decision assignments in $\boldsymbol{A}$ after decision level $j$. These ideas form the basis for defining and using cached solutions.

Define the assignment set associated with each $\boldsymbol{P}_j$ and each $\boldsymbol{T}_j$ as follows:

$$\begin{aligned} \boldsymbol{A}(\boldsymbol{P}_j) &= \{(y, \nu(y)) | \eta(y) \in \boldsymbol{P}_j\} \\ \boldsymbol{A}(\boldsymbol{T}_j) &= \{(y, \nu(y)) | \eta(y) \in \boldsymbol{T}_j\} \end{aligned} \tag{3.30}$$

Assume a solution $\boldsymbol{A_S}$ (from (3.26)) to a query, and define each $\boldsymbol{P}_j$ and $\boldsymbol{T}_j$ accordingly. Next, assume a new query, such that the associated search process is currently at decision level $c$. Then, a solution can be identified if the following condition holds for one of the decision levels $j$ of the previous solution, with $1 \leq j \leq K$, $\boldsymbol{P}_j \neq \emptyset$ and $\boldsymbol{T}_j \neq \emptyset$:

$$C(j) = \prod_{(y,\, v_y) \in \boldsymbol{A}(\boldsymbol{T}_j)} (\nu(y) = v_y) \cdot \prod_{l\,=\,j+1}^{K} \left[ \prod_{(y,\, v_y) \in \boldsymbol{A}(\boldsymbol{P}_l)} (\nu(y) \neq \bar{v}_y) \right] \tag{3.31}$$

that basically requires that, under the current partial node assignment, the node assignments of the elements of the level cut $\boldsymbol{T}_j$ to be matched, and assignments of the previous solution, defined at decision levels greater than $j$, not be contradicted. The computed solution to the current query is thus defined by:

$$\boldsymbol{A_S}' = \{(x, \nu(x)) \,|\, x \in PI \wedge \nu(x) \neq X\} \cup \{(x, v_x) \,|\, x \in PI \wedge \nu(x) = X \wedge (x, v_x) \in \boldsymbol{A_S}\} \tag{3.32}$$

where $\boldsymbol{A_S}$ is the solution to the previous query, given by (3.26), that is associated with each $C(j)$. The following result guarantees that a solution can indeed be found if one of the conditions $C(j)$ is matched:

**Theorem 3.9.** If one of the conditions identified by (3.31) holds, then $\boldsymbol{A_S}'$ given by (3.32) is a solution to the query.

The implementation of the above ideas requires maintaining a database of solutions, $\Sigma_\varphi$, where each entry of $\Sigma_\varphi$ is a 2-tuple ($C$, $\boldsymbol{A}$), such that $C$ encodes one of the conditions given by (3.31) and $\boldsymbol{A}$ identifies the solution assignment set associated with the primary inputs that defines $C$. Whenever a new query is being processed, and after deriving logical implications, the database of solutions is consulted. If (3.31) holds for a condition $C$ of some entry ($C$, $\boldsymbol{A}$) of $\Sigma_\varphi$, then (3.32) is used, along with $\boldsymbol{A}$, to construct the primary input assignment representing the solution for the current query.

The definition of $\boldsymbol{T}_j$ can be simplified by not considering nodes in the j-frontier, i.e. by *only* considering, in (3.28), nodes with outgoing edges to nodes assigned at decision levels greater than $j$. In such a situation, each condition $C(j)$ must require the j-frontier of the current query to be

| Decision level | $JF(j)$ | $\boldsymbol{A}(\boldsymbol{T}_j)$ | Assignments at higher decision levels |
|---|---|---|---|
| $j = 0$ | $\{ z_1, z_2 \}$ | $\varnothing$ | |
| 1 | $\{ z_1, z_2 \}$ | $\{(x_1, 0)\}$ | |
| 2 | $\{ u \}$ | $\varnothing$ | $w_1, w_2, w_3, w_4, t_1, t_2, t_3, t_4, t_5$ |
| 3 | $\{ u \}$ | $\{(w_1, 0)\}$ | $w_2, w_3, w_4, t_1, t_2, t_3, t_4, t_5$ |
| 4 | $\{ u \}$ | $\{(t_1, 0)\}$ | $w_3, w_4, t_2, t_3, t_4, t_5$ |
| 5 | $\{ u \}$ | $\{(w_3, 0)\}$ | $w_4, t_2, t_3, t_4, t_5$ |
| 6 | $\varnothing$ | $\varnothing$ | none |

**Table 3.2:** Conditions for matching cached solution

included in the j-frontier of the cached solution. Let $JF_j$ denote the j-frontier at decision level $j$ for the previous query. Then (3.31) can be rewritten as follows:

$$C(j) \;=\; (JF(c) \subseteq JF_j) \cdot \prod_{(y,\, v_y) \,\in\, \boldsymbol{A}(\boldsymbol{T}_j)} (\nu(y) = v_y) \cdot \prod_{l \,=\, j+1}^{K} \left[ \prod_{(y,\, v_y) \,\in\, \boldsymbol{A}(\boldsymbol{P}_l)} (\nu(y) \neq \bar{v}_y) \right] \qquad (3.33)$$

This condition has some advantages over (3.31), particularly because we can just request the current j-frontier to be included (and not to exactly match) the j-frontier, at decision level $j$, of the previous query. (Note, however, that the above condition can still be somewhat restrictive with respect to the requirements on $\boldsymbol{A}(\boldsymbol{P}_j)$ and $\boldsymbol{A}(\boldsymbol{T}_j)$.)

**Example 3.26.** In order to illustrate how solutions are cached and used to reduce the amount of search for subsequent queries, we consider again the example of Figure 3.24. The analysis assumes (3.33), i.e. $\boldsymbol{T}_j$ does not include nodes in the j-frontier. After the solution to the first query (i.e. $z_1 = 1$ and $z_2 = 0$) is computed, the constructed decision tree and resulting node justification graph yield the data shown in Table 3.2. In particular, we record the j-frontier and the assignment set $\boldsymbol{A}(\boldsymbol{T}_j)$ associated with each element $\boldsymbol{P}_j$ of the partition of $V_J$. Consider now the second query (i.e. $z_3 = 1$ and $z_4 = 0$). The first two decision assignments are $y_1 = 1$ and $y_2 = 1$. At this point, the j-frontier becomes $\{u\}$. Furthermore, none of the nodes with assignments in $\boldsymbol{A}(\boldsymbol{P}_j)$, with $j > 2$, is assigned. Hence, (3.33) is satisfied for decision level $j = 2$ of the previous query. Using (3.32), the

solution to the current query becomes:

$$A_S' = \{(y_1, 1), (y_2, 1), (w_1, 0), (w_2, 0), (w_3, 0), (w_4, 0)\}$$

and hence the search effort of processing the last four decision assignments is saved. □

Note that caching solutions poses a tradeoff between the number of saved decision assignments and the overhead to manipulate the solutions database. Hence, caching solutions is only useful when the effort to compute a related solution is significant. For example, we can just cache solutions for which a significant number of backtracks is required or a large decision tree is constructed.

Note that redundancy removal for $k \geq 2$ is not guaranteed to increase the ability to match solutions in the solution database. Reducing the size of a solution implies that some later decision assignments now guarantee the implications previously derived by earlier decision assignments. Hence, the sequence of implications that allow finding a solution is changed. This may affect negatively the ability to match partial node assignments with minimized (and cached) solutions. Consequently, caching solutions must be handled independently of redundancy removal from solutions.

**Perspective**

The proposed procedure for caching solutions is inspired by Giraldi and Bushnell's work in test pattern generation [70, 71], even though the idea of caching solutions to search problems had been proposed before in other areas (see for example [57]). Nevertheless, our procedure introduces two improvements. First, it is independent of the test pattern generation problem representation. Second, and more important, the node justification graph eliminates some of the redundant decision assignments, whereas in Giraldi and Bushnell's work all decision assignments are considered[11]. Results reported in [71] indicate that partial solutions are often matched, thus allowing the

[11.] The description of Giraldi and Bushnell's algorithm (EST) [70, 71] does not specify how the j-frontier is handled. Clearly, the information associated with a given stage of the search process (in EST referred to as a state) must contain the j-frontier, since otherwise the procedure would be incorrect. Assuming that the j-frontier is properly encoded, then each state representation in EST includes some $C(j)$ (in our approach) as well as some superfluous conditions on other node assignments.

search process to terminate earlier. The extension of the method we propose to test pattern generation (see Chapter V) is guaranteed to identify no fewer matches than that of [71].

## 3.9 Decision Making Procedures

In this section we study techniques for guiding the search process. As with the analysis of other techniques, we emphasize the *non-heuristic* aspects of decision making. Consequently, we start by studying techniques for reducing the number of decision variables. We then study procedures for selecting decision assignments. In the last subsection, we describe heuristic techniques to choose decision variables in SAT algorithms which can be related to well-known heuristic principles of search. As in the previous section, we explicitly assume a clause database associated with a combinational circuit. Hence, for each clause database a set of primary inputs is well-defined.

### 3.9.1 Reducing the Number of Decision Variables

Procedures for reducing the number of decision variables in circuit satisfiability procedures date back to FAN [62], where the concept of (static) *head line* was first proposed. A head line is the output of a fanout-free sub-circuit [1, pp. 208-209], and can thus be satisfied to any logic value in linear time in the size of its transitive fanin[12]. During the course of our work, we proposed the concept of *dynamic* head line [155]. A dynamic head line is a circuit node that becomes the output of a fanout-free sub-circuit due to assignments to some of the remaining circuit nodes. While static head lines are computed before starting the search process, dynamic head lines are updated dynamically, as the search process evolves.

Another related concept is the notion of a (topological) *don't care* node, first proposed in [109]. Don't cares denote circuit nodes whose logic value is irrelevant for the satisfiability (or path sensitization) problem being solved. The identification of don't care nodes is useful because it may prevent long implication sub-sequences bearing no relevancy to the query being solved. An integrated algorithm for the dynamic identification of don't care nodes and head lines is described in [155], and it is basically based on counting, either statically or dynamically, the number of effec-

[12] Although not specified in [1], the function of each gate is assumed to be *reasonable*. For example, if the circuit is composed of gates whose computed output function is always identically 0 (or 1), then the output of a fanout free sub-circuit is not be satisfiable in linear time.

tive fanout nodes of each node, and on relating fanout-free head lines.

In this section we propose to relate the identification of head lines and don't cares in combinational circuits with the application of the consensus operation and the *pure literal rule* [38] on clause databases. This relationship then provides a formal justification to the identification of head lines and don't cares; it also provides new insights on how to simplify the search space by further reducing the number of decision variables in the clause database. The analysis is restricted to circuits composed of simple gates, where simple gates are represented with the template CNF formulas of Table 2.1 (see page 35).

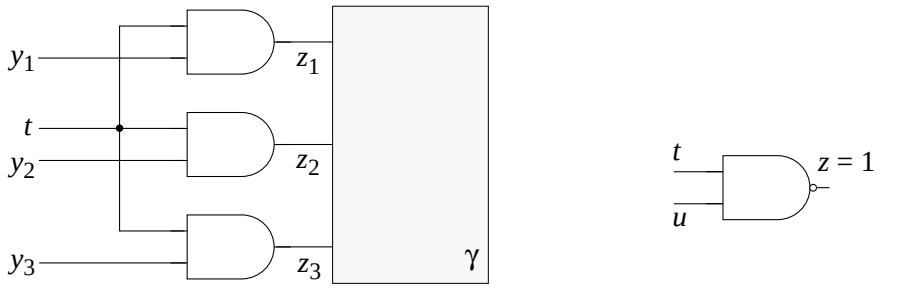
**Example 3.27.** Let us assume a combinational circuit and let $z = \text{AND}(x_1, x_2, x_3)$, such that $x_1$, $x_2$ and $x_3$ are *fanout-free head lines*. The corresponding CNF formula is given by:

$$(x_1 + \neg z) \cdot (x_2 + \neg z) \cdot (x_3 + \neg z) \cdot (\neg x_1 + \neg x_2 + \neg x_3 + z) \cdot \gamma \tag{3.34}$$

Thus, each of the variables $x_1$, $x_2$ and $x_3$ participates in exactly two clauses, in one as a positive literal, and in the other as a negative literal. Let $\varphi' \leftarrow \texttt{Consensus}(\varphi, x_1)$ denote the operation of Davis-Putnam resolution (see Figure 2.10 on page 48); then $\varphi'$ is independent of $x_1$. (Note that since $x_1$ is a fanout-free head line, then $x_1$ is only input to $z$.) Furthermore, in $\varphi'$, literals on $x_2$ and $x_3$ only appear as positive. Recall, from Section 2.5.4 on page 52, that the pure literal rule states that if a variable appears in only one literal form (either positive or negative), then all clauses containing such a literal can be removed, because assigning that literal satisfies those clauses without affecting any of the other clauses. Hence, the clauses containing a literal in $x_2$ or in $x_3$ can be removed. After this sequence of operations, the resulting clause database does not contain literals on $x_1$, $x_2$ and $x_3$. Furthermore, by proper bookkeeping, $z$ can now be identified as a new head line.

The application of restricted forms of consensus and the pure literal rule also permit the identification of dynamic headlines. Let us assume that for the above AND gate $x_1$ is not a headline. Further assume that $x_1$ is assigned value 1. Consequently, $(x_1 + \neg z)$ is satisfied and literal $\neg x_1$ is set to 0. The application of consensus with respect to $x_2$ and the subsequent application of the pure literal rule allow defining $z$ as a new head line. □

The above examples illustrate how the application of a restricted form of Davis-Putnam




(a) With fanout (b) Early justification

**Figure 3.25:** Simplification of clause databases

resolution and the pure literal rule can be used to describe the static or dynamic identification of head lines. The same techniques can be applied to a node that does not contribute to satisfying the set of objectives, i.e. a don't care node. For example, consider a fanout-free primary output $z$. Consensus with respect to $z$ just yields tautologous clauses. This fact is justified by the CNF formula of each gate. This formula has clauses with negative and positive literals on $z$. However, a clause with a positive literal in $z$ necessarily contains a literal $l$ whose complement also appears in the clause with the negative literal in $z$. Hence consensus yields a tautologous clause. We can thus conclude that by iterated application of restricted consensus, don't care variables are removed from the clause database.

**Example 3.28.** Consider again the example gate, $z = \text{AND}(x_1, x_2, x_3)$, but now $z$ is a don't care variable (without fanout nodes) and $x_1$, $x_2$ and $x_3$ are internal circuit nodes. Let us consider consensus with respect to $z$. The application of (3.34) only generates tautologous clauses, and thus all references to $z$ are erased from the clause database. □

The application of consensus and the pure literal rule can be used to describe further simplifications to the clause database that find an equivalent in its circuit counterpart.

**Example 3.29.** Consider the examples of Figure 3.25. For the example circuit of Figure 3.25-a, the derived clause database is:

$$
\begin{aligned}
&(y_1 + \neg z_1) \cdot (t + \neg z_1) \cdot (\neg t + \neg y_1 + z_1) \cdot \\
&(y_2 + \neg z_2) \cdot (t + \neg z_2) \cdot (\neg t + \neg y_2 + z_2) \cdot \\
&(y_3 + \neg z_3) \cdot (t + \neg z_3) \cdot (\neg t + \neg y_3 + z_3) \cdot \gamma
\end{aligned}
\tag{3.35}
$$

where $\gamma$ is a CNF sub-formula on $z_1$, $z_2$, $z_3$ and other variables, and where input variables $t$, $y_1$, $y_2$ and $y_3$ are known to be head lines. Furthermore, $y_1$, $y_2$ and $y_3$ are fanout-free, and $t$ fans out only to $z_1$, $z_2$, and $z_3$. Consider the application of consensus with respect to $y_1$, $y_2$ and $y_3$. Thus, $(t + \neg z_1) \cdot (t + \neg z_2) \cdot (t + \neg z_3) \cdot \gamma$ becomes the new clause database, where clauses in $y_1$, $y_2$ and $y_3$ are removed. Now we observe that $t$ only appears as a positive literal and thus, by the pure literal rule, the clause database is further simplified to $\gamma$. Consequently, even with $t$ not being fanout-free, we are able to simplify the clause database and base further decision assignments on the new head lines $z_1$, $z_2$ and $z_3$.

With respect to the example of Figure 3.25-b, let $t$ be a fanout-free head line. The original CNF formula for the NAND gate is given by $(t + z) \cdot (u + z) \cdot (\neg t + \neg u + \neg z)$. Since $z = 1$, then the binary clauses are satisfied, and the CNF formula reduces to $(\neg t + \neg u)$. In this situation, the value of $z$ can be justified by $t$, since clause $(\neg t + \neg u)$ can be satisfied by the pure literal rule (note that $t$ does not participate in any other clause because it is a fanout-free head line). In terms of node assignments, we say that since $t$ is a fanout-free head line, then we are free to assign to it the necessary value to justify $z$. Finally, we conclude that the set of objectives (which includes $z$) is satisfiable if and only if it is satisfiable without objective $z = 1$. This operation is *not* a decision, and hence it is no longer necessary to decide the assignment on $t$. □

Simplification of the clause database, with the goal of identifying don't care nodes and head lines, is organized as follows:

1. For each variable with either only one positive or negative literal, apply Davis-Putnam consensus.
2. Apply the pure literal rule to each variable whose literals are either all positive or all negative.
3. Repeat while the restricted consensus operation can be applied. Record changes to the set of head lines.

This procedure can be readily applied to a clause database, and taking into consideration the CNF formula of each gate, it can also be directly applied to structural circuit representations composed of simple gates. For more general combinational circuits, the algorithm of [155] can be used, and completed with the above procedure whenever it applies.

### 3.9.2 Deciding Assignments

The simplest decision making procedure is to assume a fixed order of the decision variables, and default assignments, and use that fixed order to guide the search process. It is commonly accepted that such procedures may lead to decision trees larger than necessary (on average problems), and are seldom used. Some exceptions do exist however. In test pattern generation, examples of static ordering decision making procedures can be found in the work of Cox and Rajski [37] and Larrabee [106]. Cox and Rajski's decision making procedure just uses the original ordering of the variables and always starts by assigning 0 to the decision variable. In [37], the large number of aborted faults for some of the benchmark circuits may be justified by this decision making procedure. Larrabee's algorithm orders variable assignments statically, by the expected number of implications they can (statically) cause. Different orders are proposed in [105]; for example, start by assigning variables to true with the largest expected number of implications. This variable order is then used to search for a solution.

Decision making procedures based on dynamic ordering basically choose the next decision assignment based on feedback from the search process. We distinguish two broad categories:

1. *Trace based*. These procedures use information, regarding the structure of the problem representation and the objective to be satisfied, to decide the next decision. Examples are *simple backtracing* [72, 160] and *multiple backtracing* [62, 144]. In all cases, a tentative goal (set of goals) is specified, which we refer to as the *decision objective*(*s*). Backtracing then chooses a decision assignment *likely* to satisfy the decision objective(s). Trace based procedures usually use topological measures to decide which paths to trace and which nodes to assign.
2. *Greed based*. These procedures try to make decision assignments that satisfy an immediate goal, while expecting that such decision simplifies identifying a solution to the query. Several mechanisms exist to choose the (greedy) decision assignment. [162] proposes choosing a de-

cision assignment that satisfies at least one clause. A simple extension would consist of estimating the number of satisfied clauses or assigned nodes and pick the decision assignment that maximizes the estimate. Some other greedy procedures attempt to satisfy the smallest clause in the clause database [64, 125].

Admittedly, one can devise a wealth of ways for deciding the next decision assignment. Experimental evidence in test pattern generation tends to support the impression that almost all reasonable decision making procedures are useful and that none is sufficient [26, 85, 123]. Usually, most decision making procedures are able to handle a large number of faults. However, most decision making procedures also have tremendous difficulty with a small number of faults, which vary with each procedure. As a result, in recent years some authors have proposed to dynamically switch between decision making procedures whenever a threshold on the number of backtracks is reached. Examples of the application of this idea can be found in [105, 123, 162, 174]. Experimental results proposed in [105, 123, 162, 174] suggest that dynamic switching can be particularly helpful.

Note that GRASP can readily allow for dynamic switching between decision making procedures, assuming that conflicts are diagnosed with `Diagnose`(). With `Diagnose`(), all conflict information is recorded as implicates of the clause database. Whenever the search process switches to a new decision making procedure, all relevant information regarding conflict diagnosis has been incorporated into the clause database, and can be used to prune the search for the new decision making procedure. Note that Diagnose_*Pm*() can also be used with dynamic switching between decision making procedures, but in this case the information of level conflicting assignment sets must be erased every time a new decision making procedure is chosen. Assuming `Diagnose_`*C*(), then we may define different search contexts for each decision making procedure, and switch between contexts after a pre-defined number of backtracks. This solution allows for dynamic switching while ensuring some form of conflict diagnosis.

### 3.9.3 Increasing the Number of Expected Implications

The *fail-first principle* [169, pp. 178-179] basically states that one should try to find conflicts as soon as possible in order to prevent the decision tree from becoming too hard to implicitly

enumerate. This principle can be used to justify decision making procedures proposed by some authors in test pattern generation, as for example the one in [167].

Assume an instance of SAT and an associated set of decision variables; either the set of primary inputs or the set of head lines. Given certain conditions, choosing other nodes to elect a decision assignment may imply the assignment of several head lines.

**Example 3.30.** Let us consider once more the example of Figure 3.25-a, and let the inputs $y_1$, $y_2$ and $y_3$ be head lines but not fanout-free. Suppose we are interested in satisfying an objective for which $z_1 = 1$ is traced, using either simple or multiple backtracing. Note that the assignment $z_1 = 1$ would immediately imply the assignment of two head lines, $y_1$ and $t$. Consequently, assigning $z_1$ can be preferable than making two decision assignments ($y_1$ and $t$) in order to attain the same traced objective. □

We refer to nodes that imply the assignment of several head lines as *covering lines*. (The concept was first proposed in [167] under the name *implying node*, but restricted to static head lines. The generalization to dynamic head lines is straightforward and only requires the necessary bookkeeping to relate each covering line to each updated set of head lines.) Covering lines are used to increase the average number of implications that result from a decision assignment. Results reported in [167] for test pattern generation indicate that a significant reduction on the number of decisions is obtained when covering lines are used as decision nodes. The major drawback of electing decision assignments on covering lines is that it may increase the size of the decision tree if backtracking is required. For the example above, if the search process backtracks to the decision level associated with the decision assignment on $z_1$, then $z_1 \leftarrow 0$, that does not imply the assignment of any head line. As a result, more decision assignments may now be required.

An extension of the previous technique is to predict, where possible, which covering lines set a node $x$ to either 1 or 0, referred to as the ON-SET and the OFF-SET of $x$, respectively. The derivation of ON and OFF sets can be used by decision making procedures that decide assignments on covering lines. In trace-based decision making procedures, the identification of ON and OFF sets can help in making decisions that imply assignments close to the decision objective. The (heuristic) motivation is that since more assignments are implied *close* to the traced objective, then

we are more *likely* to identify conflicts; this technique can be viewed as another example of applying the fail-first principle.

In the more general domain of SAT (in particular CNF-SAT), some decision making procedures have focused on the problem of satisfying a clause, and organizing the decision making procedure accordingly [125, 135]. For example, Monien and Speckenmeyer's [125] clause-based decision making procedure chooses decision assignments at a given decision level such that a chosen clause $\omega$ is to be satisfied. $\omega$ is associated with the decision level, and further assignments at the same decision level specifically discard previous assignments that satisfy $\omega$.

**Example 3.31.** Consider the formula $(x + \neg y + w) \cdot \gamma$, where $\gamma$ is a CNF sub-formula. Let us assume that decisions are to be made such that $(x + \neg y + w)$ is satisfied. Then either $x = 1$ satisfies the clause, or $y = 1$ satisfies the clause, *and* $x$ can be set to 0, or $w = 1$ satisfies the clause, *and* $x$ can be set to 0 *and* $y$ can be set to 1. Consequently, several decision assignments are collapsed into one decision having $m$ branches for a clause with $m$ literals. □

Experimental results on random instances of SAT suggest that this approach is more efficient than variable-based decision making [64]. It is interesting to note that in TPG algorithms, a related form of clause-based decision making for combinational circuits was proposed by Cha, Donath and Özgüner in [23] in 1978 (this technique is also analyzed in [1, p. 195]). According to [64], the notion of clause-based decision making can be traced back to the early 1970's in algorithms for solving the traveling salesman problem.

## 3.10 Summary and Perspective

In this chapter we described GRASP, a search algorithm for solving SAT, that can be configured by defining several types of engines that implement different tasks of the search process. The most significant contributions of GRASP with respect to other SAT algorithms can be summarized as follows:

1. The development of a formal framework supporting the application of *non-chronological* backtracking in search algorithms for SAT, based on the definition of implicates of the consistency function associated with each clause database. The definition of implicates also per-

mits the identification of *equivalent conflicting conditions* and the derivation of *failure-driven assertions*. Even though the application of non-chronological backtracking with Boolean constraint propagation was first suggested in 1980 by D. McAllester [115], and also described in [60, pp. 265-308], in the context of truth maintenance systems, our algorithm is the first to describe the integrated application of non-chronological backtracking, conflict-based equivalence and failure-driven assertions (and other pruning methods described in the previous sections) to solving SAT. In contrast with [60, 115], the framework we propose is directed towards exploiting the *structure* of implication sequences, by imposing partial orders on assigned nodes and thus defining *stronger* conflicting clauses in terms of these partial orders.

2. Development of several methods that exploit the structure of conflicts to further prune the search. First, we described how the structure of implication sequences leading to conflicts can be used for identifying *unique implication points* (UIPs). This fact increases the number of identified failure-driven assertions and associated conflicting conditions. Second, we described how UIPs permit implementing *iterated conflicts*, which can be used to identify more aggressive backtracking decision levels. Third, we showed that the identification of *multiple conflicts* can also increase the number of derived conflicting conditions and help identify lower backtracking decision levels.
3. Analysis of several tradeoffs in implementing conflict diagnosis that guarantee constant or polynomial increases in the size of the clause database.
4. The definition of a hierarchy of SAT algorithms based on *scalable deduction and diagnosis abilities*, respectively `Deduce_`*k*() and `Diagnose_`*j*(). Advanced deduction engines have been proposed before [24, 101, 145], but were not defined within a search framework that implements non-chronological backtracking and the remaining pruning methods. Furthermore, the proposed advanced deduction engines are based on prime implicate generation methods for identifying reduced implicates of the consistency function. The search framework supporting GRASP allows for different tradeoffs between deduction and diagnosis ability. For example, any degree of deduction ability can be defined, within a search context that implements several pruning methods for diagnosing conflicts.
5. The description of a *scalable preprocessing procedure* that completes the clause database

with implicates prior to answering queries. The proposed procedure expands preprocessing procedures developed by other authors [102, 144, 162], in that any degree of consistency can be attained. Its implementation follows that of advanced deduction engines, but the structure of implication sequences is analyzed with the goal of deriving more implicates of the consistency function.

6. A general procedure for *postprocessing*, in particular the removal of redundant decisions from solutions and the caching of signatures of solutions. The notion of caching solutions in search problems has been proposed before in other applications [57, 71]. [71] proposes caching solutions for test pattern generation, but as illustrated in Section 3.8.2, the information associated with each solution that is used in [71] can be somewhat redundant.

GRASP defines the basic algorithmic framework that is used in subsequent chapters for solving more specific forms of satisfiability problems, particularly those associated with path sensitization.

# CHAPTER IV

# A MODEL AND ALGORITHM FOR PATH SENSITIZATION

## 4.1 Introduction

Path sensitization is the problem of identifying assignments to the primary inputs of a combinational circuit in order to make some form of information observable at the primary outputs. The information to observe is application-dependent, and we abstractly refer to it as a *perturbation*. In test pattern generation a perturbation denotes an error signal (D or $\overline{D}$), whereas in timing analysis a perturbation denotes a signal transition.

### 4.1.1 Motivation

As mentioned earlier in Chapter I, path sensitization can be cast as a SAT problem and solved with SAT algorithms, and examples of this approach can be found in [24, 105, 120, 152]. Consequently, the algorithmic techniques described in the previous chapter could readily be applied to instances of SAT encoding instances of path sensitization. However, by representing instances of path sensitization as instances of SAT, the intrinsic topological structure of path sensitization is lost, and more search effort may then be required to derive inferences otherwise clear from the analysis of this structure. The ability to exploit this structure is then the main motivation for developing dedicated models and algorithms for path sensitization. Throughout this chapter, we show how search algorithms for path sensitization can use the structure of the problem in pro-

cesses of inference, either for deduction or for diagnosis purposes. Furthermore, structure also plays a crucial role in the definition of general conflicting conditions that are used to prune the amount of search.

### 4.1.2 Chapter Objectives

The first objective of this chapter is to introduce a new model for path sensitization, which is referred to as the *perturbation propagation* (or *p-propagation*) *model*. The main characteristic of the model is that it uncouples the logic value assumed by each node from the path sensitization properties associated with that node. We note that this uncoupling is natural, since path sensitization is a circuit analysis task that only seeks to identify valid conditions for propagating a perturbation to a primary output; this task can be tackled orthogonally to the task of assigning consistent logic values to the circuit nodes.

Besides providing a more natural representation for the path sensitization problem, the p-propagation model offers the following additional advantages:

- It provides a common framework for representing path sensitization in distinct applications.
- It allows most pruning methods described in Chapter III to be extended to path sensitization.
- It provides new insights that lead to the development of pruning methods specific to path sensitization.
- It permits pruning methods to be interchangeably used in different target applications.

The second objective of this chapter is to describe LEAP, a generic search-based algorithm for path sensitization based on the p-propagation model, that follows the organization of GRASP. Several concepts associated with the p-propagation model in the context of search are defined and their application motivated. Furthermore, the major ideas regarding the engines associated with the search algorithm for path sensitization are described. The description of conflict diagnosis procedures is emphasized, since path sensitization algorithms have seldom implemented conflict analysis techniques. Moreover, as mentioned above and as illustrated in the sequel, the p-propagation model permits most pruning methods described in Chapter III to be naturally extended to the path sensitization problem.

### 4.1.3 Chapter Outline

We start in Section 4.2 by introducing some additional definitions that are used in describing the model and algorithm for path sensitization. Section 4.3 describes the p-propagation model and emphasizes the properties of the model that are common to all target applications. Section 4.4 describes the organization of implications with the p-propagation model, with the purpose of motivating the implementation of search algorithms for path sensitization.

LEAP is described in Section 4.5. Its organization follows that of GRASP. In particular, conflict diagnosis can be adapted from GRASP, with some required modifications for handling conflicts due to propagation conditions.

## 4.2 Definitions

To facilitate the discussion of path sensitization we augment the definitions associated with combinational circuits given in Chapter II with a few additional concepts:

- Controlling value of a node $x$, $c(x)$ which is 0 for (N)AND, 1 for (N)OR, and inapplicable for the remaining node types. A node that has a controlling value is said to be *controllable*; otherwise, it is *uncontrollable*.
- Inversion polarity of a node $x$, $i(x)$, which is 0 for AND, OR, and BUFFER, and 1 for NAND, NOR, and NOT. It is inapplicable for XOR and XNOR.
- Edge predicates $c(y, x)$ and $nc(y, x)$ indicating, respectively, the presence or absence of a controlling value on node $y$ with respect to node $x$.When $x$ is controllable, $c(y, x) = [\nu(y) = c(x)]$ and $nc(y, x) = [\nu(y) = \neg c(x)]$. When $x$ is uncontrollable, $c(y, x) = 0$, and $nc(y, x) = [\nu(y) \neq X]$.
- Controlling inputs $C(x) = \{y \in I(x) | c(y, x)\}$ .
- Unassigned inputs $U(x) = \{y \in I(x) | \nu(y) = X\}$ .
- Node predicate $Cont(x)$ indicating if node $x$ is "controlled." $Cont(x) = [\nu(x) = c(x) \oplus i(x)]$ when $x$ is a controllable node; $Cont(x) = 0$ when $x$ is uncontrollable.
- Predicate $Just(x)$ was introduced in Section 2.3.2.1 and is defined to hold true if and only if $x$ is justified, i.e. $(x \neq X) \wedge (I(x) = \varnothing \vee x = g_x(I(x)))$. Node predicate $Unjust(x)$ is defined by $Unjust(x) = \neg Just(x)$.
- Relevant outputs:

$$R(x) = \{ y \in O(x) \mid c(x, y) \vee [\nu(x) = X] \vee \neg Cont(y) \} \tag{4.1}$$

For timing analysis purposes an edge $(x, y)$ between nodes $x$ and $y$ is characterized by a fixed nonnegative propagation delay $D(x, y)$. A path $\boldsymbol{P} = \langle s_1, s_2, \ldots, s_k \rangle$ is a sequence of connected nodes; $\boldsymbol{P}$ is a *partial* path if it does not start at a primary input or end at a primary output. The delay of a path $\boldsymbol{P}$, $D(\boldsymbol{P})$, is the sum of its edge delays:

$$D(\boldsymbol{P}) = \sum_{i=1}^{k-1} D(s_i, s_{i+1}) \tag{4.2}$$

Primary outputs are distinguished as separate circuit nodes of type OUT. The set of primary outputs *PO* is then defined as the set of all circuit nodes of type OUT. For example, for a gate whose output is a primary output $z$, the circuit graph will now contain a node $z'$ of type OUT, whose only fanin node is $z$. Primary inputs are defined to be of type IN. The introduction of node types IN and OUT allows us to add additional special purpose nodes to the circuit graph and still be able to identify primary inputs and outputs.

A test $\boldsymbol{T}$ denotes a set of logic assignments to the primary inputs of a combinational circuit. The path sensitization problem involves computing tests for several distinct purposes, e.g. test pattern generation, timing analysis and delay fault testing.

## 4.3 The Perturbation Propagation Model

### 4.3.1 Objectives of Perturbation Propagation

The p-propagation model seeks to identify primary input assignments that permit a perturbation to be observed at a primary output. This is achieved by uncoupling the information associated with the propagation of a perturbation from the logic value assumed by each node. As a result, in the p-propagation model, each circuit node is characterized by a logic value and a *propagation status* (or *p-status*). Logic values have their usual semantic meaning and the definitions of Chapter II apply, i.e. for a node $x$, $\nu(x) \in \{ 0, 1, X \}$. On the other hand, the p-status of a node $x$ is represented by the symbol $\pi(x)$, assuming values $\nu(\pi(x)) \in \{ 0, 1, X \}$, and denotes whether *x propa-*

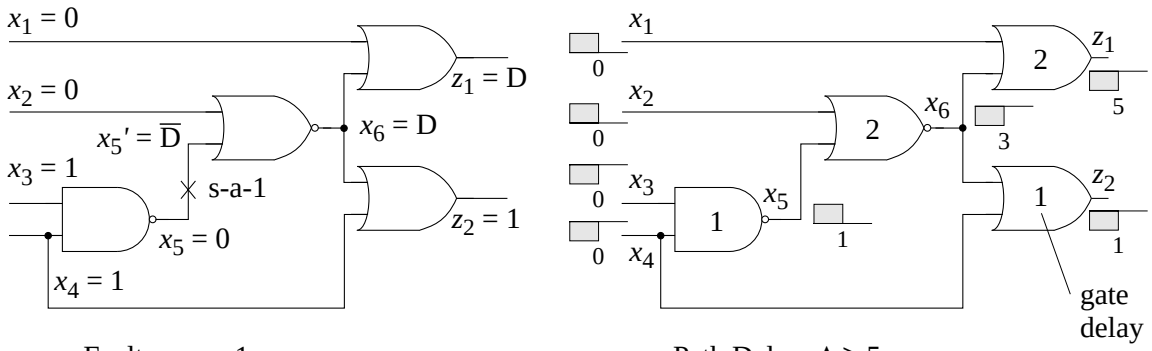
*gates a perturbation*. (When clear from the context, $\pi(x)$ is used instead of $\nu(\pi(x))$.) We say that a node propagates a perturbation if the primary input logic assignments permit the intended perturbation to propagate to the node. The p-status of a node $x$ is defined as follows:

1. $\pi(x) = 0$, also referred to as a p-false (or p-*F*) node, indicates that node $x$ cannot propagate the perturbation. This should be interpreted to mean that $x$ is not part of a sensitizable path given the current set of assignments to the circuit nodes.
2. $\pi(x) = 1$, also referred to as a p-true (or p-*T*) node, indicates that a perturbation propagates to node $x$. In general, $\pi(x) = 1$ if and only if under the current assignment to the primary inputs, $x$ is included in a sensitizable path. However, in the context of search this definition is relaxed, and $\pi(x) = 1$ is to be understood as signifying that, under the current assignments, one cannot conclude that $x$ is not part of a sensitizable path, and that conditions for propagating a perturbation to the node have been established.
3. $\pi(x) = X$, also referred to as a p-*X* node, indicates that the p-status of $x$ is *unassigned*. An unassigned p-status is to be understood as a potential ability to propagate a perturbation. Hence, in the context of a search process, $\pi(x) = X$ indicates that the current partial node assignment does not allow us to conclude that a perturbation cannot propagate to $x$.

**Example 4.1.** Figure 4.1 illustrates how the p-status of each node can be defined. For test pattern generation (Figure 4.1-a), the primary input assignments represent a test for fault $x_5$ s-a-1. The nodes that propagate the error signal are $x_5$, $x_6$ and $z_1$, which are then said to be p-*T*.

For timing analysis (Figure 4.1-b), the same primary input assignments cause node $z_1$ to stabilize after 5 time units. Hence, there exists at least one floating-mode sensitizable path in the circuit with delay no less than 5. The nodes that propagate such signal transition are $x_3$, $x_4$, $x_5$, $x_6$ and $z_1$. For the specific case of timing analysis, note that not all signal transitions correspond to p-*T* nodes; only nodes with transitions that *cause* primary output transitions at delay times no less than 5 are defined p-*T*. □

The p-propagation model can be viewed as considering two dimensions for characterizing the state of each node. A logical dimension that represents the logic value assumed by the node, and a propagation dimension that represents its propagation status. However, in contrast to the log-




Fault: $x_5$ s-a-1

p-*F* nodes: $x_1, x_2, x_3, x_4, z_2$

p-*T* nodes: $x_5, x_6, z_1$

(a) Test pattern generation

Path Delay: $\Delta \geq 5$

p-*F* nodes: $x_1, x_2, z_2$

p-*T* nodes: $x_3, x_4, x_5, x_6, z_1$

(b) Timing analysis

**Figure 4.1:** Representing path sensitization with the p-propagation model

ical dimension, the propagation dimension also assigns a p-status to the graph edges. For each edge $(x, y)$, $\nu(\pi(x, y)) \in \{ 0, 1, X \}$. (When clear from the context, $\pi(x, y)$ is used instead of $\nu(\pi(x, y))$.) The semantics of p-status for an edge $(x, y)$ is defined as follows:

1. $\pi(x, y) = 0$, also referred to as a p-false (or p-*F*) edge, indicates that a perturbation cannot propagate from node *x* to node *y*. This may be due to the fact that $\pi(x) = 0$, or due to side input conditions that block propagation from *x* to *y*.
2. $\pi(x, y) = 1$, also referred to as a p-true (or p-*T*) edge, indicates that a perturbation on node *x* propagates to node *y*. We note that this definition is relaxed similarly to the definition of a p-*T* node in the context of search.
3. $\pi(x, y) = X$, also referred to as a p-*X* edge, indicates that the p-status of the edge is unassigned. As with node p-status, and in the context of a search process, $\pi(x, y) = X$ means that the current partial node assignment does not allow us to conclude that a perturbation cannot propagate from *x* to *y*.

**Example 4.2.** For the example of Figure 4.1-a, $\pi(x_6, z_2) = 0$ because the error signal on $x_6$ does not propagate through this edge. $\pi(x_6, z_1) = 1$, because the error signal reaches $z_1$ through $(x_6, z_1)$. For the example of Figure 4.1-b, $\pi(x_6, z_1) = 1$, since the signal transition propagates from $x_6$ to $z_1$ with delay greater than or equal to 5. Conversely, $\pi(x_6, z_2) = 0$, since $z_2$ does not propagate a signal tran-

sition on a path whose delay is greater than or equal to 5. □

For each primary output $z$, let the $\Pi(z)$ denote the set of partial paths $\boldsymbol{P} = \langle s_1, s_2, \ldots, s_k \rangle$, with $z = s_k$, that can potentially propagate a perturbation from the source of the perturbation $s = s_1$. Then, the p-status of primary output $z$ is given by:

$$\pi(z) = \sum_{\langle s_1, \ldots, s_k \rangle \in \Pi(z)} \left[ \prod_{j=1}^{k-1} (\pi(s_j, s_{j+1}) = 1) \right] \tag{4.3}$$

Hence, a perturbation propagates to a primary output $z$ if it propagates along the edges of a partial path connecting the source of the perturbation $s$ to primary output $z$. The objective of the path sensitization problem is, then, to satisfy the condition,

$$\sum_{z \in PO} \pi(z) = 1 \tag{4.4}$$

subject to the logic assignments being consistent. This condition identifies the satisfiability problem associated with path sensitization. Any primary input assignment for which (4.4) holds is said to be a solution to that satisfiability problem.

A path $\boldsymbol{P} = \langle s_1, s_2, \ldots, s_k \rangle$ is said to *propagate a perturbation* if and only if the following holds:

$$\left[ \prod_{i=1}^{k-1} [(\pi(s_i) = 1) \cdot (\pi(s_i, s_{i+1}) = 1)] \right] \cdot (\pi(s_k) = 1) \tag{4.5}$$

If condition (4.5) holds then $\boldsymbol{P}$ is said to be a *sensitizable* path.

While in the logical dimension the consistent assignments are defined by a clause database $\varphi$ that identifies a consistency function $\xi$, in the propagation dimension the consistent assignments are captured implicitly by the structure of the combinational circuit and by condition (4.4). The assignments for which (4.4) is satisfied define a propagation consistency function $\xi_\pi$. Any primary input assignment for which (4.4) does not hold is said to identify a *propagation implicate* of $\xi_\pi$.

We note the *global* character of propagation implicates; *only if all propagation options are blocked does a conflict exist*. Propagation implicates can be derived and used while searching for a solution to a given instance of the path sensitization problem. Other, more general forms of propagation implicates, denoting localized structural and functional blocking conditions, are described in the sequel while studying search algorithms for path sensitization. In general, identified propagation implicates are referred to as *p-clauses*, and are maintained in a *p-clause database* $\varphi_\pi$.

Path sensitization can thus be viewed as the process of identifying a consistent assignment to the global consistency function $\xi_{PS} = \xi_\pi \cdot \xi$, where $\xi$ denotes the consistency function associated with the logical dimension. As with SAT, the search process can create implicates of $\xi_{PS}$, which can either be *logical implicates* (of $\xi$) or *propagation implicates* (of $\xi_\pi$). Such implicates can be used to reduce the amount of search during the search process. Furthermore, the applicability of each propagation implicate across path sensitization applications defines the degree of *pervasiveness* of the implicate. Different degrees of pervasiveness can be defined, as will be described in this and subsequent chapters. Note, however, that logical implicates can be used in *any* circuit analysis task, and thus are defined as *pervasive*.

### 4.3.2 Definition of Propagation Status

With each node $x$ we associate two predicates, $B(x)$ and $P(x)$, which respectively identify *blocking* and *propagation* conditions to node $x$. $B(x)$ identifies conditions under which it can be established that the perturbation cannot propagate to node $x$. $P(x)$ identifies conditions that permit the perturbation to propagate to node $x$, *given* that we cannot conclude that propagation of a perturbation to that node is blocked. Consequently, the p-status of node $x$ is defined as follows:

$$\pi(x) = \begin{cases} 0, \text{ if } (B(x)) \\ 1, \text{ if } (\neg B(x) \wedge P(x)) \\ X, \text{ if } (\neg B(x) \wedge \neg P(x)) \end{cases} \tag{4.6}$$

$\pi(x) = 0$ if $B(x)$ holds, i.e. if the blocking conditions to node $x$ have been established. Similarly, $\pi(x) = 1$ if $B(x)$ does not hold and $P(x)$ holds. This definition is intended to simplify the definition of $P(x)$, given that $P(x)$ is only considered whenever $B(x)$ does not hold true. The p-status of a node

is $X$ while both $B(x)$ and $P(x)$ do not hold.

A similar definition applies for the p-status of an edge $(x, y)$:

$$\pi(x, y) = \begin{cases} 0, \text{ if } (B(x, y)) \\ 1, \text{ if } (\neg B(x, y) \wedge P(x, y)) \\ X, \text{ if } (\neg B(x, y) \wedge \neg P(x, y)) \end{cases} \tag{4.7}$$

Each target application involving path sensitization is required to specify the conditions under which a node or an edge becomes p-false or p-true. These conditions involve logic values of other nodes as well as the p-status of other nodes and edges, and thus define the predicates $B$ and $P$. In general, the blocking predicates $B(x)$ and $B(x, y)$ are defined as follows,

$$\begin{aligned} B(x) = & \left[ \prod_{y \in I(x)} (\pi(y, x) = 0) \right] + \left[ \prod_{y \in O(x)} (\pi(x, y) = 0) \right] + \\ & \left[ \sum_{y \in I(x)} ((\pi(y, x) = 0) \cdot c(y, x)) \right] + B_C(x) \end{aligned} \tag{4.8}$$

and,

$$B(x, y) = [\pi(x) = 0] + [\pi(y) = 0] + B_C(x, y) \tag{4.9}$$

where $B_C(x)$ and $B_C(x, y)$ are defined to be application-dependent. Given that some components of $B(x)$ and $B(x, y)$ are defined to be application dependent, the propagation predicates $P(x)$ and $P(x, y)$ must also be defined to be application-dependent.

### 4.3.3 General Blocking Conditions

Given the above definition of the blocking predicates several application-independent blocking conditions become apparent.

$$[\pi(x) = 0] \Rightarrow \left[ \prod_{y \in O(x)} B(x, y) \right] \Rightarrow \left[ \prod_{y \in O(x)} (\pi(x, y) \leftarrow 0) \right] \tag{4.10}$$

indicates that if the p-status of a node $x$ is 0, then a perturbation cannot propagate from $x$ to any of

its fanout nodes. Hence, the p-status of each fanout edge of $x$ can be assigned p-*F*.

$$[\pi(x) = 0] \Rightarrow \left[ \prod_{y \in I(x)} B(y, x) \right] \Rightarrow \left[ \prod_{y \in I(x)} (\pi(y, x) \leftarrow 0) \right] \tag{4.11}$$

indicates that if a node $x$ cannot propagate a perturbation, then a perturbation that propagates to any of its fanin nodes does not propagate further through $x$. Consequently, the p-status of each of the fanin edges of $x$ can be assigned p-*F*.

Conversely, edge p-status can be used to derive node p-status:

$$\left[ \prod_{y \in I(x)} (\pi(y, x) = 0) \right] \Rightarrow B(x) \Rightarrow [\pi(x) \leftarrow 0] \tag{4.12}$$

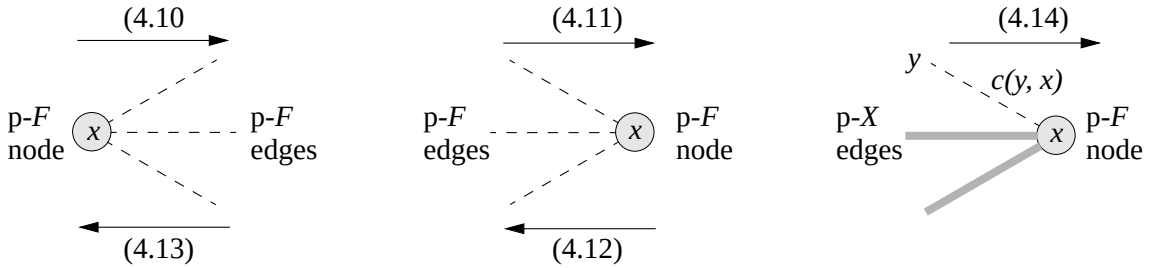
$$\left[ \prod_{y \in O(x)} (\pi(x, y) = 0) \right] \Rightarrow B(x) \Rightarrow [\pi(x) \leftarrow 0] \tag{4.13}$$

which signify, respectively, that propagation of a perturbation to a node is blocked if all of its fanin edges or all of its fanout edges are blocked. For simple gates, another blocking condition is defined when the fanin edge from node $y$ to $x$ is p-*F* and $y$ assumes the controlling value of $x$:

$$\left[ \sum_{y \in I(x)} ((\pi(y, x) = 0) \cdot c(y, x)) \right] \Rightarrow B(x) \Rightarrow [\pi(x) \leftarrow 0] \tag{4.14}$$

The above blocking conditions are illustrated in Figure 4.2, where the equation numbers identify the direction in which blocking is specified.

As mentioned above, propagation predicates are defined to be application-dependent since they depend on how $B_C(x)$ and $B_C(x, y)$ are defined. The definition of these predicates for test pattern generation and timing analysis is given in subsequent chapters. Nevertheless, in the remainder of this chapter we need, in some cases, to illustrate how a node or edge can be set to p-*T*. Consequently, the following definitions of propagation predicates are assumed:




(a) Fanout conditions (b) Fanin conditions (c) p-*F* blocking

**Figure 4.2:** Application-independent blocking conditions

$$P(x) = \prod_{y \in I(x)} [(\pi(y, x) = 1) \cdot (v(y) \neq X) + (\pi(y, x) = 0) \cdot nc(y, x)] \quad (4.15)$$

which states that the propagation condition to a node holds if all input nodes are assigned, and are either associated with p-*T* edges or assume non-controlling values. For an edge,

$$P(x, y) = [\pi(x) = 1] \quad (4.16)$$

which only requires the fanin node to be set to p-*T*. These conditions identify a valid subset of the propagation conditions for both path sensitization applications (test pattern generation and timing analysis) given the different definitions of $B_C(x)$ and $B_C(x, y)$ in the following chapters.

### 4.3.4 Search Framework

Path sensitization can be cast as a search problem. Besides requiring the definition of blocking and propagation conditions associated with each target application, the p-propagation model assumes an initialization phase that determines which nodes and edges are initially set to p-true, to p-false and to p-*X*, where the initial set of p-*T* nodes and edges defines the *source(s) of the perturbation*. This initialization phase characterizes the perturbation to be propagated and how propagation can be attained, and thus defines how the search is to be conducted.

Changes to the p-status of each node and edge are characterized by how the search process evolves. In general, as the search process evolves, p-*X* nodes or edges are allowed to be *down-graded* to p-*F* or *upgraded* to p-*T*. The evolution of the p-status (for nodes and edges) with the

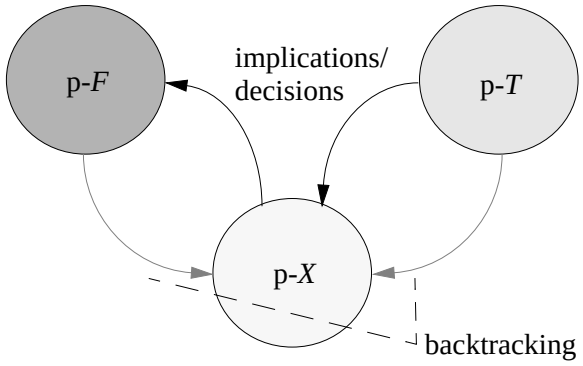



**Figure 4.3:** Evolution of the p-status with the search process

search process is shown in Figure 4.3. As the search process evolves, decision assignments and implications cause p-*X* nodes to either be downgraded to p-*F* or upgraded to p-*T*. Whenever the search process backtracks, p-*T* and p-*F* nodes are reset to p-*X*. (Note that the definition of p-status of a node/edge would allow a p-*T* node/edge to be downgraded to p-*F*. Although valid, this p-status modification is not necessary from a search perspective, and it is only performed after a solution to the path sensitization problem is found. If considered, these implications would increase the processing overhead without pruning the search and would unnecessarily complicate the search algorithms.)

We say that a *propagation conflict* is identified whenever no primary output can be set to p-*T* (conversely, when all primary outputs are set to p-*F*). As a result, in the p-propagation model there can be two types of conflicts: logical conflicts involving inconsistent gate input output assignments, and propagation conflicts denoting the impossibility to propagate a perturbation to a primary output. It is important to note that while logical conflicts have a *local* characterization due to inconsistent node assignments, propagation conflicts are characterized *globally*, denoting the impossibility to propagate a perturbation (over all potential propagation paths) to a primary output.

#### 4.3.4.1 Propagation Cuts

We define a *propagation cut* (or *p-cut*) as a set of p-*T* nodes of which at least one *must* be included in a sensitizable path. Propagation cuts identify the possible propagation options for a perturbation, and in general we allow for several propagation cuts to exist at any given stage of the search process. The set of p-*X* nodes driven by a propagation cut defines a *propagation frontier* (or

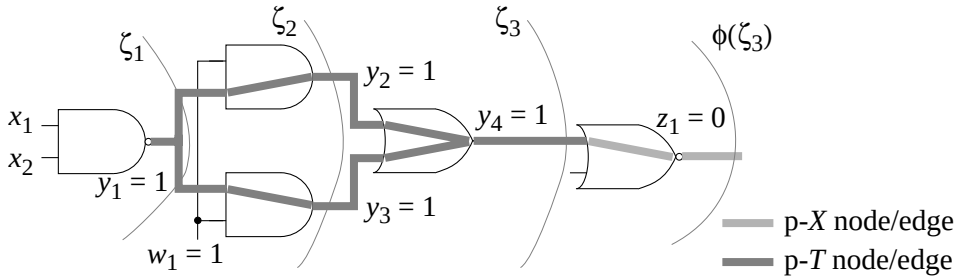



**Figure 4.4:** Examples of propagation cuts

*p-frontier*). Given a propagation cut $\zeta$, its associated p-frontier is specified by $\phi(\zeta)$. As with propagation cuts, we allow for several p-frontiers to exist at any given stage of the search process.

**Example 4.3.** Examples of p-cuts are shown in Figure 4.4, where $y_1$ is assumed to be the source of the perturbation. $\zeta_1 = \{ y_1 \}$ identifies a propagation cut, and so do $\zeta_2 = \{ y_2, y_3 \}$ and $\zeta_3 = \{ y_4 \}$. In general the number of propagation cuts can be exponential in the number of circuit nodes. However, only a few p-cuts are of interest, as is described below. □

With each set of nodes $\zeta$ we associate a function *pcut*($\zeta$) that is true whenever $\zeta$ is a p-cut, false whenever $\zeta$ cannot be a p-cut, and *X* whenever $\zeta$ may become a p-cut. In this situation, we allow for the pair ($\zeta$, 1) to be included in assignment sets to denote the fact that the set of nodes $\zeta$ has been identified as a p-cut and so *pcut*($\zeta$) holds. Whenever appropriate, $\nu(\zeta)$ assumes the value of *pcut*($\zeta$).

Propagation cuts can require justification of two different types. *Fanin justification* of a cut $\zeta$ denotes the process of propagating a perturbation to a node in $\zeta$ and which has propagated from the initial source of a perturbation. *Fanout justification* denotes the process of propagating a perturbation in a cut $\zeta_a$ to another cut $\zeta_b$ in the transitive fanout of $\zeta_a$, or to propagate a perturbation to a primary output. The original set of p-*T* nodes identifies a propagation cut $\zeta_i$, that only requires fanout justification. (In case there are no initial p-*T* nodes, the set of primary inputs is connected to a source node $\sigma_i$, that defines a propagation cut $\zeta_i$, i.e. $\zeta_i = \{ \sigma_i \}$, and which only requires fanout justification.) The primary outputs of the circuit fan out to a sink node $\sigma_o$ that defines a propagation cut $\zeta_o$, i.e. $\zeta_o = \{ \sigma_o \}$ and which only requires fanin justification. Consequently, the search entailed by path sensitization can be associated with the process of fanin and fanout justification of propagation cuts. Propagation frontiers, defined above, are only associated

with propagation cuts that require fanout justification. The set of propagation cuts requiring fanin justification defines a *propagation justification frontier* (or *pj-frontier*). The set of propagation cuts that require any form of justification is represented by $\Phi$, and denotes the set of p-cuts considered at any stage of the search process.

Given the above formulation of path sensitization in terms of justification of propagation cuts, the process of fanin justifying a propagation cut $\zeta_1$ corresponds in any situation to the process of fanout justifying another propagation cut $\zeta_2$ in the transitive fanout of $\zeta_1$. It is important to note that the process of fanin and fanout justification of p-cuts can be used in the definition of different types of propagation conflicts, as will be illustrated in the following sections.

Let us assume a propagation cut $\zeta_s$ with another propagation cut $\zeta_t$ in its transitive fanout and another cut $\zeta_r$ in its transitive fanin. Fanin and fanout justification can be respectively formalized as follows:

$$fi\_just(\zeta_s) = [\exists \zeta_r \in \Phi, \exists x \in \zeta_r, \exists y \in \zeta_s, (x \in I(y)) \wedge \neg B(y) \wedge P(y) \wedge (\pi(x, y) = 1)] \quad (4.17)$$

and

$$fo\_just(\zeta_s) = [\exists \zeta_s \in \Phi, \exists x \in \zeta_t, \exists y \in \zeta_s, (y \in I(x)) \wedge \neg B(x) \wedge P(x) \wedge (\pi(y, x) = 1)] \quad (4.18)$$

Fanin justification of a p-cut $\zeta_s$ requires a fanin p-cut $\zeta_r$ to be adjacent to $\zeta_s$, and propagation from $\zeta_r$ to $\zeta_s$ to be possible. Fanout justification requires the same property to hold but with respect to a fanout cut $\zeta_t$.

**Example 4.4.** For the example circuit in Figure 4.4, p-cuts $\zeta_1$ and $\zeta_2$ are fanout justified, since (4.18) holds for both. (Observe that p-cut $\zeta_1$ identifies the source of the perturbation and consequently only requires fanout justification.) p-cuts $\zeta_2$ and $\zeta_3$ are fanin justified since (4.17) holds for both. Finally, p-cut $\zeta_3$ requires fanout justification. □

#### 4.3.4.2 Unique Sensitization Points

A node that must propagate a perturbation for such perturbation to reach a primary output is referred to as a *unique sensitization point* (USP) [62, 92, 145, 155]. This concept has been

extensively used in algorithms for test pattern generation, and only recently was it applied to timing analysis [156]. By definition, a USP *x must* propagate a perturbation, and hence it defines a propagation cut $\zeta = \{ x \}$, *that requires both fanin and fanout justification*. Whenever a USP $x$ is identified by the search process, a new set of p-*T* nodes (of size 1) exists that must be in a path that propagates a perturbation to a primary output. This new propagation cut is then added to the set of propagation cuts $\Phi$. Besides creating the propagation cut $\zeta$, node $x$ is assigned p-*T*, since it must propagate a perturbation and is included in a propagation cut.

For the search algorithms described in the sequel, fanin justification is always restricted to p-cuts of size 1, that are derived from identification of USPs. Consequently, the process of fanin justifying p-cuts becomes simplified, since only one p-*T* node is involved.

Associated with USPs we have unique sensitization implications (USIs). A USI is defined as the assignment of a node, in the logical dimension, that is required for the fanin justification of the propagation cut (of size 1) associated with the USP.

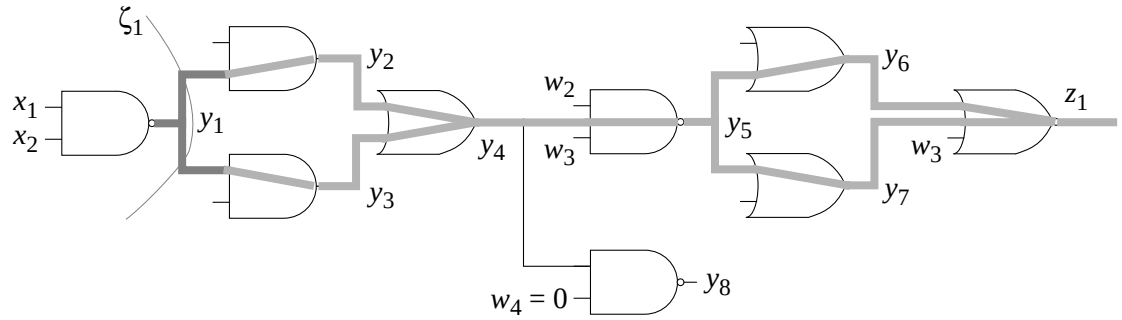
**Example 4.5.** Examples of USPs and associated USIs are shown in Figure 4.5. Given that $\zeta_1 = \{ y_1 \}$ is a p-cut, then $y_4$, $y_5$ and $z_1$ are USPs, and consequently define p-cuts of size 1, which require fanin and fanout justification. The resulting USIs are, respectively, none for $\zeta_2 = \{ y_4 \}$, $w_1 = w_2 = 1$ for $\zeta_3 = \{ y_5 \}$ and $w_3 = 0$ for $\zeta_4 = \{ z_1 \}$. Given the definition of $P(x)$ and $P(x, y)$ in (4.15) and (4.16), then $\zeta_2$ becomes fanout justified and $\zeta_3$ becomes fanin justified. □

It is important to note that USPs and USIs can be viewed as unique implication points (UIPs), that were defined in Section 3.4.2 (see page 69), but which can be identified as implications by the deduction engine and need not be identified by the diagnosis engine after conflicts are found. Hence, identification of USPs can contribute to preventing conflicts from being identified.

Given the definition of p-cuts and USPs, and in terms of a given stage of the search process, there may exist alternating sequences of sets of p-*T* nodes and sets of p-*X* nodes. Each set of p-*T* nodes is associated with a p-cut that can require fanin or fanout justification.

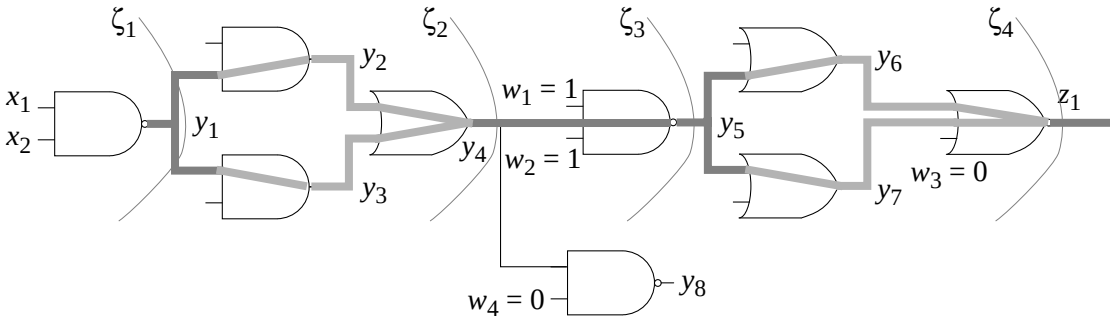
## 4.4 Derivation of Implications

The goal of path sensitization is to identify a consistent primary input assignment for




(a) Before USP/USI identification

p-*X* node/edge
p-*T* node/edge




(b) After USP/USI identification and resulting implications

**Figure 4.5:** Example USPs and USIs

which (4.4) is satisfied, thus requiring the value of the propagation consistency function $\xi_{PS}$ to be 1. Hence, elective assignments in either the logical or propagation dimensions can imply other assignments that are involved in satisfying (4.4).

The set of symbols ***V***, defined in Chapter II, is now extended to include two copies of each circuit node $x$ (one for each dimension, $x$ and $\pi(x)$) and one copy for each edge $(x, y)$, $\pi(x, y)$. Accordingly, assignment sets are defined over the extended set of symbols ***V***.

The derivation of implications in the p-propagation model is illustrated in Figure 4.6. Either a logic assignment or a propagation assignment is assumed, which triggers a sequence of implications, in either dimension. The pseudo-code for deriving implications is given in Figure 4.7. It consists of a loop that repeatedly invokes boolean constraint propagation (BCP), for logical implications, and perturbation constraint propagation (PCP), for perturbation propagation implications. While deriving implications, updating application-specific state information is optionally allowed with `Target_Application_Update`(). Procedure `BCP`() is defined in Figure 2.6 on

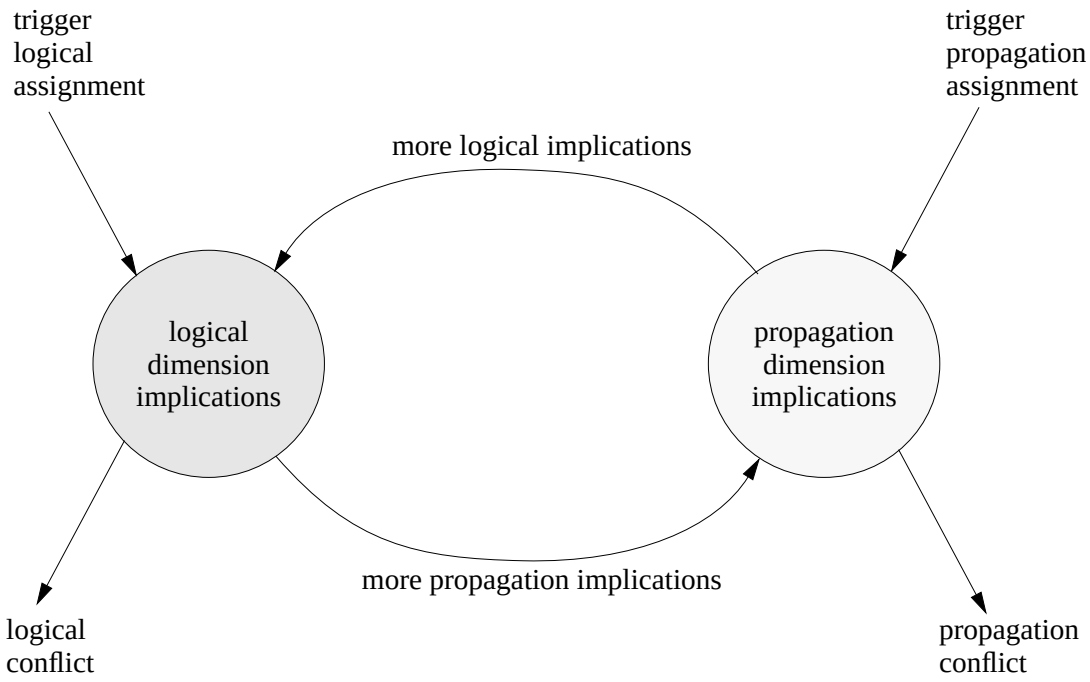



**Figure 4.6:** Derivation of implications in the p-propagation model

```
// Input arguments:    The initial assignment set
// Output arguments:   status ∈ { SUCCESS, CONFLICT }
// Return value:       The final assignment set
//
PS_Constraint_Propagation(A, &status)
{
   status = SUCCESS;
   do {
      A ← BCP(A, status);
      if (status == CONFLICT) return A;
      Target_Application_Update();   // Application-specific update
      A ← PCP(A, status);
      if (status == CONFLICT) return A;
   }
   while (changes to assignments in either dimension);
   return A;
}
```

**Figure 4.7:** Pseudo-code for derivation of implications

page 39, whereas propagation implications are described below.

The basic implication procedure for the derivation of propagation implications is shown in

```
// Input arguments:    The initial assignment set A_f
// Output arguments:   status ∈ { SUCCESS, CONFLICT }
// Return value:       The final assignment set A_f
//
PCP (A_i, &status)
{
   status = SUCCESS;
   A_f ← A_i;                          // Initialize final assignment set
   commit assignment A_f;  // Set initial partial variable assignment
   Let θ be a node or an edge;
   while (θ is p-X and (B(θ) or P(θ)) or all POs are p-F) {
      if (all POs are p-F) {
         status = CONFLICT;
         return A_f;
      }
      else if (B(θ)) π(θ) ← 0;
      else if (P(θ)) π(θ) ← 1;
   }
   return A_f;
}
```

**Figure 4.8:** Perturbation constraint propagation (PCP)

Figure 4.8. The implication procedure is adapted from BCP() for the logical dimension, but where changes to p-*X* nodes either block or propagate a perturbation to a node/edge or identify a propagation conflict. (Note that only p-*X* nodes can be assigned to either p-*T* or p-*F*, that guarantees that a p-*T* node will not be re-assigned to p-*F*.) Assigning a node or an edge to p-*T* can be viewed as the process of updating an existing propagation cut for the purpose of fanout justifying that cut. A conflict is identified whenever all primary outputs become p-*F*, denoting that a perturbation cannot reach any primary output.

When describing the deduction engine in the context of search, the implication procedure shown in Figure 4.8 requires manipulating additional information as Deduce() (Section 3.5 on page 74) does with respect to BCP().

**Example 4.6.** An example of an implication sequence is shown in Figure 4.9. The assignment $w_2 = 0$ implies $y_3 \leftarrow 1$ with BCP() and $\pi(y_3) \leftarrow 0$ with PCP(), since $\pi(w_2, y_3) = 0$ and $c(w_2, y_3)$ holds, and consequently $\pi(y_3, y_4) \leftarrow 0$. Because $c(y_3, y_4)$ holds and $\pi(y_3, y_4) = 0$, then $\pi(y_4) \leftarrow 0$,

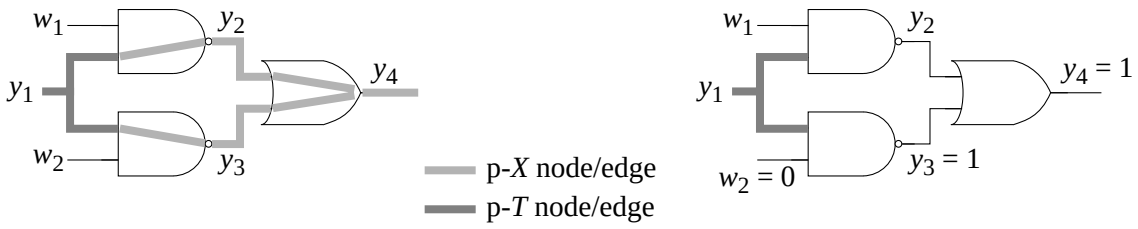



**Figure 4.9:** Propagation implication sequence

which blocks propagation to a primary output. Further note that the example is independent of any target application, since only general blocking conditions were considered. □

The description of the procedure for derivation of implications does not manipulate p-cuts and does not identify USPs. Hence, conflicts are defined in terms of having all primary outputs set to p-*F*. The application of p-cuts and USPs, and the definition of other types of conflicts will be considered while describing the search algorithm.

## 4.5 Search Algorithms for Path Sensitization

Search algorithms for path sensitization based on the p-propagation model follow the general structure proposed in Figure 3.2 on page 62. The description of LEAP is shown in Figure 4.10. The main difference with respect to the organization of GRASP (shown in Figure 3.2) is that we may need to downgrade some p-*T* nodes to p-*F* after a solution is computed. Furthermore, an application-dependent problem specification is abstractly defined by $\psi$, that encodes the initialization associated with the given instance of path sensitization, and that includes the p-clause database of propagation implicates.

The organization of the search algorithm for path sensitization, `PS_Search`(), is equivalent to the `Search`() procedure described in Figure 3.2 (see page 62) for SAT. By properly defining the different engines, the search algorithm solves path sensitization. Moreover, most algorithmic techniques described in Chapter III to prune the search find direct application in path sensitization. In particular, the search algorithm for path sensitization (in both testing and timing) can implement *conflict-directed backtracking*, *failure-driven assertions* and *conflict-based equivalence*.

The selection engine is restricted to making decision assignments on the logical dimen-

```
// Global variables:      Clause database φ
//                        Formulation of path sensitization ψ
//                        Partial variable assignment A
// Return value:          FAILURE or SUCCESS
// Auxiliary variables:   Backtracking decision level β_L
//
LEAP ()
{
   if (Preprocess() == SUCCESS and PS_Search (0, β_L) == SUCCESS){
         Postprocess();
         Toggle_Propagation_Values();        // p-T to p-F downgrade
         return SUCCESS;
   }
   return CONFLICT;
}

// Input argument:        Current decision level c
// Output argument:       Backtracking decision level β_L
// Return value:          CONFLICT or SUCCESS
//
PS_Search (c, &β_L)
{
   if (Select (VAR+VAL) == SUCCESS)              // Make decision
      return SUCCESS;
   while (TRUE) {
      if (PS_Deduce() != CONFLICT) {            // Imply assignments
         if (PS_Search (c + 1, β_L) == SUCCESS) return SUCCESS;
         else if (β_L != c) { Erase(); return CONFLICT; }
      }                                         // Diagnose conflict
      if (PS_Diagnose (c, β_L) == CONFLICT) { Erase(); return CONFLICT; }
      Erase();
      Select (VAL);                  // Modify decision assignment
   }
}
```

**Figure 4.10:** Description of LEAP

sion and with respect to the primary inputs (or head lines when possible). The formulation of the p-propagation model guarantees that by assigning all primary inputs, the p-status of all nodes and edges either becomes p-*T* or p-*F*, and hence the set of logic assignments over all primary inputs is

sufficient to implicitly enumerate the search space for path sensitization.

The selection, implication and diagnosis engines for path sensitization are necessarily different from those of SAT. In the remainder of this chapter we describe the general structure and operation of the implication and diagnosis engines, but abstractly, without describing application-dependent details. The implementation of these engines must necessarily reflect the two-dimensional properties of the p-propagation model. In the following chapters, each of these engines, as well as the selection, preprocessing and solution engines are detailed for each target application.

The soundness and completeness of the search algorithm for each target application are argued in Appendix B, assuming the results of Appendix A for SAT and the algorithmic framework described in this and the following chapters.

### 4.5.1 Search Structures

LEAP assumes the same definitions used for GRASP, but extends them to cover a few additional search structures. The search information associated with logic assignments in GRASP is now associated with nodes in both dimensions, edges in the propagation dimension and propagation cuts. Any of these entities is referred to by $\theta$, and thus $\theta$ can represent a node in the logical dimension, $x$, a node in the propagation dimension, $\pi(x)$, the p-status of an edge, $\pi(x, y)$, or a p-cut $\zeta$. Consequently, for each $\theta$ we define $\nu(\theta)$, $\boldsymbol{A}(\theta)$, $\alpha(\theta)$, $\delta(\theta)$ and $\iota(\theta)$. The decision and implication levels are given by (3.3) on page 60. The notation $\theta = v \mathbin{@} d / i$ denotes that symbol $\theta$ is assigned value $v$ at decision level $d$ and implication level $i$. The antecedent assignment and antecedent set of p-status assignments are defined by how a node or edge is downgraded to p-*F* or upgraded to p-*T*. For common blocking conditions, the antecedent assignment is defined by the set of assignments involved in blocking, which is also used for defining the antecedent set. The antecedent assignment and antecedent set of a p-cut are formally defined below. In general the antecedent assignment of a p-cut is characterized by the assignments that lead to its definition.

**Example 4.7.** For the example circuit of Figure 4.5 (see page 148), the antecedent assignment of $\zeta_2$ is given by $\boldsymbol{A}(\zeta_2) = \{ (\zeta_1, 1) \}$, since no node or edge assignments are involved in defining $\zeta_2$ as a p-cut given that $\zeta_1$ is also a p-cut. On the other hand, $\boldsymbol{A}(\zeta_3) = \{ (\zeta_2, 1), (\pi(y_4, y_8), 0) \}$, since $\zeta_3$ can be defined as a propagation cut (and as a USP) because $\zeta_2$ is a propagation cut and because

propagation from $y_4$ to $y_8$ is blocked. □

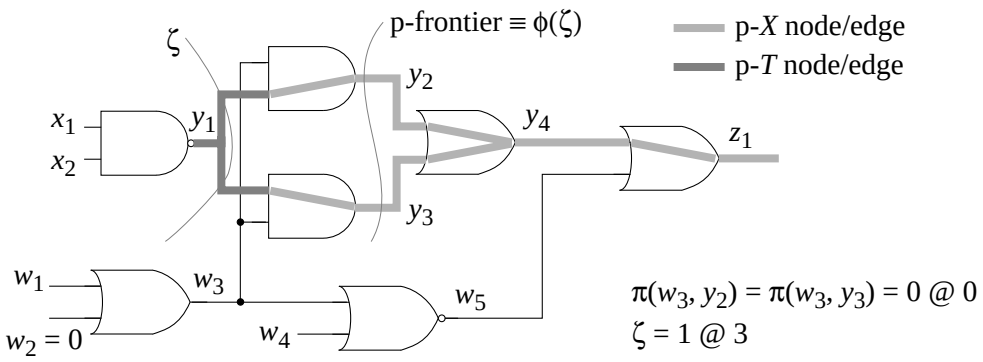
**Definition of Conflicts**

Throughout the description of search algorithms for path sensitization, instead of defining propagation conflicts by the condition of all primary outputs being p-*F*, we say that a propagation conflict is identified whenever a propagation cut cannot be either fanout justified or fanin justified. If a propagation cut cannot be fanout justified, then a perturbation cannot reach the next propagation cut, and consequently propagation of a perturbation to a primary output is blocked. A propagation cut that cannot be fanout justified is identified whenever its associated p-frontier becomes empty. On the other hand, a propagation cut that cannot be fanin justified identifies a situation where propagation of a perturbation becomes blocked at the input nodes to a p-cut. Both types of propagation conflicts capture the same global path blocking condition.

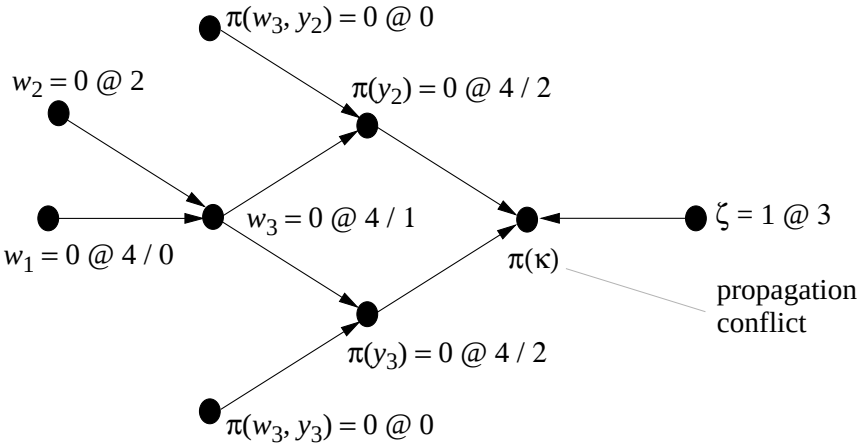
Propagation conflicts are represented in the implication graph by *propagation conflict nodes* $\pi(\kappa)$, such that the antecedent assignment of $\pi(\kappa)$ corresponds to the causes that directly create the conflict. These causes may involve propagation cuts and node/edge assignments.

**Example 4.8.** An example of an implication sequence and resulting propagation conflict is shown in Figure 4.11. Let us assume the propagation cut $\zeta = \{ y_1 \}$, with $\delta(\zeta) = 3$, that requires fanout justification. Further assume that the current decision level is 4. The decision assignment $w_1 = 0$ blocks propagation to $y_2$ and to $y_3$. Since all nodes in the p-frontier become p-*F*, then a propagation conflict is identified. Conversely, note that eventually the only primary output $z_1$ becomes p-*F*, which would also identify a propagation conflict. The antecedent assignment of $\pi(\kappa)$ is given by the nodes in the propagation frontier that become p-*F* and by the propagation cut that defines the propagation options that were blocked. Before the decision assignment, $\zeta = \{ y_1 \}$ identifies all propagation options for a perturbation to reach a primary output. Since the assignment $w_1 = 0$ blocks all propagation options, then propagation of a perturbation is blocked due to the fact that some nodes and edges become p-*F*, and to the fact that $\zeta$ is a propagation cut. Note that if p-cut $\zeta$ contained more elements, the propagation conflict would not be detected. □

By allowing the existence of multiple propagation cuts and associated p-frontiers, we can identify multiple propagation conflicts, as the following example illustrates.




(a) Example circuit




(b) Implication sequence

**Figure 4.11:** Example of an implication sequence

**Example 4.9.** An example of multiple propagation conflicts is illustrated in Figure 4.12. The assignments $x_1 = w_1 = w_2 = 0$ and $w_3 = w_4 = 1$ block propagation of a perturbation that can be attributed to four distinct causes: fanout justification of $\zeta_2$ and of $\zeta_3$ and fanin justification of $\zeta_2$ and of $\zeta_3$.

It is interesting to analyze the propagation conflict associated with the fanin justification of $\zeta_2$. We have $\pi(x_2) = 1$, $\pi(x_2, y_1) = 1$ and $\pi(y_1) = 1$. However, the assignment $\pi(y_1) = 1$ resulted from $\zeta_2$ being a propagation cut, and propagation conditions to that node had not been established yet. As a result, propagation of a perturbation to $y_1$ is blocked by $x_1$, $\zeta_2$ cannot be fanin justified, and a propagation conflict $\pi(\kappa)$ is defined. The antecedent assignment of $\pi(\kappa)$ is given by $\alpha(\pi(\kappa)) = \{ (\zeta_1, 1), (x_1, 0), (\pi(x_1, y_1), 0) \}$; propagation of a perturbation is blocked because $\zeta_1$ is a

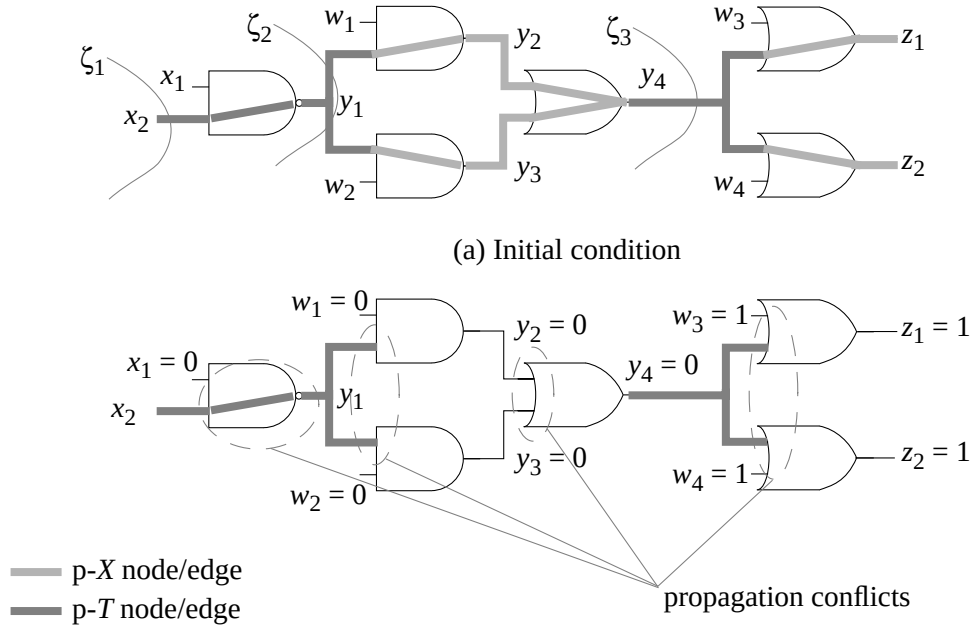



**Figure 4.12:** Different types of propagation conflicts

propagation cut, $x_1$ assumes a controlling value, and $\pi(x_1, y_1) = 0$.

Assuming that the diagnosis engine can handle multiple conflicts, then the propagation conflict yielding the best pruning conditions among the four conflicts can be chosen. □

### 4.5.2 Basic Deduction Engine

The basic deduction engine for a search algorithm based on the p-propagation model is shown in Figure 4.13 and is based on the procedure for derivation of implications described in Figure 4.7. After each successful implication sequence a new set of propagation cuts is created, each of which is defined at the current decision level. The logical deduction engine corresponds to `Deduce`() described in Figure 3.6 on page 77, whereas the propagation deduction engine is described in Figure 4.14, and is referred to as `Propagation_Deduce`(). Given that a p-clause database $\varphi_\pi$ is maintained, `Deduce`() is reformulated to also identify unsatisfied p-clauses of $\varphi_\pi$.

`Propagation_Deduce`() basically implements `PCP`() as described in Figure 4.8, but incorporates additional functionality associated with the search process. In particular, it implements defining antecedent sets, decision and implication levels for each assigned variable, identifying USPs, and related USIs, detecting propagation conflicts with `Propagation_Blocked`(),

```
// Global variables:   Implication graph I_C
// No input or output arguments
// Return value:       CONFLICT or SUCCESS
//
PS_Deduce()
{
   do {
      if (Logical_Deduce() == CONFLICT)
         return CONFLICT;
      Target_Application_Update();
      if (Propagation_Deduce() == CONFLICT)
         return CONFLICT;
   }
   while (changes to assignments in either dimension);
   Create_Propagation_Cuts();       // Create new p-cuts; add to Φ
   return SUCCESS;
}
```

**Figure 4.13:** Deduction engine for path sensitization

and maintaining propagation cuts. Procedure `Propagation_Blocked`() identifies inconsistent fanin and fanout justification conditions for propagation cuts.

The outer loop of the deduction engine repeatedly identifies USPs, with procedure `Identify_USPs`(). While new USPs are identified, additional propagation assignments are implied. (Note that even though the pseudo-code of all procedures assumes manipulation of antecedent sets and all algebraic manipulation assumes antecedent assignments, conversion between the two is straightforward.)

#### 4.5.2.1 Maintenance of Propagation Cuts

Derivation of implications causes propagation cuts to change, and may yield propagation conflicts. For each propagation cut $\zeta_i$, any p-*T* node *x* in $\zeta_i$ that propagates to a fanout node *y*, causes *y* to be added to $\zeta_i$. Whenever all fanout nodes of *x* are either p-*T* or p-*F*, then *x* is removed from $\zeta_i$. Any propagation cut $\zeta_i$ that is modified at the current decision level *c*, causes a new propagation cut $\zeta_f$ to be created (after all implications are identified) with $\delta(\zeta_f) = c$, and such that $\boldsymbol{A}(\zeta_f)$ is given by $\zeta_i$ and by the nodes/edge assignments that block propagation from nodes in $\zeta_i$:

```
// Global variables:   Implication graph I_C
// No input or output arguments
// Return value:       CONFLICT or SUCCESS
//
Propagation_Deduce()
{
   do {
      Let θ be a node or an edge;
      while (θ is p-X and (B(θ) or P(θ)) or Propagation_Blocked()) {
         if (Propagation_Blocked()) {
            define new conflict node π(κ);
            define α(π(κ));          // With either (4.22) or (4.23)
            return CONFLICT;
         } else if (B(θ)) {
            π(θ) ← 0;
            set α(θ) as elements causing B(θ) to hold;
            set δ(θ) and ι(θ);
         } else if (P(θ)) {
            π(θ) ← 1;
            set α(θ) as elements causing P(θ) to hold;
            set δ(θ) and ι(θ);
         }
         Update_Propagation_Cuts();  // Update composition of p-cuts
      }
   }
   while (Identify_USPs()== SUCCESS);  // Iterate USP identification
   return SUCCESS;
}
```

**Figure 4.14:** Basic propagation deduction engine

$$\boldsymbol{A}(\zeta_f) = \{(\zeta_i, 1)\} \cup blockedby(\zeta_i) \tag{4.19}$$

where $blockedby(\zeta_i)$ identifies the set of nodes/edges assignments that directly block propagation from p-cut $\zeta_i$. In general,

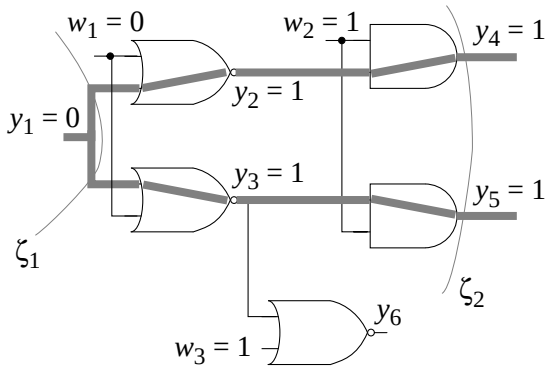



**Figure 4.15:** Identification of blocking conditions

$$blockedby(\zeta) = \bigcup_{x \in \zeta} blockedby(x) \tag{4.20}$$

where,

$$blockedby(x) = \bigcup_{y \in O(x)} \begin{cases} blockedby(y), & \text{if } (\pi(x, y) = 1) \wedge (\pi(y) = 1) \wedge \neg pcut(\{y\}) \\ \{(\pi(x, y), 0)\}, & \text{if } (\pi(x, y) = 0) \\ \{(\pi(y), 0)\}, & \text{if } (\pi(x, y) = 1) \wedge (\pi(y) = 0) \\ \varnothing, & \text{otherwise} \end{cases} \tag{4.21}$$

where $pcut(\{y\})$ tests whether node $y$ identifies a propagation cut. Note that in the proposed search algorithm, only p-cuts of size 1 require fanin justification.

**Example 4.10.** Let us consider the example circuit of Figure 4.15, where $\zeta_1$ is the initially defined p-cut. Assume that at the current decision level $c$, $w_1$ and $w_2$ are assigned as shown, whereas $w_3$ had been assigned at a lower decision level. The assignments permit propagation from $y_1$ to $y_4$ and $y_5$. As a result, a new propagation cut $\zeta_2 = \{ y_4, y_5 \}$ can be created, such that $\delta(\zeta_2) = c$ and $\boldsymbol{A}(\zeta_2) = \{ (\zeta_1, 1) \} \cup \{ (\pi(y_3, y_6), 0) \}$, and where $blockedby(\zeta_1) = \{ (\pi(y_3, y_6), 0) \}$. □

Let us assume first that a p-frontier becomes empty. In such a situation, a propagation cut $\zeta$ cannot be fanout justified and a propagation conflict $\pi(\kappa)$ is defined. The antecedent assignment of $\pi(\kappa)$ is given by,

```
Propagation_Blocked()
{
   if (exists ζ ∈ Φ such that φ(ζ) = ∅)
      return TRUE;
   else if (exists ζ ∈ Φ that cannot be fanin justified)   // (4.17)
      return TRUE;
   else if (exists unsatisfied propagation implicate)
      return TRUE;
   return FALSE;
}
```

**Figure 4.16:** Identification of propagation conflicts

$$\boldsymbol{A}(\pi(\kappa)) = \{(\zeta, 1)\} \cup blockedby(\zeta) \tag{4.22}$$

which follows from (4.19). Conversely, let us assume that the propagation conflict results from fanin justification. The only situation in which an empty p-frontier is not identified is when, in (4.17), either $B(y)$ holds or $P(y)$ does not hold. In such a situation, let $\boldsymbol{A}_B$ denote the set of assignments that cause the blocking situation to be identified. As a result, the antecedent assignment of $\pi(\kappa)$ is given by,

$$\boldsymbol{A}(\pi(\kappa)) = \{(\zeta, 1)\} \cup blockedby(\zeta) \cup \boldsymbol{A}_B \tag{4.23}$$

$\boldsymbol{A}_B$ depends on the causes that lead (4.17) not to hold and, in some situations, it can be application-dependent (even though that is not the case with Example 4.9). $\zeta$ is the p-cut in the fanin of the p-cut that yields the conflict.

The pseudo-code description of `Propagation_Blocked()` is given in Figure 4.16. Besides testing for p-cuts that cannot be fanout justified or fanin justified (and which require that type of justification), the existence of unsatisfied propagation implicates is also tested. The construction of these implicates is handled by the diagnosis engine.

#### 4.5.2.2 Identification of Unique Sensitization Points

While deriving propagation implications, USPs denote the unique form for propagation cuts that can be identified by the proposed deduction engine. Several algorithms have been proposed for the identification of USPs in test pattern generation. In FAN [62] and TOPS [92], prepro-

cessing techniques are proposed, which require worst-case quadratic-time on the size of the circuit, and do not identify dynamic USPs. In SOCRATES [145], a dynamic procedure is proposed that is based on the intersection of lists of dynamic dominators for each node in the D-frontier. This procedure requires worst-case quadratic time in the size of the circuit. In [87] a linear-time procedure is claimed, but it is only sketched, and its actual complexity is difficult to assess. LEAP [155] proposes a simple worst-case linear-time algorithm for identification of USPs based on levelized breadth-first traversals.

`Identify_USPs`() follows the main ideas of the procedure described in [155]. However, given that several p-frontiers may exist, the identification of USPs must be accordingly adapted. In addition, USPs correspond to implied p-cuts, and consequently, the antecedent assignments of such p-cuts must be defined.

The pseudo-code description of `Identify_USPs`() is given in Figure 4.17. This procedure identifies all USPs with respect to each p-frontier. No propagation conflicts need be detected, since at least one potential propagation path exists to the primary outputs; otherwise a blocked p-cut would have been detected by the deduction engine. Given that each p-frontier drives a disjoint set of p-*X* nodes, then the above procedure runs in linear time in the number of circuit nodes. However, the deduction engine iterates calls to `Identify_USPs`(), and thus an upper bound on the worst-case running time is $O(N^2)$ (whereas the procedure proposed in SOCRATES would require $O(N^3)$ time). The worst-case bound of $O(N^2)$ on the run time should hardly be exercised, since it would require a decision level where $O(N)$ USPs would be identified. Although possible, it is hard to find in practice path sensitization problems with $O(N)$ USPs identified at a given decision level. This observation is supported by experimental data given in [154] and replicated in Chapter VII; the number of USPs is usually significant, but it is far from being on the order of the number of circuit nodes (or even the largest depth in the circuit).

Nevertheless, a few optimizations can be implemented. For example, we can restrict the number of calls to `Identify_USPs`() to a fixed number *k*. This solution guarantees a worst-case linear time, but it does not necessarily identify all USPs. Another solution is to restrict traversals to sets of p-*X* nodes that were subject to changes to the p-status of some nodes or edges. Even though this solution does not improve the worst-case time bound, it prevents useless traversals over sets of

```
Identify_USPs()
{
   status = FAILURE;
   for each (p-cut ζo in Φ with p-frontier φ(ζo)) {
      schedule nodes in p-frontier φ(ζo) for levelized traversal;
      α ← { ζo } ∪ blockedby(ζo);              // Initial causes for first USP
      stop = FALSE;
      while (x is next node to visit with lowest topological level
               && !stop) {
         if (x is the only node to visit) {
            define new propagation cut ζ = { x };
            α(ζ) = α;  compute δ(ζ) = c and ι(ζ);
            φ(ζ) = { y | y ∈ O(x) and π(x, y) = X };
            //
            set x to p-T;
            α(π(x)) = { ζ };  δ(π(x)) = c; ι(π(x)) = ι(ζ) + 1;
            α ← { ζ };            // Initialize antecedent set of next cut
            status = SUCCESS;              // USPs have been identified
         }
         for each (fanout node y of x) {
            if (π(x, y) == 0) add π(x, y) to α;
            else if (π(y) == 0) add π(y) to α;
            else if (y is p-X) schedule y to be visited;
            else if (y is in a propagation cut ζy)
               { stop = TRUE; break; }    // Process another p-frontier
         }
         set x as unscheduled;
      }
   }
   return status;
}
```

**Figure 4.17:** Identification of USPs

p-*X* nodes where additional USPs are known not to be found.

The definition of the antecedent assignment of a newly created p-cut $\zeta$ as a function of another p-cut $\zeta_s$ in the transitive fanin of $\zeta$, is defined as follows:

$$\boldsymbol{A}(\zeta) = \{(\zeta_s, 1)\} \cup \boldsymbol{A} \tag{4.24}$$

such that $\boldsymbol{A}$ is computed by procedure `Identify_USPs`(), while traversing the p-*X* nodes, and identifies assignments that block propagation.

### 4.5.3 Basic Diagnosis Engine

In this section we describe how conflicts involving propagation information are diagnosed. In particular, the details for implementing conflict-directed backtracking, failure-driven assertions and conflict-based equivalence are given. We note, however, that failure-driven assertions are exclusively defined for the logical dimension. Assertions on the propagation dimension would require a significantly more complex algorithmic framework, as is suggested below.

Basic conflict diagnosis, involving the propagation dimension, is based on the same principles of conflict analysis for the logical dimension, but the global nature of propagation conflicts requires some modifications. After each conflict, implication sequences are analyzed, dependencies are recorded, and implicates of the propagation consistency function $\xi_\pi$ are created. These propagation implicates identify a sufficient set of conditions for a propagation conflict to be identified. The format of *propagation implicates* (or *p-clauses*) is as follows:

1. A *propagation cut* $\zeta$ that is associated with the propagation conflict.
2. A set of assignments that, given the propagation cut, identifies sufficient conditions for a propagation conflict to be identified.

Consequently, a p-clause $\omega_\pi$ is defined as a 2-tuple,

$$\omega_\pi = \langle \zeta, \omega \rangle \tag{4.25}$$

such that whenever $\zeta$ identifies a propagation cut, then $\omega$ is an implicate of the consistency function of the path sensitization problem, that would cause (4.4) not to hold or a logical conflict to be identified. The propagation cut of $\omega_\pi$ is assumed to be defined at a decision level less than *c*. A p-clause $\omega_\pi = \langle \zeta, \omega \rangle$ is said to be *unsatisfied* whenever $\zeta$ is a propagation cut and $\omega$ is unsatisfied. A p-clause $\omega_\pi = \langle \zeta, \omega \rangle$ is satisfied whenever either $\omega$ is satisfied or $\zeta$ cannot be a propagation cut.

The definition of $\omega$ follows the construction of conflicting clauses in Chapter III. A con-

flicting assignment set is defined by,

$$\boldsymbol{A}_{CS} = causesof(\pi(\kappa)) \tag{4.26}$$

where *causesof*($\pi(\kappa)$) is defined by (3.7) on page 67 extended to all elements of $\boldsymbol{V}$ and the set of all p-cuts.

In the search framework based on the p-propagation model, conflicting assignment sets can contain node and edge assignments, and propagation cut assignments. In the proposed conflict diagnosis procedure, however, a conflicting assignment set is required to contain at most *one* p-cut assignment. Let $S$ denote a set of nodes, edges and one propagation cut, and let *cutof*($S$) denote the only element $\theta$ of $S$ for which *pcut*($\theta$) holds. Further let $cutof^{C}(S) = S - \{\ cutof(S)\ \}$. As a result, $\omega$ for each p-clause is given by a modified form of (3.8) on page 68:

$$\omega = \{\theta^{\nu(\theta)} \,|\, (\theta, \nu(\theta)) \in cutof^{C}(\boldsymbol{A}_{CS})\} \tag{4.27}$$

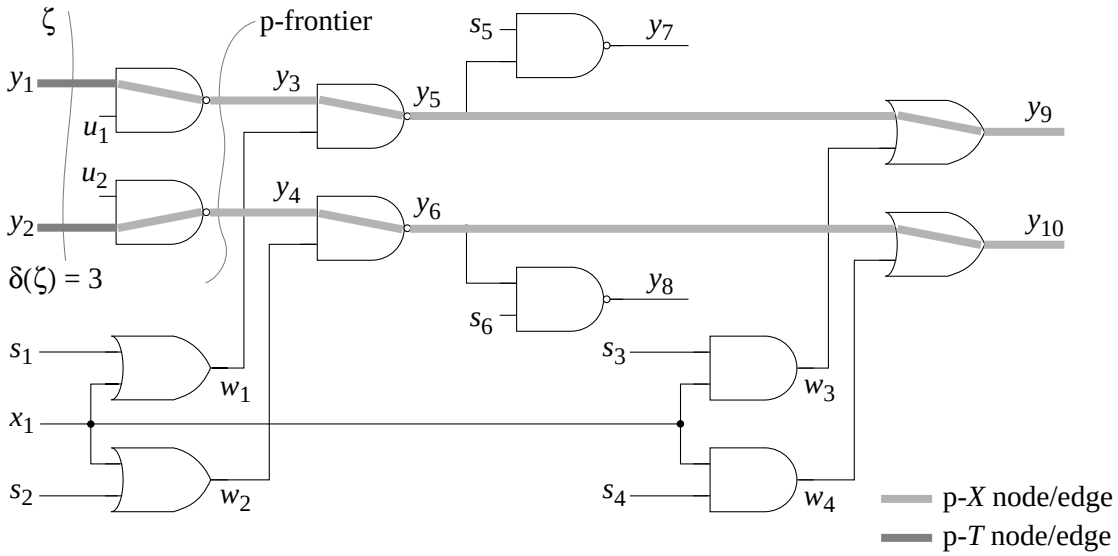
and $\zeta$ is defined to be *cutof*($\boldsymbol{A}_{CS}$).

Since the characterization of propagation cut follows that of nodes and edges, the conflict diagnosis structures used in the logical dimension can be extended to the propagation dimension. As mentioned earlier, identified p-clauses are maintained in a dedicated clause database, $\varphi_{\pi}$, and the logical deduction engine can use $\varphi_{\pi}$ to derive additional implications in the logical dimension.

**Example 4.11.** Let us consider the example circuit of Figure 4.18-a. The current decision level is 7, $\delta(\zeta) = 3$, and the decision assignment is $x_1 = 0$. The resulting implication sequence is shown in Figure 4.5-b, and yields a propagation conflict because all nodes in the p-frontier become p-false. As a result, we can create a p-clause $\omega_{\pi,\,1} = \langle \zeta_1, \omega_1 \rangle$ that identifies a propagation implicate of $\xi_{\pi}$. $\boldsymbol{A}(\pi(\kappa))$ is given by $\{(\zeta, 1), (\pi(y_3), 0), (\pi(y_4), 0)\}$ and from (4.26),

$$\boldsymbol{A}_{CS} = \{(\zeta, 1), (s_1, 0), (s_2, 0), (x_1, 0), (\pi(w_1, y_5), 0), (\pi(w_2, y_6), 0)\}$$

Consequently, the propagation cut of $\omega_{\pi,\,1}$ is given by $\zeta_1 = \zeta = \{\ y_1, y_2\ \}$ and the conditional clause $\omega_1$ is defined by,

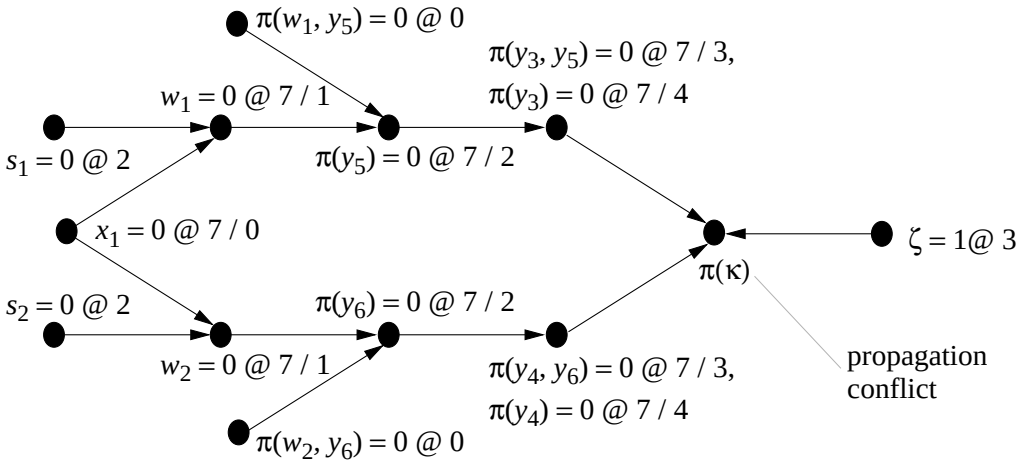



- Assignments:

Unless otherwise stated, logic values are *X*. For nodes and edges, by default the p-status is 0, assigned at decision level 0. p-*X* and p-*T* nodes and edges are shown. The current decision level is 7 and the decision assignment considered is $x_1 = 0$.

| node | $y_1$ | $y_2$ | $s_1$ | $s_2$ | $s_3$ | $s_4$ | $y_7$ | $y_8$ |
|---|---|---|---|---|---|---|---|---|
| logical | — | — | 0 @ 2 | 0 @ 2 | 0 @ 3 | 0 @ 3 | — | — |
| propagation | 1 @ 1 | 1 @ 1 | — | — | — | — | 0 @ 5 | 0 @ 3 |

(a) Example circuit




(b) Implication subgraph for $x_1 = 0$

**Figure 4.18:** Example of p-clause definition

$$\omega_1 = (s_1 + s_2 + x_1 + \pi(w_1, y_5) + \pi(w_2, y_6))$$

which signifies that whenever $\{ y_1, y_2 \}$ identifies a propagation cut, $s_1 = s_2 = x_1 = \pi(w_1, y_5) = \pi(w_2, y_6) = 0$ yield a propagation conflict. Finally, observe that for this example the propagation conflict is due to an empty p-frontier. □

It interesting to note that the p-clause defined in the above example can now be applied during the search to prevent a known propagation conflict from occurring, thus permitting the early identification of propagation conflicts due to the same conditions.

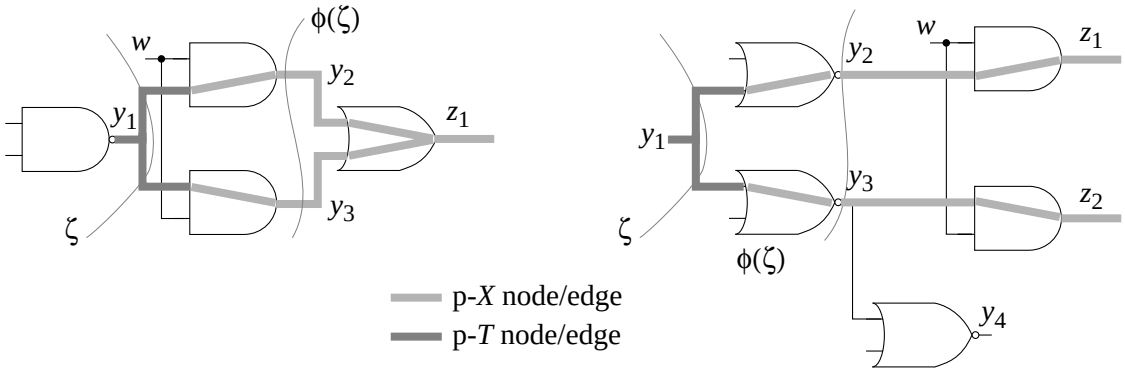
It is more interesting to note that *no* target application was actually specified. The p-clause was derived based on general blocking properties, that apply in both testing and timing. Consequently, the derived p-clause is said to be *pervasive across path sensitization applications* and can thus be used in *both* applications. This example also shows that in the propagation dimension, it is possible in some cases to define pervasive clauses, in addition to the pervasive clauses identified in the logical dimension. In general, however, not all propagation clauses are pervasive across path sensitization applications, or even in the same application. In the following chapters examples of such clauses will be described. Whenever the causes of a propagation conflict can be traced to blocking conditions that are pervasive across path sensitization problems, then the defined p-clauses are also pervasive across path sensitization problems. That is the case with the previous example.

Propagation implicates derived during the search can be used in a variety of situations and identify conditions for which the basic deduction engine might not prevent conflicts from being identified.

**Example 4.12.** Examples of propagation conflicts are shown in Figure 4.19. For the example circuit of Figure 4.19-a, let us assume the assignment $w = 0$. Then $\phi(\zeta)$ eventually becomes empty and a propagation conflict is identified. The resulting p-clause is given by,

$$\omega_\pi = \langle \zeta, \omega \rangle = \langle \{ y_1 \}, (w) \rangle$$

which signifies that whenever $\zeta = \{ y_1 \}$ denotes a p-cut, then $w$ must assume value 1. For the example circuit of Figure 4.19-b, the same assignment is assumed and a propagation conflict is also identified. The obtained p-clause is given by,




(a) Example 1 (b) Example 2

**Figure 4.19:** More examples of p-clause definition

$$\omega_\pi = \langle \zeta, \omega \rangle = \langle \{ y_1 \}, (\pi(y_3, y_4) + w) \rangle$$

which means that whenever $\zeta = \{ y_1 \}$ is a propagation cut and propagation of a perturbation from $y_3$ to $y_4$ is blocked, then $w$ must assume value 1. □

**Backtracking Decision Level**

The techniques used in SAT, for creating assertions and computing backtracking decision levels, can be extended to path sensitization. The main differences stem from handling different conflict types and propagation cuts. In path sensitization, backtracking can result from three possible conflict interactions:

1. Decision assignment and resulting assertion result in logical conflicts.
2. Decision assignment and resulting assertion result in propagation conflicts.
3. Either the decision assignment or resulting assertion results in a logical conflict, and the other in a propagation conflict.

The situation where both conflicts are logical, and the identified dependencies correspond solely to the logical dimension, was addressed in Chapter III. The situation where a logical and a propagation conflict are identified can be viewed, without loss of generality, as a special case of the situation where both conflicts are propagation conflicts. Consequently, in the following analysis we assume that a given decision level *c*, two propagation conflicts are identified. Manipulation of p-clauses is also required for any logical conflict whose diagnosis yields a p-clause.

Let $\omega_{\pi,\,1} = \langle \zeta_1, \omega_1 \rangle$ denote the first p-clause identified at a decision level *c*, and let some variable *w* be asserted due to $\omega_{\pi,\,1}$. Now let us assume that another propagation conflict is identified. The resulting p-clause could contain two propagation cuts $\zeta_1$ and $\zeta_2$, one from asserting *w* and the other from the last conflict. However, by definition, p-clauses and conflicting assignment sets can have at most one propagation cut. Hence, we must express both propagation cuts in terms of another common propagation cut $\zeta$. Let $\zeta_1$ and $\zeta_2$ be two propagation cuts. Then, a common p-cut is computed by $join(\zeta_1, \zeta_2)$, where:

$$join(\zeta_1, \zeta_2) = \begin{cases} (\zeta_1, 1), & \text{if } \zeta_1 = \zeta_2 \\ join(cutof(\boldsymbol{A}(\zeta_1)), \zeta_2) \cup cutof^C(\boldsymbol{A}(\zeta_1)), & \text{if } \delta(\zeta_1) \geq \delta(\zeta_2) \\ join(\zeta_1, cutof(\boldsymbol{A}(\zeta_2))) \cup cutof^C(\boldsymbol{A}(\zeta_2)), & \text{otherwise} \end{cases} \tag{4.28}$$

$join(\zeta_1, \zeta_2)$ recursively searches for the case where $\zeta_1 = \zeta_2$, while adding node and edge assignments associated with each visited propagation cut. The set of node/edge assignments computed by $join(\zeta_1, \zeta_2)$ includes the definition of the common propagation cut $\zeta$. Hence, $join(\zeta_1, \zeta_2)$ identifies a set of conditions under which a propagation cut $\zeta \in join(\zeta_1, \zeta_2)$ causes both $\zeta_1$ and $\zeta_2$ to be created. Given the definition of a conflicting assignment set, and assuming a computed conflicting assignment set $\boldsymbol{A}_{CS}^{i}$ containing two conflicting assignment sets $\zeta_1$ and $\zeta_2$, we can use $join(\zeta_1, \zeta_2)$ to obtain a conflicting assignment set $\boldsymbol{A}_{CS}$ containing just one p-cut as follows:

$$\boldsymbol{A}_{CS} = [\boldsymbol{A}_{CS}^{i} - \{(\zeta_1, 1), (\zeta_2, 1)\}] \cup join(\zeta_1, \zeta_2) \tag{4.29}$$

The backtracking decision level is then defined as follows:

$$\beta_L = max\{\delta(\theta) | (\theta, \nu(\theta)) \in \boldsymbol{A}_{CS}\} \tag{4.30}$$

that basically generalizes (3.17) on page 86.

Besides computing the backtracking decision level, a p-clause $\omega_{\pi,\,2}$ is created, $\omega_{\pi,\,2} = \langle \zeta_2, \omega_2 \rangle$, such that $\zeta_2$ is defined by $cutof(\boldsymbol{A}_{CS})$ and $\omega_2$ is given by (4.27). At the backtracking decision

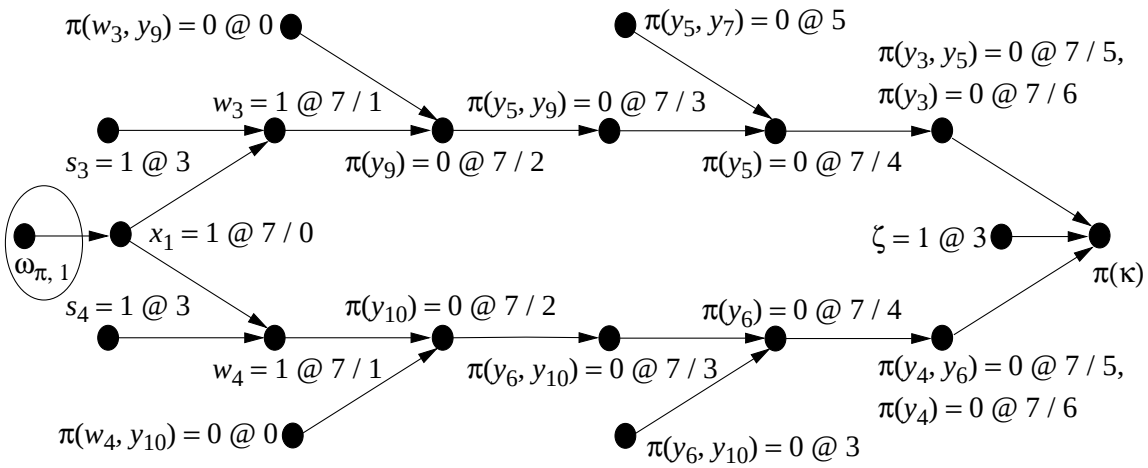



**Figure 4.20:** Implication sequence triggered by $x_1 \leftarrow 1$ for the circuit of Figure 4.5

level $\beta_L$, a propagation conflict is now forced by $\omega_{\pi,\,2}$, that serves to either derive additional p-clauses or compute another backtracking decision level.

Another source of conflicts arises from unsatisfied p-clauses, which are detected by procedure `Propagation_Blocked()` that is described in Figure 4.16. The antecedent assignment of the conflict node due to an unsatisfied p-clause $\omega_\pi = \langle \zeta, \omega \rangle$ is then defined by,

$$\boldsymbol{A}(\pi(\kappa)) \;=\; \{(\zeta, 1)\} \cup \{(\theta, \nu(\theta)) | \theta^{\nu(\theta)} \in \omega\} \tag{4.31}$$

where $\theta$ in this case can either represent a node in either dimension or an edge in the propagation dimension. It is worth noting that most of the concepts described in Section 3.6 for (logical) conflict diagnosis find application while diagnosing propagation conflicts, the major difference being the manipulation of propagation cuts.

**Example 4.13.** To illustrate the computation of the backtracking decision level, let us continue studying the example circuit of Figure 4.18. The generated p-clause (see Example 4.11) yields the assertion $x_1 \leftarrow 1$. The resulting implication sequence is shown in Figure 4.20 and it yields another propagation conflict. Taking into consideration the antecedent assignment of $x_1$, the p-clause $\omega_{\pi,\,2} = \langle \zeta_2, \omega_2 \rangle$ that is created includes the same propagation cut as $\omega_{\pi,\,1}$, i.e. $\zeta_2 = \zeta$, since $\zeta$ is common to both conflicts and $\delta(\zeta) < 7$, and the associated conditional clause $\omega_2$ is defined as follows:

$$\omega_2 = (s_1 + s_2 + \neg s_3 + \neg s_4 + \pi(w_1, y_5) + \pi(w_2, y_6) + \pi(y_5, y_7) + \pi(y_6, y_8) + \pi(w_3, y_9) + \pi(w_4, y_{10}))$$

From (4.30), the backtracking decision level is computed to be 5, due to the assignment of $\pi(y_5, y_7)$ to p-*F* at decision level 5. Hence, the search process backtracks to decision level 5. (Note that at decision level 5, $\pi(y_5, y_7) = 0$ would now denote a unique implication point (UIP) and so the assignment $\pi(y_5, y_7) = 1$ could be used to trigger the second branch at decision level 5. However, assertions in the p-dimension are disallowed. This type of assertion would require justification for p-status assignments, and consequently lead to a more involved formulation.) The newly derived p-clause $\omega_{\pi,\,2} = \langle \zeta_2, \omega_2 \rangle$ states that if $s_1 = s_2 = 0$ and $s_3 = s_4 = 1$ there can be no propagation if the possible propagation paths are as shown in Figure 4.5. □

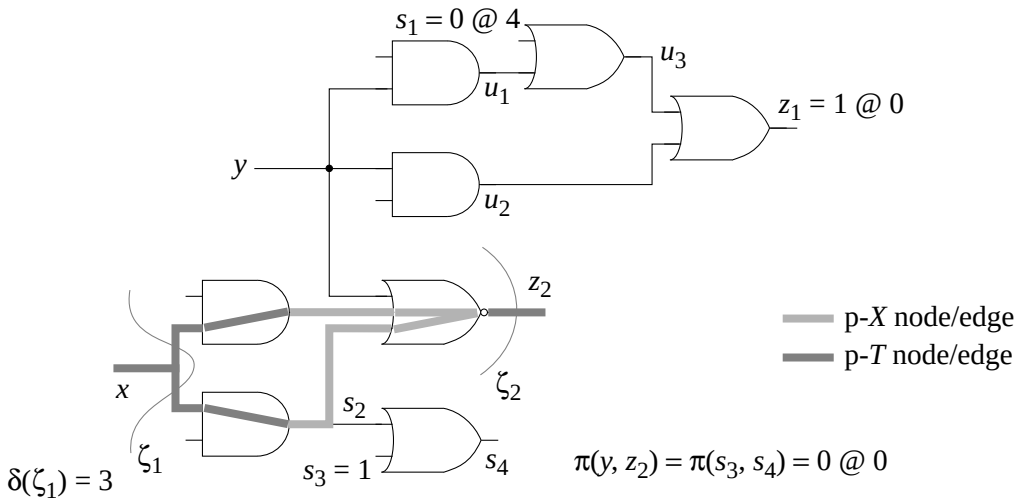
The conflict diagnosing equations proposed above for handling propagation conflicts can also be used in situations where a logical conflict is identified, but which involves a propagation cut in its definition. In these situations, even though a logical conflict is identified, the result of diagnosing the conflict is a p-clause.

**Example 4.14.** An example of a logical conflict whose diagnosis yields a p-clause is shown in Figure 4.21. $\zeta_1 = \{ x \}$ identifies a propagation cut created at decision level 3. At decision level 5, the assignment $s_3 = 1$ triggers an implication sequence, which creates a new p-cut $\zeta_2 = \{ z_2 \}$. The process of fanin justifying $\zeta_2$ implies $y \leftarrow 0$, which eventually yields a logical conflict with $z_1 = 1$. Even though a logical conflict is identified, its causes are a direct consequence of fanin and fanout justification of p-cuts. As a result, the derived p-clause is given by,

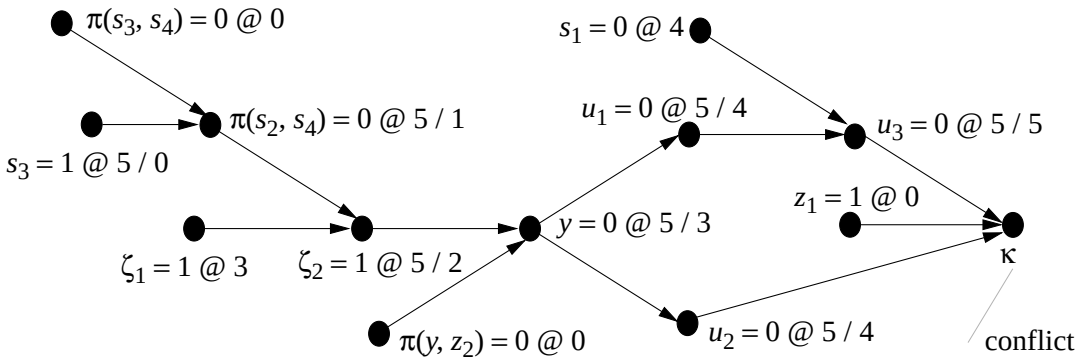
$$\omega_{\pi,\,1} = \langle \zeta_1, \omega_1 \rangle = \langle \{x\}, (\neg z_1 + s_1 + \pi(y, z_2) + \pi(s_3, s_4) + \neg s_3) \rangle$$

which states that if $\zeta_1 = \{ x \}$ is a propagation cut, and $\pi(y, z_2) = \pi(s_3, s_4) = 0$, then either $z_1 = 0$ or $s_1 = 1$ or $s_3 = 0$. For this example, we could avoid creating a p-clause by taking UIPs into account. The derived clause would then be $(\neg z_2 + s_1)$. This clause would block propagation to $z_2$, and set $\{ s_4 \}$ as a new propagation cut, which would imply $s_3 \leftarrow 0$. □

As the above example illustrates, we can use UIPs in the logical dimension. However, in




(a) Example circuit




(b) Implication sequence

**Figure 4.21:** Logical conflict that yields a p-clause

order to guarantee that the decision assignment is implied to the complemented value (as the second branch at the current decision level), a global clause must be defined. This fact results from implications due to propagation cuts being unidirectional, and thus may not be re-created if derivation of p-clauses implements UIPs.

**Organization of the Diagnosis Engine**

The diagnosis engine for path sensitization, `PS_Diagnose`() is shown in Figure 4.22, where `Logical_Diagnose`() corresponds to the basic diagnosis engine for the logical dimension described in Chapter III, whereas the basic diagnosis engine for propagation conflicts, `Propagation_Diagnose`(), is described in Figure 4.23. The procedure basically follows the steps described above. As mentioned earlier, p-clauses are maintained in a separate clause data-

```
// Global variables:   Implication graph I_C
//                     Clause databases φ and φ_π
// Input variable:     Current decision level c
// Output variable:    Backtracking decision level β_L
// Return value:       CONFLICT or SUCCESS
//
PS_Diagnose (c, &β_L)
{
   if (conflict uniquely dependent on logic values)
      return Logical_Diagnose (c, β_L);
   return Propagation_Diagnose(c, β_L);
}
```

**Figure 4.22:** Diagnosis engine for path sensitization

base, which is manipulated by the diagnosis engine to add new clauses, by the deduction engine for deriving logical implications and by procedure `Propagation_Blocked`() for identifying equivalent propagation blocking conditions.

As in the case of SAT algorithms, improvements and simplifications to conflict diagnosis can be defined. For example, straightforward extensions allows us to compute UIPs (in the logical dimension) and identify multiple conflicts (either logical or propagation). Simplifications to conflict diagnosis are also possible and are described in the following chapters. The simplified diagnosis engines bound the growth of the clause databases $\varphi$ and $\varphi_\pi$.

We can also identify conditions under which *consensus* operations between p-clauses can be defined. Let us consider two p-clauses $\omega_{\pi,1} = \langle \zeta_1, \omega_1 \rangle$ and $\omega_{\pi,2} = \langle \zeta_2, \omega_2 \rangle$ such that $\zeta_1 = \zeta_2$, and that consensus between $\omega_1$ and $\omega_2$ with respect to a variable $x$ is defined. Then the resulting consensus p-clause becomes $\omega_{\pi,3} = \langle \zeta_3, \omega_3 \rangle = \langle \zeta_1, c(\omega_1, \omega_2, x) \rangle$. (It is worth noting that consensus between p-clauses is also suggested by how propagation conflicts are diagnosed.) Furthermore, $\omega_{\pi,3}$ is an implicate of the consistency function $\xi_\pi$. If both $\omega_{\pi,1}$ and $\omega_{\pi,2}$ are satisfied, then $\omega_{\pi,3}$ must also be satisfied. Conversely, if $\omega_{\pi,3}$ is unsatisfied then either $\omega_{\pi,1}$ or $\omega_{\pi,2}$ is unsatisfied. These facts necessarily hold since all three clauses share the same propagation cut.

## 4.6 Summary and Perspective

This chapter introduces the perturbation propagation model, an abstract model for path

```
// Global variables:   Implication graph I_C
//                     Clause databases φ and φ_π
// Input variable:     Current decision level c
// Output variable:    Backtracking decision level β_L
// Return value:       CONFLICT or SUCCESS
//
Propagation_Diagnose (c, &β_L)
{
   A_CS = Compute_Conflicting_Assignment_Set();        // Using (4.26)
   if (A_CS contains assignments of two p-cuts ζ_1 and ζ_2)

        A_CS ← [A_CS − {(ζ_1, 1), (ζ_2, 1)}] ∪ join(ζ_1, ζ_2) ;        // Using (4.28)

   ζ = cutof(A_CS);
   ω = Create_Conflicting_Clause (cutof^C(A_CS));        // Using (4.27)
   ω_π = ⟨ζ, ω⟩;                                          // New p-clause
   Update_P_Clause_Database (ω_π);          // Add clause to database
   β_L = Compute_Max_Level (A_CS);                       // Using (4.30)
   if (β_L != c) {
      define new conflict node π(κ);    // Set up new conflict node
      define α(π(κ));                                    // Using (4.31)
      return CONFLICT;
   }
   return SUCCESS;
}
```

**Figure 4.23:** Basic propagation diagnosis engine

sensitization that can be used in different target applications. A search algorithm for path sensitization based on the p-propagation model is described. The algorithm implements most of the pruning methods described in Chapter III. In particular, the algorithm allows implementing conflict-directed backtracking, conflict-based equivalence and failure-driven assertions in the context of path sensitization.

The p-propagation model facilitates the implementation of several algorithmic techniques described in the previous chapter in the context of SAT. For this purpose, the model and algorithm allow the definition and manipulation of propagation implicates associated with the consistency function for path sensitization. Propagation implicates can describe powerful properties of path sensitization, and in some situations they can be application-independent.

Throughout the description of the search algorithm, the details of the target application were deliberately overlooked, being our goal to just describe properties intrinsic to the p-propagation model that are valid for path sensitization in general. In the following chapters we address two target applications, test pattern generation and timing analysis. We describe how to represent path sensitization for those applications with the p-propagation model, and detail the search algorithm, as well as its improvements and simplifications. In this analysis, we illustrate which differences must be considered when implementing test pattern generation or timing analysis tools.

Besides the details of the target application, other aspects of the search-based path sensitization algorithm were also skipped and must be further analyzed:

- We must specify which p-clauses can be considered pervasive, and how pervasiveness holds across different target applications.
- The procedure for toggling p-*T* nodes into p-*F* nodes must be formalized for both test pattern generation and timing analysis.

Another topic that must be addressed is comparing the p-propagation model with other path sensitization models in each target application. For the particular case of test pattern generation, where more models have been proposed, we show that the p-propagation model can be scaled to achieve any degree of precision that is achieved by other path sensitization models. We further show that the p-propagation model can actually be made more precise than any other path sensitization model for test pattern generation proposed in the past.

One limitation of the proposed search algorithm for path sensitization is that assertions in the propagation domain are disallowed. The difficulty of handling assertions in the propagation domain is due to the computational overhead associated with maintaining a large number of propagation cuts, requiring fanin and fanout justification, and in relating these propagation cuts whenever fanin and fanout justification takes place. The implementation of assertions in the propagation dimension is left as future research work.

# CHAPTER V

# PATH SENSITIZATION FOR TEST PATTERN GENERATION

## 5.1 Introduction

The purpose of this chapter is to detail, using test pattern generation as the target application, the path sensitization model and algorithm described in the previous chapter. The p-propagation model is defined abstractly, and so it is independent of any specific target application. Accordingly, we must define how the model is specified for path sensitization in test pattern generation.

Moreover, the path sensitization algorithm, LEAP, needs to be configured for test pattern generation. The basic deduction and diagnosis engines are described and, as was done in Chapter III for SAT, more advanced engines are analyzed. Simplifications to conflict diagnosis are proposed and analyzed in some detail, since they represent the core of the experimental results reported in Chapter VII. The description of the search algorithm is concluded with a brief discussion of solution and selection engines. Furthermore, different configurations of LEAP() are compared with algorithms proposed by other authors in the context of test pattern generation.

The formulation of the p-propagation model for test pattern generation is scalable, i.e. depending on the amount of computational effort one is willing to spend, the precision with which path sensitization facts are deduced can be increased. We show that the degrees of precision proposed by other path sensitization models for test pattern generation can also be attained with an

adequate formulation of the p-propagation model. Moreover, we propose using the edge p-status for increasing the reasoning ability of the model. This allows the precision of the p-propagation model to surpass the precision of other models for path sensitization.

**Outline**

The top-level description of test pattern generation is given in Section 5.2. Section 5.3 describes how the p-propagation model can be used to represent path sensitization for test pattern generation. Section 5.4 is dedicated to detailing the basic deduction and diagnosis engines, taking into consideration the target application, and it is followed by an analysis of advanced deduction engines in Section 5.5. Advanced deduction engines are based on the identification of propagation implicates and application of consensus, and so we have to formalize consensus operations over p-clauses characterized by different p-cuts.

Engines for postprocessing and for selecting decision assignments are studied in Section 5.6 and Section 5.7, respectively. These engines are first described in Chapter III for SAT, being our goal in the present chapter to describe the modifications required for test pattern generation.

Section 5.8 analyzes possible formulations of fault detection with the p-propagation model. The emphasis is on how to increase the reasoning ability provided by the model, while guaranteeing that the proposed pruning methods can still be implemented. This section provides the basis for comparing, in Section 5.9, LEAP with algorithms proposed by other authors.

## 5.2 Fault Detection in Test Pattern Generation

The top-level algorithm for fault detection in test pattern generation in shown in Figure 5.1. A list of target faults is assumed. Random test pattern generation can be optionally invoked. The algorithm then processes each fault until all faults are detected, proved redundant or aborted due to resource constraints. For a detectable fault, and after a solution is identified, fault simulation can be optionally executed. In general fault simulation is invoked, the exception being when the goal is to exclusively evaluate the path sensitization algorithm. After all faults are detected, reverse-order fault simulation can be optionally executed to reduce the test set size [144]. Algorithms for random test pattern generation and fault simulation can be found in [1] and are not fur-

```
// Global variables:      List of faults fault_list
//                        List of detected faults detected_list
//                        List of redundant faults redundant_list
//                        List of aborted faults aborted_list
// Auxiliary variables:   outcome ∈ { FAILURE, SUCCESS }
//
Test_Pattern_Generation()
{
   define φ for circuit and initialize φ_π;
   if (RANDOM_TPG) Generate_Random_Tests (fault_list);
   while (fault in fault_list) {
      Delete_From_List (fault_list, fault);
      outcome = Detect_Fault (fault);
      if (outcome == SUCCESS) {                    // Detected fault
         Add_To_List (detected_list, fault);
         if (FAULT_SIMULATION) Fault_Simulation (fault_list);
      }
      else if (outcome == FAILURE)                 // Redundant fault
         Add_To_List (redundant_list, fault);
      else if (outcome == ABORTED)                 // Aborted fault
         Add_To_List (aborted_list, fault);
   }
   if (FAULT_SIMULATION) Reverse_Order_Fault_Simulation();
}
```

**Figure 5.1:** Procedure for test pattern generation

ther considered in the present dissertation.

The procedure for detecting each fault is described in Figure 5.2, and its main purpose is to invoke LEAP() (described in Figure 4.10 on page 152). Note that given the definition of PS_Search(), invoked by LEAP(), no faults are aborted. However, in practical implementations, PS_Search() controls either the run time or the number of backtracks in order to decide whether a given fault is deemed too hard to detect or prove redundant.

Internal to LEAP(), procedure Toggle_Propagation_Values() is invoked whenever the search process terminates successfully. Assuming a complete node and edge assignment in both dimensions, this procedure sets to p-*F* nodes and edges assigned p-*T*, but which are not included in sensitizable paths. Taking into consideration that nodes or edges that may be down-

```
// Global structures:  p-propagation model initialization ψ
// Input arguments:    Fault specification fault
// Return value:       status ∈ { FAILURE, SUCCESS }
//
Detect_Fault (fault)
{
   define ψ for TPG given fault;     // Initialize p-status for fault
   status = LEAP();
   return status;   // It can either be SUCCESS, CONFLICT or ABORTED
}
```

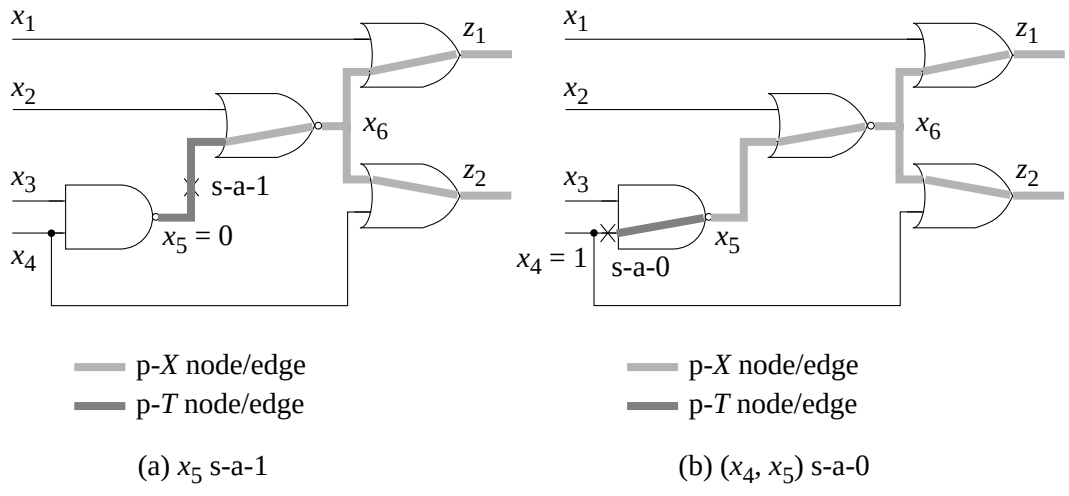
**Figure 5.2:** Procedure for detecting each fault

graded from p-*T* to p-*F* are the result of fanout blocking conditions, then a simple levelized backward circuit graph traversal can be used to visit nodes whose p-status must be toggled. It is clear that such traversal does not change the p-status of any p-*T* primary output. This procedure is only valid for test pattern generation, since for timing analysis additional conditions are involved in downgrading p-*T* to p-*F* nodes and edges. Note that toggling the p-status of nodes and edges does not affect the validity of computed tests, and it is only of interest if the sensitizable paths are to be reported; otherwise toggling values may be skipped since it just increases the computational overhead.

## 5.3 Modeling Fault Detection in Test Pattern Generation

In this section we detail how to represent path sensitization for test pattern generation using the p-propagation model. The single stuck-at line fault model is assumed [1, pp. 110-118] and two types of faults are distinguished. The stem fault *x* s-a-*v* to denote a node *x* whose logic value is fixed to a logic value *v*, and a fanout-branch fault (*x*, *y*) s-a-*v* denoting that the connection between *x* and *y* always assumes a fixed logic value *v*. Although the formulation of the model is the same for both types of faults, the initialization phase differs.

Let us assume a fault *x* s-a-*v*. Node *x* is said to be the source of the perturbation. Hence, *x* is initialized to p-*T*, all nodes and edges in its transitive fanout are initialized to p-*X*, because they may propagate the perturbation, and the remaining nodes and edges are set to p-*F*. Consequently, a perturbation can reach a primary output through any partial path connecting *x* to the primary out-




(a) $x_5$ s-a-1 (b) ($x_4$, $x_5$) s-a-0

**Figure 5.3:** Initialization of p-status for test pattern generation

puts. In addition, the fault must be activated, and so $x$ must actually assume value $\bar{v}$, thus defining the original j-frontier.

For fault ($x$, $y$) s-a-$v$, the j-frontier is initialized in the same manner, i.e. $x = \bar{v}$. However, the p-status initialization differs. Edge ($x$, $y$) is set to p-$T$, all nodes and edges in the transitive fanout of $y$ are set to p-$X$, and the remaining nodes and edges are set to p-$F$.

**Example 5.1.** Examples illustrating the initialization of the p-status for both types of faults are shown in Figure 5.3. For fault $x_5$ s-a-1, the initialization of the p-propagation model sets node $x_5$ to p-$T$ and nodes and edges in its transitive fanout to p-X. The remaining nodes and edges are set to p-$F$. Node $x_5$ is assigned value 0 to denote activation of the fault.

For fault ($x_4$, $x_5$) s-a-0, edge ($x_4$, $x_5$) is set to p-$T$, the nodes and edges in the transitive fanout of $x_5$ (including $x_5$) are set to p-$X$, and the remaining nodes and edges are set to p-$F$. Node $x_4$ is assigned value 1 to activate the fault. □

Following the definition of p-status in Section 4.3.2, the formulation of the p-propagation model for test pattern generation will be completed by specifying predicates $B_C(x)$, $B_C(x, y)$, $P(x)$ and $P(x, y)$.

Observe that the blocking predicate defined in (4.8) (see page 141) accounts for all possible blocking situations that can occur in test pattern generation with the exception of error signal cancellation. Consequently, predicate $B_C(x)$ is defined as follows:

$$B_C(x) = \sum_{y,\, w \in I(x)} ((\pi(w) = 1) \cdot (\pi(y) = 1) \cdot (\nu(w) \oplus \nu(y) = 1)) \tag{5.1}$$

which states that if two fanin nodes of $x$ are p-*T*, and the two nodes are assigned and assume opposite values, then error cancellation takes place, and propagation to $x$ is blocked. This situation would correspond, for example, to having a node $y$ set to D and a node $w$ set to $\overline{\mathrm{D}}$ at the input of a gate, thus cancelling propagation of the error signal to the output $x$. (Note that the above condition explicitly assumes simple gates, but can be easily extended to XOR/XNOR gates.) From (4.8) with $B_C(x)$ replaced with (5.1) we can define the blocking condition for test pattern generation as follows:

$$\begin{aligned} B(x) = & \left[ \prod_{y \in I(x)} (\pi(y, x) = 0) \right] + \left[ \prod_{y \in O(x)} (\pi(x, y) = 0) \right] + \\ & \left[ \sum_{y \in I(x)} ((\pi(y, x) = 0) \cdot c(y, x)) \right] + \\ & \left[ \sum_{y,\, w \in I(x)} ((\pi(w) = 1) \cdot (\pi(y) = 1) \cdot (\nu(w) \oplus \nu(y) = 1)) \right] \end{aligned} \tag{5.2}$$

and whenever $B(x)$ holds, then the p-status of $x$ becomes p-*F*. Condition $B_C(x, y)$ is defined to be identically 0 and so $B(x, y)$ is given by (4.9) (see page 141).

The node propagation predicate is defined as follows:

$$P(x) = \prod_{y \in I(x)} [(\pi(y, x) = 1) \cdot (\nu(y) \neq X) + (\pi(y, x) = 0) \cdot nc(y, x)] \tag{5.3}$$

that requires each fanin edge to either be p-*T* and the associated fanin node be assigned, or to be p-*F* and the associated fanin node to assume a non-controlling value. The formulation of $P(x)$ does not require p-*T* edges to be driven by nodes assuming the same logic values, since the blocking conditions take that into consideration, and the p-propagation model formulation gives preference to blocking. Finally, $P(x, y)$ is defined as follows:

$$P(x, y) = [\pi(x) = 1] \tag{5.4}$$

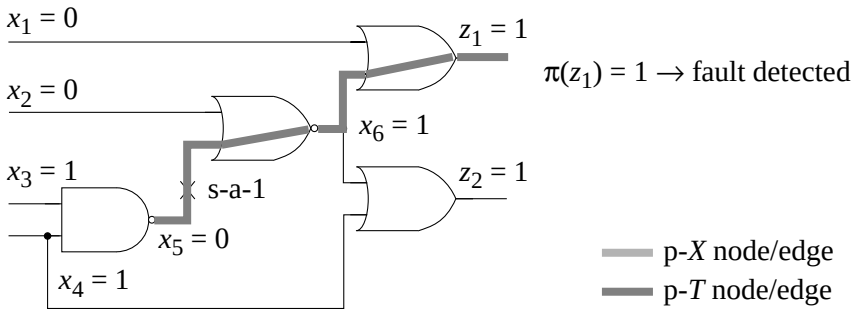



**Figure 5.4:** Applying the p-propagation model to test pattern generation

which states that an edge becomes p-*T* provided its fanin node becomes p-*T*.

It is worth noting that the definition of the edge p-status in test pattern generation is nonessential, and one can develop a formulation just based on node p-status. However, the proposed formulation illustrates the relationship between different path sensitization applications. In Section 5.8, the edge p-status definition is reformulated to allow for improved pruning ability.

Given the proposed configuration of the p-propagation model for test pattern generation the following holds:

**Theorem 5.1.** Given a SSF fault in a combinational circuit, a test ***T*** detects the fault if and only if under the p-propagation model ***T*** sets at least one primary output to p-*T*.

The above result is independent of how the computation of ***T*** is actually performed. Thus, an immediate corollary is:

**Corollary 5.1.** A sound and complete search algorithm, based on the p-propagation model, computes a test ***T*** for a given fault if and only if such test exists.

**Example 5.2.** The operation of the p-propagation model for test pattern generation is illustrated with the example of Figure 5.3-a. First, the assignment $x_5 = 0$ implies $x_3 \leftarrow 1$ and $x_4 \leftarrow 1$. As a result $B(z_2)$ holds, since $\pi(x_4, z_2) = 0$ and $c(x_4, z_2)$ holds. From (4.14) and (4.13), respectively, $\pi(z_2) \leftarrow 0$ and $\pi(x_6, z_2) \leftarrow 0$. In the logical dimension, $z_2 \leftarrow 1$ is implied by the assignment to $x_4$. Because $\pi(x_5) = 1$, then $\pi(x_5, x_6) \leftarrow 1$ from (5.4).

Now let $x_2 \leftarrow 0$. Hence, $P(x_6)$ holds and so $\pi(x_6) \leftarrow 1$. From (5.4), $\pi(x_6, z_1) \leftarrow 1$. Finally, let $x_1 \leftarrow 0$, that causes $P(z_1)$ to hold and so $\pi(z_1) \leftarrow 1$. Figure 5.4 illustrates the logic values and

propagation status after the above sequence of assignments. The assignment set { ($x_1$, 0), ($x_2$, 0), ($x_3$, 1), ($x_4$, 1) } identifies a test for fault $x_5$ s-a-1. For this example the existence of OUT nodes is only implicitly assumed, since for test pattern generation if a perturbation reaches the output of a gate connected to a OUT node *z*, then *z* also propagates a perturbation. □

**Perspective**

The p-propagation model is significantly different from the D-calculus and derived algebras. Nevertheless, a few other path sensitization models share similarities with the p-propagation model, in particular the SPLIT model [33] and the SAT-directed models of [24] and [162]. All these models characterize the state of each circuit node with three values. The semantics of each value differ slightly between models.

In the SPLIT model the values of the good and faulty circuits are kept. In addition, a third value, referred to as the difference value, identifies whether the good and faulty values are or can be different. Hence, the third value is defined locally, as is the p-status of nodes and edges in the p-propagation model. The definition of the difference value in the SPLIT model does not account for fanout blocking information, as the formulation of the p-propagation model does.

The SAT-directed models of [24] and [162] also maintain the node values of the good and faulty circuits. The third node value (referred to as the D-chain variable in [162] and the path variable in [24]) indicates whether the node is part of a sensitizable path. This definition immediately implies that a node is only said to be part of a sensitizable path after all logic assignments to the relevant primary inputs have been made. Consequently, these two models require decision assignments on some of the path variables in order to identify sensitizable paths, which may increase the size of the decision trees for some instances of path sensitization.

The p-propagation model is unique in that the notion of error signal is only implicitly considered. This formulation has the following advantages:

- The p-propagation model avoids some of the redundant information used by the SPLIT model and SAT-directed models.
- The model is scalable (as described in Section 5.8) and can be adapted such that it supersedes other path sensitization models for test pattern generation. It also suggests applications for the edge propagation status with the goal of increasing the reasoning precision of the model.

- In the context of search, as described in the previous chapter, it permits several pruning methods to be defined independently of the target application. In addition, it allows the diagnosis engines described in Chapter III to be straightforwardly adapted.

The p-propagation model for test pattern generation is compared in more detail with other models in Section 5.9.

## 5.4 Basic Deduction and Diagnosis Engines

The basic deduction and diagnosis engines for the propagation dimension are given by the procedures described in the previous chapter. In order to formalize the description of these engines for test pattern generation we only need to specify how antecedent assignments are established due to blocking and propagation conditions. Moreover, implementation tradeoffs of the diagnosis engine are described, which can be used to bound the maximum growth of the database of p-clauses.

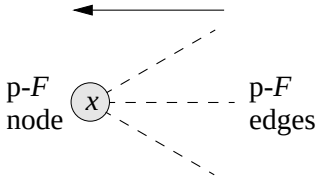
### 5.4.1 Deduction Engine

The basic deduction engine was described in Figure 4.13 (see page 157). In this section we formalize the definition of antecedent assignments for each possible implied assignment. Let us assume that a node $x$ becomes p-$F$. Hence, (5.2) holds and the antecedent assignment of assigning p-$F$ to $x$ is defined by one of the terms of (5.2) that holds true. Precedence is given to structural blocking conditions (the first and second terms) followed by the controlling value condition and then the error cancellation condition. The chosen precedence relation guarantees that antecedent assignments common to several path sensitization applications are identified whenever possible. Consequently, this contributes to defining propagation implicates common to other path sensitization applications.

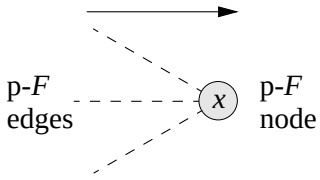
**Example 5.3.** The definition of the antecedent assignment for several blocking conditions is illustrated in Figure 5.5. In Figure 5.5-a, the antecedent assignment of $\pi(x)$ is given by all fanout edges:

$$\boldsymbol{A}(\pi(x)) = \{(\pi(x, y), 0) \mid y \in O(x)\} \tag{5.5}$$

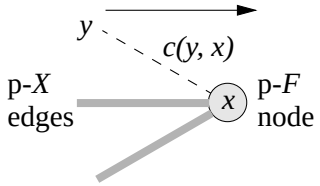
In Figure 5.5-b, the antecedent assignment is given by all fanin nodes in the propagation dimen-



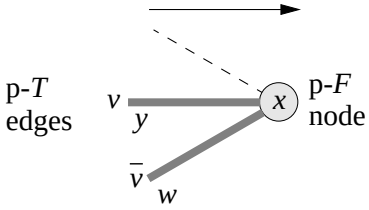

(a) All fanout edges blocked




(b) All fanin edges blocked




(c) p-*F* side input with controlling value




(d) Error cancellation

**Figure 5.5:** Blocking antecedents in test pattern generation

sion:

$$\boldsymbol{A}(\pi(x)) = \{(\pi(x, y), 0) | y \in I(x)\} \tag{5.6}$$

For the case of Figure 5.5-c, the antecedent assignment is given by the node that assumes the controlling value and the associated edge:

$$\boldsymbol{A}(\pi(x)) = \{(y, \nu(y)), (\pi(y, x), 0)\} \tag{5.7}$$

Finally, for error cancellation, the antecedent assignment is defined by the two nodes whose p-status cancel each other:

$$\boldsymbol{A}(\pi(x)) = \{(y, \nu(y)), (\pi(y, x), 1), (w, \nu(w)), (\pi(w, x), 1)\} \tag{5.8}$$

Thus, all antecedent assignments for blocking propagation to a node are defined. □

Assuming that $B(x)$ does not hold and $P(x)$ holds, then $x$ becomes p-*T*. The antecedent assignment of $\pi(x)$ is defined as follows:

$$\boldsymbol{A}(\pi(x)) = \bigcup_{y \in I(x)} \{(y, \nu(y)), (\pi(y, x), \nu(\pi(y, x)))\} \tag{5.9}$$

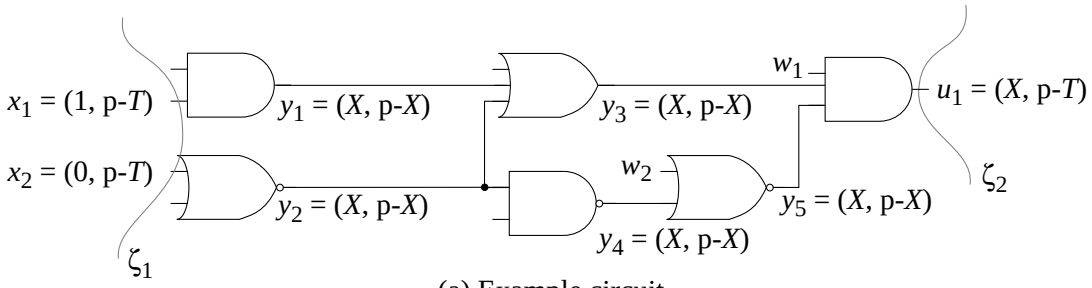
The antecedent assignments for the case of edge p-status are defined accordingly for both blocking and propagation conditions. In all cases, the antecedent set is readily obtained from the antecedent assignment with (2.14) on page 40.

We finally note that given the definition of antecedent assignment for propagation cuts, introduced in the previous chapter, and for logic assignments, defined in Section 3.2, the antecedent assignment of any element included in another antecedent assignment is well-defined, and so conflict analysis can be implemented.

**Value Probing**

While identifying USPs we can check whether the logic value assumed by each USP can be uniquely defined. If all propagation scenarios to a USP $x$ require the same logic value to be assumed by $x$, then $x$ must be assigned that value, in order to fanin justify the p-cut associated with the USP. The process of defining the logic value assumed by each USP is referred to as *value probing*. The antecedent assignment of this logic assignment is defined by the p-cut in the transitive fanin of the p-cut associated with the USP, the values assumed by the nodes in that fanin p-cut, and the assignments traced while defining the admissible values at each node in the transitive fanin of the USP.

**Example 5.4.** An example circuit where the logic value assumed by a USP can be identified is shown in Figure 5.6-a. The admissible pairs of logic and propagation values imposed solely by forward conditions are shown for each node, and where $X$ or p-$X$ denotes that both values are admissible in that dimension. While traversing the circuit graph for defining USPs, the admissible pairs of values are propagated forward and related with other pairs of values, with the goal of identifying the possible combinations of values at each node. When node $u_1$ is visited (denoting a propagation cut that requires fanin justification) we can immediately conclude that the combination of values (0, p-$F$) for $u_1$ would be the result of a propagation conflict and so the only admissible combination of values for $u_1$ is (1, p-$T$). Consequently, $u_1 \leftarrow 1$ is implied with antecedent assignment defined by $\{ (\zeta_1, 1) , (x_1, 1), (x_2, 0), (\pi(x_1), 1), (\pi(x_2), 1) \}$. The assignment of $u_1$ in the logical dimension implies several other assignments, which further constrain the search as shown in Figure 5.6-c. It is interesting to note that for this example, the final p-status of $y_3$ is actually

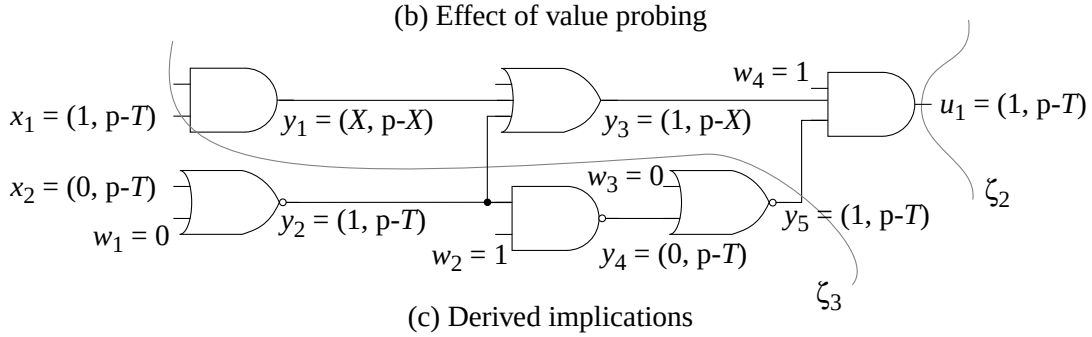



(a) Example circuit

| Node | $y_1$ | $y_2$ | $y_3$ | $y_4$ | $y_5$ | $u_1$ |
|---|---|---|---|---|---|---|
| Admissible values | (1, p-$T$)<br>(0, p-$F$) | (1, p-$T$)<br>(0, p-$F$) | (1, p-$F$)<br>(0, p-$F$)<br>(1, p-$T$) | (1, p-$F$)<br>(0, p-$T$) | (1, p-$T$)<br>(0, p-$F$) | ~~(0, p-$F$)~~<br>(1, p-$T$) |

- Pairs of values not shown are assumed to be ($X$, p-$F$).

(b) Effect of value probing




(c) Derived implications

**Figure 5.6:** Example of application of value probing

irrelevant for propagating the perturbation, and $\zeta_2$ could be said to be fanin justified, and $\zeta_3$ (derived from $\zeta_1$) to be fanout justified. However, by definition of the model, the final p-status of $y_3$ is required to be known. Nevertheless, we are guaranteed that no conflict will be identified between $\zeta_3$ and $\zeta_2$.

□

The implementation of value probing can be incorporated into the algorithm for identification of USPs, as was suggested by the previous example. Starting from a given p-frontier, all admissible pairs of values, which can be defined only by forward propagation, are associated with each visited node. For any node in a p-cut requiring fanin justification, for which only one logic value is admissible with a p-$T$ status, then that assignment must be made. The antecedents of the node assignment are defined by the initial p-cut, by the nodes in the initial p-cut (in both dimen-

sions), and by the edges blocking propagation (which contribute to defining the p-cut that requires fanin justification).

It is important to note that value probing does not always identify the logic value of a USP even if that value can be uniquely determined. On the other hand, whenever the logic value of a USP is identified with value probing, then that assignment is guaranteed to be a necessary condition for the perturbation to propagate to a primary output, given the current stage of the search process.

### 5.4.2 Diagnosis Engine

The basic diagnosis engine was described in Section 4.5.3, and can be readily implemented given the antecedent assignment definitions of the previous section. In the remaining of this section we focus on improvements to the diagnosis engine that can be implemented in linear time. We start by reviewing the improvements described in Section 3.6.2. Afterwards, application-specific techniques are described. In particular, the notion of *subleveling* is introduced, which proposes to diagnose conflicts taking into account that implications are identified by two distinct deduction engines, the logic and the propagation deduction engines.

The formulation of conflict diagnosis developed in the Section 4.5.3 allows straightforward extensions of UIPs, multiple conflicts and iterated conflicts to path sensitization. The implementation of each of these pruning methods increases the number of identified implicates. However, for path sensitization the relevance of each of these implicates depends on its composition.

Logical implicates are associated with the consistency function of the circuit and consequently are defined as *pervasive*. Hence, logical implicates can be derived within any target application and applied to any other target application. For example, logical implicates identified in logic verification can be used in timing analysis and in test pattern generation, or the ones derived in timing analysis can be used while performing satisfiability tests to the primary outputs of the circuit.

Propagation implicates are only valid where a propagation consistency function is defined, i.e. in target applications involving path sensitization. For the applications described in this disser-

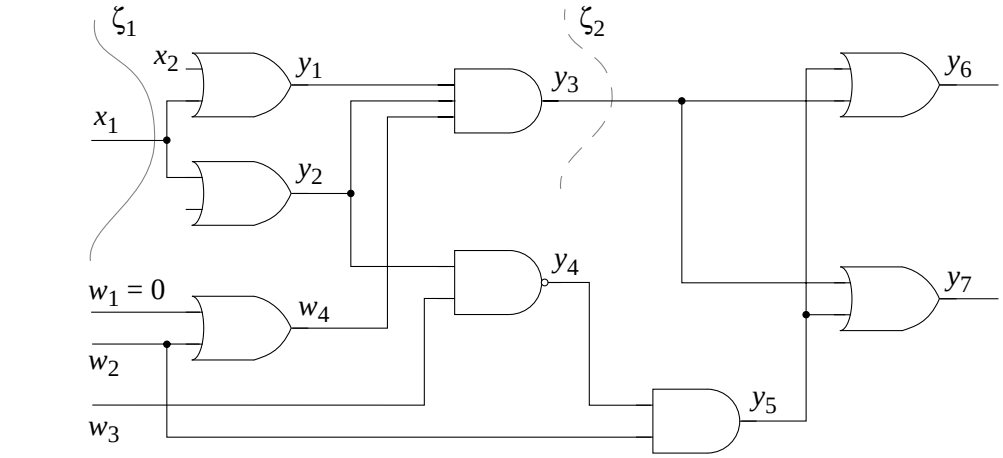
tation, and whenever propagation implicates are solely derived from common blocking conditions, then the propagation implicates are pervasive across path sensitization applications. This was the case with every example described in the previous chapter. On the other hand, and for the specific case of test pattern generation, whenever a propagation implicate results from application specific blocking or propagation conditions (e.g. error cancellation), then the derived propagation implicates are no longer pervasive across path sensitization applications. Nevertheless, these implicates are still pervasive for queries regarding test pattern generation. Thus, propagation implicates based on blocking and propagation conditions specific to the test pattern generation are defined as pervasive within test pattern generation.

**Diagnosis with Subleveling**

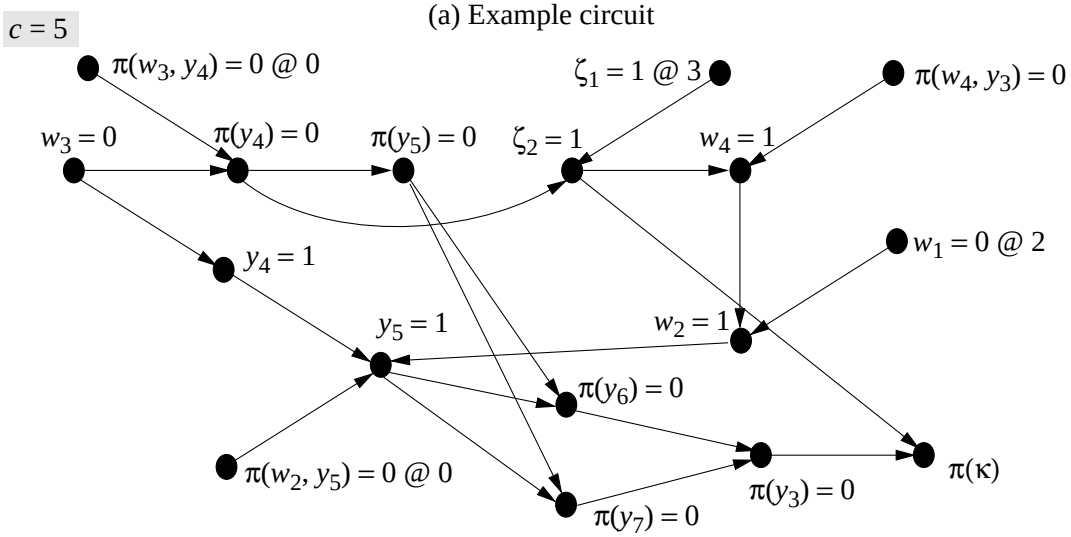
Derivation of implications with the path sensitization deduction engine is divided into two main phases: the identification of logical implications and the identification of propagation implications. Whenever a conflict is identified, the causes of the conflict are traced back to the decision assignment. The general improvements mentioned above consider exploiting some of the structure with which implications are derived. However, it is possible to identify other forms of structure that result from having two cooperating deduction engines. This can be attained by forcing additional structure on how implications are identified. The basic idea is to define a *decision sublevel* each time a new propagation cut is identified. In the presence of conflicts, the decision sublevels define an order on how to trace the causes of the conflict to the decision assignment. This order can be used to identify additional and stronger propagation implicates.

In order to implement subleveling, the state of each assigned symbol $\theta$ (node, edge or p-cut definition) is characterized by a new term, $\sigma(\theta)$, referred to as the sublevel of that symbol. Sublevels can then be used to identify portions of the implication sequence that include sufficient conditions for blocking propagation of a perturbation.

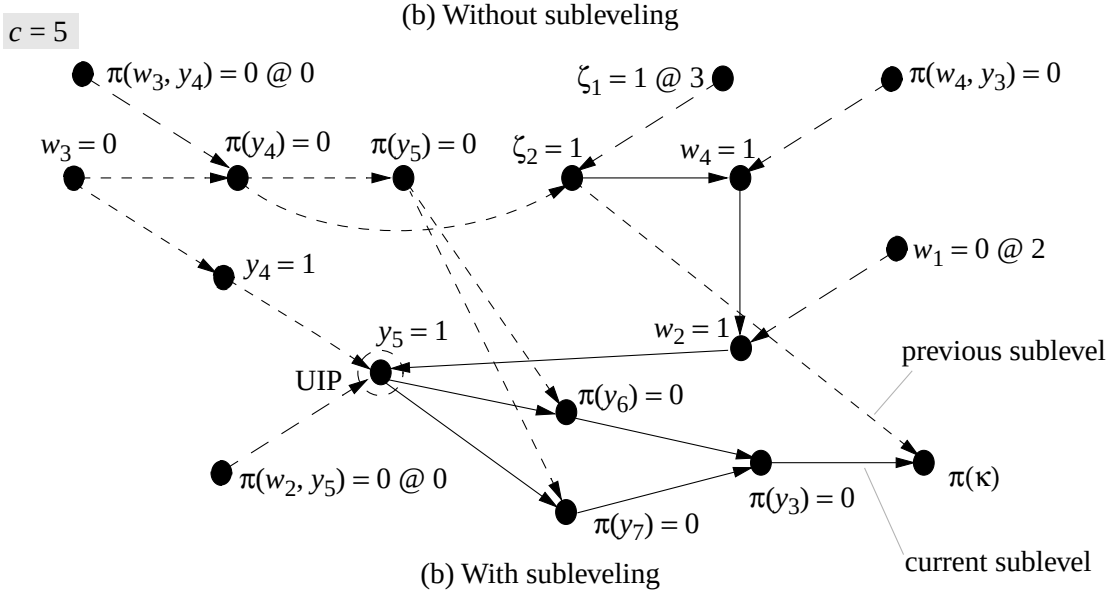
**Example 5.5.** The application of subleveling is illustrated with the example circuit of Figure 5.7-a. The current propagation cut is assumed to be $\zeta_1$, and the decision assignment is $w_3 = 0$. The resulting implication sequence is shown in Figure 5.7-b, where edge assignments are omitted for simplification purposes. Conflict diagnosis yields the p-clause:




(a) Example circuit




(b) Without subleveling




(b) With subleveling

**Figure 5.7:** Example of application of subleveling

$$\langle \zeta_1, (w_1 + w_3 + \pi(w_3, y_4) + \pi(w_4, y_3) + \pi(w_2, y_5)) \rangle$$

If subleveling is considered, then the definition of $\zeta_2$ creates a new decision sublevel. The implication sequence at the new sublevel leads to a conflict, for which $y_5$ is now a unique implication point (UIP). Consequently, from Figure 5.7-c, the following propagation implicate can be defined,

$$\langle \zeta_2, (\pi(y_5, y_6) + \pi(y_5, y_7) + \neg y_5) \rangle \tag{5.10}$$

that defines blocking conditions with respect to $\zeta_2$, which are independent of node assignments in the fanin of $\zeta_2$. The implicate derived with subleveling identifies blocking conditions local to $\zeta_2$, which can be used to constrain the value assumed by $y_5$.

The above implicate, derived with subleveling, can now be used in other instances of path sensitization. For example, let us assume a different stage of the search process with one p-cut $\zeta_3 = \{ x_2 \}$. In this situation, the p-cut $\zeta_2 = \{ y_3 \}$ is created due to USP $y_3$. Consequently, the p-clause of (5.10) is considered and $y_5 \leftarrow 0$ is implied. Note that $\pi(y_5, y_6)$ and $\pi(y_5, y_7)$ must be p-*F*, due to the composition of p-cut $\zeta_3$. □

The structure of the implication and diagnosis engines can be modified so as to permit the implementation of subleveling. A distinct decision sublevel is associated with assignments implied by each newly defined p-cut. The decision sublevel associated with each symbol θ is defined as follows:

$$\sigma(\theta) = max\{\sigma(\theta') | (\theta' \in \alpha(\theta)) \wedge (\delta(\theta') = \delta(\theta))\} \tag{5.11}$$

After a conflict is detected, the sublevel of each element in the implication graph provides a partition of the implication subgraph at the current decision level, that can then be used to create localized p-clauses.

### 5.4.3 Implementation Tradeoffs

As with SAT, the path sensitization algorithm proposed in the previous chapter may face efficiency problems if the number of derived implicates becomes too large. In addition, maintenance of p-cut information can introduce significant computational overhead, since p-cuts have to be updated after each implication sequence. In this section we propose to adapt the simplified

diagnosis engine described in Section 3.6.3 on page 97 to path sensitization for test pattern generation. The major advantage of the simplified diagnosis engine is that is guarantees a constant size clause database. Diagnosis engines that lead to worst-case polynomial size growths of the clause database are also reviewed.

**Constant Size Clause Database**

The basic idea is to maintain global conflicting assignment set information with level conflicting assignment sets associated with each decision level. Each time a conflict is identified, a conflicting assignment set is created, which is then used to update the level conflicting assignment sets. Assertions are defined as in Section 3.6.3 (see page 97), but are restricted to the logical dimension. The backtracking decision level is defined as the highest level of a non-empty level conflicting assignment set.

Additional complexity reduction is attained by simplifying the definition of the antecedent assignments of USPs. Every time a node or edge p-status becomes p-*F*, a global antecedent assignment *U* for USPs is updated. Consequently, the antecedent assignment of any USP is contained in *U*, and the antecedent assignments of USPs are only implicitly manipulated with references to set *U*. This fact also implies that the computational effort to manipulate USPs can then be associated solely to the performed graph traversals. All blocking conditions that define USPs and reduce the potential propagation paths are maintained in set *U*, hence the computation of functions *blockedby*() (see page 158) and *causesof*() (see page 67) is implicitly maintained by set *U* for every p-cut.

Since set *U* records all blocking conditions, p-cuts *need not* be explicitly maintained; *only* USP indications are required to be known, and their manipulation guarantees fanin and fanout justification of the associated p-cuts. Consequently, we can conclude that simplified conflict diagnosis and simplified p-cut maintenance can be implemented with small computational overhead when compared to the basic diagnosis engine and associated antecedent assignment manipulation.

The above approximations trade off some pruning precision with a potential reduction of the computational overhead involved in processing each decision level. As with the logical dimension situation (described in Section 3.6.3) we can easily construct examples where the effect of these approximations leads to an increase in the computed backtracking decision level.

We finally note that simplified conflict diagnosis does not allow for conflict-based equivalence, and so conflicts due to the same conditions may be identified more than once during the search process.

**Polynomially Bounded Clause Database**

Diagnosis engines, where recorded implicates are bounded in size, can also be devised. For p-clauses, two degrees of freedom exist. We may eliminate p-clauses with large p-cuts, or with large conditional clauses. In any situation, the growth is guaranteed to be polynomial in the size of the original representation for a maximum p-clause size of $m$. Note, however, that these diagnosis engines will require the manipulation of p-cuts whereas the constant size diagnosis engine deals with p-cuts implicitly.

Polynomially bounded diagnosis engines can be particularly useful in identifying and recording small p-clauses, which can be used often to imply assignments or to find equivalent conflicting conditions, and in discarding large p-clauses, which are necessarily harder to be subsequently used for deriving implications and for finding equivalent conflicting conditions.

Finally, note that we can also allow for different growths in the logical and propagation clause databases. For example, we may allow for a polynomial growth of the logical clause database and restrict the propagation clause database to a constant size, which then avoids the overhead of explicitly manipulating p-cuts.

## 5.5 Advanced Deduction Engines

In this section we describe advanced deduction engines for path sensitization, that extend the ideas described in Section 3.5.3 (see page 77) to the p-propagation model, but which restrict the subsets of variables to be tested to the logical dimension. This restriction is justified by the fact that p-$T$ assignments in the propagation dimension would have to be viewed as p-cuts, and in such a situation the manipulation of p-cuts would become more complex. (This is the same reason why assertions in the p-dimension are disallowed.) Note, however, that if the size of the subsets of variables is restricted to 1, then the requirement to manipulate several p-cuts no longer holds, and a restricted form of deduction engine can then be defined, which assigns values in the propagation dimension. Nevertheless, this restricted form of deduction engine is not extensible to larger sets of

variable assignments tested by the advanced deduction engine.

Given the above restriction, the basic ideas described in Section 3.5.3 can be used (with the algorithm described in Figure 3.8 on page 80) to define `Propagation_Deduce_`*k*(). The main difference results from the need to compute consensus between p-clauses. Since consensus of two p-clauses is defined only when the two p-clauses have the same p-cut, the *join* operation, defined in (4.28) on page 168, can be utilized to relate two p-clauses to a common p-cut.

Let $\omega_{\pi,1} = \langle \zeta_1, \omega_1 \rangle$ and $\omega_{\pi,2} = \langle \zeta_2, \omega_2 \rangle$ be p-clauses such that $\omega_1$ contains a literal $\theta$ and $\omega_2$ contains a literal $\neg\theta$. Furthermore, let us assume that $\omega_{\pi,1}$ and $\omega_{\pi,2}$ have been identified by diagnosing conflicts associated with a given set of variables, i.e. $\zeta_1$ and $\zeta_2$ currently define propagation cuts. We start by computing an assignment set that results from finding a p-cut common to $\zeta_1$ and $\zeta_2$:

$$\boldsymbol{A}_J = join(\zeta_1, \zeta_2) \tag{5.12}$$

Now, let $\zeta = cutof(\boldsymbol{A}_J)$. Then the resulting p-clause is defined as follows:

$$\omega_\pi = \langle \zeta, c(\omega_1, \omega_2, \theta) \cup \{ \mu^{\nu(\mu)} \,|\, (\mu, \nu(\mu)) \in cutof^C(\boldsymbol{A}_J) \} \rangle \tag{5.13}$$

which adds the literals of the symbols included in $cutof^C(\boldsymbol{A}_J)$ to the consensus of the conditional clauses. Note that $\theta$ cannot be included in $\boldsymbol{A}_J$, since it is unassigned at the current decision level and only becomes assigned due to the assignments tested by the deduction engine, whereas *join*() is computed without these assignments being defined.

**Preprocessing**

Different degrees of preprocessing can be implemented in test pattern generation. The logical clause database can be preprocessed with the objective of identifying additional implicates of the logical dimension consistency function. As described in Section 3.7 (see page 106), different preprocessing engines can be applied, which identify different sets of implicates.

For each fault, and before starting the search process, preprocessing for the path sensitization problem can be invoked. At this stage the objective is to identify propagation conflicts so that incorrect decision assignments can be prevented while searching. Any advanced deduction engine

`Propagation_Deduce_`*k*() can be used for preprocessing, but for practical purposes *k* has to be kept small. Note that the derived propagation implicates are pervasive for test pattern generation and consequently may be applied to the detection of other faults. The same necessarily holds true for all derived logical implicates.

## 5.6 Postprocessing Engine

Solution processing involves two orthogonal activities: removing redundancies from solutions and caching solutions for simplifying subsequent queries. The implementation of these techniques follows the description given in Section 3.8 (see page 111), but the existence of propagation information must be taken into consideration.

### 5.6.1 Removing Redundancies from Solutions

Computed solutions for path sensitization can include some redundancies, i.e. decision assignments that are not relevant for satisfying the original objectives. The approach for removing redundancies from path sensitization solutions is based on constructing the node justification graph (defined in Section 3.8.1), now referred to as the *variable justification graph*. For path sensitization, each p-*T* primary output defines by itself a sufficient condition for propagating a perturbation, i.e. for sensitizing a path. Hence, we can create the variable justification graph with respect to any single p-*T* primary output, and to the initial logical objective that activates the fault. Nevertheless, as we show below, for caching solutions the complete justification graph is useful, and so we propose to construct the variable justification graph for all p-*T* primary outputs, and remove redundancies with respect to a randomly chosen p-*T* primary output. (In practice, all p-T primary outputs can be analyzed and the one with the least number of decision assignments can be chosen.)

The definition of sets $M(y)$ is extended to node and edge p-status. Hence $M(\theta)$ identifies the conditions that (fanin) justify $\theta$. In is important to note that any traced assignment in the propagation dimension cannot be assigned due to fanout conditions, because such an assignment is in the transitive fanin justification chain of a p-*T* primary output. For a p-*T* / p-*F* node or edge $\theta$, set $M(\theta)$ is defined by the antecedent assignment of $\theta$. With respect to definitions for the node justification graph $J_G = (V_J, E_J)$ given in Section 3.8.1 that apply for logic assignments, the following

modifications are required for the variable justification graph:

1. Every p-$T$ primary output $z$ corresponds to a vertex $\eta(\pi(z))$ in $V_J$. The initial logical objective $y = v_y$ corresponds to a vertex $\eta(y)$ in $V_J$.
2. For each vertex $\eta(\theta)$ in $V_J$, denoting the assignment of $\theta$ and such that $\eta(\theta)$ has no incoming edges and $y$ is not a primary input, identify $M(\theta)$. For each node $\mu$ in $M(\theta)$, add $\eta(\mu)$ to $V_J$ and let $(\eta(\mu), \eta(\theta)) \in E_J$ .

The next step is to choose one of the p-$T$ primary outputs. Let $z$ be such a primary output and let $J_z = (V_z, E_z)$ be the subgraph of $J_G$ defined from sink vertex $\eta(\pi(z))$ and from the initial logical objective. Then, the set of primary inputs assignments to be considered as the solution to the path sensitization problem is given by,

$$\boldsymbol{A_S}' = \{(x, \nu(x)) | x \in PI \wedge \eta(x) \in V_z\} \tag{5.14}$$

that is necessarily included in the solution assignment set $\boldsymbol{A_S}$. Any primary output $z$ can be chosen for defining $\boldsymbol{A_S}'$.

**Example 5.6.** Redundancy removal from solutions is illustrated with the example of Figure 5.8-a, which is based on the example circuit of Figure 3.21 (see page 112). The decision tree for the search process is shown in Figure 5.8-b, and the corresponding variable justification graph is shown in Figure 5.8-c. It is immediate that the assignments $x_1 = 1$ and $x_2 = 1$ are redundant. Consequently the assignment set obtained from the variable justification graph becomes $\boldsymbol{A} = \{ (x_3, 0), (x_4, 0), (x_5, 1), (x_6, 1) \}$, thus reducing the computed decision assignment set. Finally, note that $\boldsymbol{A}$ can be further simplified (as was described in Section 3.8.1). □

Another approach for removing redundancies from solutions is to consider subsets of the assignments to the primary inputs, and test whether any subset identifies a solution to the path sensitization problem. The complexity of this approach is given in Section 3.8.1 and depends on the amount of solution simplification attempted.

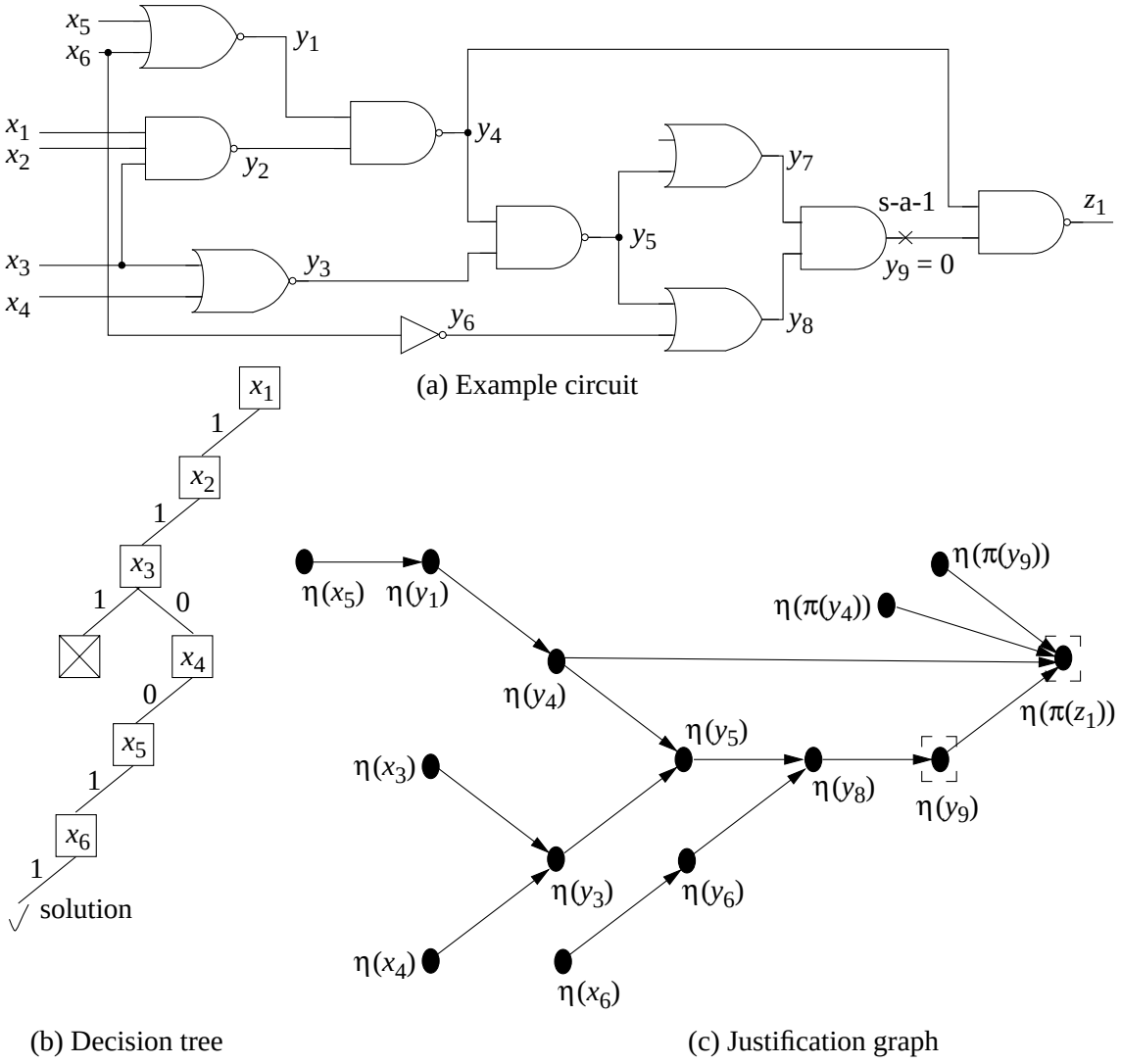



**Figure 5.8:** Removing redundancies from solutions

### 5.6.2 Caching Solutions

The node justification graph forms the basis upon which relevant information about computed solutions can be identified so that it may be applied to simplifying the search for solutions of subsequent queries. The construction of the variable justification graph includes all p-*T* primary outputs. In the following, the restriction of the previous section is assumed, i.e. one primary output is randomly chosen and a subgraph of that primary output and of the original logical objective is created. The procedure described below can then be applied to *each* individual p-*T* primary output.

The steps described in Section 3.8.2 (see page 116) are implemented, such that the sets

$\boldsymbol{P}_0$, …, $\boldsymbol{P}_K$ denote a partition of $V_J$ and the level cuts $\boldsymbol{T}_j$ associated with each decision level are defined using the partition and (3.28) on page 118. In such a situation, the conditions for matching cached solutions are given by (3.31), whereas the resulting assignment set to the primary inputs is defined by (3.32).

Note that since the variable justification graph may not include information regarding some decision levels and is based on a sequence of required justification assignments, the conditions created for solution caching are necessarily less redundant than those of EST [70, 71], which creates conditions based on node assignments of cuts of the circuit graph and current D-frontiers, and thus does not consider any dependency information for simplifying the created conditions.

## 5.7 Decision Making Procedures

The decision making procedures (or selection engines) described in Section 3.9 can be used with the path sensitization algorithm, where decision assignments are restricted to the logical dimension. Nevertheless, a few modifications must be introduced. First, the definition of new head lines cannot include p-$X$ nodes, since these nodes may become involved in propagating the perturbation and so nodes in their transitive fanin must be assigned in the logical dimension. Second, the definition of don't care nodes cannot involve p-$X$ nodes, since these nodes can potentially propagate a perturbation. Finally, objectives for backtracing can be defined from any propagation frontier as well as the justification frontier, whereas in the logical dimension, objectives are always drawn from the j-frontier.

The selection engine can be organized in several different ways. For example, decision assignments may always be based on the same procedure for making decisions, e.g. simple or multiple backtracing. Note, however, that the search framework also permits several decision making procedures to be iteratively applied after a given threshold on the number of backtracks. Since the search process records logical and propagation implicates, the search effort spent on a given decision making procedure provides additional information that can be used by subsequent decision making procedures for pruning the search. We can thus conclude that most selection engines can be used in LEAP().

The implementation of simple and multiple backtracing in the p-propagation model must

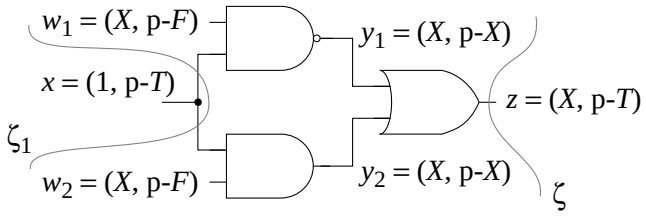
specify which nodes are to be traced. For test pattern generation, tracing is restricted to p-*F* nodes and is conducted until elements in the current set of head lines are visited. Different controllability and observability measures can be implemented [1, pp. 213-220], which can take into account the potential existence of several p-frontiers. For the results reported in Chapter VII for test pattern generation, the implemented selection engine can use simple and multiple backtracing and is solely based on topological controllability and observability measures. Since the selection engine is orthogonal to how the search process is implemented, other more elaborated measures can be implemented, some of which can be based on dynamic testability considerations derived from the test pattern generation process [26, 85].

## 5.8 Scaling the Perturbation Propagation Model

Path sensitization models based on the D-calculus have attempted over the years to increase the reasoning ability on the cone of influence of the fault effect. The motivation for the added reasoning ability is that conflicting conditions can be more easily identified, and therefore the amount of search can be reduced. The quest for added reasoning ability has led to 5, 9, 10, 11 and 16-valued algebras (among others) [1, 2, 23, 33, 37] derived from the D-calculus.

We propose to show that the p-propagation model can incorporate any such degree of reasoning ability, without compromising any of the features that allow the search algorithm to implement conflict diagnosis.

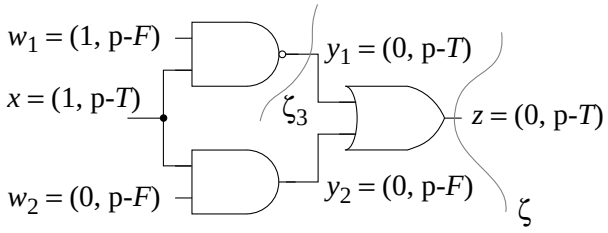
As noted by other authors [37], *m*-valued algebras are approximations to considering all possible values at each node in the cone of influence of the fault effect, and this corresponds to considering a 16-valued algebra. Thus, let us consider for each node all admissible pairs of values it can assume in both dimensions. For a complete node assignment, each node must assume one out of four admissible pairs of values: (0, p-*F*), (1, p-*F*), (0, p-*T*) and (1, p-*T*). Let us assume a p-cut $\zeta$ and for each node $x$ in $\zeta$ let us consider all possible values it can assume; since $x$ is a p-*T* node, and assuming that it can be part of a sensitizable path, then at most two combinations of values are possible, (0, p-*T*) and (1, p-*T*), or ($v$, p-*T*) if $x$ is already assigned value $v$. Afterwards, we propagate through the circuit the admissible pairs of values for each node, relate those pairs of values with the ones of other nodes and compute the admissible pairs of values for the fanout nodes.




(a) Example circuit

| Node | $x_1$ | $w_1$ | $w_2$ | $y_1$ | $y_2$ | $z$ |
|---|---|---|---|---|---|---|
| Admissible values | (1, p-$T$) | (0, p-$F$)<br>(1, p-$F$) | (0, p-$F$)<br>(1, p-$F$) | (1, p-$F$)<br>(0, p-$T$) | (1, p-$T$)<br>(0, p-$F$) | ~~(1, p-$F$)~~<br>(0, p-$T$) |

(b) Admissible values




(c) Result of $z = 0$

**Figure 5.9:** Potential value propagation

Whenever only one value in either dimension becomes admissible, an assignment is implied. The antecedent assignment for such assignment is defined by all assignments in either dimension that constrain the admissible pairs of values from the p-cut until the node is visited. Moreover, it should be noted that this procedure is a generalization of value probing described in Section 5.4.1, where probing is extended to any circuit node.

**Example 5.7.** Propagation of admissible values is illustrated in Figure 5.9. Let $x = 1$ and $\pi(x) = 1$, and let $\pi(z) = 1$ (i.e. $z$ is a USP). Now let us consider estimating the admissible logic and propagation values at each node in the transitive fanout of $x$, as shown in Figure 5.9-b. $y_1$ can either assume the pairs of values (1, p-$F$) or (0, p-$T$) and $y_2$ can assume (1, p-$T$) or (0, p-$F$). As a result, the admissible pairs of values at $z$ would be (1, p-$F$) or (0, p-$T$). However, $z$ must be p-$T$ since it denotes a USP, and consequently the assignment $z \leftarrow 0$ is implied. The antecedent assignment of $z$ is given by $\boldsymbol{A}(z) = \{ (\zeta_1, 1), (x, 1), (\pi(x), 1) \}$. The consequences of this assignment are shown in Figure 5.9-c, and consequently $\zeta$ becomes fanin justified. □

Observe that for each node all possible cases for propagating a perturbation are considered, and so the formulation of the model identifies no fewer implications that the 16-valued D-calculus based algebra. Furthermore, advanced deduction engines can also use the above analysis to identify additional implications.

**Changing the Semantics of Edge Propagation Status**

The p-propagation model can be further improved. As mentioned earlier, the edge p-status can be made redundant, provided blocking conditions are expressed solely in terms of node values. We start by showing how edge p-status may not be considered in the formulation of the p-propagation model for test pattern generation. Afterwards, the semantics of the edge p-status is redefined in order to increase the reasoning ability in the propagation dimension.

In test pattern generation if two connected nodes $x$ and $y$ are p-$X$, then $\pi(x, y)$ is also p-$X$. Consequently, the edge p-status can be made redundant, and the blocking condition is defined as follows:

$$\begin{aligned} B(x) = & \left[ \prod_{y \in I(x)} (\pi(y) = 0) \right] + \left[ \prod_{y \in O(x)} (\pi(y) = 0) \right] + \\ & \left[ \sum_{y \in I(x)} ((\pi(y) = 0) \cdot c(y, x)) \right] + \\ & \left[ \sum_{y, w \in I(x)} ((\pi(w) = 1) \cdot (\pi(y) = 1) \cdot (\nu(w) \oplus \nu(y) = 1)) \right] \end{aligned} \tag{5.15}$$

The propagation condition becomes,

$$P(x) = \prod_{y \in I(x)} [(\pi(y) = 1) \cdot (\nu(y) \neq X) + (\pi(y) = 0) \cdot nc(y, x)] \tag{5.16}$$

where with respect to (4.19) and (4.20) on page 159, each edge p-status reference has been replaced by the appropriate node p-status reference.

Edge p-status can now be used to identify situations where a perturbation may reach a node but cannot propagate to a primary output. Given the stage of the search process we may not be able yet to assign p-$F$ to the fanout node, but we may use the edge information to focus the

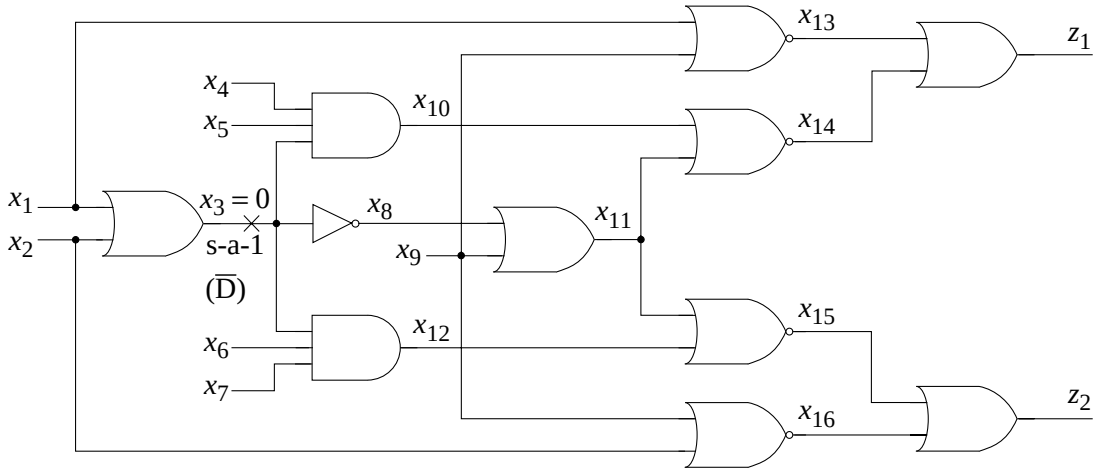
search on other potential propagation paths. The p-status of an edge is now defined as follows:

1. $\pi(x, y) = 0$ if either the edge cannot propagate a perturbation, or even if it propagates the perturbation, the edge will not be part of a sensitizable path.
2. $\pi(x, y) = 1$ the edge propagates a perturbation.
3. $\pi(x, y) = X$ if the edge can potentially propagate a perturbation.

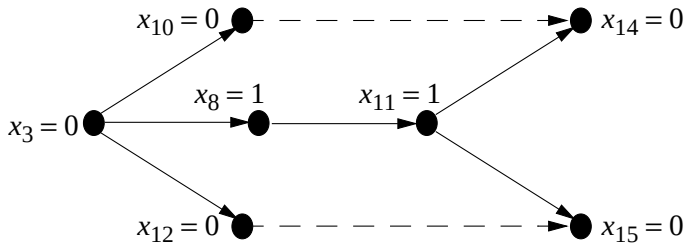
This modified formulation of the edge p-status can lead to situations where a p-*F* edge connects two p-*X* nodes *x* and *y*. This then signifies that a perturbation does not propagate from *x* to *y*, even if both nodes may propagate a perturbation. With the new formulation, edge p-status is maintained separately from node p-status and logic value relations can be used to identify edges that must be p-*F*. This is the case, for example, of every edge whose fanin node assumes a non-controlling value of a controlled fanout node; a perturbation in the fanin node cannot propagate to the fanout node and so the edge p-status can be assigned p-*F*.

**Example 5.8.** The application of the new definition of edge p-status is illustrated with the example circuit of Figure 5.10-a. Let us assume the objective is to compute a test for fault $x_3$ s-a-1. Logical implications create the implication subgraph shown in Figure 5.10-b. Next we note that since the value of $x_{10}$ is 0, and the value of $x_{11}$ is 1, then a perturbation in $x_{10}$ will not propagate to $x_{14}$; it either cancels propagation from $x_{11}$, if it becomes p-*T*, or allows perturbation to $x_{14}$, if it becomes p-*F*. The same holds true with respect to propagation from $x_{12}$ to $x_{15}$. These relations are highlighted in Figure 5.10-b, We can then say that $\pi(x_{10}, x_{14}) = 0$, and similarly $\pi(x_{12}, x_{15}) = 0$. As a result, USP computation (over p-*X* nodes and edges) identifies $x_8$ and $x_{11}$ as USPs, since these nodes must indeed propagate the perturbation for it to reach a primary output. The new p-cut $\{ x_{11} \}$ requires fanin justification and hence $x_9 \leftarrow 0$ is implied. The consequence of these implications is shown in Figure 5.10-c, and a propagation conflict is identified. ($\sigma_o$ identifies a p-cut driven by the primary outputs, which requires fanin justification as defined in Section 4.3.4.1.) Since no decisions have been made, the fault is proved redundant. Observe that a crucial step is to set the edge p-status to p-*F*, so that the USPs can be defined and the propagation conflict identified. □

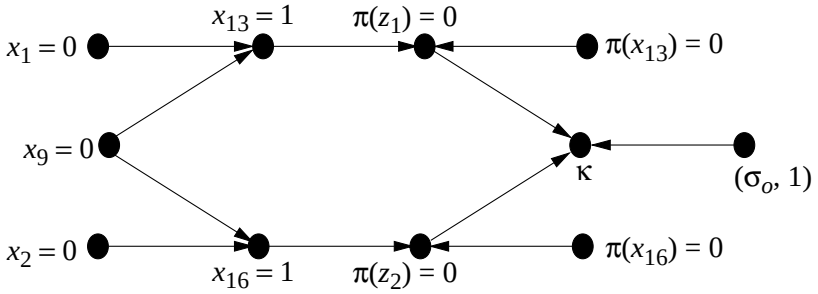
Antecedent assignments for edge p-status assignments are defined by the assignments that




(a) Example circuit




(b) Blocking implication sequence




(c) Propagation blocked

**Figure 5.10:** Application of redefined edge p-status

are involved in downgrading or upgrading the edge p-status. For the above example, $\boldsymbol{A}(\pi(x_{10}, x_{14}))$ = { $(x_{10}, 0)$, $(x_{14}, 0)$ }.

For test pattern generation we can either use the basic definition of edge p-status, not consider edge p-status at all, or use the modified definition of edge p-status with the goal of further pruning the search. The first option simplifies the integration of test pattern generation with timing

analysis in a common path sensitization framework. The second simplifies the implementation of test pattern generation, and necessarily reduces the overhead of maintaining path sensitization information. Finally, the third option increases the model precision which can be useful for difficult faults.

## 5.9 Comparison with other Test Pattern Generation Algorithms

The proposed test pattern generation algorithm introduces the following improvements over most path sensitization algorithms:

- The development of diagnosis engines, that allow several pruning methods to be defined and applied in an integrated procedure for handling conflicts. Most test pattern generation algorithms such as PODEM [72], FAN [62], SOCRATES [144, 145], TOPS [92], QUEST [37], TAGUS [162], SSR [167], TRAN [24] and recursive learning [101] do not provide *any* form of conflict diagnosis. For combinational circuits, EST allows defining equivalent conflicting conditions but, as shown earlier, the recorded conditions can be significantly redundant since they do not directly reflect the causes of a conflict. For sequential circuits, the algorithms of [113] and [114] propose restricted forms of non-chronological backtracking, but are only informally sketched and, given the descriptions in [113] or [114], are incomplete.
- The definition of deduction and preprocessing engines that can identify implicates of the consistency function (in both dimensions) with different degrees of precision. These engines extend other procedures for derivation of implications [24, 37, 101, 145] in that any degree of deduction can be achieved and implicates are added to the clause databases. Furthermore, in some cases these implicates are identified as pervasive. In such a situation, derived implicates can be *permanently* added to the clause databases and applied for subsequent problem instances. They can also be potentially used in different target applications.
- The introduction of postprocessing engines that allow processing computed solutions for path sensitization problems. Redundancy removal from solutions is a new concept, whereas solution caching has been proposed before, with a different formulation, in EST [70, 71]. As mentioned earlier in this chapter, the proposed procedure for caching solutions is necessarily more precise than that of EST.

- The definition of a scalable path sensitization model, where different degrees of propagation reasoning precision can be implemented within the same search framework. The most precise formulation supersedes existing models for path sensitization in test pattern generation.

Conflict diagnosis offers other interesting possibilities. As noted earlier, derived logical implicates are pervasive across circuit analysis applications. Propagation implications are pervasive within test pattern generation and, under appropriate conditions, are pervasive across path sensitization applications.

Finally we note that the proposed test pattern generation algorithm can be configured to realize a large number of other test pattern generation algorithms, some of which have been proposed by other researchers. For example, with an adequate formulation of the diagnosis and deduction engines, and no realization of the preprocessing and postprocessing engines, we can readily implement PODEM and FAN. SOCRATES can be implemented by allowing for restricted preprocessing, and identification of USPs. (Our algorithm for identification of USPs is more efficient than that of SOCRATES, and this leads to a more efficient implementation of SOCRATES.) In order to emulate the aforementioned algorithms, the formulation of the p-propagation just needs not consider fanout blocking conditions in order to emulate the D-calculus.

## 5.10 Summary

This chapter details the application of the path sensitization model and algorithm to test pattern generation. Most of the concepts regarding the path sensitization algorithm were previously described in Chapter III and Chapter IV, and the purpose of this chapter is solely to describe the necessary modifications given that the problem being solved is path sensitization for test pattern generation.

The engines associated with the search algorithm for path sensitization are described, and emphasis is given to conflict diagnosis. The concepts of subleveling and value probing are introduced, which can be respectively used to derive smaller propagation implicates in the presence of conflicts and identify more implications. Simplifications to conflict diagnosis are proposed, which provide computationally inexpensive methods to diagnose conflicts and guarantee constant size clause databases. Diagnosis engines with worst-case polynomial size growth of the clause data-

base are reviewed, which are based on equivalent engines described in Chapter III. Engines for advanced deduction, postprocessing and selection are analyzed in the context of path sensitization for test pattern generation.

The chapter concludes with a study of accuracy tradeoffs provided by the p-propagation model, and describes how accuracy can be improved. In addition, the path sensitization algorithm is compared with algorithms proposed by other authors.

# CHAPTER VI

# PATH SENSITIZATION FOR TIMING ANALYSIS

## 6.1 Introduction

As mentioned earlier in Chapter I, timing analysis poses key challenges to search algorithms, since the final result of the search is most often to prove unsatisfiability of a given path sensitization goal. Instances of path sensitization for timing analysis thus pose interesting test cases for pruning methods associated with diagnosis engines.

This chapter describes how to apply the p-propagation model and LEAP to path sensitization for timing analysis. The organization of LEAP proposed in Chapter IV is assumed. As in Chapter V, the emphasis is how to solve a target application that is based on path sensitization. In particular, we focus on how to represent path sensitization for timing analysis with the p-propagation model and how to configure LEAP() for timing analysis.

Simplifications to conflict diagnosis are described in some detail, since they represent the core of the experimental results described in Chapter VII and can be particularly useful in timing analysis, where delay-based dependencies can lead to large propagation implicates.

**Outline**

The general procedure for circuit delay computation is described in Section 6.2, where several techniques for iterating threshold delays are analyzed. Section 6.3 describes how to represent path sensitization in timing analysis with the p-propagation model. Afterwards, in Section 6.4, the basic deduction and diagnosis engines are detailed for timing analysis. For these engines, iden-

```
// Output argument:    status ∈ { ABORTED, FAILURE, SUCCESS }
// Return value:       Computed circuit delay
//
Circuit_Delay_Computation (&status)
{
   define φ for circuit;
   Δ = LTP;          // Initial path delay is largest topological path
   while (Δ > 0) {
      define ψ for timing analysis given Δ;
      status = LEAP();          // Attempt to sensitize path delay Δ
      if (status == SUCCESS && Circuit_Delay_Identified())
         return Define_Sensitizable_Delay ();          // Return Δ
      else if (status == ABORTED)
         return −1;               // Computational resources exceeded
      Δ = Iterate_Next_Delay();           // Identify next path delay
   }
   return Δ;
}
```

**Figure 6.1:** Circuit delay computation procedure

tification of antecedent assignments due to delay-based blocking conditions plays a key role. Other engines that can be used for implementing timing analysis are described in Section 6.5. A comparison of LEAP with other timing analysis algorithms is conducted in Section 6.6.

## 6.2 Circuit Delay Computation in Timing Analysis

The main objective of timing analysis is circuit delay computation, that entails computing the maximum delay $\Delta_C$ over the sensitizable paths of the circuit. Such delay is referred to as the *sensitizable path delay* (or *circuit delay*). The general procedure for circuit delay computation is shown in Figure 6.1. A procedure for iterating threshold delays is assumed. The search for the largest sensitizable path iteratively defines the next threshold delay and invokes procedure LEAP() that was described in Figure 4.10 on page 152. The process is iterated until a sensitizable path is found for a chosen threshold delay and such that this delay is declared to be the last threshold delay by procedure `Circuit_Delay_Identified`(). Note that this implementation permits several procedures for iterating threshold delays to be modeled and used. Furthermore, the pro-

posed circuit delay computation procedure is said to implement *concurrent path sensitization* [152] since it considers the sensitization of all paths within a given delay range. In contrast with the path-by-path analysis of initial solutions for circuit delay computation [8, 15, 31, 55, 117, 129, 149], concurrent path sensitization entails some mechanism to iterate path delays or delay thresholds and, in each case, to specify the associated path sensitization problem. For example, threshold delay iteration procedures can perform a binary search over the possible range of path delays or enumerate threshold delays in decreasing order starting from the largest topological path delay.

One procedure for iterating path delays is described in [152]. However, if the number of distinct path delays is large[1], then the time to find the largest sensitizable path can become unacceptable. Another procedure is to choose a fixed delay decrement $d$ and at each iteration decrement the target threshold delay by $d$. In such a situation, if a sensitizable path is found with delay $\Delta$, then the result reported must be $\Delta_C = \Delta + d$, and the largest delay error is $d$. Note, however, if the least path delay above $\Delta$ is larger than or equal to $\Delta + d$, then the delay value returned is $\Delta_C = \Delta$; this is the case, for example, whenever unit delays are assumed and $d = 1$.

More precise approaches can be developed. If $\varepsilon$ is the allowed error in computing the largest sensitizable path, then the following procedure can be used:

1. Let $\Delta$ be the first threshold delay for which a sensitizable path is found, i.e. for $\Delta + d$ no sensitizable path was found.
2. Perform binary search in the delay range $(\Delta, \Delta + d)$, starting with $\Delta' = \Delta / 2$, until the delay different between iterations is less than $\varepsilon$. Report delay $\Delta' + \varepsilon$.

In such a situation, the computed delay is at most off by an excess of $\varepsilon$ with respect to the largest sensitizable path delay in floating mode operation. The number of iterations in the range $(\Delta, \Delta + d)$ is then given by $O(\log_2(d/\varepsilon))$. Assuming that $\Delta$ is computed by iterated decrements of $d$ with respect to the original longest topological path (LTP), then the number of iterations is bounded by:

$$O(\mathrm{LTP}/d + \log_2(d/\varepsilon)) \tag{6.1}$$

On the other hand, if binary search is used to compute the first sensitizable delay $\Delta$ above, then the

[1] In the worst-case the number of path delays is exponential in the number of circuit nodes.

number of iterations is bounded by:

$$O(\log_2(\mathrm{LTP}/d) + \log_2(d/\varepsilon)) = O(\log_2(\mathrm{LTP}/\varepsilon)) \quad (6.2)$$

Note, however, that in most cases the largest sensitizable path delay is close to LTP, and consequently the procedure based on iterated decrements requires fewer iterations. This is the case, for example with all the ISCAS'85 benchmark circuits [156]. On the other hand, carry-skip adders contain a large number of path delays for which no sensitizable paths exist. For some of these circuits, binary search requires fewer iterations than iterated decrements of the delay threshold.

In the remaining of this chapter a threshold delay $\Delta$ is assumed to be defined prior to creating the path sensitization problem, with any of the above procedures.

Procedure `Toggle_Propagation_Values`() (invoked from procedure `LEAP`() in Figure 4.10 on page 152) must also be defined. After a solution to the circuit delay computation problem is identified, the set of sensitizable paths must satisfy (4.5) on page 139. All other p-*T* edges and nodes are downgraded to p-*F*. A backward traversal from the primary outputs can be used to visit each circuit node / edge and downgrade those for which the blocking condition holds.

## 6.3 Modeling Circuit Delay Computation in Timing Analysis

In this section we describe how the p-propagation model can be used to model the path sensitization problem associated with the following question: *Given a threshold delay* $\Delta$*, are there any floating-mode sensitizable paths with delay no less than* $\Delta$? Recall from Chapter I that a path is said to be floating-mode sensitizable if and only if for a given primary input assignment, each node on the path stabilizes as a direct consequence of its fanin node on the path also stabilizing [31, 50, 151, 153].

The definition of the p-propagation model for timing analysis assumes a set of delay estimates at each node and at each edge. In particular, the following delay estimates are assumed:

1. *DTo*(*x*) denotes the estimate of the maximum propagation delay for a signal transition to propagate from a primary input to *x*.
2. *DFrom*(*x*) denotes the estimate of the maximum propagation delay from *x* to any primary output.

3. *DThru*($x$) denotes the estimate of the maximum propagation delay for a signal transition to propagate from a primary input to a primary output if that signal transition propagates through $x$.
4. *DThru*($x$, $y$) denotes the estimate of the maximum propagation delay for a signal transition to propagate from a primary input to a primary output such that the signal transition propagates through edge ($x$, $y$).

Each delay estimate is computed as follows:

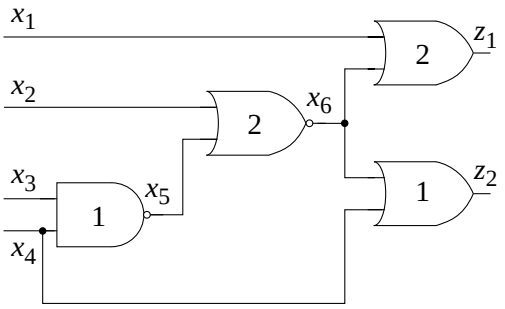
$$DTo(x) = \begin{cases} 0, & \text{if } x \in PI \\ max\{DTo(y) + D(y, x) | y \in I(x)\}, & \text{if } \neg Cont(x) \\ max\{DTo(y) + D(y, x) | y \in U(x)\}, & \text{if } Cont(x) \wedge Unjust(x) \\ min\{DTo(y) + D(y, x) | y \in C(x)\}, & \text{if } Cont(x) \wedge Just(x) \end{cases} \tag{6.3}$$

The propagation delay estimate to a primary input is defined to be 0[2]. If $x$ is not controlled, then the maximum propagation delay estimate to $x$ is given by the maximum of the delay estimates to its fanin nodes added with the corresponding edge delays. Otherwise, if $x$ is controlled but unjustified, then the maximum delay estimate to $x$ is given by the maximum of the delay estimates to its unassigned fanin nodes added with the corresponding edge delays. Finally, if $x$ is controlled and justified, then the propagation delay to $x$ is given by the minimum of the delay estimates to its controlling fanin nodes added with the corresponding edge delays. Note that from a simulation perspective, (6.3) corresponds to modeling floating-mode operation and *DTo* identifies the stable time of each node.

$$DFrom(x) = \begin{cases} 0, & \text{if } x \in PO \\ max\{D(x, y) + DFrom(y) | y \in R(x)\}, & \text{if } R(x) \neq \emptyset \\ -\infty, & \text{if } R(x) = \emptyset \end{cases} \tag{6.4}$$

The propagation delay from a primary output is defined to be 0. If the set of relevant outputs of $x$ is not empty, then the maximum propagation delay estimate from $x$ to a primary output is the maxi-

[2] If each primary input $x$ has a distinct arrival time, then *DTo*($x$) is defined to be that arrival time.




| Assignment | $DTo(x_6)$ | $DFrom(x_6)$ |
|---|---|---|
| none | 3 | 2 |
| $x_1 = 1$ | 3 | 1 |
| $x_2 = 1$ | 2 | 1 |

**Figure 6.2:** Example of updating propagation delay estimates

mum delay over the outputs for which $x$ can be relevant added with the corresponding edge delays. If the set of relevant outputs is empty, then the propagation delay estimate from $x$ to a primary output is $-\infty$, which means that a signal transition that reaches $x$ does not propagate to a primary output.

$$DThru(x) = DTo(x) + DFrom(x) \tag{6.5}$$

and finally,

$$DThru(x, y) = \begin{cases} -\infty, & \text{if } y \notin R(x) \\ DTo(x) + DFrom(y) + D(x, y), & \text{if } y \in R(x) \end{cases} \tag{6.6}$$

since $x$ can propagate to $y$ provided it can be relevant to the propagation delay to $y$.

**Example 6.1.** An example of how delay estimations are updated is shown in Figure 6.2. For node $x_6$, with no logic assignments, $DTo(x_6) = 3$ and $DFrom(x_6) = 2$. The assignment $x_2 = 1$, guarantees that $x_6$ stabilizes no later than time unit 2, i.e. $DTo(x_6) = 2$. After all assignments are made, $DThru(x_6, z_1) = -\infty$ and $DThru(x_6, z_2) = 3$. □

For each circuit, and with all nodes unassigned, the delay estimates can be initially computed with two levelized breadth-first traversals of the circuit graph; one forward traversal for computing *DTo* estimates, and one backward for computing *DFrom* as well as *DThru* estimates. Given the initial delay estimates for each node and edge, the initialization of the p-propagation model consists in setting to p-$X$ all nodes $x$, with $DThru(x) \geq \Delta$, and edges $(x, y)$, with $DThru(x, y)$

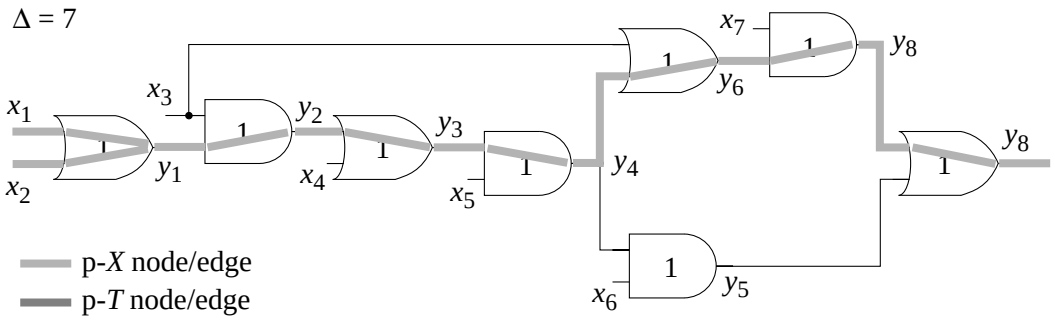



**Figure 6.3:** Initialization of the p-propagation model for timing analysis

$\geq \Delta$.

**Example 6.2.** The example circuit (adapted from [83]) shown in Figure 6.3 illustrates the initialization of the p-propagation model for timing analysis. For $\Delta = 7$, only two paths can propagate a perturbation, respectively $\langle x_1, y_1, y_2, y_3, y_4, y_6, y_7, y_8\rangle$ and $\langle x_2, y_1, y_2, y_3, y_4, y_6, y_7, y_8\rangle$. These paths define the set of p-*X* nodes and edges. Note that since delay computation entails a given threshold delay, the p-status definitions for each node or edge must include conditions on delay estimates that will contribute to defining the final p-status of that node or edge. □

The condition for the initialization of the p-status of each node, implies that every node in each path $\boldsymbol{P}$, such that $D(\boldsymbol{P}) \geq \Delta$, is initialized to p-*X*. The same holds true for every edge associated with each such path $\boldsymbol{P}$. Conversely, we may have paths $\boldsymbol{Q}$, with $D(\boldsymbol{Q}) < \Delta$, such that all its nodes and edges are initialized to p-*X*. The algorithmic framework must then guarantee that no such path $\boldsymbol{Q}$ is identified as sensitizable.

In timing analysis, and besides the common blocking conditions, propagation of a perturbation to a node $x$ becomes blocked if $DThru(x) < \Delta$, meaning that a perturbation cannot reach a primary output with propagation delay no less than $\Delta$ if it propagates through $x$. This condition defines $B_C(x)$; whereas condition $DThru(x, y) < \Delta$ defines $B_C(x, y)$ for blocking propagation to each edge $(x, y)$. Consequently,

$$B(x) = \left[\prod_{y \in I(x)} (\pi(y, x) = 0)\right] + \left[\prod_{y \in O(x)} (\pi(x, y) = 0)\right] + \left[\sum_{y \in I(x)} ((\pi(y, x) = 0) \cdot c(y, x))\right] + [DThru(x) < \Delta] \tag{6.7}$$

and,

$$B(x, y) = [\pi(x) = 0] + [\pi(y) = 0] + [DThru(x, y) < \Delta] \tag{6.8}$$

The conditions for propagating a perturbation to a node and edge are similar to the test pattern generation case, and (5.3) and (5.4) (see page 180) can be adapted:

$$P(x) = [x \in PI \wedge \nu(x) \neq X] + \prod_{y \in I(x)} [(\pi(y, x) = 1) \cdot (\nu(y) \neq X) + (\pi(y, x) = 0) \cdot nc(y, x)] \tag{6.9}$$

and,

$$P(x, y) = [\pi(x) = 1] \tag{6.10}$$

Thus, propagation to a p-*X* primary input occurs when its logic value is assigned (i.e. a signal transition is defined). Edge propagation is the same as for test pattern generation.

Observe that we may have a gate output *z* set to p-*T*, that drives a primary output, but such that the primary output is p-*F*. This situation can happen when the gate output drives other nodes, some of which are or can become p-*T*. By specifically considering OUT nodes, a primary output that becomes p-*T* indicates in fact the existence of a floating-mode sensitizable path with delay no less than $\Delta$.

**Example 6.3.** For the example circuit of Figure 6.3, let us consider the logic assignments $x_1 = 0$ and $x_2 = 1$. Immediately, $DFrom(x_1) = -\infty$, $DThru(x_1) = -\infty$ and $DThru(x_1, y_1) = -\infty$. Hence, $\pi(x_1) \leftarrow 0$, $\pi(x_1, y_1) \leftarrow 0$, and $\pi(x_2, y_1) \leftarrow 1$. Since no blocking condition applies to $y_1$, we have $\pi(y_1) \leftarrow 1$. Finally, no blocking conditions can be derived for $(y_1, y_2)$ and so $\pi(y_1, y_2) \leftarrow 1$. The new assignments are shown in Figure 6.4. Note that the final p-status of $x_1$ could also be p-*T*, if the

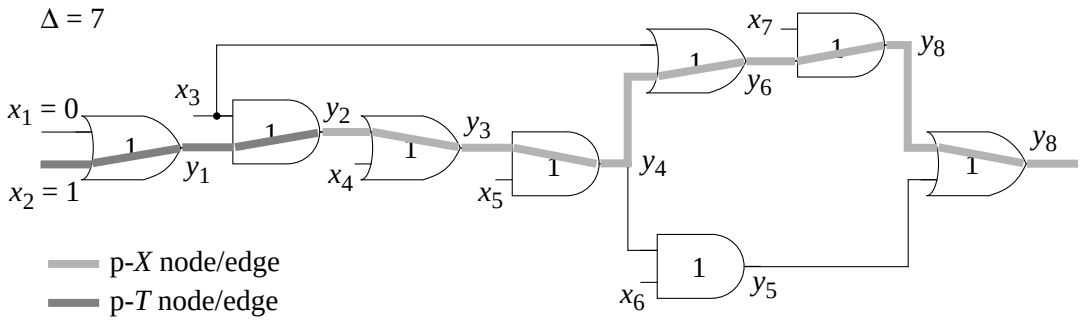



**Figure 6.4:** Updating p-status for timing analysis

p-status was updated before updating delay information. However, such change to the p-status of $x_1$ is not relevant for the goal of path sensitization, and would eventually be toggled to p-*F* provided some other path was proved to be sensitizable. The order of implications in this example assumes the implementation of the deduction engine described in Chapter IV. □

Correctness of the p-propagation model for path sensitization in timing analysis is guaranteed by the following:

**Theorem 6.1.** A combinational circuit contains a floating-mode sensitizable path of delay no less than Δ, for a test ***T***, if and only if under the p-propagation model such test ***T*** sets a primary output to p-*T*.

As in the case of test generation, an immediate corollary follows:

**Corollary 6.1.** A sound and complete search algorithm, based on the p-propagation model, computes a test ***T*** that sensitizes a path with delay no less than Δ if and only if such test exists.

**Perspective**

Other models for path sensitization based on differently formulated delay estimates are described in [29, 30, 50, 52]. While in these models only delay information is involved, in the p-propagation model additional blocking conditions based on structural and functional relations are defined, which permit, in the case of search-based algorithms, pruning the amount of search with delay-independent information. As described in Chapter IV, the implementation of some of the pruning methods becomes greatly simplified if conflicts can be associated with structural and

functional blocking conditions. We further note that relying on structural and functional blocking conditions can be particularly useful for the timing analysis of practical circuits, since circuits are designed subject to structure and function and not necessarily signal delay interactions.

## 6.4 Basic Deduction and Diagnosis Engines

In contrast with test pattern generation, the implementation of a timing analysis tool based on the p-propagation model requires defining `Target_Application_Update`(), with the purpose of updating delay estimates after each implication sequence in the logical dimension is derived. After each decision assignment causes an implication sequence in the logical dimension, propagation delay estimates can be updated. This is done by traversing (on a per need basis) nodes / edges whose propagation delay estimates change. First a forward levelized traversal on these nodes updates the *DTo* propagation delay estimates. Afterwards, a backward levelized traversal updates the *DFrom* and *DThru* estimates.

Observe that for p-*X* nodes and edges, changes to the propagation delay estimates *are necessarily* the result of some other p-*X* nodes / edges becoming p-*F*, since delay estimates of p-*X* nodes are defined from propagation delay estimates of other p-*X* nodes. Furthermore, delay estimates for p-*F* nodes need not be updated. First, because these delay estimates do not contribute for the delay estimates of p-*X* nodes and edges. Second, because updating such delay estimates would just increase the computational overhead. Consequently, whenever a node or edge is downgraded to p-*F* all its delay estimates are set to $-\infty$. Since the delay estimates of p-*F* nodes are fixed at $-\infty$, and only required delay estimates are otherwise updated, the computational overhead of updating delay is reduced, in explicit contrast with other algorithms for timing analysis, which base all deductive reasoning on delay estimates [29, 30, 50, 52].

### 6.4.1 Deduction Engine

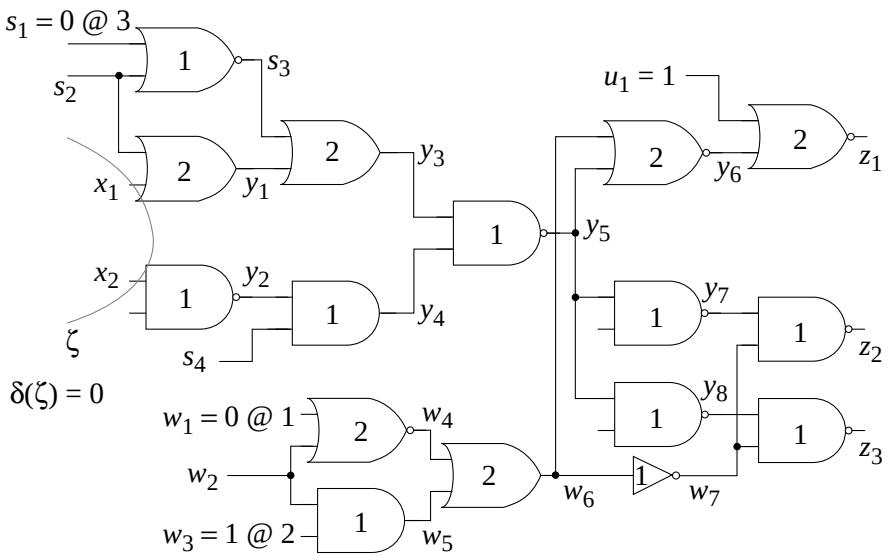
The deduction engine described in Figure 4.14 (see page 158) is used for deriving implications in timing analysis. We only need to define how the antecedent assignments of propagation assignments are defined.

For blocking propagation to a node, the antecedent assignment definitions of Section 5.4.1

(see page 183) are applied with the exception of error cancellation, which is not applicable to timing analysis. On the other hand, we need to define the antecedent assignments of nodes and edges that are downgraded to p-*F* due to delay conditions.

Some facts are useful for identifying the direct causes of delay changes. First, as mentioned above, delay changes are a direct consequence of downgraded nodes and edges, which may not be connected to the nodes and edges upon which they cause changes to the delay estimates. Second, delay changes may require several updates before a node/edge is finally downgraded to p-*F*. The identification of the specific set of downgraded nodes and edges that cause blocking due to delay conditions is complicated by all the above facts. Our solution is to maintain global information of the nodes and edges that are downgraded and consequently cause changes to the delay estimates of other nodes. Each time a node or edge is downgraded and causes a change to the delay estimates of any other node or edge, such node or edge is added to a set $U$ of nodes directly affecting the delay estimates. Any node or edge that becomes downgraded due to delay conditions assumes set $U$ as its antecedent assignment. (Note that this definition of antecedent assignment for blocking due to delay estimates is an intermediate step to simplifying antecedent manipulation, as was proposed in Chapter V (see page 191) for test pattern generation, but for timing analysis the coupling introduced by delay estimates can only be efficiently handled by assuming a global definition of the antecedent assignment.)

**Example 6.4.** The effect of delay conditions on the node and edge p-status is illustrated with the example circuit of Figure 6.5. The specified threshold delay is $\Delta = 17$, and the current decision assignment implies $u_1 \leftarrow 1$. As a result, $\pi(y_6, z_1) = 0$, and this then implies $\pi(y_6) \leftarrow 0$ and $\pi(y_5, y_6) \leftarrow 0$. Symbol $\pi(y_6)$ is added to set $U$; because the p-status of edge $(y_5, y_6)$ is downgraded, the delay estimates of the nodes in the transitive fanin of $y_5$ are modified as shown in Figure 6.5-b. The resulting delay estimates cause $y_2$, $y_4$, $(y_2, y_4)$ and $(y_4, y_5)$ to be downgraded to p-*F*. The antecedent assignment of $\pi(y_4, y_5)$ is defined by set $U$ (thus including $\pi(y_6)$), whereas structural constraints define the antecedent assignments for the remaining nodes and edges. The example illustrates how delay-based blocking conditions may cause the antecedent assignment of nodes or edges to include unconnected nodes and edges. □




- $\Delta = 17$
- $DTo(x_1) = DTo(x_2) = 10$; $x_1$ and $x_2$ are p-$T$ and define a p-cut $\zeta$
- $y_1$ through $y_8$, $z_1$, $z_2$ and $z_3$ are the only p-$X$ nodes in the circuit
- Current decision assignment implies $u_1 \leftarrow 1$

(a) Example circuit

| | Initial estimates | | | | | Final estimates | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $y_1$ | $y_2$ | $y_3$ | $y_4$ | $y_5$ | $y_1$ | $y_2$ | $y_3$ | $y_4$ | $y_5$ |
| *DTo* | 12 | 11 | 14 | 12 | 15 | 12 | 11 | 14 | 12 | 15 |
| *DFrom* | 7 | 6 | 5 | 5 | 4 | 5 | 4 | 3 | 3 | 2 |
| *DThru* | 19 | 17 | 19 | 17 | 19 | 17 | 15 | 17 | 15 | 17 |

(b) Delay estimates after assigning $u_1$

**Figure 6.5:** Example of delay-based blocking conditions

For a node that is upgraded, the antecedent assignment if given by (5.9) on page 184 if the node is not a primary input. Otherwise, and due to (4.27) on page 164, the antecedent assignment of a p-$T$ primary input $x$ is given by $\{ (x, \nu(x)) \}$. For upgraded edges, the antecedent assignment is given by the assignment to p-$T$ of the fanin node.

Having defined the antecedent assignments for all types of implications, the deduction engine of Figure 4.14 can now be applied to timing analysis.

**Reverse Value Probing**

Value probing, as described in Chapter V, can be extended to backward analysis of admissible pairs of values. This technique can be of interest for timing analysis since the logic values of USPs may help identifying the logic values of USPs in their transitive fanin; for test pattern generation this situation can never occur. Reverse value probing operates in the same way value probing does. However, the definition of admissible pairs of values is done backwards, after reaching each USP $x$ with a specified logic value, and with respect to USP $y$ in the transitive fanin of $x$. The antecedent assignments of implied assignments due to reverse value probing are defined the same way as for (direct) value probing.

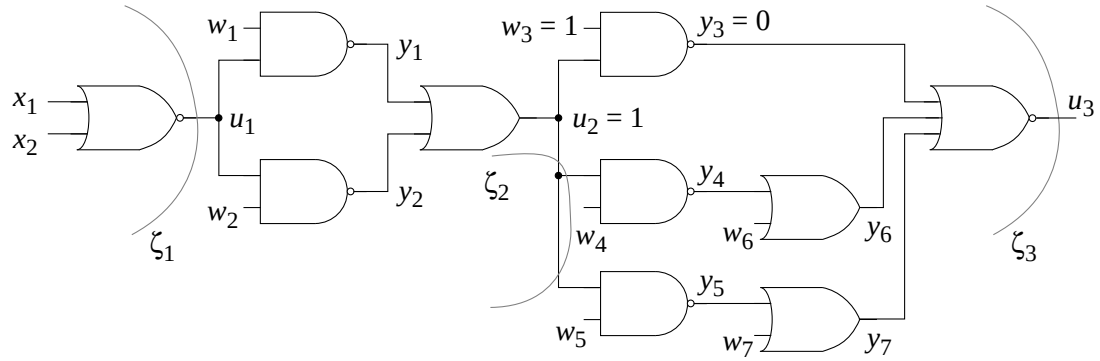
**Example 6.5.** An example of applying reverse value probing is shown in Figure 6.6-a. The derivation of values is due to the assignment $u_2 = 1$, which is implied because $u_3$ is a USP and $y_3 \leftarrow 0$. Propagation of values from $u_1$ to $u_2$ can only occur if $u_1$ assumes a value that propagates to $u_2$ and is compatible with the value of $u_2$. Hence, $u_1 \leftarrow 0$ is implied with antecedent assignment { ($\zeta_2$, 1), ($u_2$, 1) }. Furthermore, if a perturbation in $u_2$ propagates to $u_3$, then the value of $u_3$ must be 1, and thus $u_3 \leftarrow 1$ is implied. The resulting assignments are shown in Figure 6.6-b. Note that p-cut $\zeta_3$ becomes fanin justified and $\zeta_4$ (that results from $\zeta_2$) becomes fanout justified. □

### 6.4.2 Diagnosis Engine

The basic diagnosis engine is described in Figure 4.21 (see page 159), and can readily be applied to timing analysis given the above definition of antecedent assignments. Note, however, that propagation implicates involving nodes or edges downgraded due to delay estimate conditions can become significantly large.

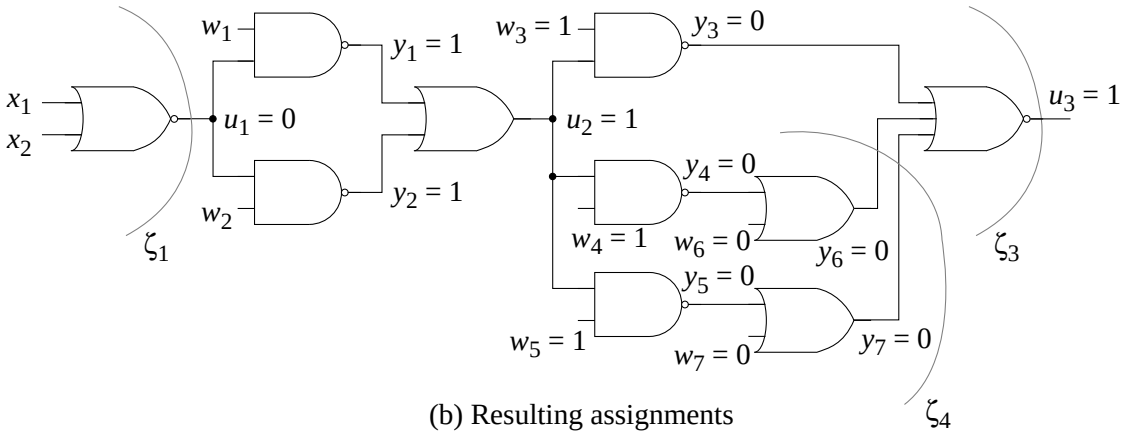
As with the test pattern generation case, improvements to the basic diagnosis engine are possible. For example, one can implement UIPs, multiple conflicts and iterated conflicts. In addition, subleveling (described in Section 5.4.2 (see page 187)) can be applied whenever several USPs are identified.

Implicates derived with conflict diagnosis in timing analysis can also be defined as pervasive under some conditions. Logical implicates are defined as pervasive across all circuit analysis applications that use the circuit's consistency function. Propagation implicates derived with block-




- $\pi(u_2, y_3) = 0$ due to delay constraints
- $u_2$ is USP and implies $y_3 \leftarrow 0$, which implies $u_2 \leftarrow 1$and $w_3 \leftarrow 1$

(a) Example circuit

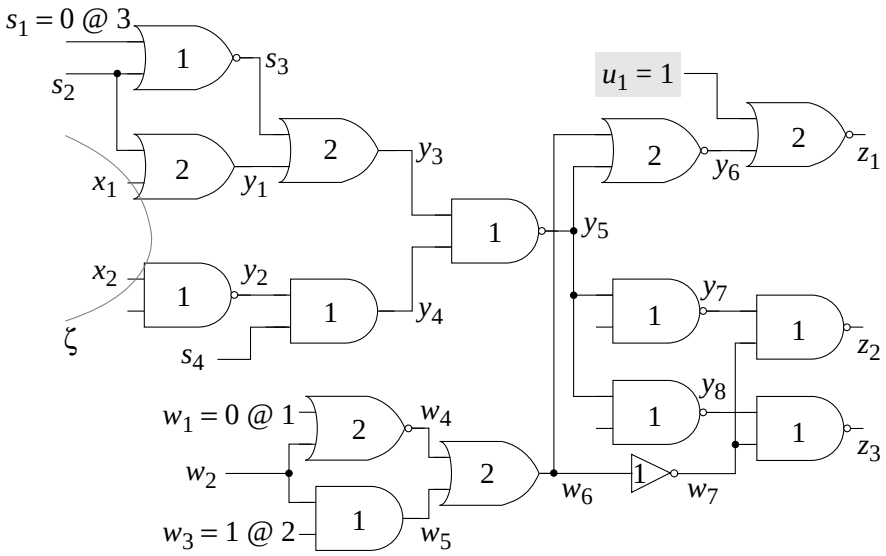



(b) Resulting assignments

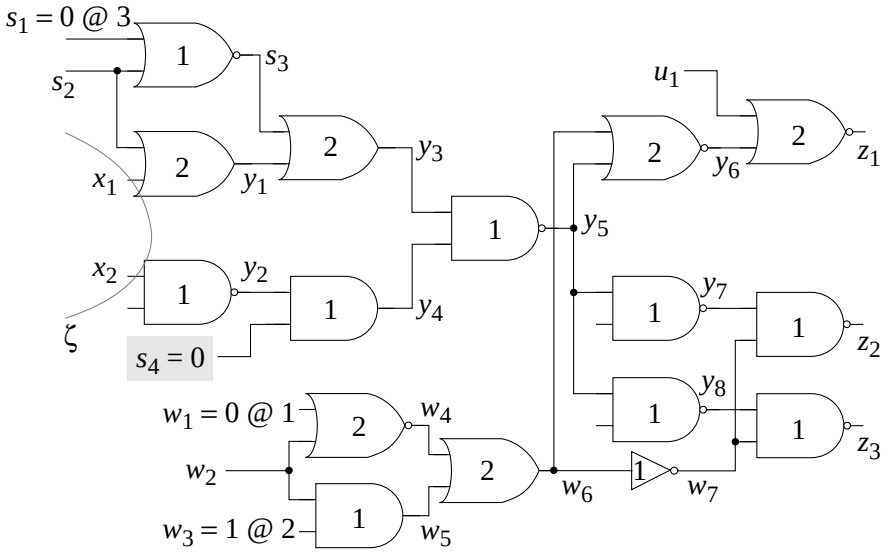
**Figure 6.6:** Reverse value probing

ing conditions common to both path sensitization applications are defined as pervasive across path sensitization applications. Finally, propagation implicates derived from delay estimate blocking conditions are defined as *non-pervasive*. Delay conditions are a function of the threshold delay associated with each instance of the path sensitization problem. While computing the longest sensitizable path, the delay threshold changes at every iteration; hence the propagation implicates based on delay estimates cannot be applied in different iterations and must be declared non-pervasive. Note, however, that these implicates can still be used within the current instance of the path sensitization problem.

**Example 6.6.** The derivation of propagation implicates for timing analysis is illustrated with the




(a) Non-pervasive implicate




(b) Pervasive implicate

**Figure 6.7:** Derivation of propagation implicates in timing analysis

example circuit of Figure 6.7 (that is the same as in Figure 6.5). For Figure 6.7-a, let us assume the assignment $u_1 = 1$. As a result (from Example 6.4), $y_1$ becomes a USP and the assignment $s_2 = 0$ results. This in turn implies $s_3 = 1$, which blocks propagation to $y_3$ and a propagation conflict is identified. Diagnosis of the conflict yields the p-clause,

$$\omega_{\pi,1} = \langle \{x_1, x_2\}, (\neg u_1 + s_1 + \pi(u_1, z_1) + \pi(s_2, y_1)) \rangle \quad (6.11)$$

which is declared as non-pervasive, since propagation from $x_2$ to $z_2$ or $z_3$ is blocked due to delay conditions. Hence, this implicate is only valid for the current path sensitization problem and for the specified threshold delay. If the threshold delay is decreased, the above condition is not necessarily a propagation implicate.

For the example of Figure 6.7-b, let us assume that the assignment $s_4 = 0$ replaces the assignment $u_1 = 1$ of the previous case. Once again a propagation conflict is detected, since $s_4$ blocks propagation to $y_4$, which is assigned value 0 and consequently blocks propagation to $y_5$. Conflict diagnosis yields the p-clause,

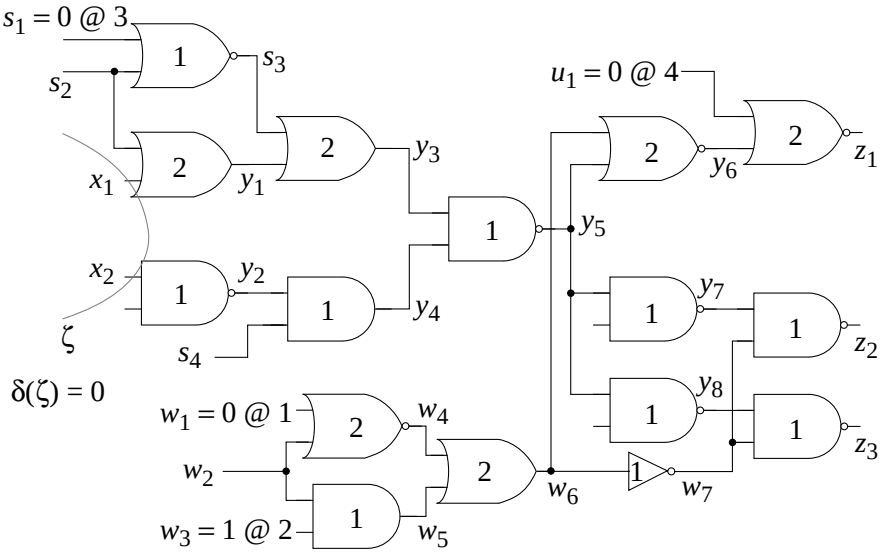
$$\omega_{\pi,2} = \langle \{x_1, x_2\}, (s_4 + \pi(s_4, y_4)) \rangle \quad (6.12)$$

which immediately requires $s_4 = 1$ whenever $\pi(s_4, y_4)$ is p-*F* and $\{ x_1, x_2 \}$ is a p-cut. This last propagation implicate is pervasive across path sensitization applications, since only general blocking conditions are involved in its derivation. □

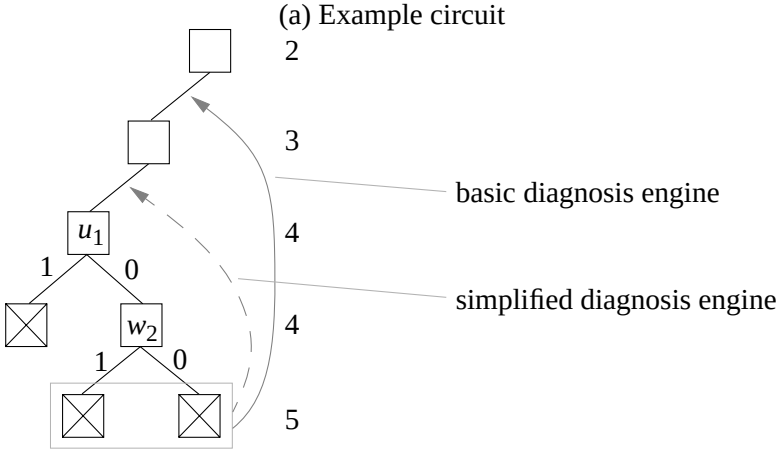
### 6.4.3 Simplifications to the Diagnosis Engine

As with test pattern generation, we can simplify the diagnosis engine by not creating implicates and by maintaining potentially large antecedent assignments implicitly defined. The proposed engine follows the procedures described in Section 3.6.3 and Section 5.4.3, in that level conflicting assignment sets are defined and updated each time a conflict is identified. As in Section 5.4.3, the antecedent assignments of USPs are maintained in a set of dependencies *U*, which records all downgraded nodes and edges. (Note that this is the same set that is used for recording dependencies for delay-based blocking conditions, but now no propagation implicates are created. Furthermore, the manipulation of set *U* can be optimized by being ordered by decision levels. Each level conflicting assignment set is then directly related to a corresponding partition of set *U*.)

**Example 6.7.** Simplifications to conflict diagnosis trade off pruning precision for the guarantee of




(a) Example circuit




(b) Backtracking decision levels

**Figure 6.8:** Pruning precision tradeoffs

fixed size clauses databases. The example circuit of Figure 6.8 illustrates how pruning precision may decrease when simplified conflict diagnosis is considered. Let us assume that the current decision level is 5, and that at decision level 4, $u_1 = 1$ led to a conflict whose conflicting assignment set is specified by (6.11), thus yielding the assertion $u_1 = 0$. Consequently, the level conflicting assignment set $\boldsymbol{A}_{CS}[3]$ now contains entry ($s_1$, 0), whereas the remaining dependencies are assumed at decision level 0 and update $\boldsymbol{A}_{CS}[0]$. At decision level 5, let the first assignment be $w_2 = 1$, which implies $w_6 \leftarrow 1$, and consequently a propagation conflict is identified. Conflict diagnosis causes level conflicting assignment set $\boldsymbol{A}_{CS}[2]$ to be updated with ($w_3$, 1), and the other p-status dependencies to be added to $\boldsymbol{A}_{CS}[0]$. As a result, $w_2$ is asserted to 0 at decision level 5. The

resulting implication sequence also yields a propagation conflict, whose diagnosis causes $\boldsymbol{A}_{CS}[1]$ to be updated with $(w_1, 0)$, and other p-status dependencies to be added to $\boldsymbol{A}_{CS}[0]$. Since $w_2$ is asserted and yielded a conflict, it is necessary to backtrack, and the computed backtracking decision level (using (3.21) on page 99) is 3 due to $s_1$ in $\boldsymbol{A}_{CS}[3]$. If propagation implicates were explicitly created the backtracking decision level would be 2 due to p-clause,

$$\omega_\pi = \langle \{x_1, x_2\}, (\neg w_3 + w_1 + \pi(w_6, y_6) + \pi(w_7, z_2) + \pi(w_7, z_3)) \rangle$$

which for this example would be pervasive across path sensitization applications. □

Completeness of simplified conflict diagnosis is guaranteed because all assignments that contribute to conflicts are recorded in the level conflicting assignment sets. Hence, whenever a conflict is identified, the union of the level conflicting assignment sets defines an implicate of the propagation consistency function. Furthermore, simplified conflict diagnosis yields backtracking decision levels that are always no less than the backtracking decision level computed with the basic diagnosis engine.

For simplified conflict diagnosis, and as the above example suggests, dependencies on the p-status that result from the initialization phase (i.e. at decision level 0) need not update $\boldsymbol{A}_{CS}[0]$, since no p-clauses are to be created and these dependencies are constant throughout the search process. This fact allows further simplifying the manipulation of dependencies associated with p-status assignments for simplified conflict diagnosis.

Diagnosis engines with worst-case polynomial size growth of the clause databases can also be implemented. For a maximum propagation implicate size of *m*, the implementation can decide whether to give preference to large p-cuts or to large conditional clauses. Furthermore, given that propagation implicates due to delay-based blocking conditions are non-pervasive, we may not add them to the p-clause database, and given preference to pervasive implicates.

## 6.5 Other Engines of LEAP

In this section other engines that are required to implement timing analysis are described. Note that postprocessing engines are not relevant for timing analysis; redundancy removal is of

reduced interest for circuit delay computation, and so is solution caching, since circuit delay computation is to be executed only once for a given circuit.

Advanced deduction engines can be implemented for path sensitization in timing analysis, and follow the implementation proposed in Section 3.5.3 and Section 6.5 for SAT and for test pattern generation, respectively. However, implication sequences require updating the delay estimates for each node and edge. Consensus of p-clauses is defined based on the *join*() operation and is given by (5.13) (see page 193). For large p-clauses, derived from conflicts involving delay-based blocking conditions, the application of advanced implications engines may produce implicates of a reasonably large size that are not pervasive and that will hardly contribute to pruning the search. Consequently, advanced deduction engines can be useful for identifying logical implicates and propagation implicates in circuits where delay-based blocking seldom occurs.

Preprocessing engines can also be implemented. If applied to the logical clause database, then a more complete database is used for circuit delay computation. If preprocessing is invoked after the path sensitization goal is specified, then an advanced deduction engine can be used for identifying propagation implicates.

The decision making procedures described in Section 3.9 and in Chapter V can be straightforwardly adapted to timing analysis. As in test pattern generation, the existence of p-$X$ nodes constrains the definition of head lines and don't care nodes, and constrains how backtracing-based procedures can trace objectives. We have implemented simple and multiple backtracing procedures based on topological controllability and observability relations, and the results are given in Chapter VII.

## 6.6 Comparison with other Timing Analysis Algorithms

The proposed path sensitization algorithm for timing analysis is unique in that it implements conflict diagnosis. Existing timing analysis tools do not implement any form of conflict diagnosis [5, 29, 30, 50, 52, 119, 120]. In addition, we allow for structural properties of the circuit to be identified and used to prune the search, whereas other search-based approaches are exclusively based on delay considerations [29, 30, 50, 52]. For SAT-based formulations of timing analysis, there can be worst-case exponential size problem instance representations [117, 119]. In

contrast, the p-propagation model guarantees representations linear in the size of the combinational circuit.

Constructive approaches, which compute all logical conditions for a number of (potentially all) path delays, perform reasonably well in certain forms of regular circuits (e.g. carry-skip adders), but perform particularly poorly in random circuits such as the ISCAS'85 benchmark circuits [7]. The advantage of these approaches is that all the path sensitization information is computed in one step; the problem being that the size of the representation may be unacceptably large.

The potential advantages of the proposed algorithm can be characterized as follows:

- Framework for implementing conflict diagnosis, where different pruning methods can be integrated and several forms of implicates, some of which pervasive across several applications, can be identified.
- Specific consideration of the structure of the path sensitization problem, in particular USPs, which permit identifying more logical implications and consequently further prune the search.
- Configurable path sensitization algorithm, where different degrees of deduction and diagnosis ability can be implemented and applied to timing analysis.
- Linear size representation of the path sensitization problem, which makes the representation of the problem independent of the distribution of delays in a given combinational circuit.

## 6.7 Summary

In this chapter the path sensitization algorithm for timing analysis is detailed. It mostly follows the algorithm delineated in Section 4.5, with suitable modifications to take into consideration the formulation of the p-propagation model for timing analysis. Simplifications to conflict diagnosis were described, which guarantee constant size clauses databases, and a reduction in the overhead for manipulating large antecedent assignments.

Other engines, required for implementing the timing analysis tool were also described. Finally, we compared the proposed path sensitization algorithm with algorithms proposed by other authors. The main difference resides in the ability of the proposed algorithm to exploit the structure of the problem and to diagnose the causes of conflicts.

# CHAPTER VII

# EXPERIMENTAL RESULTS

The algorithms proposed in the past chapters constitute the main components of the GRASP+LEAP toolset for the analysis of combinational circuits, proposed in Chapter I and shown in Figure 7.1. In this chapter we describe the implementation details of some of those algorithms, and study experimental results obtained with each tool.

## 7.1 Tool Implementation

At the time of this writing, the following tools have been implemented:

1. A test-pattern generation tool, TG-LEAP, first described in [155].
2. A timing analysis tool, TA-LEAP, first described in [156].
3. An experimental SAT algorithm, associated with the kernel of GRASP, that can be interfaced with a front-end for solving SAT problems on CNF formulas. This algorithm is based on the simplified diagnosis engine. Results of a preliminary implementation were described in [150, 152] where the generated instances of SAT were related to circuit delay computation.

**Implementation of TG-LEAP**

The test pattern generation tool (TG-LEAP) follows the implementation described in Chapter V but with the following configuration:

- Use of a simplified diagnosis engine, that is based on level conflicting assignment sets and records blocked nodes and edges in a dedicated set $U$. This engine, as described in Chapter V, guarantees a constant size representation of the path sensitization problem and avoids the

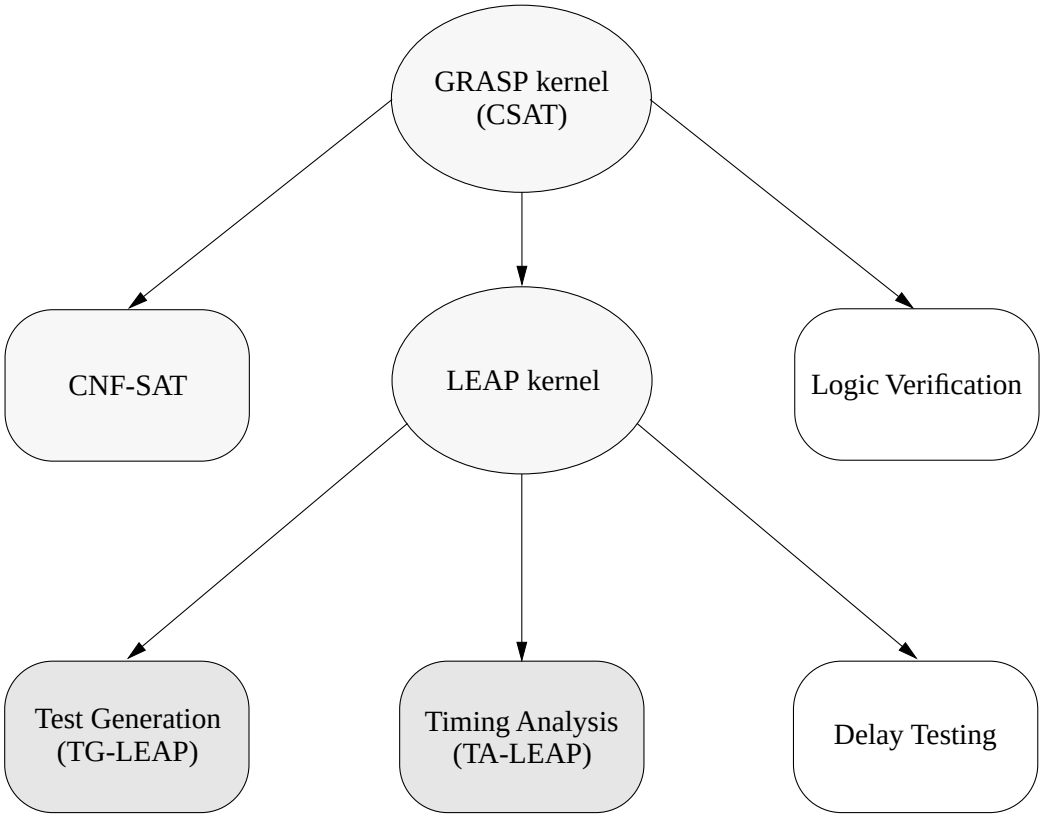



**Figure 7.1:** The GRASP+LEAP toolset

explicit manipulation of propagation cuts and multiple p-frontiers. The diagnosis engine creates assertions based on unique implication points.

- Implementation of the basic deduction engine. Logical implications and propagation implications are identified directly on the circuit structure, and the logical clause database needs not be constructed. Furthermore, manipulation of propagation cuts is simplified due to using the simplified conflict diagnosis engine. Unique sensitization points (USPs) are identified and used to derive unique sensitization implications (USIs) in the logical dimension.
- Preprocessing restricted to the logical dimension and restricted to `Preprocess_1()`, i.e. implicates of size two are derived (in the absence of other assigned nodes). The implementation of `Preprocess_1()` adds implicates to the clause database as they are identified. (Hence the final set of identified implicates depends on the order in which assignments are tested.)
- No postprocessing engine has been implemented. In a practical tool, the postprocessing

engine can be useful for reducing the test set size, and for simplifying the search for some instances of path sensitization.

- The decision making engine can implement simple and multiple backtracing based on topological controllability and observability measures.

**Implementation of TA-LEAP**

The timing analysis tool (TA-LEAP) implements the circuit delay computation procedure described in Chapter VI but configured as follows:

- The delay iteration procedure iteratively decrements threshold delays starting from the longest topological path (LTP).
- Use of a simplified conflict diagnosis engine equivalent to the one of TG-LEAP.
- Implementation of the basic deduction engine, but with the simplifications to the deduction engine described for TG-LEAP.
- Preprocessing solely based on `Preprocess_1()`.
- The same decision making procedure of TG-LEAP is used, i.e. simple and multiple backtracing can be used.

## 7.2 Results

The tools described above have been implemented in the C++ programming language, and all the results were obtained on a DEC 5000/240 workstation with 32 MByte of RAM using the ATT C++ compiler (version 3.0.1). The ISCAS'85 benchmark circuits [17] are used for all test pattern generation results. For timing analysis, the same circuits are used, but other circuits proposed in [50, 52] are also tested.

### 7.2.1 Results for Test Pattern Generation

Some statistics of the ISCAS'85 benchmark circuits are shown in Table 7.1. Of interest are the total number of faults for each circuit, as well as the number of redundant faults. The set of faults considered corresponds to the faults specified in the original distribution of the ISCAS’85 benchmark circuits [17]. The number of implicates identified with `Preprocess_1()` for each benchmark circuit is also shown. These implicates are commonly referred to as non-local (or glo-

| Circuit | Gates | PIs | POs | Faults | Redundant faults | Pre-processing time (in sec) | Implicates of size 2 |
|---|---|---|---|---|---|---|---|
| C432 | 160 | 36 | 7 | 524 | 4 | 0.082 | 138 |
| C499 | 202 | 41 | 32 | 758 | 8 | 0.176 | 40 |
| C880 | 383 | 60 | 26 | 942 | 0 | 0.207 | 116 |
| C1355 | 546 | 41 | 32 | 1574 | 8 | 0.902 | 208 |
| C1908 | 880 | 33 | 25 | 1879 | 9 | 1.644 | 1310 |
| C2670 | 1193 | 233 | 140 | 2747 | 117 | 2.421 | 1951 |
| C3540 | 1669 | 50 | 22 | 3428 | 137 | 14.600 | 6906 |
| C5315 | 2307 | 178 | 123 | 5350 | 59 | 4.137 | 3609 |
| C6288 | 2406 | 32 | 32 | 7744 | 34 | 0.832 | 830 |
| C7552 | 3512 | 207 | 108 | 7550 | 131 | 12.060 | 10139 |

**Table 7.1:** Statistics for the ISCAS'85 benchmark circuits

bal) implications [106, 144].

In the following, the results shown correspond to detecting every specified fault for each circuit. The purpose of the experiments is to evaluate the path sensitization algorithm, and hence we are interested in the largest number of faults. In a practical test pattern generation tool, fault simulation would be employed to detect some faults and reduce the test set size.

#### 7.2.1.1 Benchmarking Run Time Options

The test pattern generation tool can be configured to implement several algorithms for test pattern generation. In the following tests, ten different algorithms were tested as described in Table 7.2, which correspond to different combinations of the following options:

1. How to manipulate head lines. Head lines may either not be computed (option N), computed statically (option S), or computed dynamically (option D).
2. How to preprocess the circuit structure (to identify implicates of size 2, also referred to as non-local implications). A circuit may either not be (option N) or be (option Y) preprocessed.
3. The computation of unique sensitization points (USPs). USPs may either not be computed (option N), computed only when the size of p-frontier is 1 (option Y1, solely used to emulate

| Algorithms | $A_0$[a] | $A_1$ | $A_2$ | $A_3$ | $A_4$[b] | $A_5$ | $A_6$[c] | $A_7$ | $A_8$ | $A_9$[d] |
|---|---|---|---|---|---|---|---|---|---|---|
| Head lines [N/S/D] | N | S | D | D | S | S | S | S | S | S |
| Preprocessing [N/Y] | N | N | N | Y | N | Y | Y | Y | Y | Y |
| USPs [N/Y1/Y] | N | N | N | N | Y1 | Y1 | Y | Y | Y | Y |
| Backtracking [C/N] | C | C | C | C | C | C | C | N | C | N |
| Assertions [N/Y] | N | N | N | N | N | N | N | N | Y | Y |

**Table 7.2:** Combinations of options tested

a. PODEM*
b. FAN*
c. SOCRATES*
d. TG-LEAP

FAN), or be dynamically computed (option Y).

4. The backtracking option. Backtracking can either be chronological (option C) or non-chronological (option N).
5. The failure-driven assertions option. Assertions can either identified (option Y) or not be identified (option N). With option Y, unique implication points are computed.

Decision making procedures rely on either simple or multiple backtracing. Because our main goal is to compare the pruning ability of each configuration of the algorithm, only structural controllability/observability measures are used [1]. The tested *backtracing* schemes were the following:

1. Simple backtracing, starting by trying to satisfy the most difficult controllability problems and afterwards trying to satisfy the most simple observability problems.
2. Multiple backtracing, as proposed in [62], but using structural controllability/observability measures.

It is important to note that in TG-LEAP backtracing is *always* performed to a head line in opposition to the backtracing schemes used in FAN and SOCRATES, where backtracing can stop at fanout points [62, 144]. Our goal is to guarantee that decision assignments are restricted to head lines, even though this may increase the size of the decision tree is some cases.

Some of the algorithms shown in Table 7.2 can be viewed as customized implementations of well-known test pattern generation algorithms. In particular, $A_0$ is a modified implementation of

PODEM, which we refer to as PODEM*, $A_4$ corresponds to FAN, referred to as FAN*, and $A_6$ corresponds to SOCRATES, referred to as SOCRATES*. Finally, $A_9$ denotes the actual implementation of TG-LEAP.

The customized implementations of PODEM [72], FAN [62] and SOCRATES [144] have some differences with respect to the original algorithmic descriptions. PODEM* can perform both forward and backward implications, and thus must maintain a j-frontier. FAN* identifies unique sensitization points whenever the size of the (only) p-frontier is one, using the algorithm described in Section 4.5.2.2 (see page 160). SOCRATES* implements the concepts described in [144] and also computes dynamic unique sensitization points, but using the algorithm proposed in this dissertation. Hence, SOCRATES* corresponds to a more efficient implementation of the pruning methods described in [144] and [145] until phase DYN_1, but without the implementation of instruction 2 of the unique sensitization procedure[1]. Moreover, the results given for SOCRATES* are based on the preprocessing with `Preprocess_1()`.

The results shown below assume multiple backtracing for all configurations and that the backtracking limit is 500. Table 7.3 contains the total CPU times for each algorithmic configuration and for each benchmark circuit. The total number of aborted faults for each benchmark circuit is given in Table 7.4. The total number of backtracks and of decisions are given in Table 7.5 and in Table 7.6, respectively.

From the number of aborted faults for each algorithm, we can conclude that the identification of unique sensitization points (USPs), failure-driven assertions (FDAs) and non-chronological backtracking (CDB) are of key significance for pruning the search in test pattern generation. Note that by applying USPs and then by applying FDAs, the number of aborted faults decreases sharply. It is worth noting that preprocessing may not always perform well. Algorithm $A_3$ uses preprocessing with respect to $A_2$ and performs worse. This fact is justified due to non-local implications creating larger j-frontiers, which may cause multiple backtracing to make incorrect decision assignments. Without pruning methods that can handle these incorrect decisions, larger j-frontiers may in some cases lead to increased backtracking.

---

[1] Phase DYN_2 [145] corresponds to dynamic learning, which in our framework can be modeled with `Deduce_1()`, whereas instruction 2 attempts to identify logic assignments which, if do not hold, block propagation [144].

| Circuit | $A_0$ | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ | $A_6$ | $A_7$ | $A_8$ | $A_9$ |
|---|---|---|---|---|---|---|---|---|---|---|
| C432 | 30.63 | 26.94 | 27.86 | 28.36 | 29.80 | 22.14 | 21.57 | 25.72 | 13.39 | 13.55 |
| C499 | 118.00 | 102.60 | 110.60 | 108.70 | 71.26 | 68.77 | 70.13 | 71.86 | 70.86 | 70.55 |
| C880 | 23.24 | 27.92 | 30.67 | 30.58 | 27.68 | 27.00 | 27.42 | 27.16 | 27.44 | 27.49 |
| C1355 | 402.00 | 397.70 | 414.90 | 411.50 | 365.20 | 357.20 | 362.50 | 360.40 | 360.60 | 360.50 |
| C1908 | 306.10 | 286.80 | 304.80 | 265.60 | 208.50 | 209.20 | 174.10 | 173.90 | 173.60 | 174.1 |
| C2670 | 625.60 | 616.10 | 763.20 | 966.7 | 508.40 | 515.90 | 526.00 | 584.70 | 359.20 | 358.10 |
| C3540 | 738.80 | 676.80 | 541.00 | 641.9 | 430.60 | 392.50 | 400.40 | 401.70 | 390.40 | 389.80 |
| C5315 | 541.70 | 696.70 | 798.20 | 799.30 | 645.00 | 647.30 | 667.60 | 649.60 | 643.00 | 642.30 |
| C6288 | 3446.00 | 3443.00 | 3694.00 | 5003.00 | 3405.00 | 4912.00 | 4617.00 | 3760.00 | 3278.00 | 3275.00 |
| C7552 | 2784.00 | 2852.00 | 3223.00 | 3250.00 | 2416.00 | 2504.00 | 2455.00 | 2609.00 | 2056.00 | 2051.00 |

**Table 7.3:** CPU times for test pattern generation

| Circuit | $A_0$ | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ | $A_6$ | $A_8$ | $A_8$ | $A_9$ |
|---|---|---|---|---|---|---|---|---|---|---|
| C432 | 3 | 3 | 4 | 4 | 3 | 2 | 2 | 2 | 0 | 0 |
| C499 | 8 | 8 | 8 | 8 | 0 | 0 | 0 | 0 | 0 | 0 |
| C880 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| C1355 | 8 | 8 | 8 | 8 | 0 | 0 | 0 | 0 | 0 | 0 |
| C1908 | 12 | 14 | 13 | 11 | 6 | 2 | 0 | 0 | 0 | 0 |
| C2670 | 49 | 51 | 51 | 64 | 33 | 26 | 24 | 12 | 0 | 0 |
| C3540 | 33 | 33 | 30 | 25 | 12 | 0 | 0 | 0 | 0 | 0 |
| C5315 | 7 | 7 | 6 | 6 | 0 | 0 | 0 | 0 | 0 | 0 |
| C6288 | 7 | 7 | 7 | 16 | 2 | 15 | 15 | 4 | 0 | 0 |
| C7552 | 119 | 188 | 187 | 184 | 124 | 120 | 105 | 95 | 2 | 0 |
| Total | 246 | 319 | 314 | 326 | 180 | 165 | 146 | 113 | 2 | 0 |

**Table 7.4:** Number of aborted faults

Finally, we note the variation in the total number of backtracks over all benchmark circuits. The identification of head lines leads to mixed results, which we conjecture to be related to the decision making procedure chosen. Over all algorithms, the total number of backtracks decreases significantly as more pruning methods are considered. For TG-LEAP (i.e. $A_9$) the total

| Circuit | $A_0$ | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ | $A_6$ | $A_7$ | $A_8$ | $A_9$ |
|---|---|---|---|---|---|---|---|---|---|---|
| C432 | 2041 | 2002 | 2379 | 2379 | 1876 | 1376 | 1376 | 1376 | 90 | 76 |
| C499 | 4078 | 4078 | 4078 | 4078 | 74 | 74 | 74 | 74 | 74 | 74 |
| C880 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| C1355 | 4002 | 4002 | 4002 | 4002 | 0 | 0 | 0 | 0 | 0 | 0 |
| C1908 | 8223 | 9160 | 9009 | 5920 | 3308 | 1091 | 59 | 51 | 57 | 53 |
| C2670 | 27585 | 28615 | 28592 | 32876 | 16563 | 13037 | 12647 | 8191 | 666 | 254 |
| C3540 | 18479 | 18479 | 17643 | 14684 | 6769 | 699 | 763 | 687 | 216 | 206 |
| C5315 | 5558 | 5537 | 5321 | 4961 | 1091 | 1029 | 1636 | 263 | 181 | 125 |
| C6288 | 8717 | 8717 | 8717 | 13318 | 3405 | 10661 | 10661 | 4184 | 2092 | 1796 |
| C7552 | 65576 | 101069 | 100929 | 100151 | 62733 | 62381 | 54925 | 49734 | 14275 | 12507 |
| Total | 144259 | 181659 | 180670 | 182369 | 95819 | 90348 | 82141 | 64560 | 17651 | 15091 |

**Table 7.5:** Total number of backtracks

| Circuit | $A_0$ | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ | $A_6$ | $A_7$ | $A_8$ | $A_9$ |
|---|---|---|---|---|---|---|---|---|---|---|
| C432 | 13466 | 13356 | 13905 | 13820 | 12197 | 9458 | 9458 | 9458 | 5765 | 5765 |
| C499 | 36960 | 36960 | 36944 | 36944 | 26620 | 26620 | 26620 | 26620 | 26618 | 26618 |
| C880 | 8251 | 8149 | 8103 | 8077 | 6953 | 6930 | 6965 | 6965 | 6965 | 6965 |
| C1355 | 59537 | 59537 | 59537 | 58951 | 52182 | 51426 | 51426 | 51426 | 51426 | 51426 |
| C1908 | 42746 | 42584 | 42128 | 36733 | 29147 | 26068 | 24420 | 24420 | 24199 | 24199 |
| C2670 | 89388 | 89816 | 90347 | 115033 | 68568 | 72982 | 71112 | 84996 | 47884 | 47834 |
| C3540 | 76559 | 76556 | 73704 | 69461 | 45052 | 34714 | 34662 | 34522 | 33566 | 33566 |
| C5315 | 70885 | 70742 | 70350 | 69863 | 57244 | 57413 | 58415 | 55798 | 55463 | 55411 |
| C6288 | 200043 | 200028 | 200021 | 242541 | 188644 | 235028 | 235022 | 221051 | 214785 | 214785 |
| C7552 | 283499 | 325693 | 329831 | 322738 | 244017 | 247798 | 222420 | 235219 | 168314 | 170012 |

**Table 7.6:** Total number of decisions

number of backtracks reaches a minimum, far from the total number of backtracks of the other algorithms.

Another data point of interest is to identify which types of faults are actually aborted. For this purpose, PODEM*, FAN*, SOCRATES* and TG-LEAP were run with simple and multiple backtracing and a backtracking limit of 500. The results obtained with simple backtracing are

| Circuit | PODEM* | | | FAN* | | | SOCRATES* | | | TG-LEAP | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | #D | #R | #A | #D | #R | #A | #D | #R | #A | #D | #R | #A |
| C432 | 519 | 0 | 5 | 519 | 1 | 4 | 519 | 2 | 3 | 520 | 4 | 0 |
| C499 | 750 | 0 | 8 | 750 | 8 | 0 | 750 | 8 | 0 | 750 | 8 | 0 |
| C880 | 942 | 0 | 0 | 942 | 0 | 0 | 942 | 0 | 0 | 942 | 0 | 0 |
| C1355 | 1566 | 0 | 8 | 1566 | 8 | 0 | 1566 | 8 | 0 | 1566 | 8 | 0 |
| C1908 | 1864 | 6 | 9 | 1868 | 9 | 2 | 1868 | 9 | 2 | 1868 | 9 | 2 |
| C2670 | 2626 | 62 | 59 | 2630 | 86 | 31 | 2630 | 93 | 24 | 2630 | 117 | 0 |
| C3540 | 3282 | 114 | 32 | 3287 | 132 | 9 | 3291 | 137 | 0 | 3291 | 137 | 0 |
| C5315 | 5291 | 55 | 4 | 5291 | 59 | 0 | 5391 | 59 | 0 | 5291 | 59 | 0 |
| C6288 | 7675 | 34 | 35 | 7700 | 34 | 10 | 7710 | 34 | 0 | 7710 | 34 | 0 |
| C7552 | 7375 | 62 | 113 | 7388 | 73 | 89 | 7390 | 77 | 83 | 7417 | 131 | 2 |
| Total | | | 273 | | | 145 | | | 112 | | | 4 |

**Table 7.7:** Test generation results with simple backtracing

shown in Table 7.7. Note that TG-LEAP aborts 4 detectable faults (given a backtracking limit of 500 and simple backtracing). As can be seen the remaining algorithms abort a much larger number of faults. The results obtained with multiple backtracing are shown in Table 7.8. In this situation, TG-LEAP aborts no faults. With respect to the simple backtracing case, the number of aborted faults decreases for PODEM* and SOCRATES*, but increases for FAN*. With multiple backtracing, and for C6288, FAN* performs better than SOCRATES*. The reason for this result is attributed to the larger j-frontiers created by SOCRATES* with non-local implications. As mentioned earlier, the existence of larger j-frontiers may cause some wrong initial decisions, which are difficult to correct when the size of the decision tree becomes large. Although TG-LEAP uses the same decision assignments as SOCRATES*, the initial wrong assignments are overcome by conflict diagnosis; conflict-directed backtracking and failure-driven assertions.

For both tests above, the average run times per fault (in seconds) are shown in Table 7.9, where columns labeled **S** denote simple backtracing, and columns labeled **M** denote multiple backtracing. For circuits where several faults are aborted by the other algorithms, TG-LEAP performs better. However, in circuits where SOCRATES* does not abort any fault (e.g. C499, C880,

| Circuit | PODEM* | | | FAN* | | | SOCRATES* | | | TG-LEAP | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | #D | #R | #A | #D | #R | #A | #D | #R | #A | #D | #R | #A |
| C432 | 520 | 1 | 3 | 520 | 1 | 3 | 520 | 2 | 2 | 432 | 4 | 0 |
| C499 | 750 | 0 | 8 | 750 | 8 | 0 | 750 | 8 | 0 | 750 | 8 | 0 |
| C880 | 942 | 0 | 0 | 942 | 0 | 0 | 942 | 0 | 0 | 942 | 0 | 0 |
| C1355 | 1566 | 0 | 8 | 1566 | 8 | 0 | 1566 | 8 | 0 | 1566 | 8 | 0 |
| C1908 | 1861 | 6 | 12 | 1866 | 7 | 6 | 1870 | 9 | 0 | 1870 | 9 | 0 |
| C2670 | 2630 | 68 | 49 | 2628 | 86 | 33 | 2630 | 93 | 24 | 2630 | 117 | 0 |
| C3540 | 3281 | 114 | 33 | 3284 | 132 | 12 | 3291 | 137 | 0 | 3291 | 137 | 0 |
| C5315 | 5290 | 53 | 7 | 5291 | 59 | 0 | 5291 | 59 | 0 | 5291 | 59 | 0 |
| C6288 | 7703 | 34 | 7 | 7708 | 34 | 2 | 7695 | 34 | 15 | 7708 | 34 | 0 |
| C7552 | 7369 | 62 | 119 | 7349 | 77 | 124 | 7368 | 77 | 105 | 7419 | 131 | 0 |
| Total | | | 246 | | | 180 | | | 146 | | | 0 |

**Table 7.8:** Test generation results with multiple backtracing

| Circuit | PODEM* | | FAN* | | SOCRATES* | | TG-LEAP | |
|---|---|---|---|---|---|---|---|---|
| | S | M | S | M | S | M | S | M |
| C432 | 0.054 | 0.058 | 0.025 | 0.057 | 0.023 | 0.041 | 0.013 | 0.026 |
| C499 | 0.066 | 0.156 | 0.038 | 0.094 | 0.038 | 0.093 | 0.039 | 0.093 |
| C880 | 0.020 | 0.025 | 0.025 | 0.029 | 0.025 | 0.029 | 0.025 | 0.029 |
| C1355 | 0.123 | 0.255 | 0.103 | 0.232 | 0.102 | 0.230 | 0.103 | 0.229 |
| C1908 | 0.120 | 0.163 | 0.113 | 0.111 | 0.084 | 0.093 | 0.090 | 0.093 |
| C2670 | 0.166 | 0.228 | 0.109 | 0.185 | 0.112 | 0.191 | 0.072 | 0.130 |
| C3540 | 0.164 | 0.216 | 0.093 | 0.126 | 0.079 | 0.117 | 0.080 | 0.114 |
| C5315 | 0.069 | 0.101 | 0.097 | 0.121 | 0.099 | 0.125 | 0.099 | 0.120 |
| C6288 | 0.324 | 0.445 | 0.199 | 0.440 | 0.221 | 0.596 | 0.210 | 0.423 |
| C7552 | 0.218 | 0.369 | 0.206 | 0.320 | 0.211 | 0.332 | 0.186 | 0.272 |

**Table 7.9:** Run time per fault with simple and multiple backtracing

C1355, C1908, C3540 and C5315), the average running time per fault for TG-LEAP can be

slightly larger. The main reason for the larger run times is related to the overhead of diagnosing conflicts, that leads to larger run times when the number of backtracks of SOCRATES* and TG-LEAP are similar. For circuits where SOCRATES* aborts faults, TG-LEAP has smaller run times. However, if the backtrack limit is reduced, SOCRATES* would eventually have smaller run times at the cost of some aborted faults.

As Table 7.9 indicates, the average CPU times using the multiple backtracing scheme are most often larger than the average CPU times using simple backtracing. In several of the circuits, the average run time per fault almost doubles with multiple backtracing. The main reason for this discrepancy in run times is due to the difference in the number of nodes traversed by multiple backtracing and by simple backtracing. Consequently, simple backtracing is most often the better option, the problem being that TG-LEAP can abort a few faults with simple backtracing. These results immediately suggest using one the algorithms with simple backtracing to detect most faults, and use another algorithm (e.g. TG-LEAP) to detect or prove redundant the remaining faults.

#### 7.2.1.2 Asymptotic Behavior With Time and Backtrack Limits

In general, limited resources are available for detecting each fault or proving that the fault is redundant. Consequently, we tested how hard can it be for detecting each fault with each of the above algorithms. PODEM*, FAN*, SOCRATES* and TG-LEAP were run, using multiple backtracing, with increasing backtracking limits and with increasing maximum allowed CPU time per fault.

The results of running the algorithms with increasing maximum allowed time per fault are shown in Figure 7.2. As can be concluded, TG-LEAP may require a reasonable amount of time for detecting a few faults. On the other hand, the remaining algorithms abort a significant number of faults even when the maximum CPU time per fault is on the order of hundreds of seconds.

The results of running the different algorithms with increasing backtrack limits are shown in Figure 7.3. Once more, we can conclude that TG-LEAP may require a reasonably large number of backtracks for detecting a few faults. On the other hand, the remaining algorithms abort a large number of faults even if tens of thousands of backtracks are allowed. Further note that after a back-

**Figure 7.2:** Number of aborted faults versus CPU time per fault

track limit of 50, the number of aborted faults for PODEM*, FAN* and SOCRATES* decreases slightly with increases in the number of allowed backtracks per fault.

#### 7.2.1.3 Handling Difficult Faults

TG-LEAP is primarily targeted at difficult faults, either hard to detect or redundant. With the purpose of comparing TG-LEAP with the other algorithms on difficult faults, a small set of redundant and hard to detect faults was chosen from some of the benchmark circuits. The results obtained are shown in Table 7.10; columns labeled **#B** denote the number of backtracks and the column labeled **#A** denotes the number of assertions identified by TG-LEAP. The backtrack limit was set to 50000 for all algorithms and for all faults. For all the redundant faults, TG-LEAP proves redundancy with a small number of backtracks. On the other hand, the other algorithms cannot prove redundancy in some cases, given a backtrack limit of 50000. The difference in the number of backtracks between SOCRATES* and TG-LEAP illustrates the strength of the pruning methods incorporated into TG-LEAP. For both algorithms, the decision tree created for each fault is the

**Figure 7.3:** Number of aborted faults versus number of backtracks

same until backtracking is required. Afterwards, while SOCRATES* usually requires a very large number of backtracks, TG-LEAP manages to derive the information required to skip several decision assignments, thus proving redundancy with a very small number of backtracks. Furthermore, in each of the examples shown, which require backtracking, several assertions are determined by analyzing the causes of conflicts.

PODEM*, FAN* and SOCRATES* abort fault 2282 s-a-1. It interesting to analyze the difference of run times for each of these algorithms. PODEM* is the simplest and executes 50000 backtracks faster than the others. SOCRATES* is the slowest. The reasons are the non-local implications that must be processed, and which introduce some overhead, and the requirement to compute unique sensitization points at each decision level. Even though the procedure for computing unique sensitization points has linear time complexity, its overhead increases the run time, which

| Circuit R: redundant D: detectable | PODEM* | | FAN* | | SOCRATES* | | TG-LEAP | | |
|---|---|---|---|---|---|---|---|---|---|
| | #B | Time | #B | Time | #B | Time | #B | #A | Time |
| C432 (R) 259gat s-a-1 | > 50000 | 92.86 | 5618 | 50.09 | 793 | 4.29 | 35 | 66 | 0.33 |
| C432 (R) 347gat s-a-1 | > 50000 | 127.90 | 5740 | 38.20 | 921 | 5.92 | 11 | 20 | 0.09 |
| C1908 (R) 565 s-a-1 | > 50000 | 331.80 | > 50000 | 569.30 | 0 | 0.03 | 0 | 0 | 0.03 |
| C2670 (R) 2282 s-a-1 | > 50000 | 460.50 | > 50000 | 650.80 | > 50000 | 882.50 | 9 | 16 | 0.20 |
| C2670 (R) 2417 s-a-1 | > 50000 | 476.80 | 1872 | 41.54 | 15592 | 242.20 | 4 | 6 | 0.12 |
| C7552 (D) 3695 s-a-1 | > 50000 | 438.00 | > 50000 | 390.00 | > 50000 | 397.50 | 107 | 48 | 1.80 |

**Table 7.10:** Handling difficult faults

becomes noticeable when a large number of backtracks is required. FAN* does not process non-local implications, and hence the excess of run time is only due to the procedure for the identification of unique sensitization points (when the size of the p-frontier is 1).

For fault 2417 s-a-1 of C2670, FAN* manages to prove redundancy with fewer backtracks than SOCRATES*. As mentioned earlier, the reason is conjectured to be the larger j-frontier in SOCRATES* caused by non-local implications, that in some situations may cause the multiple backtracing scheme to make several wrong assignments, and which can result in SOCRATES* not being able to either detect the fault or prove the fault redundant. For the particular case of fault 2417 s-a-1 of C2670, this fact causes SOCRATES* to require an order of magnitude more backtracks than FAN*.

For fault 3695 s-a-1 in circuit C7552, although LEAP requires 107 backtracks to find a test pattern to detect the fault, none of the other algorithms is able to detect the fault in less than 50000 backtracks. This example further illustrates the applicability of the pruning methods used in TG-LEAP when compared to SOCRATES*.

| Circuit | Decisions | | Backtracks | | Assertions | | USIs | | Head lines | |
|---|---|---|---|---|---|---|---|---|---|---|
| | S | M | S | M | S | M | S | M | S | M |
| C432 | 12.49 | 11.00 | 0.170 | 0.145 | 1.05 | 0.48 | 3.98 | 3.96 | 2.02 | 1.41 |
| C499 | 34.10 | 35.12 | 0.169 | 0.098 | 2.43 | 2.46 | 0.61 | 0.93 | 0.51 | 0.07 |
| C880 | 10.87 | 7.39 | 0.015 | 0.000 | 0.04 | 0.00 | 2.60 | 2.10 | 3.60 | 2.04 |
| C1355 | 32.88 | 32.67 | 0.000 | 0.000 | 1.91 | 0.20 | 0.71 | 0.59 | 0.02 | 0.01 |
| C1908 | 18.32 | 12.88 | 0.850 | 0.028 | 1.52 | 0.27 | 1.87 | 2.10 | 0.71 | 0.44 |
| C2670 | 16.93 | 17.41 | 0.088 | 0.092 | 0.66 | 0.19 | 2.83 | 2.84 | 12.71 | 12.71 |
| C3540 | 11.56 | 9.79 | 0.009 | 0.060 | 0.52 | 0.47 | 2.80 | 2.86 | 0.43 | 0.38 |
| C5315 | 9.77 | 10.36 | 0.024 | 0.023 | 0.25 | 0.09 | 2.20 | 2.23 | 0.88 | 0.83 |
| C6288 | 26.94 | 27.74 | 0.034 | 0.232 | 0.35 | 0.69 | 0.31 | 0.32 | 0.01 | 0.01 |
| C7552 | 33.73 | 22.52 | 1.165 | 1.657 | 1.00 | 1.38 | 2.99 | 3.35 | 1.03 | 1.01 |

**Table 7.11:** Statistics for TG-LEAP (average numbers per fault)

#### 7.2.1.4 Statistics for TG-LEAP

Some of the relevant statistics of running TG-LEAP with simple and multiple backtracing on each of the benchmark circuits are shown in Table 7.11; columns labeled **S** denote simple backtracing whereas columns labeled **M** indicate multiple backtracing. The results were obtained with dynamic head line evaluation, in order to also obtain data with respect to head lines. For TG-LEAP, the number of decision assignments is usually small when compared with the number of primary inputs of each circuit. The average number of decision assignments depends on the backtracing scheme chosen, and none of the backtracing schemes implemented seems to be definitely better. On average, the number of backtracks per fault is negligible; the only exception being C7552. The same holds true with the number of failure-driven assertions. Since the average number of backtracks is small, the number of assertions is also necessarily small. Except for specific circuits (C499, C1355 and C6288), a reasonable number of unique sensitization implications is determined for each fault. These implications are crucial for pruning the amount of search, as the results of Table 7.4 show.

For several circuits, static as well as dynamic head lines are found and reduced. C432,

| Circuit | PODEM* (backtrack limit of 5) | | | | TG-LEAP (backtrack limit of 500) | | | | Time/Fault |
|---|---|---|---|---|---|---|---|---|---|
| | #T | #D | #R | #A | #T | #D | #R | #A | |
| C432 | 524 | 510 | 0 | 14 | 14 | 10 | 4 | 0 | 0.022 |
| C499 | 758 | 750 | 0 | 8 | 8 | 0 | 8 | 0 | 0.040 |
| C880 | 942 | 940 | 0 | 2 | 2 | 2 | 0 | 0 | 0.020 |
| C1355 | 1574 | 1566 | 0 | 8 | 8 | 0 | 8 | 0 | 0.105 |
| C1908 | 1879 | 1818 | 6 | 55 | 55 | 52 | 3 | 0 | 0.059 |
| C2670 | 2747 | 2624 | 49 | 74 | 74 | 6 | 68 | 0 | 0.053 |
| C3540 | 3428 | 3262 | 100 | 66 | 66 | 29 | 37 | 0 | 0.071 |
| C5315 | 5350 | 5268 | 46 | 36 | 36 | 23 | 13 | 0 | 0.059 |
| C6288 | 7744 | 7534 | 34 | 176 | 176 | 176 | 0 | 0 | 0.163 |
| C7552 | 7550 | 7364 | 52 | 134 | 134 | 55 | 79 | 0 | 0.146 |
| Total | 32496 | 31636 | 287 | 573 | 573 | 353 | 220 | 0 | |

**Table 7.12:** Results using PODEM* followed by TG-LEAP

C880, and C2670 appear to be specially suited for producing head lines. For circuits C499, C1355 and C6288, on the contrary, the number of head lines is small. As noted before, dynamic identification of head lines only contributes with noticeable overhead and leads to an increase on the run times in most cases.

#### 7.2.1.5 Optimizing Test Pattern Generation

The results of the previous sections suggest that TG-LEAP introduces unnecessary overhead for easy to detect faults. Furthermore, the use of multiple backtracing also introduces significant overhead and should be used only when required. In general, PODEM*, with simple backtracing, can be used to detect most of the faults with very reduced computational overhead. On the other hand, using TG-LEAP is preferable to detect or prove redundant the more difficult faults. Hence, we ran PODEM*, with simple backtracing and a backtrack limit of 5, on all the benchmark circuits. Afterwards, we ran TG-LEAP, with multiple backtracing and a backtrack limit of 500, on the set of faults aborted by PODEM*. The results obtained are shown in Table 7.12. The total number of faults analyzed by each algorithm is denoted by **#T**. The number of detected,

redundant and aborted faults is denoted by #**D**, **#R** and **#A**, respectively. PODEM* detects a total 31636 detectable faults from a total of 32496 faults, proves redundant 287 faults, and aborts 573 faults. Afterwards, TG-LEAP detects 353 faults from an initial total of 573, proves redundant 220 faults and aborts no faults. For C499, C880 and C1355 some of the algorithms described earlier can perform better alone without aborting faults. For the remaining benchmark circuits, using the combination of PODEM* followed by TG-LEAP achieves a much better performance than any of the other algorithms alone. Furthermore, as expected no fault is aborted, since TG-LEAP with multiple backtracing aborts no faults.

Even though the integrated application of the two algorithms yields promising run times, better results are to be expected by tuning the implementation of TG-LEAP.

**Perspective**

The results presented in this section are intended only to illustrate the effectiveness of TG-LEAP for difficult faults, both redundant and detectable. In a complete test pattern generation system (as proposed for example in [37, 144, 174]), fault simulation would be employed to reduce the test set size, and to randomly detect some difficult detectable faults. We further note that our implementation of SOCRATES* has some relevant differences with respect to the original algorithm [144, 145]. SOCRATES uses an improved multiple backtracing procedure as well as improved controllability/observability measures to guide the decision procedure. Furthermore, SOCRATES* only implements one of the unique sensitization procedures of SOCRATES. These differences justify the differences in results observed between SOCRATES* and SOCRATES.

### 7.2.2 Results for Timing Analysis

The results obtained for timing analysis assume a unit delay for every circuit gate. The delay iteration procedure decrements a unit delay at each iteration. The final circuit delay reported corresponds to the threshold delay of the last iteration.

The results for the timing analysis tool are shown in Table 7.13, and are compared with the results obtained with TrueD-F [50, 52][2]. The ISCAS'85 benchmark circuits, as well the bench-

[2.] We choose to compare TA-LEAP with TrueD-F since the circuit delay results of TrueD-F are consistent with ours, and are based on a detailed experimental procedure. TrueD-F is implemented in the C programming language and the results were obtained on a SUN 4 workstation.

| Circuit | LTP | Delay Δ | TrueD-F [50, 52] | | TA-LEAP (multiple) | | TA-LEAP (simple) | |
|---|---|---|---|---|---|---|---|---|
| | | | #B | Time | #B | Time | #B | Time |
| C432 | 17 | 17 | — | — | 0 | 0.18 | 0 | 0.08 |
| C499 | 11 | 11 | — | — | 0 | 0.52 | 0 | 0.40 |
| C880 | 24 | 24 | — | — | 0 | 0.13 | 0 | 0.07 |
| C1355 | 24 | 24 | — | — | 0 | 0.86 | 0 | 0.90 |
| C1908 | 40 | 37 | — | — | 23 | 2.90 | 23 | 2.00 |
| C2670 | 32 | 30 | — | — | 17 | 3.12 | 12 | 1.40 |
| C3540 | 47 | 46 | — | — | 12 | 2.65 | 9 | 1.40 |
| C5315 | 49 | 47 | — | — | 256 | 31.23 | 428 | 43.21 |
| C6288 | 124 | 123 | — | — | 5713 | 1981.00 | 2663 | 679.20 |
| C7552 | 43 | 42 | — | — | 2383 | 282.60 | 27 | 4.17 |
| C432 (N) | 19 | 19 | 1 | 0.06 | 0 | 0.18 | 0 | 0.09 |
| C499 (N) | 25 | 25 | 1 | 0.37 | 0 | 0.84 | 0 | 0.77 |
| C880 (N) | 20 | 20 | 31 | 0.59 | 0 | 0.14 | 0 | 0.11 |
| C1355 (N) | 27 | 27 | 0 | 0.39 | 0 | 1.03 | 0 | 1.05 |
| C1908 (N) | 34 | 31 | 89436 | 3674.52 | 6 | 1.41 | 9 | 1.40 |
| C2670 (N) | 25 | 24 | 5306 | 200.23 | 0 | 0.73 | 1 | 0.44 |
| C3540 (N) | 41 | 39 | 3941 | 181.63 | 0 | 0.46 | 1 | 0.58 |
| C5315 (N) | 46 | 45 | 116 | 5.15 | 62 | 7.61 | 5 | 0.85 |
| C6288 (N) | 123 | 122 | 10345 | 802.60 | 5970 | 2352.00 | 1068 | 293.80 |
| C7552 (N) | 38 | 37 | 61 | 5.89 | 2 | 0.93 | 13 | 1.35 |
| CBP.12.2 | 40 | 23 | 31134 | 233.27 | 1522 | 46.08 | 887 | 21.58 |
| CBP.16.4 | 44 | 27 | 33454 | 238.24 | 420 | 18.96 | 191 | 7.11 |
| CLA.16 | 34 | 34 | 0 | 0.04 | 0 | 0.02 | 0 | 0.02 |
| TAU92EX1 | 27 | 24 | 33530 | 217.48 | 23 | 2.43 | 22 | 2.26 |
| TAU92EX2 | 93 | 42 | 12413 | 2210.10 | 2177 | 885.00 | 2588 | 918.80 |
| MULT-CSA | 78 | 78 | 5972 | 1352.55 | 135 | 60.04 | 126 | 41.42 |
| MULT-RPL | 107 | 106 | 3692 | 544.15 | 5052 | 8651.00 | 688 | 1044.00 |
| MULT-WALL | 52 | 51 | 22474 | 9334.50 | 31610 | 11200.00 | 12306 | 3482.00 |
| Total | | | 251907 | | 55383 | | 21067 | |

**Table 7.13:** Results for TA-LEAP: number of backtracks and CPU time

mark circuits of [50], are used to evaluate TA-LEAP. Results with simple and multiple backtracing

are shown. As can be concluded, simple backtracing performs better than multiple backtracing over all circuits, even though for some cases it requires more backtracks. When compared with TrueD-F, TA-LEAP requires fewer backtracks for all benchmark circuits for which backtracking is required. For these circuits, only for MULT-RPL does TA-LEAP require more CPU time[3]. For some benchmark circuits, TA-LEAP requires several orders of magnitude fewer backtracks than TrueD-F. Moreover, for most benchmark circuits the number of backtracks required by TA-LEAP is negligible. Further note that over all benchmark circuits, TA-LEAP with simple backtracing requires an order of magnitude fewer backtracks than TrueD-F, even though TA-LEAP is applied to a larger number of benchmark circuits.

It is interesting to note that for circuits where simple and multiple backtracing require similar number of backtracks, multiple backtracing requires significantly more time. This fact shows that a reasonable amount of time is spent on backtracing, which in some cases can introduce more overhead than conflict diagnosis.

#### 7.2.2.1 Statistics for TA-LEAP

Table 7.14 contains the statistics of running TA-LEAP on the benchmark circuits with simple (**S**) and multiple (**M**) backtracing. As can be concluded, a large number of logical implications is identified due to unique sensitization points and implications (USPs and USIs). These implications are particularly effective in reducing the amount of search and have not been used by previous algorithms for circuit delay computation. The number of assertions is also significant, which illustrates that the structure of implication sequences often contain several unique implication points (UIPs), and which can be used to reduce the amount of search.

For circuits where a large number of backtracks is required, the number of USIs and assertions can become particularly large. This fact results from USIs and assertions being rediscovered a large number of times, since the search process visits related stages of the search several times. For USPs this fact indicates that logical conditions may constrain the set of propagation paths such that USPs are defined. For assertions the results suggest that conflict-based equivalence might pre-

[3] The performance of the two machines differs. For integer processing, a program should on average take 30 to 40% more time on a SUN 4 than on a DEC 5000/240. Note, however, that C++ code also runs slower than C code, and so the results for TA-LEAP would improve if it was coded in C.

| Circuit | TA-LEAP | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Decisions | | Backtracks | | Assertions | | USIs | | Head Lines |
| | S | M | S | M | S | M | S | M | |
| C432 | 16 | 35 | 0 | 0 | 0 | 7 | 10 | 4 | 0 |
| C499 | 40 | 41 | 0 | 0 | 4 | 0 | 3 | 0 | 0 |
| C880 | 15 | 28 | 0 | 0 | 2 | 0 | 13 | 6 | 0 |
| C1355 | 38 | 32 | 0 | 0 | 0 | 1 | 11 | 17 | 0 |
| C1908 | 48 | 58 | 23 | 23 | 3 | 35 | 167 | 184 | 0 |
| C2670 | 68 | 68 | 12 | 17 | 19 | 61 | 45 | 41 | 9 |
| C3540 | 30 | 34 | 9 | 12 | 27 | 25 | 250 | 277 | 0 |
| C5315 | 959 | 536 | 428 | 256 | 602 | 286 | 24437 | 18253 | 0 |
| C6288 | 4534 | 12337 | 2663 | 5713 | 6390 | 20133 | 27339 | 52795 | 0 |
| C7552 | 126 | 4858 | 27 | 2383 | 116 | 3055 | 2812 | 193899 | 0 |
| C432 (N) | 10 | 24 | 0 | 0 | 0 | 0 | 11 | 9 | 0 |
| C499 (N) | 35 | 40 | 0 | 0 | 0 | 0 | 6 | 7 | 0 |
| C880 (N) | 23 | 29 | 0 | 0 | 0 | 0 | 14 | 15 | 0 |
| C1355 (N) | 36 | 32 | 0 | 0 | 0 | 2 | 11 | 17 | 0 |
| C1908 (N) | 35 | 34 | 9 | 6 | 2 | 4 | 70 | 63 | 0 |
| C2670 (N) | 56 | 36 | 1 | 0 | 25 | 27 | 27 | 25 | 0 |
| C3540 (N) | 18 | 8 | 1 | 0 | 32 | 2 | 62 | 63 | 0 |
| C5315 (N) | 32 | 282 | 5 | 62 | 5 | 72 | 230 | 2612 | 0 |
| C6288 (N) | 1977 | 12906 | 1068 | 5970 | 4306 | 26736 | 8087 | 69676 | 0 |
| C7552 (N) | 40 | 33 | 13 | 2 | 3 | 3 | 421 | 142 | 0 |
| CBP.12.2 | 1156 | 2347 | 887 | 1522 | 473 | 2232 | 13587 | 33627 | 0 |
| CBP.16.4 | 251 | 653 | 191 | 420 | 107 | 408 | 5949 | 15039 | 0 |
| CLA.16 | 13 | 13 | 0 | 0 | 0 | 0 | 30 | 30 | 0 |
| TAU92EX1 | 68 | 68 | 22 | 23 | 32 | 34 | 772 | 1149 | 2 |
| TAU92EX2 | 4344 | 4370 | 2588 | 2177 | 1994 | 2361 | 72018 | 48418 | 0 |
| MULT-CSA | 241 | 228 | 126 | 135 | 301 | 293 | 2620 | 7264 | 0 |
| MULT-RPL | 1383 | 8792 | 688 | 5052 | 1334 | 7740 | 1722 | 47473 | 0 |
| MULT-WALL | 22408 | 57272 | 12306 | 31610 | 33590 | 83732 | 361935 | 938278 | 0 |

**Table 7.14:** Statistics for TA-LEAP with simple and multiple backtracing

vent some of the conflicts that yield failure-driven assertions, and consequently reduce the number

| Block Size | Number of bits | | | | | | |
|---|---|---|---|---|---|---|---|
| | **8** | **12** | **16** | **20** | **24** | **32** | **64** |
| **2** | (25; 14) | (37; 18) | (49; 22) | (61; 26) | (73; 30) | (97; 38) | (193; 70) |
| **4** | (21; 18) | (31; 20) | (41; 22) | (51; 24) | (61; 26) | (81; 34) | (161; 46) |
| **8** | (19; 19) | — | (37; 34) | — | (55; 36) | (73; 38) | (145; 46) |
| **16** | — | — | (35; 35) | — | — | (69; 66) | (137; 70) |
| **32** | — | — | — | — | — | (67; 67) | (133; 130) |
| **64** | — | — | — | — | — | — | (131; 131) |
| **128** | — | — | — | — | — | — | — |

**Table 7.15:** Delay values (LTP; $\Delta_C$) for carry-skip adders

of assertions.

The contribution of head lines is negligible. The results shown just include static head line identification. From our experiments we have concluded that dynamic head line identification only increases the run times without significantly reducing the amount of search.

#### 7.2.2.2 Analysis of Carry-Skip Adders

Even though the results of the previous section appear to indicate that TA-LEAP performs particularly well for most practical circuits, this may not always be the case. For some instances of carry-skip adders (CSAs), the search space can become too large for TA-LEAP to manage to identify a solution.

Table 7.15 contains delay information for several carry-skip adders with varying total number of bits and block sizes. In particular, the longest topological path delay (LTP) as well as the longest sensitizable path delay ($\Delta_C$) are shown. (The organization of carry-skip adders as well as the different delay expressions are described in Appendix C, where CSAs of fixed block sizes are assumed.)

The results obtained with TA-LEAP using simple backtracing and a backtrack limit of 50000 are shown in Table 7.16. Whenever TA-LEAP is unable to identify a solution, the largest

| Block Size | Number of bits | | | | | | |
|---|---|---|---|---|---|---|---|
| | **8** | **12** | **16** | **20** | **24** | **32** | **64** |
| **2** | 14 | 18 | 22 | ≤ 29 | ≤ 41 | ≤ 65 | ≤ 161 |
| **4** | 18 | 20 | 22 | 24 | ≤ 28 | ≤ 48 | ≤ 128 |
| **8** | 19 | — | 34 | — | 36 | 38 | ≤ 97 |
| **16** | — | — | 35 | — | — | 66 | 70 |
| **32** | — | — | — | — | — | 67 | 130 |
| **64** | — | — | — | — | — | — | 131 |
| **128** | — | — | — | — | — | — | — |

**Table 7.16:** Computed delay Δ for carry-skip adders

path delay proved false is included; otherwise the circuit delay Δ computed by TA-LEAP is shown, which is always the same as predicted by the analysis of CSAs in Appendix C. As can be concluded, TA-LEAP aborts circuit delay computation for CSAs with a large number of bits and with small block sizes. For each number of bits of a CSA we can define an optimal design as the one that minimizes area while guaranteeing minimum circuit delay. The optimal design for each number of bits is marked in Table 7.16. As the number of bits in the carry-skip adder increases, TA-LEAP becomes unable to compute the circuit delay for these optimal designs. We can thus conclude that for a large number of combinational circuits (e.g. the benchmark circuits), TA-LEAP performs well in almost all cases. On the other hand, we have constructed examples of regular circuits where TA-LEAP is unable (under the backtrack limit specified) to compute the circuit delay.

For carry-skip adders, structural information is useful for delays close to the largest topological path delay because a large number of USPs and USIs is identified. For smaller delays, structural properties can no longer be exploited since the carry bypass logic begins to be involved in potential sensitizable paths, thus reducing the number of USPs and associated USIs. Given that there are more options on how to sensitize a path, TA-LEAP needs to consider more primary inputs assignments to prove all paths false. For this reason the search space grows significantly as the threshold delay approaches the sensitizable path delay, and too large a number of backtracks

may be required to prove intermediate delays as false. CSAs denote a type of circuits where pruning methods based on delay information can be particularly useful. A large number of USPs in CSAs can be shown not to be sensitizable with delays corresponding to the longest topological path. This information can be used to prune the considered threshold delays without having to use search to prove that such threshold delays cannot be sensitized. Further research work is needed on how to incorporate delay-based pruning methods within the search framework proposed in this dissertation.

## 7.3 Conclusions

For both circuit analysis tools described in the present chapter, the obtained experimental results are promising. As noted above, the tools have yet to be subject to implementation fine-tuning, and the processing overhead is likely to be reduced. Furthermore, not all algorithmic functionality has been implemented and tested. Additional search pruning is to be expected with a more complete implementation of the path sensitization algorithms for both tools. In particular, the application of diagnosis engines with bounded growth of the clause database, permit restricted forms of conflict-based equivalence, and may prove useful in identifying strong search pruning conditions.

The timing analysis tool performs well for unstructured logic circuits. In contrast, an analysis of several carry-skip adder configurations showed that TA-LEAP is unable to compute circuit delay for CSAs with large number of bits and with small block sizes. The implementation of the basic diagnosis engine, or diagnosis engines with bounded growth of the clause database, can be of use for these circuits. Furthermore, we briefly mentioned how delay-based pruning methods can be applied to some circuits, CSAs included.

The results presented in this chapter for both tools are comparable to, when not better than, most results obtained in recent years with other test pattern generation and timing analysis tools. This fact justifies and motivates the integrated development of circuit analysis tools, where common pruning methods can be implemented and shared among the different tools.

# CHAPTER VIII

# CONCLUSIONS

## 8.1 Contributions

The contributions of our work can be divided among the following main areas:

- Satisfiability algorithms.
- Path sensitization model.
- Path sensitization algorithms.

The following sections review the contributions in each of these areas.

### 8.1.1 Search-Based Satisfiability Algorithms

We developed a search algorithm for satisfiability, GRASP, that can be configured with different engines: selection, deduction, diagnosis, preprocessing and postprocessing engines.

The description of diagnosis engines was emphasized since conflict diagnosis has seldom been applied to SAT. Conflict diagnosis is based on several methods to prune the amount of search. We described *conflict-directed backtracking*, *conflict-based equivalence* and *failure-driven assertions*. Moreover, *unique implications points*, *multiple conflicts* and *iterated conflicts* were proposed as additional techniques for conflict diagnosis. Engines for conflict diagnosis can also be simplified, whenever the computational overhead at each decision level is important, or improved whenever precise diagnosis is the goal. We proposed a hierarchy of simplified diagnosis engines that permit a wide spectrum of diagnosis ability versus clause database growth complexity.

Deduction engines are characterized by their deduction ability. We described a hierarchy

of deduction engines, and associated deduction abilities, based on testing combinations of assignments and applying consensus operations. These deduction engines supersede deduction procedures proposed by other authors. The same techniques were applied to preprocessing and a hierarchy of preprocessing engines was also described.

The objectives of postprocessing engines can be divided into *redundancy removal* and *solution caching* and were described solely for combinational circuits. Redundancy removal allows deleting from a computed solution decision assignments that are irrelevant for the goal to be satisfied. Solution caching permits identifying signatures of computed solutions that can be used to reduce the amount of search for other queries on the clause database.

Finally, we reviewed decision making procedures and described a relationship between head line identification and consensus on a clause database.

### 8.1.2 Path Sensitization Model

The perturbation propagation (p-propagation) model allows capturing path sensitization in different applications. The main characteristic of the model is that the logic value assumed by each node is uncoupled from the path sensitization information associated with the node. We illustrated how path sensitization for test pattern generation and for timing analysis could be represented with adequate formulations of the model.

By formulating path sensitization in terms of the p-propagation model, the following advantages can be identified:

- The logical clause database is common to different path sensitization applications and the notion of pervasive implicate is introduced to justify sharing identified implicates *across* different circuit analysis tools.
- Most propagation reasoning is similar for different path sensitization applications. This also leads to the derivation of implicates that are valid *across* different path sensitization applications.
- Path sensitization specific pruning methods can be generalized and applied to other applications. This is the case, for example, of unique sensitization points, which were shown to be applicable to timing analysis, even though the concept on unique sensitization point was

originally developed for path sensitization in test pattern generation.

The precision of the model can be scaled and we describe several improvements for test pattern generation that extend the accuracy of models based on the D-calculus.

### 8.1.3 Search-Based Path Sensitization Algorithms

Using the ideas proposed for SAT algorithms, we developed LEAP, a generic path sensitization algorithm, and described how it can be applied to test pattern generation and timing analysis. The path sensitization algorithm can be configured in much the same way the SAT algorithm can. Once more, we emphasized conflict diagnosis, since previous path sensitization algorithms have seldom implemented conflict diagnosis techniques. Diagnosis of propagation conflicts is defined in terms of identifying *propagation implicates*, that describe conditions for blocking propagation.

The notion of *propagation* cut was introduced to formalize the evolution of the search process, the notion of *unique sensitization points* (USPs) and how to define propagation implicates. Several problem-specific pruning methods were developed, that included *direct* and *reverse value probing*, and *subleveling*.

## 8.2 Future Research Work

The problems addressed in this dissertation are known to be algorithmically hard and, consequently, we can always construct test cases for which the proposed algorithms perform poorly. The purpose of research work in these areas is to develop algorithmic techniques that reduce the number of cases where the algorithms perform poorly. We have described several ways on how this can be achieved for circuit analysis tasks, but additional work is necessary.

The following sections describe several pending research problems, as well as empirical validations, that are of interest in the continuation of the work described in this dissertation.

### 8.2.1 Satisfiability

We described a general search framework for SAT (GRASP), but emphasized its use in solving path sensitization problems. The individual analysis of the proposed techniques in the con-

text of SAT (without specifying target applications) is of interest. For example, it would be interesting to identify the best configuration of GRASP for the instances of SAT obtained by mapping circuit analysis problems into CNF (some examples of which were mentioned in Chapter I). Moreover, it would be interesting to empirically study how well GRASP performs on other instances of SAT, not necessarily related to circuit analysis.

**Logic Verification**

Among the applications of GRASP to circuit analysis tasks, logic verification is especially suitable given the formulation of the problem. We propose applying the conflict diagnosis techniques described in this dissertation to the logic verification problem. In particular, it would be of interest to evaluate whether implicate identification and controlled forms of advanced deduction engines could help inducing structure on the logic verification problem so that the application of conflict diagnosis methods could be facilitated. One possible approach is to replace nodes proved equivalent by a single copy and rearranging the clause database accordingly (by adapting ideas first proposed in [16, 100]). This solution constraints the number of implication paths, thus inducing structure that GRASP can exploit.

### 8.2.2 Path Sensitization Algorithm

The algorithmic model for path sensitization needs extending failure-driven assertions to the propagation dimension. As mentioned in earlier chapters, this type of failure-driven assertions poses algorithmic difficulties because the number of p-cuts can become significantly large. In addition, different types of p-cut interactions can exist, which makes it difficult to define an integrated procedure for maintaining p-cuts. The same problem arises in advanced deduction engines, where we restrict the sets of tested variable assignments to the logical dimension.

### 8.2.3 Test Pattern Generation

The experimental results obtained with TG-LEAP are promising, and motivate incorporating additional pruning ability into the tool. TG-LEAP was developed as a prototype and the implementation can be further optimized. An exhaustive analysis of the different pruning methods, on a benchmark set larger that the one currently available, may permit defining which pruning methods

are the most useful. The results we obtained suggest that unique sensitization points and failure-driven assertions are particularly useful, followed by conflict-directed backtracking. It may be interest to empirically study how useful constrained creation of implicates (logical and propagation) may help reducing the search on average problems.

In addition, we propose to study the usefulness of postprocessing engines for test pattern generation. In EST [70, 71] promising results were obtained with different formulations of solution caching. Since our procedure caches less information than that of EST, we expect that restricted solution caching can be of use in reducing the test sizes obtained with EST. Redundancy removal from solutions was originally proposed in this dissertation, and can be of use whenever conflicts are identified, since in this situation, some of the existing decision assignments can become redundant. Experimental evaluation of both solution processing techniques is proposed.

Finally, the p-propagation model can be automatically scaled to provide for different propagation reasoning precisions. We believe it would be of interest to study for a representative set of benchmark circuits, which formulation of the p-propagation model is best suited for test pattern generation in both test pattern generation time and test size.

### 8.2.4 Timing Analysis

The experimental results obtained with TA-LEAP fare well against those of other timing analysis tools [50, 52], even though several pruning methods have not been incorporated into TA-LEAP.

As with TG-LEAP, additional benchmarking of TA-LEAP is suggested. In addition, the implementation of a configurable tool, able to implement the whole range of proposed pruning methods might be particularly useful in ascertaining the best configuration from a practical standpoint.

TA-LEAP can perform poorly for some forms of regular circuits (e.g. some instances of carry-skip adders) as other search-based timing analysis tools do. Further insights into the problem formulation and the full implementation of the path sensitization algorithm may help curbing the difficulties faced by TA-LEAP with these types of circuits. For example, as mentioned in Chapter VII, it may be of interest to study the development of delay-based pruning techniques (by adapting

ideas first proposed in [119]), which augment structural and functional pruning methods with conditions on the maximum propagation delay to specific nodes in the circuit.

### 8.2.5 Other Applications

Several other circuit analysis tasks can benefit from the ideas described in this dissertation. Delay fault testing is another application that involves path sensitization [108, 159], and consequently, we propose studying the application of the p-propagation model and the proposed search algorithms to delay fault testing.

The analysis of sequential circuits is the next step where to try to apply the search algorithms proposed in this dissertation. In particular, we believe that path sensitization tasks may benefit from the proposed algorithmic techniques. At this level, several differences exist between path sensitization applications. Nevertheless, a few similarities also exist, since all path sensitization problems for sequential circuits must solve different formulations of the state reachability problem. For test pattern generation the existence of faults affects how fault activation is performed (and consequently how state reachability is solved). For timing analysis and for circuits with flip-flops, the path sensitization problem must only take into account that not all states of a finite state machine are reachable. For circuits with level-sensitive latches, it is not yet clear how the existence of uncertainty intervals at a combinational block inputs can affect the path sensitization problem.

Future generations of circuit analysis tools will necessarily have to manipulate some form of hierarchical circuit descriptions. We believe that algorithmic solutions for circuit analysis problems of hierarchically described circuits will have to involve some form of search procedure. In such a situation, we believe that the algorithmic framework proposed in this dissertation may be extended to the analysis of hierarchically described circuits, for verification as well as path sensitization problems.

# APPENDICES

# APPENDIX A

# FORMAL RESULTS ON SATISFIABILITY

This appendix includes the proofs for all formal results of Chapter III. We also include an analysis of the computational complexity of processing each decision level with GRASP.

## A.1 Soundness and Completeness

The purpose of this section is to prove the soundness and completeness of two configurations of GRASP. Theorem 3.1 (see page 64) establishes the soundness and completeness of GRASP configured with the basic deduction and diagnosis engines. Theorem 3.8 (see page 100) establishes the soundness and completeness of GRASP configured with the basic deduction engine and with a diagnosis engine that ensures a constant size clause database (i.e. `Diagnose_`*C*()).

In subsequent proofs involving `Diagnose`() the following configuration is assumed:

1. No clauses are subsumed or merged (i.e. REDUCE_DATABASE is false in Figure 3.12 on page 90).
2. No unique implication points are identified.
3. No implementation of iterated or multiple conflicts.

In addition, several proofs examine the effects of added clauses to the clause database. For a consistency function $\xi$, associated with an initial clause database $\varphi_i$, a modified clause database $\varphi$ is said to be *valid* if and only if $\xi|_{A} = \varphi|_{A}$.

The plain backtracking search algorithm is both sound and complete [97]. Thus, to prove that GRASP is also sound and complete, it is only necessary to prove that:

1. The basic deduction engine only implies assignments that are necessary for finding a solution to the query. This is guaranteed by Theorem 2.1 (see page 38).
2. Both conflict diagnosis engines do not affect either the soundness or completeness of the al-

gorithm. The proof of this statement is the main result of this section.

### A.1.1 Soundness

The following result establishes that the search algorithm is sound, i.e. any computed solution to a given query is indeed a solution to that query.

**Theorem A.1.** GRASP is sound.

**Proof:** By definition of clause database the original clause database can only be satisfied if the variable assignments are consistent, hence denoting a solution to a query. In addition, clauses can only be added to the clause database (with `Diagnose_`*C*() no clauses are added). Consequently, given an initial clause database $\varphi_i$ and the current clause database $\varphi$, then $\varphi_i \subseteq \varphi$. Consequently, a satisfying assignment for $\varphi$ must also be a satisfying assignment for $\varphi_i$.

Furthermore, from Theorem 2.1 on page 38, any variable assignment implied by Boolean constraint propagation is necessary for identifying a solution to a query, and any violated clause of $\varphi$ is identified by the procedure of Figure 2.6. Hence, any computed solution variable assignment for the current clause database, must indeed be a solution to the query. ■

Observe that the above proof holds independently of whether conflicting clauses are or not added to the clause database, and so it holds for GRASP configured with either `Diagnose`() or `Diagnose_`*C*().

### A.1.2 Completeness with `Diagnose()`

The proof that the search algorithm is complete hinges on the fact that any clause that is added to the clause database during the search process is an implicate of the consistency function $\xi$. In order to prove this fact, we will use induction on the number of conflicts found during the search and consequently on the number of conflicting clauses added to the clause database. As a result, we have to prove that the first conflicting clause is an implicate of $\xi$. This requires showing that if a partial variable assignment includes the conflicting assignment set associated with a given conflict, then a conflict must be identified. Using these results, we can then show that the non-chronological backtracking procedure does not skip potential solutions. Finally, the previous

results can be used to prove GRASP to be complete. Consequently, the proof is organized as follows:

1. Let $\boldsymbol{A}$ be the assignment set associated with a partial variable assignment and let $\boldsymbol{A}_{CS}$ be given by (3.6) on page 67 for some identified conflict. Then $\boldsymbol{A}_{CS} \subseteq \boldsymbol{A}$ implies that $\boldsymbol{A}$ causes a conflict.
2. The first conflicting clause is an implicate of $\xi$.
3. Every conflicting clause is an implicate of $\xi$.
4. Let $\beta_L$ be the backtracking decision level given by (3.17) (see page 86). Then a solution cannot be found by backtracking to a decision level $b$ such that $\beta_L < b \leq c$.
5. GRASP is complete.

We first show that the node assignments identified by $\boldsymbol{A}_{CS}$ (from (3.6)) imply a conflict. In order to prove this result, some additional definitions are required. Let $\boldsymbol{A}_{SG}$ denote the assigned variables of the subgraph of the implication graph assigned at decision level $c$. $\boldsymbol{A}_{SG}$ can be computed with $trace : \boldsymbol{V} \rightarrow 2^{\boldsymbol{V} \times \{0,1\}}$:

$$\boldsymbol{A}_{SG} = \bigcup_{(y, \nu(y)) \in \Sigma(\kappa)} trace(y) \tag{A.1}$$

where,

$$trace(x) = \begin{cases} (x, \nu(x)), & \text{if } \iota(x) = 0 \\ (x, \nu(x)) \cup \left[ \bigcup_{(y, \nu(y)) \in \Sigma(x)} trace(y) \right], & \text{otherwise} \end{cases} \tag{A.2}$$

The implication levels are used to partition $\boldsymbol{A}_{SG}$ as follows:

$$\boldsymbol{A}_{SG}[i] = \{(x, \nu(x)) | (x, \nu(x)) \in \boldsymbol{A}_{SG} \wedge \iota(x) = i\}, 0 \leq i \leq N \tag{A.3}$$

where the highest implication level can be no greater than the number of variables $N$. For each $\boldsymbol{A}_{SG}[i]$, let $\Omega[i]$ denote the set of clauses that imply the assignments included in $\boldsymbol{A}_{SG}[i]$. Given

the above definitions, the following properties can be established.

**Lemma A.1.** Let $(y, v_y) \in \boldsymbol{A}_{SG}[k], 0 \leq k \leq N$ be any assignment in $\boldsymbol{A}_{SG}$. In such a situation,

$$\forall((w, v_w) \in \boldsymbol{A}(y)), [(w, v_w) \in \boldsymbol{A}_{CS} \vee \exists(0 \leq j < k), ((w, v_w) \in \boldsymbol{A}_{SG}[j])] \tag{A.4}$$

**Proof:** Assume that a given assignment $(y, v_y)$ is in $\boldsymbol{A}_{SG}$, then from (A.1) and (A.2) the assignment of all its antecedents, assigned at decision level $c$, are also in $\boldsymbol{A}_{SG}$. If $(y, v_y)$ is contained in $\boldsymbol{A}_{SG}[k]$, then by definition of implication level, in (3.3), these assignments must be contained in sets $\boldsymbol{A}_{SG}[j]$, with $0 \leq j < k$. The assignments of the remaining antecedents, assigned at decision levels less than $c$, are contained in $\boldsymbol{A}_{CS}$ from (3.6) and (3.7). Hence (A.4) follows. ■

**Lemma A.2.** Assume a conflict such that $\boldsymbol{A}_{CS}$ is given by (3.6) and $\boldsymbol{A}_{SG}[k]$ is given by (A.3). Furthermore, assume a partial variable assignment $\boldsymbol{A}$, and let $\forall(1 \leq j < k), \boldsymbol{A}_{SG}[j] \subseteq \boldsymbol{A}$. Then, for each clause $\omega$ in $\Omega[k]$, either $\omega$ is satisfied, unsatisfied or it is a unit clause.

**Proof:** Given the definition of $\Omega[k]$, assuming (A.4), and since by hypothesis the condition $\forall(1 \leq j < k), \boldsymbol{A}_{SG}[j] \subseteq \boldsymbol{A}$ holds, then the following conclusions can be drawn from Lemma A.1. All literals of $\omega \in \Omega[k]$ are assigned with the possible exception of one literal $y^i$, whose assignment is implied by $\omega$. If $y$ is assigned, and $y = i$, then $\omega$ is unsatisfied and yields a conflict; if $y = \neg i$, then $\omega$ is satisfied. If $y$ is unassigned, then $\omega$ is a unit clause. ■

We can now use the previous results to show that given $\boldsymbol{A}_{CS}$, any partial variable assignment $\boldsymbol{A}$ causes a conflict if $\boldsymbol{A}_{CS} \subseteq \boldsymbol{A}$.

**Lemma A.3.** Assume a given conflict node $\kappa$ at decision level $c$. Let $\boldsymbol{A}_{CS}$ be given by (3.6) and $\boldsymbol{A}_{SG}$ be given by (A.1). In this situation, for any partial variable assignment $\boldsymbol{A}$, such that $\boldsymbol{A}_{CS} \subseteq \boldsymbol{A}$, then a conflict is detected.

**Proof:** Suppose that $\boldsymbol{A}_{SG} \subseteq \boldsymbol{A}$, since by hypothesis $\boldsymbol{A}_{CS} \subseteq \boldsymbol{A}$, then $\boldsymbol{A}_{CS} \cup \boldsymbol{A}_{SG} \subseteq \boldsymbol{A}$, and since $\boldsymbol{A}(\kappa) \subseteq \boldsymbol{A}_{CS} \cup \boldsymbol{A}_{SG}$, then a clause of the clause database is unsatisfied under $\boldsymbol{A}$, and a conflict is detected.

The next step is to show that if $\boldsymbol{A}_{CS} \subseteq \boldsymbol{A}$, then either $\boldsymbol{A}_{SG} \subseteq \boldsymbol{A}$ or a conflict is detected, and

hence a conflict is necessarily detected. We use induction of the implication level to show that for each implication level $i$, either $\boldsymbol{A}_{SG}[i] \subseteq \boldsymbol{A}$ or a conflict is identified. Consider $\boldsymbol{A}_{CS} \subseteq \boldsymbol{A}$.

*Basis step* ($k = 0$). Any node $x$ assigned to $v_x$ at implication level 0 is either asserted due to some clause or is a decision assignment. If $(x, v_x)$ is a decision assignment, then by definition $(x, v_x) \in \boldsymbol{A}_{CS}$. Otherwise $(x, v_x)$ is implied due to some clause $\omega$, such that the assignments $(y, v_y)$ of all its other literals $y^i$ are in $\boldsymbol{A}_{CS}$, again by definition of conflicting assignment set. Hence $(x, v_x)$ must be included in $\boldsymbol{A}_{SG}[0]$ or otherwise a conflict is identified. In any case $\boldsymbol{A}_{SG}[0] \subseteq \boldsymbol{A}$ holds or a conflict is identified.

*Induction hypothesis* ($k \leq m$). Assume that $\boldsymbol{A}_{SG}[k] \subseteq \boldsymbol{A}$ for $0 \leq k \leq m$.

*Induction step* ($k = m+1$). By Lemma A.2 and by the induction hypothesis, each clause $\omega$ in $\Omega[k]$ must either be satisfied, unsatisfied or be a unit clause. Note that $\omega$ cannot be a unit clause, since `Deduce()` would imply an assignment and satisfy $\omega$. If $\omega$ is unsatisfied a conflict is identified and the claim holds. If $\omega$ is satisfied, then the assignment $(y, v_y)$ implied by $\omega$ at implication level $k$ has been made. As a result, either a conflict is identified, or all assignments in $\boldsymbol{A}_{SG}[k]$ hold. Hence, $\boldsymbol{A}_{SG}[k] \subseteq \boldsymbol{A}$.

Consequently, we can conclude that if $\boldsymbol{A}_{CS} \subseteq \boldsymbol{A}$, then a conflict is detected. ■

The next step is to show that the first identified conflicting clause is an implicate of the consistency function $\xi$.

**Lemma A.4.** Assume a search process, a valid clause database and let $c$ be the current decision level. Further, let $x$ be the node assigned due to the most recent decision (i.e. $x = v_x$), and let the resulting implication sequence result in the first conflict. Then, the conflicting clause $\omega$ created with (3.8) is an implicate of $\xi$.

**Proof:** Let $\boldsymbol{A}_{CS}$ be the conflicting assignment set associated with the conflict (from (3.6)) and let $\omega$ be the conflicting clause created from $\boldsymbol{A}_{CS}$ with (3.8). Consequently, for some $\boldsymbol{A}$ if $\omega|_{\boldsymbol{A}} = 0$, then $\boldsymbol{A}_{CS} \subseteq \boldsymbol{A}$, by definition. In addition, $\boldsymbol{A}_{CS} \subseteq \boldsymbol{A}$ implies a conflict (i.e. $\xi|_{\boldsymbol{A}} = 0$) from Lemma A.3. Hence, $(\omega|_{\boldsymbol{A}} = 0) \Rightarrow (\xi|_{\boldsymbol{A}} = 0)$. Conversely, suppose $\boldsymbol{A}$ such that $\xi|_{\boldsymbol{A}} = 1$. Hence, $\boldsymbol{A}_{CS} \not\subset \boldsymbol{A}$, and so $\omega|_{\boldsymbol{A}} = 1$. As a result we can conclude $\omega|_{\boldsymbol{A}} \Rightarrow \xi|_{\boldsymbol{A}}$ is true for all $\boldsymbol{A}$, for which $\omega|_{\boldsymbol{A}} = 0$ or $\xi|_{\boldsymbol{A}} = 1$, and thus $\omega$ is an implicate of $\xi$. ■

The above result establishes that, under the assumption that the clause database is valid, the first conflicting clause is an implicate of $\xi$. We now prove that any clause added to the clause database is an implicate of $\xi$.

**Lemma A.5.** All clauses derived from conflicting assignment sets identified during the search process are implicates of the consistency function $\xi$.

**Proof:** We use induction on the number of conflicts $k$, identified during the search process, to show that every conflicting clause created after diagnosing a conflict is an implicate of $\xi$.

*Basis step* ($k = 1$). From Lemma A.4.

*Induction hypothesis* ($k = m$). The conflicting clause created after the $m^{\text{th}}$ conflict is an implicate of $\xi$.

*Induction step* ($k = m + 1$). If the $(m + 1)^{\text{th}}$ conflict results from a decision assignment, then Lemma A.4 guarantees that the identified conflicting clause is an implicate of $\xi$. Otherwise, the conflict results from an implication sequence triggered by an asserted variable $x$. Let $\omega_1$ be the conflicting clause associated with asserting $x$, and let $\omega_2$ be the conflicting clause identified from analyzing the $(m + 1)^{\text{th}}$ conflict. Further let $x^i$ be the literal of $\omega_1$ associated with $x$. Now define the clause,

$$\omega_3 = [\omega_2 - (\omega_1 - \{x^i\})] \cup \{\neg x^i\} \tag{A.5}$$

$\omega_3$ can be viewed as the conflicting clause that is created as a result of diagnosing the conflict caused by the decision assignment $x = \neg i$. Hence, from Lemma A.4 $\omega_3$ is an implicate of $\xi$ since the clause database is valid until the $m^{\text{th}}$ conflict. In addition, $\omega_2 = c(\omega_1, \omega_3, x)$, and hence it is also an implicate of $\xi$. (Observe that even though several nodes can be asserted at a given decision level, only one is associated with the decision variable of that level. The other assertions result from implicates of the clause database by the induction hypothesis.)

As a result, any conflicting clause given by (3.8) is an implicate of $\xi$. ■

The next step consists in showing that non-chronological backtracking does not skip any potential solutions to the query.

**Lemma A.6.** Let $\beta_L$ be computed with (3.17) and let $c$ be the current decision level, with $\beta_L < c$. In this situation, no solution to the query can be found by backtracking to a decision level $b$, such that $\beta_L < b \leq c$.

**Proof:** Let $\boldsymbol{A}_{CS}$ be the conflicting assignment set from (3.6). Hence, the associated conflicting clause $\omega$ is an implicate of the consistency function from Lemma A.5. Furthermore, from Theorem 3.4 on page 75, $\boldsymbol{A}_{\beta_L} \subseteq \boldsymbol{A}_c$ while the search process does not backtrack to decision level $\beta_L$. From (3.17) we can conclude that $\boldsymbol{A}_{CS} \subseteq \boldsymbol{A}_{\beta_L}$, and from Lemma A.3 $\xi|_{\boldsymbol{A}} = 0$ for all $\boldsymbol{A}_b$ such that $\boldsymbol{A}_{CS} \subseteq \boldsymbol{A}_{\beta_L} \subseteq \boldsymbol{A}_b$. Hence, a solution to the problem cannot be found until the search process backtracks to decision level $\beta_L$. ■

**Theorem A.2.** GRASP configured with `Diagnose`() is complete.

**Proof:** From [97] the plain backtracking search algorithm is known to be complete. Hence, it is necessary to show that (1) all implications derived by boolean constraint propagation are necessary conditions for a solution to the satisfiability problem to be found; and that (2) `Diagnose`() will not cause the search process to skip any possible solutions to the query. (1) follows from Lemma 2.1; (2) follows from Lemma A.6, which ensures that any decision level that is skipped by the search process cannot contribute to finding a solution to the query. Consequently, the search algorithm is complete. ■

Finally, Theorem 3.1 (see page 64) follows from Theorem A.1 and Theorem A.2.

### A.1.3 Completeness with `Diagnose_`*C*`()`

To prove that GRASP configured with based on `Diagnose_`*C*() is complete, we invoke the results of Section A.1.2. In addition, we prove that, whenever a conflict is diagnosed, the conflicting clause derived from the union of the level conflicting assignment sets is an implicate of the consistency function. This fact guarantees that the computed backtracking decision level is correct in the sense that no potential solutions are skipped. The completeness result then follows from Lemma A.6. As in the proof of Lemma A.5, we use induction on the number of conflicts to show that each created conflicting clause is an implicate of the consistency function.

**Lemma A.7.** Assume a search process, a valid clause database, let $c$ be the current decision level and let a conflict be identified. Further, define the following conflicting clause:

$$\omega = \left\{ x^{\nu(x)} \middle| (x, \nu(x)) \in \bigcup_{0 \leq i \leq c} \boldsymbol{A}_{CS}[i] \right\} \tag{A.6}$$

where $\boldsymbol{A}_{CS}[i]$ is given by (3.20) on page 98 and by applying `Diagnose_C`(). Then, $\omega$ is an implicate of the consistency function $\xi$.

**Proof:** As with the proof of Lemma A.5, we use induction on the number of conflicts and start by showing that the first clause created with (A.6) is an implicate of $\xi$. Note, however, that this follows from Lemma A.4. The next step is to prove the induction step. Assume that after the $i^{\text{th}}$ conflict, (A.6) is an implicate of $\xi$. Then we have to show that after the $(i+1)^{\text{th}}$ conflict, (A.6) yields an implicate of $\xi$. If the $(i+1)^{\text{th}}$ results from a decision assignment, then Lemma A.4 applies and (A.6) is an implicate of the consistency function. Otherwise, we can reason as in the proof of Lemma A.5, and construct a clause $\omega_3$ with (A.5) which then guarantees (A.6) to be an implicate of the consistency function. Thus, after each conflict, (A.6) always denotes an implicate of $\xi$. ■

**Theorem A.3.** GRASP configured with `Diagnose_C`() is complete.

**Proof:** The backtracking decision level computed with (3.21) is always derived from an implicate of $\xi$ due to Lemma A.7, which guarantees that Lemma A.6 holds. From Theorem A.2 the results follows. ■

**Perspective**

Even though there has been extensive work on non-chronological backtracking search algorithms in artificial intelligence, completeness proofs have seldom been established. In [19], M. Bruynooghe proposed a convincing argument for the completeness of a search algorithm for constraint satisfaction problems which implemented non-chronological backtracking but did not propagate constraints (in GRASP this would correspond to having no deduction engine). This same argument was used by M. Shanahan and R. Southwick in 1989 [148, pp. 65-66] to prove that the same algorithm and some of its variations were indeed complete. A different proof for the same

algorithm is given in [143], but under the assumption that nogoods are identified and recorded.

Although backjumping[1] was proposed by J. Gaschnig in 1979 [66], only recently an abstract definition of backjumping was shown to be complete by M. Ginsberg [68].

The proofs given in this dissertation are distinct from the aforementioned proofs. Specifically, our proofs consider the derivation of implication sequences, while the others do not. In addition, the proof for `Diagnose`() entails the construction and application of conflicting clauses which, in contrast with nogoods [54, 161], are not necessarily specified in terms of decision assignments.

## A.2 Time and Space Complexity

In this section we establish results regarding the computational complexity of processing each decision level during the search process. The clause database is assumed to be derived from a combinational circuit.

The time complexity to find a solution to an instance of SAT using GRASP is in the worst-case clearly proportional to the product of the worst-case time required to process a given decision level ($T_L$) and the exponential of the number of primary inputs (i.e. $2^{|PI|}$). This result can be concluded from the fact that the search algorithm implicitly enumerates all possible primary input assignments and each decision involves some processing (i.e. $T_L$). (This result is also expected, because SAT is an NP-complete decision problem [34, 35, 65].) If the clause database is permitted to grow without bound, then the space complexity if clearly exponential in the number of variables, because in the worst-case an exponential number of implicates can be identified and added to the clause database. As a result, the time complexity to process a given decision is in the worst-case exponential in number of variables. For a first query this leads to $T_L = O(2^N)$. For a sequence of queries, an upper bound on the growth of the clause database is $T_L = O(3^N)$. (Both bounds are somewhat loose, but suggest the computational effort that may be involved in processing a given decision level. A more accurate bound is given below.)

In practice, however, only a limited amount of computational resources can be allocated to finding the solution of a given query. Most often the bounds on these resources are specified by a

[1] Backjumping is a non-chronological backtracking procedure specifically developed for CSPs [66, 133].

bound on the amount of search (i.e. a maximum number of backtracks) and by bounds on how to update the clause database (i.e. maximum number and size of implicates that can be added to the database and what operations can be performed on those implicates). As a result, we can analyze the complexity of GRASP under the following constraints:

**Assumption A.1.** In the analysis of GRASP the following *resource constraints* are assumed:

1. The maximum number of backtracks to solve a given query is $B$.
2. After solving a query, the clause database is reset to its original structure (e.g. for a combinational circuit the clause database is given by the conjunction of clauses associated with the consistency function of each gate). (In general, we may relax this constraint by allowing a constant number of conflicting clauses to be kept in the clause database.)
3. The addition of a new clause to the clause database is not subject to subsumption or merging operations.

Given these constraints, we can improve the complexity bounds on the space required to solve SAT and on the time required to process each decision. We note that these new bounds are obtained under the assumption that if solving a query requires more than $B$ backtracks, then the algorithm declares itself incapable of finding a solution to that particular query, because the query requires more computational resources than the algorithm is allowed to spend on any given query. Furthermore, we assume an initial clause database for which $\|\varphi\| = O(N)$. This is the case with clause databases derived from combinational circuits.

At any stage of the search process, the worst-case number of backtracks already executed is $O(B)$, given the bound on the number of backtracks. As a result, the size of the clause database can have increased with clauses that contribute with at most $O(N \cdot B)$ literals, due to the creation of two conflicting clauses associated with each backtrack. Hence, the number of literals in the clause database is at most $O(\|\varphi\| + N \cdot B) = O(N \cdot B)$, which also denotes the bound on the space required by the algorithm. At any decision level, the time complexity for the derivation of implication sequences is linearly related to the size of the clause database, and thus it is $O(N \cdot B)$ given the above assumptions. Diagnosing each conflict requires computing the associated conflicting assignment set (using (3.6)), and consequently it is necessary to recursively identify antecedent

assignments. Each clause in the clause database can be associated with at most one antecedent assignment. Hence, the recursive identification of antecedent assignments is bounded by $O(N \cdot B)$, i.e. the worst-case size of the clause database. This immediately implies that a conflicting assignment set, at any decision level, is computed in $O(N \cdot B)$ time. Furthermore, since there can be at most $B$ backtracks, the total run time is in the worst-case $O(N \cdot B^2)$. The previous analysis supports the following:

**Theorem A.4.** Under Assumption A.1, and for clause databases where initially $\|\varphi\| = O(N)$, the space complexity of GRASP is $O(N \cdot B)$ and the time complexity at each decision level is $T_L = O(N \cdot B)$, where $B$ is the maximum number of allowed backtracks, and $N$ is the number of variables. Moreover, the worst-case running time of GRASP is $O(N \cdot B^2)$.

Consequently, the size of the clause database and the time required to process each decision level are bounded by the size of the problem and by the number of allowed backtracks.

## A.3 Diagnosis Engines

In this section we prove results of Chapter III regarding conflict analysis and diagnosis engines. The results are associated with unique implication points (UIPs), maintenance of the clause database and identification of multiple conflicts.

**Theorem A.5. (Theorem 3.2 on page 69)** Let a conflict be identified at decision level $c$, and let $U = \{(u_1, \nu(u_1)), \ldots, (u_n, \nu(u_n))\}$ denote the set of UIPs. Then the isolated assignment of each UIP is a sufficient condition for causing the same conflict.

**Proof:** To prove that a UIP indeed represents a sufficient condition for the same conflict to be detected, we just have to take into consideration the definition of dominator [166]. All elements in $U$ are dominators of the subgraph defined by $\boldsymbol{A}_{SG}$ (given in (A.1)). Let $(u, \nu(u))$ be a UIP, and let $\iota(u) = i$. Further, let $(y, v_y) \in \boldsymbol{A}_{SG}[k]$, $i < k \leq N$. Then for any $k > i$, (A.4) can be re-written as follows:

$$\forall((w, v_w) \in \boldsymbol{A}(y)), [(w, v_w) \in \boldsymbol{A}_{CS} \vee \exists(i \leq j < k), ((w, v_w) \in \boldsymbol{A}_{SG}[j])] \tag{A.7}$$

Suppose that (A.7) does not hold because for some assignment $(y, v_y)$ with $(\iota(y) = m) \wedge (m > i)$ there exists $(w, v_w) \in \boldsymbol{A}(y) \wedge [(w, v_w) \in \boldsymbol{A}_{SG}[j] \wedge (j < i)]$ . But then $(u, v)$ would no longer be a dominator since the subgraph at decision level $c$ would contain an edge between $(w, v_w)$ and $(y, v_y)$ (by definition of implication graph); a contradiction. Since (A.7) must hold, then each UIP must trigger an implication sequence leading to the same conflict. ■

**Theorem A.6. (Theorem 3.3 on page 74)** With the definitions of $\omega_1$, $\omega_2$ and $\omega_3$ given above (see page 74), $\omega_1 \cdot \omega_2 \leftrightarrow \omega_3$. Clause $\omega_3$ is an implicate of the consistency function $\xi$. Moreover, $\omega_1$ and $\omega_2$ can be removed from the clause database if $\omega_3$ is added to the clause database.

**Proof:** Suppose a partial variable assignment $\boldsymbol{A}$ such that $\omega_3\big|_{\boldsymbol{A}} = 1$. Then,

$$[(\omega_1 - \{l_{1,i}\})\big|_{\boldsymbol{A}} = 1] \wedge [(\omega_2 - \{l_{2,i}\})\big|_{\boldsymbol{A}} = 1]$$

Consequently, $\omega_1\big|_{\boldsymbol{A}} = 1$ and $\omega_2\big|_{\boldsymbol{A}} = 1$. Hence, $\omega_1 \cdot \omega_2 = 1$. On the other hand, if $\omega_3\big|_{\boldsymbol{A}} = 0$, then,

$$(\omega_1 - \{l_{1,i}\})\big|_{\boldsymbol{A}} = (\omega_2 - \{l_{2,i}\})\big|_{\boldsymbol{A}} = 0$$

and because $l_{1,i}$ is the complement of $l_{2,i}$, either $\omega_1\big|_{\boldsymbol{A}} = 0$ or $\omega_2\big|_{\boldsymbol{A}} = 0$. Hence, $\omega_1 \cdot \omega_2 = 0$ and $\omega_1 \cdot \omega_2 \leftrightarrow \omega_3$. As a result, $\omega_1$ and $\omega_2$ can be removed from the clause database if $\omega_3$ is added to the clause database. ■

**Theorem A.7. (Theorem 3.7 on page 96)** Diagnosis of multiple conflicts, where each conflict is separately diagnosed, and where $\|\varphi\| = O(N)$, has a lower bound on the worst-case run time of $\Omega(N^2)$.

**Proof:** The proof is based on constructing a specific circuit structure for which a given decision assignment triggers an implication sequence leading to $O(N)$ conflicts, such that the separate diagnosis of each conflict requires $O(N)$ time. Hence the worst-case lower bound of $\Omega(N^2)$ follows. Consider the circuit shown in Figure A.1. Let $m$ be a constant and let $M = \lceil N/m \rceil$. Let $x_1$

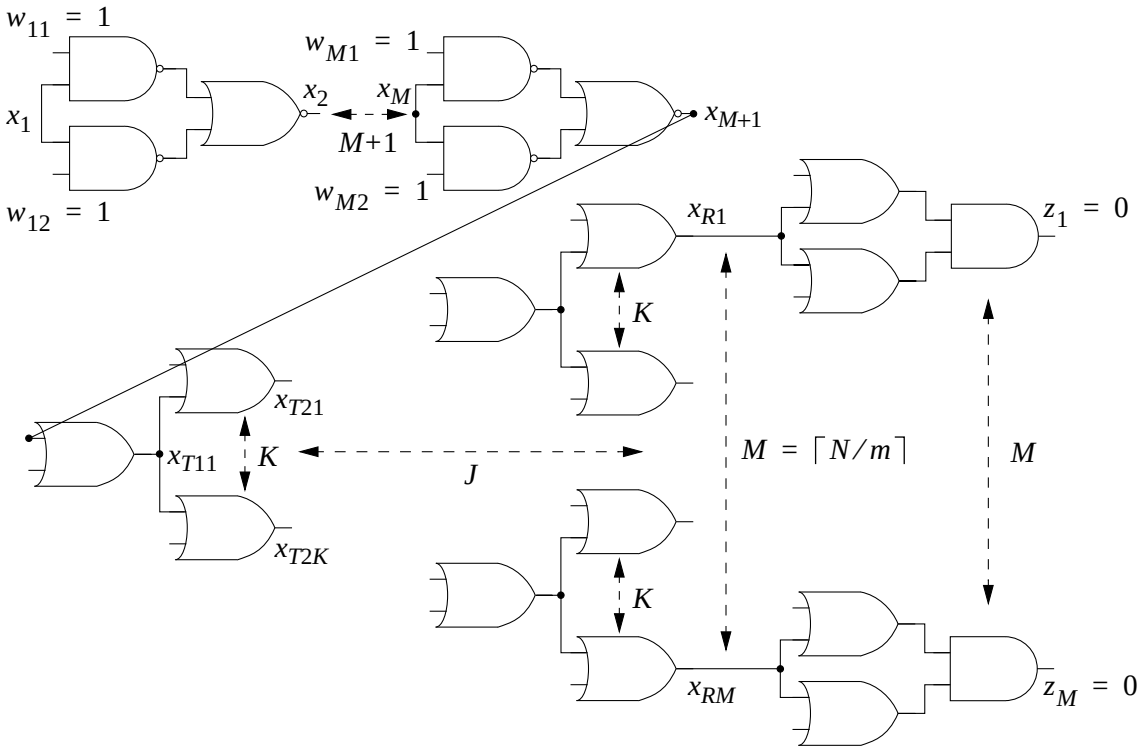



**Figure A.1:** Lower-bound on multiple conflict diagnosis

be assigned value 0. As a result, nodes $x_2$ through $x_{M+1}$ become assigned value 1 (through an implication sequence of size $M$). Afterwards, each gate output in the tree of OR gates is assigned value 1, and consequently $x_{R1}$ through $x_{RM}$ are assigned value 1. The implication of any $x_{Ri}$ to 1 causes a conflict with $z_i$ assigned value 0, for every $1 \le i \le M$. Since $M = \lceil N/m \rceil$, we have $O(N)$ conflicts. Diagnosing each conflict separately requires traversing $O(N)$ nodes (i.e. $J$ nodes in the OR tree and $\lceil N/m \rceil$ nodes from $x_{M+1}$ back to $x_1$). Hence, diagnosing $O(N)$ conflicts requires $O(N^2)$ time. To conclude the proof, we need to show that the number of circuit nodes is $\Theta(N)$, and thus the circuit can be designed with $N$ nodes given adequate constants.

The generation of $x_i$ from $x_{i-1}$ requires 5 nodes. Hence to generate $x_{M+1}$ from $x_1$ a total of $5 \cdot \lceil N/m \rceil + 1$ nodes are required. For the OR tree, to generate $\lceil N/m \rceil$ outputs it is necessary to have $\lfloor \log_K(\lceil N/m \rceil) \rfloor \le J \le \lceil \log_K(\lceil N/m \rceil) \rceil$ which contributes $L$ nodes such that,

$$L = \left\lceil 2 \times \sum_{j=0}^{J} K^j \right\rceil + 1 = \left\lceil 2 \times \frac{K^J - 1}{K - 1} + 1 \right\rceil$$

and given the bounds on $J$,

$$2 \times \frac{K^{\lfloor \log_K(\lceil N/m \rceil) \rfloor} - 1}{K-1} + 1 \leq L \leq 2 \times \frac{K^{\lceil \log_K(\lceil N/m \rceil) \rceil} - 1}{K-1} + 1$$

Finally, each module linking $x_{Ri}$ to $z_i$ requires 5 nodes for a total of $5 \cdot \lceil N/m \rceil$ nodes. As a result, with $M = \lceil N/m \rceil = O(N)$, the total number of nodes $T$ is such that,

$$10 \times M + 2 \times \frac{K^{\lfloor \log_K(M) \rfloor} - 1}{K-1} + 2 \leq T \leq 10 \times M + 2 \times \frac{K^{\lceil \log_K(M) \rceil} - 1}{K-1} + 2$$

which means that $T = \Theta(N)$. Hence, if $K$ is fixed, then we just have to choose $m$ and $N$ such that the above condition is satisfied. Since there are $\Theta(N)$ nodes and each gate has two inputs, the fanin constraints are clearly satisfied. Thus, it follows that a lower bound on the worst-case time for processing a decision level is $\Omega(N^2)$ if multiple conflicts are identified and each conflict is separately diagnosed. ■

## A.4 Deduction Engines

**Theorem A.8. (Theorem 3.5 on page 82)** Let $\varphi$ be a clause database. Let $|\boldsymbol{I_C}|$ be the current number of vertices of the implication graph, and let $N - |\boldsymbol{I_C}|$ identify the total number of unassigned nodes. Then, the worst-case run time of `Deduce_`$k$(), assuming that SIMPLIFY_$\varphi_k$ does not hold, is bounded by:

$$O\left(\binom{N - |\boldsymbol{I_C}|}{k} \cdot (\|\varphi\| \cdot 2^k + N \cdot k \cdot (3^k)^2)\right) \tag{A.8}$$

The worst-case run time of `Deduce_`$k$,$R$() is bounded by,

$$O\left(\binom{N - |\boldsymbol{I_C}|}{k} \cdot \left[\left[\|\varphi\| + N \cdot \binom{N - |\boldsymbol{I_C}|}{k} \cdot 3^k\right] \cdot 2^k + N \cdot k \cdot (3^k)^2\right] \cdot \left[\binom{N - |\boldsymbol{I_C}|}{k} \cdot 3^k\right]\right) \tag{A.9}$$

For both procedures the worst-case space is bounded by,

$$O\left(\|\varphi\| + N \cdot \binom{N - \left|\boldsymbol{I}_{\boldsymbol{C}}\right|}{k} \cdot 3^k\right) \tag{A.10}$$

and the space growth if bounded by $O(N \cdot 3^N)$.

**Proof:** Let us consider `Deduce_`$k$() as described in Figure 3.8 on page 80. A decision assignment is assumed to have been made, and the first invocation to `Deduce`() decides whether the implication sequence thus created leads to a conflict. Assuming that $\left|\boldsymbol{I}_{\boldsymbol{C}}\right|$ denotes the size of the implication graph (i.e. the number of assigned variables), then set $\Gamma$ is the set of all combinations of $k$ nodes out of $N - \left|\boldsymbol{I}_{\boldsymbol{C}}\right|$ unassigned variables, i.e. $|\Gamma| = \binom{N - \left|\boldsymbol{I}_{\boldsymbol{C}}\right|}{k}$. Let $\|\varphi\|$ denote the current size of the clause database (i.e. the total number of literals in $\varphi$). In such a situation, deriving an implication sequence with `Deduce`() requires time $O(\|\varphi\|)$. (Note that the size of the clause database remains unchanged until all sets in $\Gamma$ are analyzed.) Since there are $2^k$ node assignments associated with each set of nodes $\gamma$ of size $k$, an upper bound on the time for processing a set $\gamma$ in $\Gamma$ is $O(\|\varphi\| \cdot 2^k)$. The effect of last phase, for prime implicate generation, can be obtained by adapting (2.17) on page 45; each clause has $O(N)$ literals but consensus operations are restricted to $k$ variables. As a result, an upper bound on the run time of the $k$-consistency procedure is:

$$O\left(\binom{N - \left|\boldsymbol{I}_{\boldsymbol{C}}\right|}{k} \cdot (\|\varphi\| \cdot 2^k + N \cdot k \cdot (3^k)^2)\right) \tag{A.11}$$

and the size of the resulting clause database is $O\left(\|\varphi\| + N \cdot \binom{N - \left|\boldsymbol{I}_{\boldsymbol{C}}\right|}{k} \cdot 3^k\right)$, because $O(3^k)$ prime implicates can be generated for each subset $\gamma$, and each implicate has size $O(N)$, out of a total of $\binom{N - \left|\boldsymbol{I}_{\boldsymbol{C}}\right|}{k}$ subsets. Clearly, the size growth if bounded by $O(N \cdot 3^N)$, which denotes an upper bound on the size of the clause database.

With respect to $k$-consistency with relaxation, two additional factors constraint the run time. First, the procedure iterates over `Deduce_`$k$() while more implicates are derived. Second, the size

of the clause database changes each time Deduce_*k*() is invoked. A valid upper bound to the run is then to consider both effects for derivation of implications. Moreover the phase for the generation of prime implicates has to be considered[2]:

$$O\left(\binom{N-|\boldsymbol{I}_C|}{k}\cdot\left[\left[\|\varphi\|+N\cdot\binom{N-|\boldsymbol{I}_C|}{k}\cdot 3^k\right]\cdot 2^k+N\cdot k\cdot(3^k)^2\right]\cdot\left[\binom{N-|\boldsymbol{I}_C|}{k}\cdot 3^k\right]\right) \qquad \text{(A.12)}$$

where term $\left[\binom{N-|\boldsymbol{I}_C|}{k}\cdot 3^k\right]$ denotes the worst-case number of iterations, and the remaining terms account for the run time of Deduce_*k*() for each iteration. Deduce_*k*,*R*() exhibits the same bound on space growth that Deduce_*k*() does. Moreover, the size growth is bounded by $O(N\cdot 3^N)$, which denotes an upper bound on the size of the clause database. ■

**Theorem A.9. (Theorem 3.6 on page 84)** For each deduction engine Deduce_*k*(), with fixed *k*, it is always possible to construct a clause database for which the identification of all implications requires a deduction engine Deduce_*m*(), with $m > k$.

**Proof:** The proof consists in developing an example circuit (or generic clause database) that for any *k* can be scaled in such a way that *k*-consistency does not derive all possible logical consequences. One such example circuit is shown in Figure A.2. We assume the deduction engine to be Deduce_*k*(). It is clear that the objective $z = 1$ is not satisfiable. since $y_{m,\,1}$ and $y_{m,\,2}$ always assume the same logic value. Next, we show that Deduce_*k*() does not derive all logical consequences given the initial clause database.

The key idea is to show that for any assignment of size *k* to variables $y_{1,\,1}$ through $y_{1,\,k}$ will not create any unique assignments on the variables $y_{1,\,k+1}$ through $y_{1,\,2k}$, and consequently, no set of conflicts will be detected such that the objective is shown to be unsatisfiable.

We start by claiming that for any assignment set $\boldsymbol{A}_1$, with respect to the variables in $\{y_{1,\,1}, y_{1,\,2}, \ldots, y_{1,\,k}\}$, there exists more than one assignment set $\boldsymbol{A}_2$, with respect to the variables in $\{y_{1,\,k+1}, y_{1,\,k+2}, \ldots, y_{1,\,2k}\}$, that has the same parity as $\boldsymbol{A}_1$. Indeed, there are $2^{k-1}$ such

2. (A.12) is a loose upper bound because we assume the worst-case scenario for both the number of iterations and the size of the clause database; as future research work, a tighter bound ought to be derived.

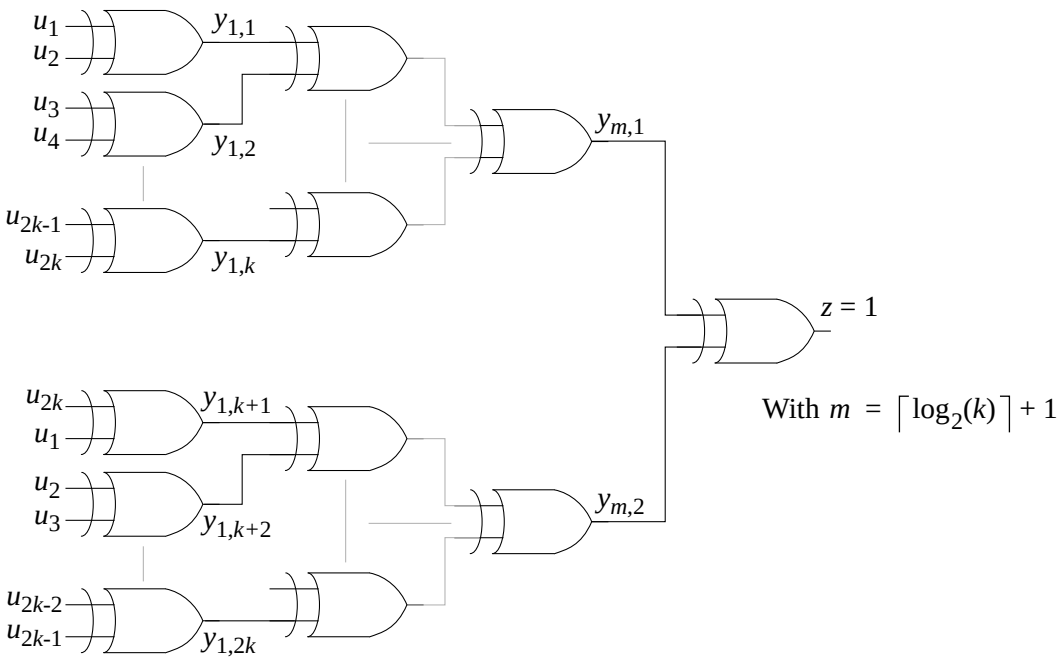



**Figure A.2:** *k*-consistency not able to prove the assignment unsatisfiable

assignments, since half of the $2^k$ assignment sets $\boldsymbol{A}_2$ have the same parity as $\boldsymbol{A}_1$, and the other half have opposite parity. Our second claim is that for each of these possible assignments $\boldsymbol{A}_2$, there exists at least one consistent assignment to the variables in $\{u_1, u_2, \ldots, u_{2k}\}$. Define the following variables:

$$\begin{aligned} v_1 &= y_{1,1} \oplus y_{1,k+1} \\ &\ldots \\ v_k &= y_{1,k} \oplus y_{1,2k} \end{aligned} \tag{A.13}$$

Then, by simple algebraic manipulation, one possible solution for $\{u_1, u_2, \ldots, u_{2k}\}$ is given by:

$$\begin{aligned} u_{2k} &= 1 \\ u_2 &= 1 \oplus [v_2 \oplus \ldots \oplus v_k] \\ u_1 &= y_{1,1} \oplus u_2 \\ &\ldots \\ u_{2j} &= 1 \oplus [v_1 \oplus \ldots \oplus v_{j-1} \oplus v_{j+1} \oplus \ldots \oplus v_k] \\ u_{2j-1} &= y_{1,2j-1} \oplus u_{2j} \end{aligned} \tag{A.14}$$

where the last two rows correspond to the general solution, $j \geq 1$. The above result proves that, given an assignment of the $k$ variables in $\{y_{1,1}, y_{1,2}, \ldots, y_{1,k}\}$, there are an exponential number of possible assignments to the variables in $\{y_{1,k+1}, y_{1,k+2}, \ldots, y_{1,2k}\}$, that yield consistent assignments to the variables in $\{u_1, u_2, \ldots, u_{2k}\}$. Consequently, for any assignments to $\{y_{1,1}, y_{1,2}, \ldots, y_{1,k}\}$, a conflict is not detected.

It is also necessary to guarantee that backward implications and other derived implicates will not imply assignments to the variables in $\{y_{1,k+1}, y_{1,k+2}, \ldots, y_{1,2k}\}$, given a set of assignments to the variables in $\{y_{1,1}, y_{1,2}, \ldots, y_{1,k}\}$. To show that this requirement holds we consider the case of $y_{m,1}$ and $y_{m,2}$. Application of `Deduce_2()` yields the implicates $(y_{m,1} + y_{m,2})$ and $(\neg y_{m,1} + \neg y_{m,2})$, which require $y_{m,1}$ and $y_{m,2}$ to always assume opposite logic values. Now consider the previous set of variables in the circuit, i.e. $\{y_{m-1,1}, y_{m-1,2}, y_{m-1,3}, y_{m-1,4}\}$. `Deduce_4()` is required to identify implicates that disallow assignments of even parity (since a conflict with $z$ would then be identified). Hence, the implicates on $y_{m,1}$ and $y_{m,2}$ do not contribute for constraining the variables at level $m-1$. A similar analysis guarantees that implicates at level 2 will not constrain the variables at level 1. It follows that `Deduce_`$k$`()` will not identify a conflict from the original objective. We further node that the example is scalable. For any inference engine `Deduce_`$k$`()`, the circuit is created given $k$, and hence `Deduce_`$k$`()` will not prove the query to be unsatisfiable without search. ■

## A.5 Postprocessing Engines

**Lemma A.8. (Lemma 3.1 on page 119)** Assume a solution to a query identified by an assignment set $\boldsymbol{A}$, and let the associated node justification graph $J_G$ be defined. For each decision level $j$, define $\boldsymbol{T}_j$ by (3.28) on page 118. In such a situation, any node $y$, such that $J(y, x)$ holds for $x$ assigned at a decision level greater than $j$, either $y$ is also assigned at a decision level greater than $j$ or is such that $\eta(y) \in \boldsymbol{T}_j$.

**Proof:** Assume otherwise. Then there would exist $s$ assigned at a decision level less than or equal to $j$ such that $\eta(s)$ would not be in $\boldsymbol{T}_j$. Let $x$ be assigned at a decision level greater than $j$, such that $J(s, x)$ holds. Then by definition of predicate $J$, there must be an edge between $\eta(s)$ and $\eta(y)$. But by definition of $\boldsymbol{T}_j$, then $\eta(s)$ must be in $\boldsymbol{T}_j$; a contradiction. ■

**Theorem A.10. (Theorem 3.9 on page 120)** If one of the conditions identified by (3.31) on page 120 holds, then $\boldsymbol{A_S}'$ given by (3.30) is a solution to the query.

**Proof:** This result basically follows from Lemma A.8. Assume that condition $C(j)$ (see (3.31) on page 120) is satisfied. Let $\boldsymbol{T}_j$ denote the level cut associated with $C(j)$. For the assignment set of the previous query, $\boldsymbol{A_S}$, consider the decision assignments after decision level $j$. Then, by Lemma A.8 and for any assigned node $x$, with $\delta(x) > j$, any node $y$, such that $J(y, x)$ holds, either is assigned at decision levels greater than $j$ or $\eta(y)$ is contained in $\boldsymbol{T}_j$. After $K - j$ additional decisions, a solution to the query is identified. Let those decision assignments be represented by $\boldsymbol{A}$.

Now consider the current query. At some decision level $i$ condition $C(j)$ is matched. Hence, the assignments associated with components in $\boldsymbol{T}_j$ are matched. Furthermore, assignments at decision levels greater than $j$ for the previous query are not contradicted. Consequently, in $K - j$ additional decisions we can identify a solution to the query. This solution consists of the union of the current assignment set $\boldsymbol{A}_c$ with $\boldsymbol{A}$. ■

# APPENDIX B

# FORMAL RESULTS ON PATH SENSITIZATION

This appendix includes the proofs for all formal results on path sensitization that are related with the validity of the p-propagation model when applied to test pattern generation and timing analysis. In addition, we use the results for GRASP to argue the soundness and completeness of LEAP.

## B.1 Test Pattern Generation

**Theorem B.1. (Theorem 5.1 on page 181)** Given a SSF fault in a combinational circuit, a test ***T*** detects the fault if and only if under the p-propagation model ***T*** sets at least one primary output to p-*T*.

**Proof:** The proof relates p-status values with D-calculus values. Since a test ***T*** detects a fault under the D-calculus if and only if the fault is detectable with ***T*** [141], then under the p-propagation model a primary output becomes p-*T* for a test ***T*** if and only if the fault is detectable with ***T***. Simple gates are assumed.

(*If part*) Let ***T*** detect a fault under the D-calculus and let *z* be a primary output that assumes value D or $\overline{D}$. Further let *s* be the site of the fault. The goal is to show that under the p-propagation model primary output *z* would be set to p-*T*. Identify the set $\Pi$ of all nodes *y*, in the transitive fanin of *z* and in the transitive fanout of *s* including *s*, such that *y* is connected to *z* by a partial path where every node assumes value D or $\overline{D}$ and sort in reverse topological the nodes in $\Pi$. Now we can traverse in reverse level order the elements of $\Pi$, starting at *z* and terminating at *s*. For each visited node *w*, let us assume that all its fanin nodes with value D or $\overline{D}$ can be set to p-*T*. Under this assumption, *B*(*w*) does not hold, and *P*(*w*) holds. Consequently. we can tentatively set *w* to p-*T*,

under the assumption that all fanin nodes assuming value D or $\overline{D}$ can also be set to p-*T*. Eventually *s* is reached, which would be p-*T* by initialization of the p-propagation model. Hence under ***T*** all nodes in Π would be set to p-*T* and the fault would be detected.

(*Only if part*) The above reasoning immediately applies, but now we establish conditions for a node to be conditionally set to D or $\overline{D}$. Consequently, if under ***T*** a primary output becomes p-*T*, then that output would assume value D or $\overline{D}$ under ***T***. ■

**Corollary B.1. (Corollary 5.1 on page 181)** A sound and complete search algorithm, based on the p-propagation model, computes a test ***T*** for a given fault if and only if such test exists.

**Proof:** Since from Theorem B.1 a test ***T*** detects a fault if and only if under the p-propagation model ***T*** sets at least one primary output to p-*T*, a sound and complete search algorithm eventually enumerates ***T*** and so a solution is found if and only if a solution exists. ■

## B.2 Timing Analysis

**Theorem B.2. (Theorem 6.1 on page 214)** A combinational circuit contains a floating-mode sensitizable path of delay no less than Δ, for a test ***T***, if and only if under the p-propagation model such test ***T*** sets a primary output to p-*T*.

**Proof:** For circuit delay computation in timing analysis, the effect of delay information must be considered. Hence the proof hinges on the conditions required for a primary output node stabilizing with delay no less than Δ.

(*If part*) Let us assume that under ***T*** a primary output *z* stabilizes with propagation delay $\Delta_C \geq \Delta$. Let *y* be the fanin node of *z* such that *z* stabilizes as a direct consequence of *y* stabilizing. By definition of the p-propagation model, then if *y* is p-*T* then *z* must also be p-*T*, since *B*(*z*) does not hold and *P*(*z*) holds. Otherwise, the propagation delay to *z* would be less than Δ. We can create a path ***P*** by traversing from *z* to a primary input *s* such that each node $w_i$ in the path defines the propagation delay to fanout node $w_{i+1}$ also in the path. For each such node B($w_{i+1}$) cannot hold, since again the propagation delay to $w_{i+1}$ would not be extensible to a propagation delay no less than Δ. Primary input *s* is eventually visited. It must be assigned and thus be set to p-*T* be definition of the p-propagation model. Consequently all nodes in ***P*** are set to p-*T*, which also includes *z*.

(*Only if part*) Just observe that by definition a primary output *z* can only become p-*T* if its delay estimate is no less than Δ. Since delay estimation *DTo* captures floating mode operation (see page 210), then a floating-mode sensitizable path is defined whenever a primary output *z* is set to p-*T* under a test ***T***. ■

**Corollary B.2. (Corollary 6.1 on page 214)** A sound and complete search algorithm, based on the p-propagation model, computes a test ***T*** that sensitizes a path with delay no less than Δ if and only if such test exists.

**Proof:** Since from Theorem B.2 a test ***T*** sensitizes a path of delay no less than Δ if and only if under the p-propagation model ***T*** sets at least one primary output to p-*T,* a sound and complete search algorithm eventually enumerates ***T*** and so a solution is found if and only if a solution exists. ■

## B.3 Soundness and Completeness

The soundness of the search algorithm for path sensitization follows from the results of the previous sections for each application. If a solution is found, then it is indeed a solution to the path sensitization problem.

We argue that the search algorithm is complete using the results of Appendix A, where it is proved that every clause derived from conflict diagnosis is an implicate of the logical consistency function. In Appendix A this fact is key to prove that the search algorithm for SAT is complete.

For path sensitization the same reasoning applies. Every clause (p-clause) derived with conflict diagnosis is an implicate of the logical (propagation) consistency function $\xi$ ($\xi_\pi$) and so it is an implicate of the path sensitization consistency function $\xi_{PS}$. We note that every p-clause is derived from a known conflict and identifies sufficient conditions for that conflict to be identified. Whenever a p-clause is unsatisfied, then propagation of a perturbation to a primary output becomes blocked and a propagation conflict is detected. Consequently, every p-clause derived with `Propagation_Diagnose`() is necessarily and implicate of $\xi_\pi$, which guarantees correct com-

puted backtracking decision levels. It can thus be concluded that the search algorithm for path sensitization is complete.

# APPENDIX C

# CARRY-SKIP ADDERS

The purpose of this appendix is to describe the organization and delay properties of carry-skip adders (CSAs) that were used in Chapter VII to evaluate TA-LEAP. The basic structure of a carry-skip adder is shown in Figure C.1, and it follows the organization of CSAs described in [91]. The number of bits is $B$, the (constant) number of bits per block is $K$ and the number of blocks is $M$ (i.e. $B = K \cdot M$). For all gates a unit delay is assumed. If $M > 1$, the longest topological path is defined by,

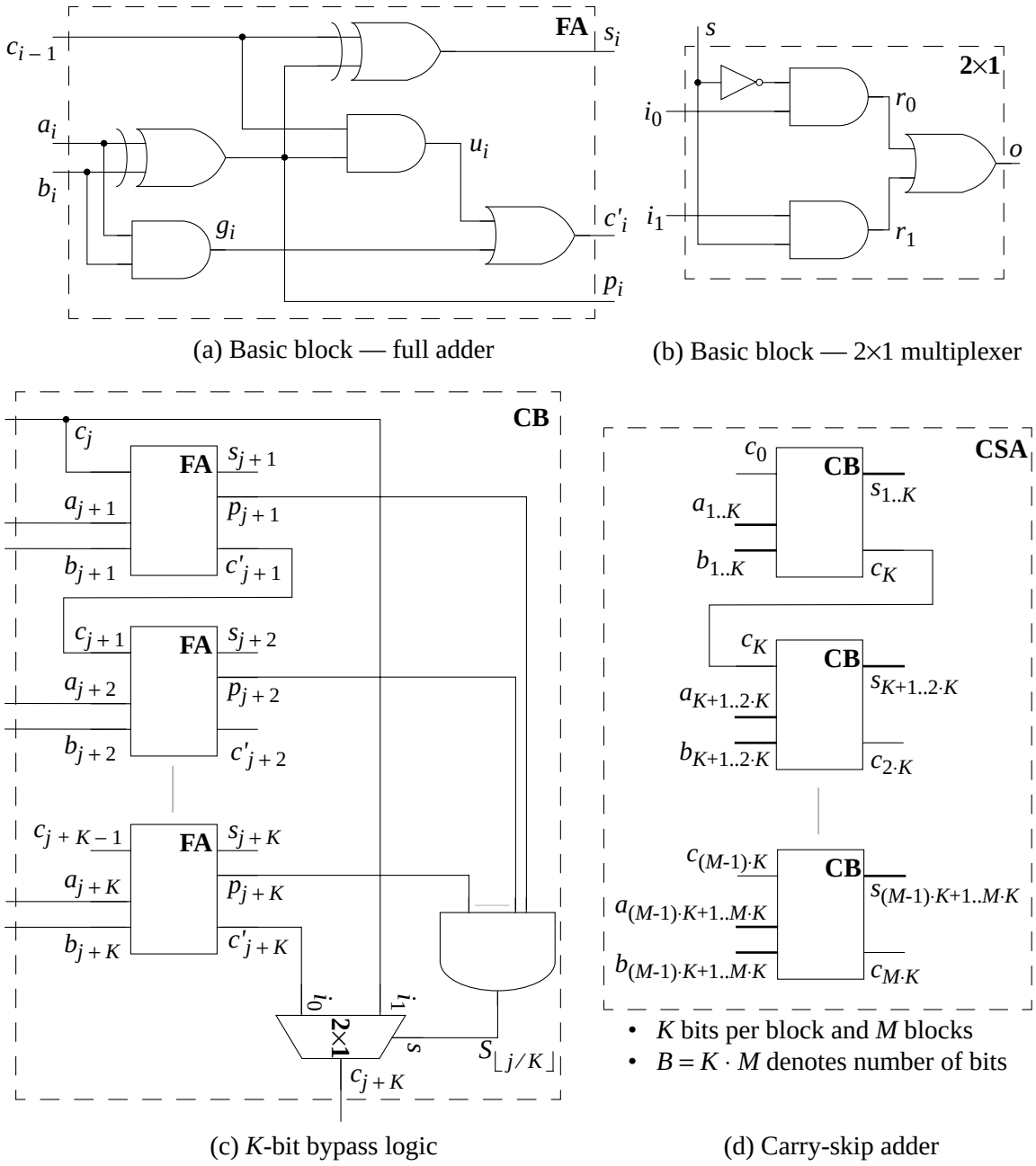
$$\begin{aligned} \text{LTP} &= [3 + (K-1) \times 2] + (K \times 2 + 2) \times (M-1) \\ &= K \times M \times 2 + M \times 2 + 1 \\ &= 2 \times B + 2 \times M + 1 \end{aligned} \tag{C.1}$$

Otherwise it is defined by,

$$\text{LTP} = 3 + 2 \cdot K \tag{C.2}$$

In both cases, the first block contributes with delay $3 + 2 \cdot (K - 1)$, due to propagation from inputs $a_1$ or $b_1$. In addition, the delay contribution of each block is defined by the delay from the input carry line to the output carry line of the block added to the delay of the multiplexer.

The longest sensitizable path cannot involve propagation along a sequence of carry bits larger than $K$–1. Hence each carry bit $c_j$ propagates to $c_{j+K}$ only along the bypass logic, which corresponds to a delay of 2 time units. In such a situation, the longest sensitizable path delay is defined by propagation from $a_0$ to $c_K$ (for example), then across the bypass logic of $M$–2 modules and finally from $c_{(M-1)\cdot K}$ to the last sum bit of the last module, i.e. $s_{M \cdot K}$. If $M > 1$, the circuit delay $\Delta$ for the carry-skip adder is given by:




(a) Basic block — full adder

(b) Basic block — 2×1 multiplexer

(c) $K$-bit bypass logic

(d) Carry-skip adder

**Figure C.1:** Description of a carry-skip adder (all gates with unit delay)

$$\begin{aligned} \Delta_{\boldsymbol{C}} &= [3 + K \cdot 2] + (M - 2) \cdot 2 + [K \times 2 - 2 + 1] \\ &= 2 \times [2 \times K + M - 1] \end{aligned} \tag{C.3}$$

otherwise it is given by,

$$\Delta_{\boldsymbol{C}} = 3 + 2 \cdot K \tag{C.4}$$

# BIBLIOGRAPHY


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

[2] S. B. Akers, "A Logic System for Fault Test Generation," *IEEE Transactions on Computers*, vol. 25, no. 6, pp. 620-630, June 1976.

[3] S. B. Akers, "Binary Decision Diagrams," *IEEE Transactions on Computers*, vol. 27, no. 6, pp. 509-516, June 1978.

[4] A. d'Anjou, M. Graña, F. J. Torrealdea and M. C. Hernandez, "Solving Satisfiability via Boltzmann Machines," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 15, no. 5, pp. 514-521, May 1993.

[5] P. Ashar, S. Malik and S. Rothweiler, "Functional Timing Analysis Using ATPG," in *Proceedings of EDAC*, pp. 506-510, 1993.

[6] B. Aspvall, M. F. Plass and R. E. Tarjan, "A Linear-Time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas," *Information Processing Letters*, vol. 8, no. 3, pp. 121-123, March 1979.

[7] R. I. Bahar, H. Cho, G. D. Hachtel, E. Macii and F. Somenzi, "Timing Analysis of Combinational Circuits Using ADD's," in *Proceedings of European Design and Test Conference*, pp. 625-629, 1994.

[8] J. Benkoski, E. Vanden Meersch, L. Claesen and H. De Man, "Efficient Algorithms for Solving the False Path Problem in Timing Verification," in *Proceedings of International Conference on Computer-Aided Design*, pp. 44-47, 1987.

[9] J. Benkoski, E. Vanden Meersch, L. Claesen and H. De Man, "Timing Verification Using Statically Sensitizable Paths," *IEEE Transactions on Computer-Aided Design*, vol. 9, no. 10, pp. 1073-1083, October 1990.

[10] A. Billionnet and A. Sutter, "An Efficient Algorithm for the 3-Satisfiability Problem," *Operations Research Letters*, vol. 12, pp. 29-36, July 1992.

[11] J. R. Bitner and E. M. Reingold, "Backtrack Programming Techniques," *Communications of the ACM*, vol. 18, no. 11, pp. 651-656, November 1975.

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

[13] A. Blake, *Canonical Expressions in Boolean Algebra*, Ph.D. Dissertation, Department of Mathematics, University of Chicago, 1937.

[14] G. Boole, *An Investigation of the Laws of Thought*, Macmillan and Co., 1854 (Reprinted by Dover Publications, 1958).

[15] D. Brand and V. S. Iyengar, "Timing Analysis Using Functional Analysis," *IEEE Transactions on Computers*, vol. 37, no. 10, pp. 1309-1314, October 1988.

[16] D. Brand, "Verification of Large Synthesized Designs," in *Proceedings of International Conference on Computer-Aided Design*, pp. 534-537, 1993.

[17] 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.

[18] F. M. Brown, *Boolean Reasoning: The Logic of Boolean Equations*, Kluwer Academic Publishers, 1990.

[19] M. Bruynooghe, "Solving Combinatorial Search Problems by Intelligent Backtracking," *Information Processing Letters*, vol. 12, no. 1, pp. 36-39, February 1981.

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

[21] M. Bruynooghe, "Intelligent Backtracking Revisited," in *Computational Logic: Essays in Honor of Alan Robinson*, pp. 166-177, J.-L. Lassez and G. Plotkin (Eds.), MIT Press, 1991.

[22] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, vol. 35, no. 8, pp. 677-691, June 1986.

[23] C. W. Cha, W. E. Donath and F. Özgüner, "9-V Algorithm for Test Pattern Generation of Combinational Digital Circuits," *IEEE Transactions on Computers*, vol. 27, no. 3, pp. 193-200, March 1978.

[24] 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.

[25] A. K. Chandra and G. Markowsky, "On the Number of Prime Implicants," *Discrete Mathematics*, vol. 24, no. 1, pp. 7-11, 1978.

[26] S. J. Chandra and J. H. Patel, "Experimental Evaluation of Testability Measures for Test Generation," *IEEE Transactions on Computer-Aided Design*, vol. 8, no. 1, pp. 93-97, January 1989.

[27] M. S. Chandrasekhar, J. P. Privitera and K. W. Conradt, "Application of Term Rewriting Techniques to Hardware Design Verification," in *Proceedings of the 24th Design Automation Conference*, pp. 277-282, 1987.

[28] C.-L. Chang and R. C.-T. Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, 1973.

[29] H. Chang and J. A. Abraham, "CHAN: An Efficient Critical Path Analysis Algorithm," in *Proceedings of EDAC*, pp. 444-448, 1993.

[30] H. Chang and J. A. Abraham, "VIPER: An Efficient Vigorously Sensitizable Path Extractor," in *Proceedings of the 30th Design Automation Conference*, pp. 112-117, 1993.

[31] H. C. Chen and D. H.-C. Du, "Path Sensitization in Critical Path Problem," in *Proceedings of the ACM Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU)*, 1990.

[32] H. C. Chen and D. H.-C. Du, "Path Sensitization in Critical Path Problem," *IEEE Transactions on Computer-Aided Design*, vol. 12, no. 2, pp. 196-207, February 1993.

[33] W. T. Cheng, "SPLIT Circuit Model for Test Generation," in *Proceedings of the 25th Design Automation Conference*, pp. 96-101, 1988.

[34] S. A. Cook, "The Complexity of Theorem-Proving Procedures," in *Proceedings of the Third Annual Symposium on Theory of Computing*, pp. 151-158, 1971.

[35] T. H. Cormen, C. E. Leiserson and R. L. Rivest, *Introduction to Algorithms*, MIT Press, 1990.

[36] O. Coudert, J. C. Madre and H. Fraisse, "A New ViewPoint on Two-Level Logic Minimization," in *Proceedings of the 30th Design Automation Conference*, pp. 625-630, 1993.

[37] 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.

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

[39] M. Davis, G. Logemann and D. Loveland, "A Machine Program for Theorem-Proving," *Communications of the ACM*, vol. 5, pp. 394-397, July 1962.

[40] M. D. Davis and E. J. Weyuker, *Computability, Complexity, and Languages*, Academic Press, 1983.

[41] R. Dechter, "Learning While Searching in Constraint-Satisfaction Problems," Technical Report CSD-860049, University of California at Los Angeles, June 1986.

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

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

[44] J. de Kleer, "Problem Solving with the ATMS," *Artificial Intelligence*, vol. 28, pp. 197-224, 1986.

[45] J. de Kleer, "A Comparison of ATMS and CSP Techniques," in *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 290-296, 1989.

[46] J. de Kleer, "Exploiting Locality in a TMS," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 264-271, 1990.

[47] J. de Kleer, "An Improved Incremental Algorithm for Generating Prime Implicates," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 780-785, 1992.

[48] S. Devadas, "Optimal Layout Via Boolean Satisfiability," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 294-297, 1989.

[49] S. Devadas, K. Keutzer and S. Malik, "Delay Computation in Combinational Logic Circuits: Theory and Algorithms," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 176-179, 1991.

[50] S. Devadas, K. Keutzer, S. Malik and A. Wang, "Computation of Floating Mode Delay in Combinational Circuits: Practice and Implementation," in *Proceedings of the ACM Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU)*, 1992.

[51] S. Devadas, K. Keutzer, S. Malik and A. Wang, "Certified Timing Verification and the Transition Delay of a Logic Circuit," in *Proceedings of the 29th Design Automation Conference*, pp. 549-555, 1992.

[52] S. Devadas, K. Keutzer, S. Malik and A. Wang, "Computation of Floating Mode Delay in Combinational Circuits: Practice and Implementation," *IEEE Transactions on Computer-Aided Design*, vol. 12, no. 12, pp. 1924-1936, December 1993.

[53] W. F. Dowling and J. H. Gallier, "Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae," *Journal of Logic Programming*, vol. 3, pp. 267-284, 1984.

[54] J. Doyle, "A Truth Maintenance System," *Artificial Intelligence*, vol. 12, pp. 231-272, 1979.

[55] D. H.-C. Du, S. H. C. Yen and S. Ghanta, "On the General False Path Problem in Timing Analysis," in *Proceedings of the 26th Design Automation Conference*, pp. 555-560, 1989.

[56] B. Dunham, R. Fridshal and G. L. Sward, "A Non-Heuristic Program for Proving Elementary Logical Theorems," in *Proceedings of the International Conference on Information Processing*, pp. 282-285, 1959.

[57] C. Elkan, "Conspiracy Numbers and Caching for Searching And/Or Trees and Theorem-Proving," in *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 341-346, 1989.

[58] S. Even, A. Itai and A. Shamir, "On the Complexity of Timetable and Multicommodity Flow Problems," *SIAM Journal on Computing*, vol. 5, no. 4, pp. 691-703, December 1976.

[59] J. Franco, "On the Probabilistic Performance of Algorithms for the Satisfiability Problem," *Information Processing Letters*, vol. 23, pp. 103-106, August 1986.

[60] K. D. Forbus and J. de Kleer, *Building Problem Solvers*, MIT Press, 1993.

[61] E. C. Freuder, "Synthesizing Constraint Expressions," *Communications of the ACM*, vol. 21, no. 11, pp. 958-966, November 1978.

[62] 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.

[63] H. Fujiwara and S. Toida, "The Complexity of Fault Detection Problems for Combinational Logic Circuits," *IEEE Transactions on Computers*, vol. 31, no. 6, pp. 555-560, June 1982.

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

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

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

[67] M. L. Ginsberg and W. D. Harvey, "Iterative Broadening," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 216-220, 1990.

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

[69] M. L. Ginsberg, *Essentials of Artificial Intelligence*, Morgan Kaufman Publishers, 1993.

[70] J. Giraldi and M. L. Bushnell, "EST: The New Frontier in Automatic Test-Pattern Generation," in *Proceedings of the 27th Design Automation Conference*, pp. 667-672, 1990.

[71] 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.

[72] 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.

[73] A. Goldberg, P. Purdom and C. Brown, "Average Time Analyses of Simplified Davis-Putnam Procedures," *Information Processing Letters*, vol. 15, no. 2, pp. 72-75, September 1982.

[74] S. W. Golomb and L. D. Baumert, "Backtrack Programming," *Journal of the Association for Computing Machinery*, vol. 12, no. 4, pp. 516-524, October 1965.

[75] S. Greenbaum, A. Nagasaka, P. O'Rorke and D. Plaisted, "Comparison of Natural Deduction and Locking Resolution Implementations," in *Proceedings of the 6th Conference on Automated Deduction*, pp. 159-171, 1982.

[76] J. Gu, "Efficient Local Search for Very Large-Scale Satisfiability Problem," ACM SIGART Bulletin, vol. 3, no. 1, pp. 8-12, January 1992.

[77] J. Gu, "Local Search for Satisfiability (SAT) Problem," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 23, no. 4, pp. 1108-1129, July/August 1993.

[78] J. P. Hayes, "On the Properties of Irredundant Logic Networks," *IEEE Transactions on Computers*, vol. 25, no. 9, pp. 884-892, September 1976.

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

[80] L. Henschen and L. Wos, "Unit Refutations and Horn Sets," *Journal of the Association for Computing Machinery*, vol. 21, no. 4, pp. 590-605, October 1974.

[81] R. B. Hitchcock, Sr., "Timing Verification and the Timing Analysis Program," in *Proceedings of the 19th Design Automation Conference*, pp. 594-603, 1982.

[82] J. N. Hooker, "Generalized Resolution and Cutting Planes," *Annals of Operations Research*, vol. 12, no. 1-4, pp. 217-239, 1988.

[83] V. M. Hrapcenko, "Depth and Delay in a Network," *Soviet Math. Dokl.*, vol. 19, no. 4, pp. 1006-1009, 1978.

[84] O. H. Ibarra and S. K. Sahni, "Polynomially Complete Fault Detection Problems," *IEEE Transactions on Computers*, vol. 24, no. 3, pp. 242-249, March 1975.

[85] A. Ivanov and V. K. Agarwal, "Dynamic Testability Measures for ATPG," *IEEE Transactions on Computer-Aided Design*, vol. 7, no. 5, pp. 598-608, May 1988.

[86] K. Iwama, "CNF Satisfiability Test by Counting and Polynomial Average Time," *SIAM Journal on Computing*, vol. 18, no. 2, pp. 385-391, April 1989.

[87] R. Jacoby, P. Moceyunas, H. Cho and G. Hachtel, "New ATPG Techniques for Logic Optimization," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 548-551, 1989.

[88] J. Jaffar and M. J. Maher, "Constraint Logic Programming: A Survey," *Journal of Logic Programming*, vol. 19/20, pp. 503-581, 1994.

[89] N. D. Jones and W. T. Laaser, "Complete Problems for Deterministic Polynomial Time," *Theoretical Computer Science*, vol. 3, pp. 105-117, 1977.

[90] R. M. Karp, "Reducibility Among Combinatorial Problems," in *Complexity of Computer Computations*, pp. 85-103, R. E. Miller and J. W. Thatcher (Eds.), Plenum Press, New York, 1972.

[91] K. Keutzer, S. Malik and A. Saldanha, "Is Redundancy Necessary to Reduce Delay?," *IEEE Transactions on Computer-Aided Design*, vol. 10, no. 4, pp. 427-435, April 1991.

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

[93] T. I. Kirkpatrick and N. R. Clark, "PERT as an Aid to Logic Design," *IBM Journal of Research and Development*, vol. 10, no. 2, March 1966, pp. 135-141.

[94] S. C. Kleene, *Mathematical Logic*, John Wiley & Sons, 1967.

[95] W. Kneale and M. Kneale, *The Development of Logic*, Clarendon Press, 1962.

[96] D. E. Knuth, *Sorting and Searching*, volume 3 of *The Art of Computer Programming*, Addison-Wesley, 1973.

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

[98] R. E. Korf, "Depth-First Iterative Deepening: An Optimal Admissible Tree Search," *Artificial Intelligence*, vol. 27, pp. 97-109, 1985.

[99] V. Kumar, "Algorithms for Constraint-Satisfaction Problems: A Survey," *AI Magazine*, vol. 13, pp. 32-43, Spring 1992.

[100] W. Kunz, "HANNIBAL: An Efficient Tool for Logic Verification Based on Recursive Learning," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 538-543, 1993.

[101] 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.

[102] 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.

[103] W. K. C. Lam, R. K. Brayton and A. L. Sangiovanni-Vincentelli, "Circuit Delay Models and Their Exact Computation Using Timed Boolean Functions," in *Proceedings of the 30th Design Automation Conference*, pp.128-134, 1993.

[104] T. Larrabee, "Efficient Generation of Test Patterns Using Boolean Difference," in *Proceedings of the International Test Conference*, pp. 795-801, 1989.

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

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

[107] D. H. Lehmer, "The Machine Tools of Combinatorics," in A*pplied Combinatorial Mathematics*, pp. 5-31, E. F. Beckenbach (Ed.), John Wiley and Sons, 1964.

[108] C. J. Lin and S. M. Reddy, "On Delay Fault Testing in Logic Circuits," *IEEE Transactions on Computer-Aided Design*, vol. 6, no. 5, pp. 694-703, January 1987.

[109] A. Lioy, "Adaptive Backtrace and Dynamic Partitioning Enhance APTG," in *Proceedings of the International Conference on Computer Design*, pp. 62-65, 1988.

[110] D. W. Loveland, *Automated Theorem Proving: A Logical Basis*, North-Holland, 1978.

[111] A. K. Mackworth, "Consistency in Network of Relations," *Artificial Intelligence*, vol. 8, pp. 99-118, 1977.

[112] S. Malik, A. R. Wang, R. K. Brayton, A. Sangiovanni-Vincentelli, "Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 6-9, 1988.

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

[114] R. Marlett, "An Effective Test Generation System for Sequential Circuits," in *Proceedings of the 23th Design Automation Conference*, pp. 250-256, 1986.

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

[116] D. A. McAllester, "Truth Maintenance," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 1109-1116, 1990.

[117] P. C. McGeer and R. K. Brayton, "Efficient Algorithms for Computing the Longest Viable Path in a Combinational Network," in *Proceedings of the 26th Design Automation Conference*, pp. 561-567, 1989.

[118] P. C. McGeer and R. K. Brayton, *Integrating Functional and Temporal Domains in Logic Design: The False Path Problem and its Implications*, Kluwer Academic Publishers, 1991.

[119] P. C. McGeer, A. Saldanha, R. K. Brayton and A. L. Sangiovanni-Vincentelli, "Delay Models and Exact Timing Analysis," in *Logic Synthesis and Optimization*, pp. 167-189, T. Sasao (Ed.), Kluwer Academic Publishers, 1993.

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

[121] P. C. McGeer, J. Sanghavi, R. K. Brayton and A. Sangiovanni-Vincentelli, "ESPRESSO-SIGNATURE: A New Exact Minimizer for Logic Functions," in *Proceedings of the 30th Design Automation Conference*, pp. 618-624, 1993.

[122] T. M. McWilliams, "Verification of Timing Constraints on Large Digital Systems," in *Proceedings of the 17th Design Automation Conference*, pp. 139-147, 1980.

[123] H. B. Min and W. A. Rogers, "Search Strategy Switching: An Alternative to Increased Backtracking," in *Proceedings of the International Test Conference*, pp. 803-811, 1989.

[124] M. Minoux, "LTUR: A Simplified Linear-Time Unit Resolution Algorithm for Horn Formulae and Computer Implementation," *Information Processing Letters*, vol. 29, no. 1, pp. 1-12, September 1988.

[125] B. Monien and E. Speckenmeyer, "Solving Satisfiability in less than $2^n$ Steps," *Discrete Applied Mathematics*, vol. 10, pp. 287-295, 1985.

[126] U. Montanari, "Networks of Constraints: Fundamental Properties and Applications to Picture Processing," *Information Sciences*, vol. 7, pp. 95-132, 1974.

[127] B. Nadel, "Constraint Satisfaction Algorithms," *Computational Intelligence*, vol. 5, pp. 188-224, 1989.

[128] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, 1984.

[129] S. Perremans, L. Claesen and H. De Man, "Static Timing Analysis of Dynamically Sensitizable Paths," in *Proceedings of the 26th Design Automation Conference*, pp. 568-573, 1989.

[130] T. Pietrzykowski and S. Matwin, "Exponential Improvement of Efficient Backtracking," in *Proceedings of the 6th Conference on Automated Deduction*, pp. 223-239, 1982.

[131] D. A. Plaisted, "Mechanical Theorem Proving," in *Formal Techniques in Artificial Intelligence, A Sourcebook*, pp. 269-320, R. B. Banerji (Ed.), North-Holland, 1990.

[132] D. A. Plaisted and S. Greenbaum, "A Structure-Preserving Clause Form Translation," *Journal of Symbolic Computation*, vol. 2, no. 3, pp. 293-304, September 1986.

[133] P. Prosser, "Hybrid Algorithms for the Constraint Satisfaction Problem," *Computational Intelligence*, vol. 9, no. 3, pp. 268-299, August 1993.

[134] P. W. Purdom Jr., C. A. Brown and E. L. Robertson, "Backtracking with Multi-Level Dynamic Search Rearrangement," *Acta Informatica*, vol. 15, no. 2, pp. 99-113, 1981.

[135] P. W. Purdom Jr., "Solving Satisfiability with less Searching," *IEEE Transaction on Pattern Analysis and Machine Intelligence*, vol. 6, no. 4, pp. 510-513, July 1984.

[136] R. Puri and J. Gu, "A Modular Partitioning Approach for Asynchronous Circuit Synthesis," in *Proceedings of the 31st Design Automation Conference*, pp. 63-69, 1994.

[137] W. V. Quine, "The Problem of Simplifying Truth Functions," *The American Mathematical Monthly*, vol. 59, no. 8, pp. 521-531, October 1952.

[138] W. V. Quine, "A Way to Simplify Truth Functions," *The American Mathematical Monthly*, vol. 62, no.9, pp. 627-631, November 1955.

[139] W. V. Quine, "On Cores and Prime Implicants of Truth Functions," *The American Mathematical Monthly*, vol. 66, no. 9, pp. 755-760, November 1959.

[140] J. A. Robinson, "A Machine Oriented Logic Based on the Resolution Principle," *Journal of the Association for Computing Machinery*, vol. 12, no. 1, pp. 23-41, January 1965.

[141] J. P. Roth, "Diagnosis of Automata Failures: a Calculus and a Method," *IBM Journal of Research and Development*, vol. 10, no. 4, pp. 278-291, July 1966.

[142] J. P. Roth, "Hardware Verification," *IEEE Transactions on Computers*, vol. 26, no. 12, pp. 1292-1294, December 1977.

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

[144] 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.

[145] 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.

[146] B. Selman, H. Levesque and D. Mitchell, “A New Method for Solving Hard Satisfiability Problems,” in *Proceedings of the National Conference on Artificial Intelligence*, pp. 440-446, 1992.

[147] B. Selman and H. Kautz, “Domain-Independent Extensions to GSAT: Solving Large Structured Satisfiability Problems,” in *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 290-295, 1993.

[148] M. Shahahan and R. Southwick, *Search, Inference and Dependencies in Artificial Intelligence*, Ellis Horwood, 1989.

[149] J. P. M. Silva, K. A. Sakallah and L. M. Vidigal, “FPD — An Environment for Exact Timing Analysis,” in *Proceedings of the International Conference on Computer-Aided Design*, pp. 212-215, 1991.

[150] J. P. M. Silva and K. A. Sakallah, “Sensitization Networks for Accurate Timing Analysis,” in *Proceedings of the ACM Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU)*, 1993.

[151] J. P. M. Silva and K. A. Sakallah, “A Comparison of Path Sensitization Criteria for Timing Analysis,” in *Proceedings of the ACM Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU)*, 1993.

[152] J. P. M. Silva and K. A. Sakallah, “Concurrent Path Sensitization in Timing Analysis,” in *Proceedings of EURO-DAC*, pp. 196-199, 1993.

[153] J. P. M. Silva and K. A. Sakallah, “An Analysis of Path Sensitization Criteria,” in *Proceedings of the International Conference on Computer Design*, pp. 68-72, 1993.

[154] J. P. M. Silva and K. A. Sakallah, “Search-Space Pruning Heuristics in Path Sensitization for Test Pattern Generation,” Technical Report CSE-TR-178-93, University of Michigan, October 1993.

[155] 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.

[156] J. P. M. Silva and K. A. Sakallah, “Efficient and Robust Test-Generation Based Timing Analysis,” in *Proceedings of the International Symposium on Circuits and Systems*, pp. 303-306, 1994.

[157] H. Simonis and M. Dincbas, "Propositional Calculus Problems in CHIP," in *Constraint Logic Programming: Selected Research*, pp. 269-285, F. Benhamou and A. Colmerauer (Eds.), MIT Press, 1993.

[158] T. Skolem, "Über die Mathematische Logik," *Norsk Matematisk Tidsskrift*, vol. 10, 125-142, 1928 (English Translation in [171]).

[159] G. L. Smith, "Model for Delay Faults Based Upon Paths," in *Proceedings of the International Test Conference*, pp. 342-349, 1985.

[160] T. J. Snethen, "Simulator-Oriented Fault Test Generator," in *Proceedings of the 14th Design Automation Conference*, pp. 88-93, 1977.

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

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

[163] M. E. Stickel, "Resolution Theorem Proving," *Annual Review of Computer Science*, vol. 3, pp. 285-316, 1988.

[164] T. Strzemecki, "Polynomial-Time Algorithms for Generation of Prime Implicants," *Journal of Complexity*, vol. 8, no. 1, pp. 37-63, March 1992.

[165] Y. Tanaka, "A Dual Algorithm for the Satisfiability Problem," *Information Processing Letters*, vol. 37, pp. 85-89, January 1991.

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

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

[168] P. Tison, "Generalization of Consensus Theory and Application to the Minimization of Boolean Functions," *IEEE Transactions on Electronic Computers*, vol. 16, no. 4, pp. 446-456, August 1967.

[169] E. Tsang, *Foundations of Constraint Satisfaction*, Academic Press, 1993.

[170] G. S. Tseitin, "On the Complexity of Derivation in Propositional Calculus," in *Studies in Constructive Mathematics and Mathematical Logic*, Part II, pp. 115-125, A. O. Silenko (Ed.), 1968.

[171] J. van Heijenoort (Ed.), *From Frege to Gödel: A Source Book of Mathematical Logic, 1879-1931*, Harvard University Press, 1967.

[172] F. Vlach, "A Tautology Checker for VLSI Applications that uses Rule-Based Simplification and Directed Backtracking," Technical Report N-90-010, Department of Computer Science, University of North Texas, May 1990.

[173] F. Vlach, “Simplification in a Satisfiability Checker for VLSI Applications,” *Journal of Automated Reasoning*, vol. 10, pp. 115-136, 1993.

[174] J. A. Waicukauski, P. A. Shupe, D. J. Giramma and A. Matin, “ATPG for Ultra-Large Structured Designs,” in *Proceedings of the International Test Conference*, pp. 44-51, 1990.

[175] R. J. Walker, “An Enumerative Technique for a Class of Combinatorial Problems,” in *Proceedings of the Symposium in Applied Mathematics*, vol 10, pp. 91-94, 1960.

[176] R. Wei and A. L. Sangiovanni-Vincentelli, “PROTEUS: A Logic Verification System for Combinational Circuits,” in *Proceedings of the International Test Conference*, pp. 350-359, 1986.

[177] H. C. Yen, S. Ghanta and H. C. Du, “A Path Selection Algorithm for Timing Analysis,” in *Proceedings of the 25th Design Automation Conference*, pp. 720-723, 1988.

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