DON'T CARES IN MULTI-LEVEL NETWORK OPTIMIZATION

by

Hamid Savoj

Memorandum No. UCB/ERL M92/122

30 October 1992

DON'T CARES IN MULTI-LEVEL NETWORK OPTIMIZATION

by

Hamid Savoj

Memorandum No. UCB/ERL M92/122

30 October 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Don't Cares in Multi-Level Network Optimization

Hamid Savoj

University of California
Berkeley, California

Department of Electrical Engineering
and Computer Sciences

Abstract

An important factor in the optimization of a multi-level circuit, modeled as a Boolean network, is to compute the flexibility for implementing each node of the network and to exploit this flexibility to get a better functional implementation at that node. A general form for describing input-output behavior of a Boolean network is a Boolean relation. This relation or a subset of it, is then used to compute the flexibility for implementing each node in the network. The nodes in the network can be either single or multiple output. In the case of a network composed of single-output nodes, this flexibility is captured by don't cares. Techniques for computing both maximum and compatible don't care sets for each node are presented. In the case of multi-output nodes, don't cares are not sufficient to express input-output behavior of the node. Thus, we present techniques to compute maximal and compatible flexibility at multi-output nodes using Boolean relations.

The current model for representing a Boolean circuit uses single output nodes. We present efficient techniques for single-output node simplification that use don't cares in terms of the fanins of node being simplified. The don't care set in terms of fanins of a node is called the local don't care set for that node; it usually has a small size and can be used to remove all the redundancies within that node. Practical issues for computing local don't cares and simplifying nodes are discussed in detail and experimental results are presented that show the effectiveness of the approach. New scripts are designed for technology independent optimization of Boolean circuits which use these new techniques.

Finally, a new Boolean matching algorithm is presented that can match two functions with given don't care sets. To prove the effectiveness of the approach, this algorithm is used within a technology mapper where matches are sought between subfunctions in the network and sets of gates in the library. The symmetries of the gates in the library are used to speed up the matching process.

R.K. Brayton

Prof. Robert K. Brayton
Thesis Committee Chairman

Don't Cares in Multi-Level Network Optimization

Hamid Savoj

Acknowledgements

The four and a half years that I have spent at the University of California at Berkeley have been a very intellectually stimulating stage of my life. Many individuals contributed to the work presented here.

I am indebted to my research advisor, Professor Robert K. Brayton, who provided guidence not only with my research but also with my personal growth. He has taught me how to develop and communicate research ideas. My discussions with him have always been motivating and enlightening. This work would have not been possible without his vision and continuous support.

Professor Alberto Sangiovanni-Vincentelli has helped me develop presentation skills and has taught me structured approaches to research.

Several people assisted me during the writing of my thesis. I am grateful to Adnan Aziz, Luciano Lavagno, Rajeev Murgai, Massoud Pedram, Narendra Shenoy, Tom Shiple, and Tiziano Villa for reading the first draft of my thesis and making valuable suggestions.

I would like to thank my colleagues for their interaction and companionship over the past few years. They are: Pranav Ashar, Adnan Aziz, Wendell Baker, Mark Beardslee, Andrea Casotto, Abhijit Ghosh, Ramin Hojati, Chuck Kring, Luciano Lavagno, Bill Lin, Abdul Malik, Sharad Malik, Rick McGeer, Cho Moon, Rajeev Murgai, Massoud Pedram, Jaijeet Roychowdhury, Rick Rudell, Alex Saldanha, Ellen Sentovich, Narendra Shenoy, Tom Shiple, Kanwar Jit Singh, Paul Stephan, Hervé Touati, Tiziano Villa, Albert Wang, Huey-Yih Wang, Yosinori Watanabe, and Greg Whitcomb. Many thanks to Flora Oviedo, Kia Cooper, and Elise Mills for the all administrative assistance provided. Brad Krebs helped with many hardware and software problems over the years.

My life at Berkeley would not have been as happy and memorable as it has been without my friends from International House. In particular, I would like to thank Barbara Calhoun, Cormac Conroy, Bijan Dastmalchi, Gustavo DeVeciana, Ami Doshi, Orla Feely, Cynthia Gaertner, Mehryar Gharakani, Peter Kennedy, George Kesidis, Davar Khoshnevisan, Carlos Kirjner, Michelle Leversee, Mary McNamara, Dariush Mirfendereski, Liam Murphy, Saeid Nazari, Kamran M. Nemati, Mehdi Nosrati, Ali Sarhaddi, Anthony Sarkis, Shahab Sheikholeslam, Ravi Subramanian, Shahram Taheri, and Farhad Zabihi.

Special thanks to my good friends of many years, Afsane Arvand, Massoud Pedram, and Khosrow Hasibi.

This thesis is dedicated to my parents, Ferdose and Abbas Savoj. Their love and support has inspired me throughout my life. Together with my brothers and sisters, they are constantly in my memory.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer Aided Design (CAD) tools are used in many fields of science to cope with complexity in the analysis or synthesis of systems. CAD tools have been developed to help chemists and biologist to study the structure of molecules, find common sub-structures among them, and predict their activities. CAD tools have also been used to analyze and predict the behavior of financial markets based on statistical models. Finally, CAD tools have been employed to automate the design of VLSI circuits. This dissertation focuses on new techniques for automated synthesis of logic circuits, which is a major division of CAD for VLSI circuits.

1.1 CAD for VLSI

The objective of VLSI CAD is to automate the design of VLSI systems. The starting point is usually a description of a system in a high-level language like ELLA [55] or VHDL [62]. The result of the design process is a layout description of that system which can be implemented as a set of integrated circuits in one or more technologies. This process is usually divided into four steps because the problem is too complex to be handled all at once. These steps are high-level synthesis, logic synthesis, physical design, and fabrication and test. A design may have to go through many iterations in one or more of the above steps before it meets the required specification. In some applications, CAD tools are employed starting at the logic or physical design level. This is the case when a design in an old technology is converted into a new technology.

We discuss the steps in VLSI design using the order in which they are applied.

High-level synthesis generates a register-transfer level structure from a behavioral description which realizes the given behavior. The temporal scheduling of operations and allocation of hardware are among the issues considered at this stage. Behavioral synthesis tools [37, 60, 80, 82, 84] can be used to reduce the amount hardware while meeting the required performance constraints. The next step is logic synthesis, where the register transfer structure is transformed into a netlist of gates in a given technology. Optimization techniques are applied to improve the area, performance, and testability of the circuit at this stage. Our focus is on new sets of optimization techniques that can be applied during logic synthesis. The third step, physical design, provides the physical realization of a net-list on a chip and includes placement, routing, and geometric artwork [16, 72, 63, 85]. As in logic synthesis, the objective is to find a layout with the best area and/or performance. The final step is fabrication and test where mask-making, fabrication, and performance test is done. Some manufactured circuits are defective because manufacturing processes cannot guarantee 100% yield. It is important to separate defective circuits from the proper ones. To detect defective circuits, test patterns are applied to the circuit and the response is compared with the expected response. Test pattern generation has been investigated for many years [2, 32, 35, 44, 48, 34, 71].

While synthesizing a given specification at the higher level of the design, one may estimate some information from the lower levels to produce a better design. For example, one may use placement and routing information during logic synthesis [61, 1].

Partitioning of a VLSI system and handling hierarchies are hard problems that need to be dealt with at all levels [28, 29, 8]. Formal verification is another area of interest where a given high-level specification of a circuit is checked for some desired behavior before any implementation is done [20, 43, 76]. This area has received considerable attention lately.

1.2 Logic Synthesis

Logic synthesis is the process of transforming a set of Boolean functions, obtained from the register transfer level structure, into a network of gates in a particular technology, while optimizing the network for performance, area, and/or testability. The best implementation of a functional specification is usually in multi-level logic. The model used for representing multiple levels of logic is a directed acyclic graph with single-output nodes, called a Boolean network, where each node of the network is a logic function itself. A set

of transformations is applied to the logic network to find the best set of nodes which give the desired input-output behavior.

1.2.1 Transformations

The synthesis process is usually divided into technology independent and technology dependent phases [10, 7]. The objective of the technology independent phase is to simplify the logic as much as possible, while the main role of the technology dependent phase is to implement the logic network using a set of well characterized gates.

In [46], it was shown experimentally that the layout area of a Boolean network implementation in standard cells correlates well with the number of literals used for representing that network in factored form (defined in [10]). This measure is used in the technology independent stage because it helps to reduce the size of the nodes in the network and therefore, the amount of logic used. After a circuit is optimized for literals in factored form, additional transformations can be applied to optimize performance, or to tune the circuit for eventual implementation in a particular technology such as, Field Programmable Gate Arrays [39, 4]. A network optimized for literals in factored form is a good starting point for these other manipulations. When the amount of logic used is reduced, there are less gates to be placed on a chip and fewer nets to be routed. As a result, performance is also improved in most cases. In the same way, a circuit can be implemented on an FPGA chip using fewer blocks, if the amount of logic is reduced.

An important transformation to reduce the number of levels in factored form is to apply a two-level logic minimizer to each node of the multi-level network to optimize the two-level function associated with the node. The input supplied to a two-level minimizer is the onset and the don't care set of the node. The onset gives the function of the node in terms of its fanins. The don't care set gives the flexibility in implementing the node, and is a combination of structural don't cares and external don't cares supplied by the user.

In the technology dependent stage, transformations are applied to implement and optimize logic for a particular technology. For standard cells, the most common approach decomposes the circuit into a set of trees and then maps each tree into a set of library gates [42, 64]. For the Xilinx FPGA architecture, the Boolean network is decomposed into a set of nodes where each node has less than 5 inputs; therefore, it can be directly mapped into a logic block [56, 57]. Don't care conditions can be used to improve the quality of the mapped

circuits.

1.2.2 Flexibility in Node Implementation

This thesis contributes to the understanding and ability to use don't care sets and Boolean relations to manipulate Boolean networks. Don't cares are used to find the maximum flexibility for implementing nodes of a network decomposed into single-output nodes. The don't care set at each node is a combination of external, observability, and satisfiability don't cares. The external don't care set for a network is a restricted form for expressing the freedom in input-output behavior of that network. These are input conditions under which the value of a particular output is not important. The observability and satisfiability don't cares are related to the structure of the network. The observability don't cares are conditions under which the value of a node is not observable at any of the outputs. Satisfiability don't cares are related to the functions at the nodes of the network. Each intermediate node of a Boolean network gets a value of 1 or 0 for a particular input. Some combinations are impossible in the Boolean space of intermediate nodes and primary inputs of the network; satisfiability don't cares capture all such impossible combinations.

A Boolean network can be also decomposed into a set of multi-output nodes. This is very useful for data path synthesis where the circuit has many multi-output blocks like multipliers and adders. These blocks must be recognized and mapped into corresponding multi-output gates in the library. The maximum flexibility for implementing a multi-output node is captured by a Boolean relation which gives the set of outputs possible for any particular input pattern to the node. A Boolean network is a multi-output node itself, and external don't cares are not sufficient to express the maximum flexibility for implementing it. Thus Boolean relations must be used to express the input-output behavior of a Boolean network. Such a relation is called an observability relation. The use of the observability relation allows the definition and computation of maximum flexibility in implementing multi-output nodes in a Boolean network.

An interesting concept which applies to both don't cares and Boolean relations is the concept of compatibility. In general, the flexibility in implementing a node in the network affects the flexibility in implementing other nodes in the network; therefore, if the function at a node is changed, don't care sets or Boolean relations at other nodes in the network have to be updated. The updating process can be costly. It is possible to order all

the nodes in the network and compute compatible flexibilities according to the ordering. In this manner, all the nodes in the network can be optimized simultaneously. Furthermore, once the function at a node is changed, the flexibility for implementing other nodes in the network need not be recomputed.

1.3 Overview

Chapter 2 presents some basic definitions used throughout this thesis. It also contains the definition of generalized cofactor and its properties. Finally, it discusses different methods for image computation and explains shortcomings and advantages of each approach.

Chapter 3 discusses maximum flexibility for implementing nodes of a Boolean network decomposed into single-output nodes. The flexibility for a single-output node can be computed in the form of don't cares. We discuss different kinds of don't cares and techniques for computing a full don't care set and approximate don't care sets at a particular node. The input-output behavior of the network itself is expressed by a Boolean relation because it can be viewed as a multi-output node. This relation is represented by a node attached to the top of the original network. This new network is called the observability network. We develop techniques for computing full and compatible don't cares from the observability network.

Techniques for computing compatible and maximal Boolean relations for a network decomposed into multi-output nodes are developed in Chapter 4. We first discuss the parallel and serial decomposition of Boolean relations and then expand this to a general decomposition into multi-output nodes.

Chapter 5 discusses the practical issues in using local don't cares for the simplification of single-output nodes in a multi-level network. The algorithms used for local don't care computation are discussed in detail. The extensions to BLIF for representing external don't cares are discussed in this chapter.

Chapter 6 discusses scripts used for technology independent optimization. The algorithms used within these scripts are discussed and improvements are provided which enable the application of these algorithms to larger circuits. Experiments are run on a large set of benchmark circuits to show the effectiveness of these improved scripts for reducing the amount of logic used and removing redundancies in each circuit.

Finally, we discuss Boolean matching in Chapter 7 and show how this can be used for technology mapping. The tree matching algorithm used to match a subfunction in the network with a gate in the library, is replaced by Boolean matching. The symmetries of the gates in the library are found and used to speed up the matching process. Gates in the library are grouped into sets using the same Boolean matching algorithm, where it is enough to check the existence of a match of a subfunction in the network with a set representative.

Chapter 2

Terminology and Background

The purpose of this chapter is to introduce some basic definitions and concepts that are essential for describing the work presented in this thesis.

2.1 Boolean Functions and Boolean Networks

2.1.1 Boolean Functions

Definition 2.1.1 *A completely specified Boolean function f with n inputs and l outputs is a mapping*

$$f : B^n \longrightarrow B^l$$

where $B = \{0,1\}$. In particular, if $l = 1$ the onset and offset of f are

$$onset = \{m \in B^n \mid f(m) = 1\}$$
$$offset = \{m \in B^n \mid f(m) = 0\}$$

Definition 2.1.2 *A minterm of a function f is a vertex $m \in B^n$ such that $f(m) = 1$.*

Definition 2.1.3 *An incompletely specified Boolean function $\mathcal{F}$ with n inputs and l outputs is a mapping*

$$\mathcal{F} : B^n \longrightarrow Y^l$$

where $Y = \{0,1,*\}$. The onset , offset , and don't care set (dcset) of $\mathcal{F} : B^n \longrightarrow Y$ are,

$$onset = \{m \in B^n \mid \mathcal{F}(m) = 1\}$$

$$offset = \{m \in B^n \mid \mathcal{F}(m) = 0\}$$
$$dcset = \{m \in B^n \mid \mathcal{F}(m) = *\}.$$

The symbol $*$ implies that the function can be either 0 or 1.

Definition 2.1.4 *A cover for the incompletely specified function $\mathcal{F} : B^n \longrightarrow Y$ is any completely specified function f such that $f(m) = 1$ if $\mathcal{F}(m) = 1$, $f(m) = 0$ if $\mathcal{F}(m) = 0$, and $f(m) = 0$ or 1 if $\mathcal{F}(m) = *$.*

$\mathcal{F} : B^n \longrightarrow Y$ is usually described by two completely specified functions f and d where f is a cover for $\mathcal{F}$ and $d(m) = 1$ if $\mathcal{F}(m) = *$, and 0 otherwise. Alternately, $\mathcal{F}$ can be described as $\mathcal{F} = (f, d, r)$ where f, d, r are respectively the onset, don't care set, and offset.

Let $x_1, x_2, \ldots, x_n$ be the variables of the space B^n. We use $\mathbf{x}$ to represent a vertex or a vector of variables in B^n.

Definition 2.1.5 *Let $A \subseteq B^n$. The characteristic function of A is the function $f : B^n \rightarrow B$ defined by $f(\mathbf{x}) = 1$ if $\mathbf{x} \in A$, $f(\mathbf{x}) = 0$ otherwise.*

Characteristic functions are nothing but a functional representation of a subset of a set. Any completely specified function $f : B^n \longrightarrow B$ is a characteristic function for its onset.

Definition 2.1.6 *A literal is a variable in its true or complement form (e.g.x_i, or $\overline{x_i}$). A product term or cube is the conjunction of some set of literals (e.g. $x_1 x_2 \overline{x_3}$).*

Definition 2.1.7 *A cube c is called a prime cube of $\mathcal{F}$ if $c \subset f \cup d$ and there is no cube c' such that $c \subset c' \subset f \cup d$.*

Definition 2.1.8 *The distance between two cubes denoted as $dist(c_1, c_2)$ is the number of literals in which c_1 contains the complement literal of c_2. Two cubes c_1 and c_2 are called orthogonal if $dist(c_1, c_2) \geq 1$ (e.g. $x_1 x_2 x_3$ is orthogonal to $x_1 \overline{x}_2$ and of distance 1).*

The onset, don't care set, and offset of an incompletely specified function $\mathcal{F} : B^n \longrightarrow Y$ can each be represented by the union of some set of cubes. This representation is called a *sum-of-products* form.

Definition 2.1.9 *Let $f : B^n \rightarrow B$ be a Boolean function, and x_i an input variable of f. The cofactor of f with respect to a literal $x_i(\overline{x_i})$, shown as $f_{x_i}(f_{\overline{x_i}})$, is a new function obtained by substituting $1(0)$ for $x_i(\overline{x_i})$ in every cube in f which contains $x_i(\overline{x_i})$.*

Definition 2.1.10 *Let $f : B^n \rightarrow B$ be a Boolean function, and x_i an input variable of f. f is monotone increasing in a variable x_i if $f_{\overline{x}_i} \subseteq f_{x_i}$. A function f is monotone decreasing in a variable x_i if $f_{x_i} \subseteq f_{\overline{x}_i}$. f is unate in variable x_i if it is monotone increasing or decreasing in that variable. f is a unate function if it is unate in all its input variables. A function is independent of x_i if $f_{x_i} = f_{\overline{x}_i}$.*

Definition 2.1.11 *Let $f : B^n \rightarrow B$ be a Boolean function, and x_i an input variable of f. The Shannon's expansion of a Boolean function f with respect to a variable x_i is*

$$x_i f_{x_i} + \overline{x_i} f_{\overline{x_i}}.$$

Lemma 2.1.1 $f = x_i f_{x_i} + \overline{x_i} f_{\overline{x_i}}$.

The iterated Shannon decomposition of a Boolean function is a Binary tree representing the function obtained by applying Shannon's expansion with respect to all the variables. The leaves are either 0 or 1. Each path of the tree represents a minterm of the function.

2.1.2 Boolean Network

Definition 2.1.12 *A Boolean network $\mathcal{N}$, is a directed acyclic graph (DAG) such that for each node in $\mathcal{N}$ there is an associated representation of a Boolean function f_i, and a Boolean variable y_i, where $y_i = f_i$. There is a directed edge (i,j) from y_i to y_j if f_j depends explicitly on y_i or $\overline{y}_i$. A node y_i is a fanin of a node y_j if there is a directed edge (i,j) and a fanout if there is a directed edge (j,i). A node y_i is a transitive fanin of a node y_j if there is a directed path from y_i to y_j and a transitive fanout if there is a directed path from y_j to y_i. Primary inputs $\mathbf{x} = (x_1, \ldots, x_n)$ are inputs of the Boolean network and primary outputs $\mathbf{z} = (z_1, \ldots, z_m)$ are its outputs. Intermediate nodes of the Boolean network have at least one fanin and one fanout. The global function f_i^g at y_i is the function at the node expressed in terms of primary inputs.*

We sometimes represent the local function at y_i by f_i^l to make a clear distinction with the global function f_i^g.

Definition 2.1.13 *The cofactor of $\mathcal{N}$ with respect to y_i denoted by $\mathbf{z}_{y_i}$ is a new network obtained from $\mathcal{N}$ by disconnecting the output edges of y_i from y_i and forcing each output edge equal to 1. The cofactor of $\mathcal{N}$ with respect to $\overline{y}_i$ denoted by $\mathbf{z}_{\overline{y}_i}$ is a new network*

obtained from $\mathcal{N}$ by disconnecting the output edges of y_i from y_i and forcing each output edge equal to 0.

Example:
If $\mathcal{N}$ has only one output z, $z_{y_i} z_{\overline{y}_i}$ is a new network representing the function obtained by ANDing outputs of z_{y_i} and $z_{\overline{y}_i}$. $z_{y_i} z_{\overline{y}_i}$ gives conditions under which the value of y_i can be changed from 0 to 1 or vice versa but output z remains equal to 1. This computation is important while computing observability don't cares discussed in Chapter 3.

Definition 2.1.14 *The support of a function f is the set of variables that f explicitly depends on.*

Example:
The support of $f = x_1x_2 + x_2\overline{x}_3$ is $\{x_1, x_2, x_3\}$.

Definition 2.1.15 *Nodes of a network are topologically ordered from outputs if each node appears somewhere after all its fanouts in the ordering. They are topologically ordered from inputs if each node appears somewhere after all its fanins in the ordering.*

2.2 Set Operations and BDD's

Set operations are essential for manipulating Boolean functions. In this section, we discuss some important set operations used in this thesis.

2.2.1 Binary Decision Diagrams

Binary Decision Diagrams [14] are compact representations of recursive Shannon decompositions. The decomposition is done with the same order along every path from the root to the leaves as shown in Figure 2.1. BDD's are unique for a given variable ordering and hence are canonical forms for representing Boolean functions. They can be constructed from the Shannon's expansion of a Boolean function by 1) deleting a node whose two child edges point to the same node, and 2) sharing isomorphic subgraphs. Technically the result is a reduced ordered BDD, (ROBDD), which we shall just call BDD.

Example:
Figure 2.1 shows the Shannon decomposition of $f = x_1x_3 + \overline{x}_1x_3$ with the ordering $x_1 \succ$

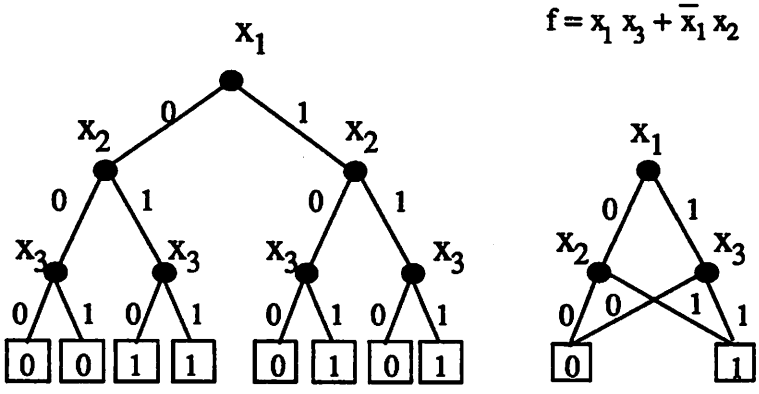

Figure 2.1: Binary Decision Diagram

$x_2 \succ x_3$ and the corresponding BDD. The unreduced one is on the left, the reduced one on the right. An example of reduction is the merging of the two right most nodes which represent the same function, namely x_3. Then the x_2 node above them has both of its children with the same node and thus can be eliminated.

2.2.2 Consensus Operator

Definition 2.2.1 *Let $f : B^n \rightarrow B$ be a Boolean function, and $\mathbf{x} = (x_{i_1}, \ldots, x_{i_k})$ a set of input variables of f. The consensus of f by $\mathbf{x}$ is*

$$\begin{aligned} C_{\mathbf{x}} f &= C_{x_{i_1}} \ldots C_{x_{i_k}} f \\ C_{x_{i_j}} f &= f_{x_{i_j}} f_{\bar{x}_{i_j}} \end{aligned}$$

This is also the largest Boolean function contained in f which is independent of $x_{i_1}, \ldots, x_{i_k}$.

2.2.3 Smoothing Operator

Definition 2.2.2 *Let $f : B^n \rightarrow B$ be a Boolean function, and $\mathbf{x} = (x_{i_1}, \ldots, x_{i_k})$ a set of input variables of f. The smoothing of f by $\mathbf{x}$ is*

$$\begin{aligned} S_{\mathbf{x}} f &= S_{x_{i_1}} \ldots S_{x_{i_k}} f \\ S_{x_{i_j}} f &= f_{x_{i_j}} + f_{\bar{x}_{i_j}} \end{aligned}$$

If f is interpreted as the characteristic function of a set, the smoothing operator computes the projection of f to the subspace of B^n orthogonal to the domain of the $\mathbf{x}$ variables. This is also the smallest Boolean function independent of $x_{i_1}, \ldots, x_{i_k}$ which contains f.

Lemma 2.2.1 *Let $f : B^n \times B^m \rightarrow B$ and $g : B^m \rightarrow B$ be two Boolean functions. Then:*

$$S_{\mathbf{x}}(f(\mathbf{x},\mathbf{y})g(\mathbf{y})) \;=\; S_{\mathbf{x}}(f(\mathbf{x},\mathbf{y}))g(\mathbf{y}) \tag{2.1}$$

where $f(\mathbf{x},\mathbf{y})g(\mathbf{y})$ is the Boolean AND of $f(\mathbf{x},\mathbf{y})$ and $g(\mathbf{y})$.

2.2.4 Boolean Difference

Let $f : B^n \rightarrow B$ be a Boolean function, and x_i an input variable of f. The *Boolean difference* of a function f with respect to a variable x_i is defined as

$$\frac{\partial f}{\partial x_i} \equiv f_{x_i}\overline{f}_{\overline{x_i}} + \overline{f}_{x_i}f_{\overline{x_i}}$$

This function gives all the conditions under which the value of f is influenced by the value of x_i. Its complement therefore is all the conditions under which f is *insensitive* to x_i. The concept of Boolean difference of a Boolean function with respect to a variable is very similar to the concept of partial derivative of a real function with respect to a variable.

Example:

Let $f = x_1x_3 + x_2\overline{x}_3$. Then, $\frac{\partial f}{\partial x_3} = \overline{x}_1x_2 + x_1\overline{x}_2$. Notice that if $x_1 = 0$ and $x_2 = 1$, which is a minterm of $\frac{\partial f}{\partial x_3}$, $f = \overline{x}_3$. As a result, the value of f is sensitive to x_3. In the same way, if $x_1 = 1$ and $x_2 = 0$, $f = x_3$ and the value of f changes with the value of x_3.

2.3 Image and Inverse Image Computations

Definition 2.3.1 *Let $f : B^n \rightarrow B^m$ be a Boolean function and A a subset of B^n. The* image *of A by f is the set $f(A) = \{\mathbf{y} \in B^m \mid \mathbf{y} = f(\mathbf{x}),\ \mathbf{x} \in A\}$. If $A = B^n$, the image of A by f is also called the* range *of f.*

Definition 2.3.2 *Let $f : B^n \rightarrow B^m$ be a Boolean function and A a subset of B^m. The* inverse image *of A by f is the set $f^{-1}(A) = \{\mathbf{x} \in B^n \mid f(\mathbf{x}) = \mathbf{y},\ \mathbf{y} \in A\}$.*

Example:

Let $f(\mathbf{x},\mathbf{i}) : B^n \times B^k \rightarrow B^n$ be the next state function of a finite state machine, where n is the number of state variables and k the number of input variables. Let c_∞ be the set of states reachable from a set of initial states c_0. c_∞ can be obtained by repeated computation

of an image as follows

$$\begin{aligned} c_{i+1} &= c_i \cup f(c_i \times B^k) \\ c_\infty &= c_i \quad \text{if} \quad c_{i+1} = c_i \end{aligned}$$

The sequence is guaranteed to converge after a finite number of iterations because $\{c_i\}$ is monotone increasing and the number of states is finite.

2.3.1 The Generalized Cofactor

The *generalized cofactor* is an important operator that can be used to reduce an image computation to a range computation. This operator was initially proposed by Coudert *et al.* in [21] and called the *constraint* operator. Given a Boolean function: $f = (f_1, \ldots, f_m) : B^n \rightarrow B^m$ and a subset of B^n represented by its characteristic function h, the generalized cofactor $f_h = ((f_1)_h, \ldots, (f_m)_h)$ is one of many functions from B^n to B^m whose range is equal to the image of h by f. An important property of this function is that $f_h(\mathbf{x}) = f(\mathbf{x})$ if $h(\mathbf{x}) = 1$. In addition, in most cases, the BDD representation of f_h is smaller than the BDD representation of f. Given a cover $f : B^n \rightarrow B$ for the onset and don't care set $d : B^n \rightarrow B$ of an incompletely specified function $\mathcal{F}$, the function $f_{\overline{d}}$ has the property that $f - d \subseteq f_{\overline{d}} \subseteq f + d$ ($f - d$ is the same as $f\overline{d}$.); therefore $f_{\overline{d}}$ is also a cover of $\mathcal{F} = (f\overline{d}, d, \overline{f}\ \overline{d})$. This is because $f_{\overline{d}}$ gives the same values as f for any minterm in $\overline{d}$. In practice the size of the BDD for $f_{\overline{d}}$ is usually smaller than that of f [78]. As a result, this is an effective way to get a cover with smaller BDD size.

The generalized cofactor f_h depends in general on the variable ordering used in the BDD representation.

Definition 2.3.3 *Let $h : B^n \rightarrow B$ be a non-null Boolean function and $x_1 \succ x_2 \succ \ldots \succ x_n$ an ordering of its input variables. We define the mapping $\pi_h : B^n \rightarrow B^n$ as follows:*

$$\begin{aligned} &\pi_h(\mathbf{x}) = \mathbf{x} && \text{if} \quad h(\mathbf{x}) = 1 \\ &\pi_h(\mathbf{x}) = \arg\min_{h(\mathbf{y})=1} d(\mathbf{x},\mathbf{y}) && \text{if} \quad h(\mathbf{x}) = 0 \\ &\text{where} \quad d(\mathbf{x},\mathbf{y}) = \sum_{1 \le i \le n} |x_i - y_i| 2^{n-i} \end{aligned}$$

Lemma 2.3.1 *π_h is the projection that maps a minterm $\mathbf{x}$ to the minterm $\mathbf{y}$ in the onset of h which has the closest distance to $\mathbf{x}$ according to the metric d. The particular form of*

the distance metric guarantees the uniqueness of $\mathbf{y}$ *in this definition, for any given variable ordering.*

Proof Let $\mathbf{y}$ and $\mathbf{y}'$ be two minterms in the onset of h such that $d(\mathbf{x},\mathbf{y}) = d(\mathbf{x},\mathbf{y}')$. Each of the expressions $d(\mathbf{x},\mathbf{y})$ and $d(\mathbf{x},\mathbf{y}')$ can be interpreted as the binary representation of some integer. Since the binary representation is unique, $|x_i - y_i| = |x_i - y'_i|$ for $1 \leq i \leq n$ and thus $\mathbf{y} = \mathbf{y}'$. ∎

Lemma 2.3.2 *Let* $h(\mathbf{x},\mathbf{y}) : B^{n+m} \rightarrow B$, *where* n *is the number of variables in* $\mathbf{x}$ *and* m *is the number of variables in* $\mathbf{y}$, *be dependent only on the* $\mathbf{x}$ *variables* ($S_{\mathbf{y}}h \equiv h$). *For every vertex* $\mathbf{x}$ *there is a vertex* $\mathbf{x}'$ *such that* $\pi_h(\mathbf{x},\mathbf{y}) = (\mathbf{x}',\mathbf{y})$ *for any ordering of* $\mathbf{x}$ *and* $\mathbf{y}$ *variables.*

Proof Any vertex $(\mathbf{x},\mathbf{y})$ must be mapped to the closest vertex $(\mathbf{x}',\mathbf{y}')$ to it according to metric d such that $h(\mathbf{x}',\mathbf{y}') = 1$. $\mathbf{y}' \neq \mathbf{y}$ is not possible because $(\mathbf{x}',\mathbf{y})$ is closer to $(\mathbf{x},\mathbf{y})$ than $(\mathbf{x}',\mathbf{y}')$ and $h(\mathbf{x}',\mathbf{y}) = 1$. ∎

Definition 2.3.4 *Let* $f : B^n \rightarrow B$ *and* $h : B^n \rightarrow B$, *with* $h \neq 0$. *The generalized cofactor of* f *with respect to* h, *denoted by* f_h, *is the function* $f_h = f \circ \pi_h$, *i.e.* $f_h(\mathbf{x}) = f(\pi_h(\mathbf{x}))$. *If* $f : B^n \rightarrow B^m$, *then* $f_h : B^n \rightarrow B^m$ *is the function whose components are the cofactors by* h *of the components of* f.

If c is a cube the generalized cofactor f_c is equal to the usual cofactor of a Boolean function, and is, in that case, independent of the variable ordering.

Lemma 2.3.3 *If* h *is a cube (i.e.* $c = c_1c_2\ldots c_n$ *where* $c_i = \{0,1,*\}$), π_c *is independent of the variable ordering. More precisely,* $\mathbf{y} = \pi_c(\mathbf{x})$ *satisfies*

$$y_i = 0 \quad \textit{if} \quad c_i = 0$$
$$y_i = 1 \quad \textit{if} \quad c_i = 1$$
$$y_i = x_i \quad \textit{if} \quad c_i = *$$

and $f_c = f \circ \pi_c$ *is the usual cofactor of a Boolean function by a cube.*

Proof Any minterm $\mathbf{y}'$ in B^n such that $c(\mathbf{y}') = 1$ is orthogonal to $\mathbf{x}$ in at least the same variables as $\mathbf{y}$. Thus $\mathbf{y}$ minimizes $d(\mathbf{x},\mathbf{y})$ over c. ∎

In addition, the generalized cofactor preserves the following important properties of cofactors:

Lemma 2.3.4 *Let $g : B^m \to B$ and $f : B^n \to B^m$. Then $(g \circ f)_h = g \circ f_h$. In particular the cofactor of a sum of functions is the sum of the cofactors, and the cofactor of an inverse is the inverse of the cofactor (e.g. $f = [f_1, f_2]$ and $g = f_1 + f_2$, then $g_h = f_{1h} + f_{2h}$).*

Proof $(g \circ f)_h = (g \circ f) \circ \pi_h$ and $g \circ f_h = g \circ (f \circ \pi_h)$. ∎

Lemma 2.3.5 *Let $f : B^n \times B^m \to B$ and $h : B^n \to B$ be two Boolean functions, with $h \neq 0$. Then:*

$$S_{\mathbf{x}}(f(\mathbf{x},\mathbf{y}) \cdot h(\mathbf{x})) \quad = \quad S_{\mathbf{x}}(f_h(\mathbf{x},\mathbf{y})) \tag{2.2}$$

Proof For every vertex $\mathbf{x} \in B^n$ if $h(\mathbf{x}) = 1$, then $f_h(\mathbf{x},\mathbf{y}) = f(\mathbf{x},\mathbf{y})$. This is from the definition of generalized cofactor. If $h(\mathbf{x}) = 0$, then $f(\mathbf{x},\mathbf{y})h(\mathbf{x}) = 0$. Therefore, $f(\mathbf{x},\mathbf{y})h(\mathbf{x}) \subseteq f_h(\mathbf{x},\mathbf{y})$ and $S_{\mathbf{x}}(f(\mathbf{x},\mathbf{y})h(\mathbf{x})) \subseteq S_{\mathbf{x}}(f_h(\mathbf{x},\mathbf{y}))$. Conversely, if vertex $\mathbf{y} \in B^m$ is such that $S_{\mathbf{x}}(f_h(\mathbf{x},\mathbf{y})) = 1$, there exists an $\mathbf{x}$ for which $f_h(\mathbf{x},\mathbf{y}) = 1$ and therefore there exists an $\mathbf{x}'$ such that $f(\mathbf{x}',\mathbf{y}) = 1$ and $h(\mathbf{x}') = 1$. This is because $f_h(\mathbf{x},\mathbf{y}) = f \circ \pi_h(\mathbf{x},\mathbf{y})$ and h is dependent only on $\mathbf{x}$ variables; therefore for each vertex $(\mathbf{x},\mathbf{y})$, $f_h(\mathbf{x},\mathbf{y}) = f(\mathbf{x}',\mathbf{y})$ and $h(\mathbf{x}') = 1$ from Lemma 2.3.2. This gives $S_{\mathbf{x}}(f_h(\mathbf{x},\mathbf{y})) \subseteq S_{\mathbf{x}}(f(\mathbf{x},\mathbf{y})h(\mathbf{x}))$ and the statement of the Lemma follows. ∎

Lemma 2.3.6 *Let f be a Boolean function, and h a non-null Boolean function. Then h is contained in f if and only if f_h is a tautology.*

Proof Suppose that h is contained in f. Let $\mathbf{x}$ be an arbitrary minterm. $\mathbf{y} = \pi_h(\mathbf{x})$ implies $h(\mathbf{y}) = 1$. Since $h \Rightarrow f$, $f_h(\mathbf{x}) = f(\mathbf{y}) = 1$. Thus f_h is a tautology. Conversely, suppose that f_h is a tautology. Let $\mathbf{x}$ be such that $h(\mathbf{x}) = 1$. Then $\pi_h(\mathbf{x}) = \mathbf{x}$ and $f(\mathbf{x}) = f(\pi_h(\mathbf{x})) = f_h(\mathbf{x}) = 1$, which proves that h is contained in f. ∎

Lemma 2.3.7 *If h is the characteristic function of a set A then $f_h(B^n) = f(A)$; that is the image of A by f is equal to the range of the cofactor f_h.*

```
function gcofactor(f, h) {
    assert (h ≠ 0) ;
    if (h ≡ 1 or is_constant(f))     return f;
    else if (h_{x̄1} = 0)     return gcofactor(f_{x1}, h_{x1});
    else if (h_{x1} = 0)     return gcofactor(f_{x̄1}, h_{x̄1});
    else return x1 gcofactor(f_{x1}, h_{x1}) + x̄1 gcofactor(f_{x̄1}, h_{x̄1});
}
```

Figure 2.2: Generalized Cofactor

Proof $\pi_h(B^n)$ is equal to the onset of h, which is A. Thus $f_h(B^n) = f \circ \pi_h(B^n) = f(A)$. ∎

The generalized cofactor can be computed very efficiently in a single bottom-up traversal of the BDD representations of f and h by the algorithm given in Figure 2.2.

Theorem 2.3.8 $f \circ \pi_h \equiv gcofactor(f, h)$.

Proof Let the variable ordering be $x_1 \succ \ldots \succ x_n$, and $m = x_1 \ldots x_n \in B^n$ be a particular vertex. We compute the value of the function $gcofactor(f, h)$ for the particular m (shown as $[gcofactor(f, h)](m)$) and show that it is equal to $f \circ \pi_h(m)$ irrespective of the choice of m. Assume h and f are not constants. The proof in such cases is trivial.

If $h(m) = 1$, it follows from Figure 2.2 that

$$gcofactor(f, h)(m) = x_1 gcofactor(f_{x_1}, h_{x_1})(m)$$

because $h_{x_1} \neq 0$. We are not interested in $\overline{x}_1 gcofactor(f_{\overline{x}_1}, h_{\overline{x}_1})$ because it gives 0 for the vertex m. In the *ith* step, if neither $f_{x_1 \ldots x_i}$ nor $h_{x_1 \ldots x_i}$ is a constant,

$$x_1 \ldots x_i gcofactor(f_{x_1 \ldots x_i}, h_{x_1 \ldots x_i})(m) = x_1 \ldots x_{i+1} gcofactor(f_{x_1 \ldots x_{i+1}}, h_{x_1 \ldots x_{i+1}})(m)$$

because $h_{x_1 \ldots x_{i+1}} \neq 0$. Eventually either $f_{x_1 \ldots x_j}$ becomes constant or $h_{x_1 \ldots x_j} \equiv 1$. In either case, the returned function has the term $x_1 \ldots x_j f_{x_1 \ldots x_j}$ which is equal to $x_1 \ldots x_j f$. As a result, $[gcofactor(f, h)](m) = x_1 \ldots x_j f(m) = f(m)$. The same reasoning holds for any m such that $h(m) = 1$; therefore, $f \circ \pi_h(m) \equiv [gcofactor(f, h)](m)$, if $h(m) = 1$.

If $h(m) = 0$, the computation is as before as long as $h_{x_1 \ldots x_{i-1}} \neq 0$. If $h_{x_1 \ldots x_i} = 0$, it follows from Figure 2.2 that

$$x_1 \ldots x_{i-1} gcofactor(f_{x_1 \ldots x_{i-1}}, h_{x_1 \ldots x_{i-1}})(m) = x_1 \ldots x_{i-1} gcofactor(f_{x_1 \ldots \overline{x}_i}, h_{x_1 \ldots \overline{x}_i})(m).$$

Eventually, either $f_{x_1 \ldots \overline{x}_i \ldots x_j}$ becomes constant or $h_{x_1 \ldots \overline{x}_i \ldots x_j} = 1$ in the jth iteration. In either case, the part of the returned function which is of interest is $x_1 \ldots x_{i-1} x_{i+1} \ldots x_j f_{x_1 \ldots \overline{x}_i \ldots x_j}$. If $c = x_1 \ldots x_{i-1} x_{i+1} \ldots x_j$, then the returned expression can be written as $c f_c$. The value of $[gcofactor(f,h)](m) = [x_1 \ldots x_{i-1} x_{i+1} \ldots x_j f_{x_1 \ldots \overline{x}_i \ldots x_j}]_m$ which is $f_{x_1 \ldots \overline{x}_i \ldots x_j x_{j+1} \ldots x_n}$. We prove that the closest vertex m' to m such that $h(m') = 1$ is in the cube $c = x_1 \ldots \overline{x}_i \ldots x_j$. Because $h_{x_1 \ldots x_{i-1}} \neq 0$, the closest vertex m' to m (distance is defined by Definition 2.3.3) such that $h(m') = 1$ must have literals $x_1, \ldots, x_{i-1}$. Because $h_{x_1 \ldots x_i} = 0$, the closest vertex m' to m such that $h(m') = 1$ must have literals $x_1, \ldots, x_{i-1}, \overline{x}_i$. This same reasoning holds for any other literal in c, therefore m' must have all the literals in c. If $c = 1$, $x_1 \ldots \overline{x}_i \ldots x_j x_{j+1} \ldots x_n$ is the closest vertex to m and $f \circ \pi_h(m) \equiv [gcofactor(f,h)](m)$. If $c \neq 1$, f_c must be constant to terminate the algorithm in Figure 2.2. f_c has the same value for all the vertices in c and again $f \circ \pi_h(m) \equiv [gcofactor(f,h)](m)$. This same reasoning holds for any m such that $h(m) = 0$. ∎

The next example shows that the BDD size of f_h is not always smaller than the BDD size of f.

Example:

Let $f = \overline{x}_2 + x_3$, $h = x_1 + x_2$, and the ordering used for computing the generalized cofactor be $x_1 \succ x_2 \succ x_3$. $gcofactor(f,h) = x_1 gcofactor(\overline{x}_2 + x_3, 1) + \overline{x}_1 gcofactor(\overline{x}_2 + x_3, x_2)$. This gives $f_h = gcofactor(f,h) = x_1 \overline{x}_2 + x_3$. The BDD for f_h with the given ordering has 3 nodes while the BDD for f has 2 nodes.

2.3.2 The Transition Relation Method

Definition 2.3.5 *Let $f : B^n \rightarrow B^m$ be a Boolean function. The transition relation associated with f, $F : B^n \times B^m \rightarrow B$, is defined as $F(\mathbf{x},\mathbf{y}) = \{(\mathbf{x},\mathbf{y}) \in B^n \times B^m \mid \mathbf{y} = f(\mathbf{x})\}$. Equivalently, in terms of Boolean operations:*

$$F(\mathbf{x},\mathbf{y}) = \prod_{1 \leq i \leq m} (y_i \overline{\oplus} f_i(\mathbf{x})) \tag{2.3}$$

We can use F to obtain the image by f of a subset A of B^n, by computing the projection on B^m of the set $F \cap (A \times B^m)$. In terms of BDD operations, this is achieved

by a Boolean AND and a smooth. The smoothing and the Boolean *and* can be done in one pass on the BDD's to further reduce the need for intermediate storage [15]:

$$f(A)(\mathbf{y}) = S_{\mathbf{x}}(F(\mathbf{x},\mathbf{y}) \cdot A(\mathbf{x})) \tag{2.4}$$

The inverse image by f of a subset A of B^m can be computed as easily:

$$f^{-1}(A)(\mathbf{x}) = S_{\mathbf{y}}(F(\mathbf{x},\mathbf{y}) \cdot A(\mathbf{y})) \tag{2.5}$$

The transition relation method allows both image and inverse image computation for a function f. However, computing the transition relation may require too much memory to be feasible in some examples. We do not need to compute the transition relation explicitly to perform an image computation as in equation 2.4. Using propositions 2.3.4 and 2.3.5, we can rewrite equation 2.4 as follows:

$$S_{\mathbf{x}}(F(\mathbf{x},\mathbf{y}) \cdot A(\mathbf{x})) = S_{\mathbf{x}}(\prod_{1 \le i \le m} y_i \overline{\oplus} f_{i\,A(\mathbf{x})})$$

One efficient way to compute the product is to decompose the Boolean AND of the m functions $(g_i(\mathbf{x},\mathbf{y}) = y_i \overline{\oplus} f_{i\,A(\mathbf{x})})$ into a balanced binary tree of Boolean AND's. Moreover, after computing every binary AND p of two partial products p_1 and p_2, we can smooth the $\mathbf{x}$ variables that appear in p and in no other partial product. As for equation 2.4, the smoothing and the AND computations can be done in one pass on the BDD's to reduce storage requirements.

2.3.3 The Recursive Image Computation Method

Coudert *et al.* [21, 22] introduced an alternate procedure to compute the image of a set by a Boolean function that does not require building the BDD for the transition relation. This procedure relies on lemma 2.3.7 to reduce the image computation to a range computation, and proceeds recursively by cofactoring by a variable of the input space or the output space. We use $range(f)$ to denote the range of a multiple output function $f = [f_1, \ldots, f_m]$. There are two techniques for doing this range computation. The first is *input cofactoring* where output functions are cofactored with respect to inputs only.

$$range(f)(\mathbf{y}) = range([f_1, \ldots, f_m]_{x_1}) + range([f_1, \ldots, f_m]_{\overline{x_1}})$$

The second technique is *output cofactoring* where output functions are cofactored with respect to other functions.

$$\begin{aligned} range(f)(\mathbf{y}) &= range([f_1, f_2, \ldots, f_m]_{f_1}) + range([f_1, f_2, \ldots, f_m]_{\overline{f_1}}) \\ &= range([1, f_2, \ldots, f_m]_{f_1}) + range([0, f_2, \ldots, f_m]_{\overline{f_1}}) \\ &= y_1 range([f_2, \ldots, f_m]_{f_1}) + \overline{y}_1 range([f_2, \ldots, f_m]_{\overline{f_1}}) \end{aligned}$$

If a particular component f_i becomes constant, the corresponding y_i in positive or complement form depending on the value of f_i replaces f_i in the range computation. The image of f for a set $A(\mathbf{x})$ is obtained by first finding $f_{A(\mathbf{x})}$ and then applying one of the input or output cofactoring techniques.

One can also apply a combination of input and output cofactoring. The ordering in which cofactoring is done is very important. The procedure can be sped up dramatically by caching intermediate range computations, and detecting the case where, at any step in the recursion, the functions $[f_1, \ldots, f_m]$ can be grouped into two or more sets with disjoint support. In this case, the range computation can proceed independently on each group with disjoint support. This reduces the worst case complexity from 2^m to $2^{s_1} + \ldots + 2^{s_k}$, where $(s_1, \ldots, s_k)$ are the sizes of the groups with disjoint support ($s_1 + \ldots + s_k = m$).

Example:

We find the range of the function $f = [f_1, \ldots, f_5]$ using the transition relation method, input cofactoring method, and output cofactoring method.

$$\begin{aligned} f_1 &= x_1 x_2 x_3 \\ f_2 &= x_1 + x_2 + x_3 \\ f_3 &= \overline{x}_1 \overline{x}_2 \overline{x}_3 \\ f_4 &= x_4 \overline{x}_5 \\ f_5 &= \overline{x}_4 x_5 \end{aligned}$$

The characteristic function for f is

$$F(\mathbf{x}, \mathbf{y}) = (y_1 \overline{\oplus} x_1 x_2 x_3)(y_2 \overline{\oplus} (x_1 + x_2 + x_3))(y_3 \overline{\oplus} \overline{x}_1 \overline{x}_2 \overline{x}_3)(y_4 \overline{\oplus} x_4 \overline{x}_5)(y_5 \overline{\oplus} \overline{x}_4 x_5).$$

Using the transition relation method the range of f is

$$f(\mathbf{y}) = S_{\mathbf{x}} F(\mathbf{x}, \mathbf{y})$$

$$= \mathcal{S}_{x_1x_2x_3}(y_1 \overline{\oplus} x_1x_2x_3)(y_2 \overline{\oplus}(x_1+x_2+x_3))(y_3 \overline{\oplus} \overline{x}_1\overline{x}_2\overline{x}_3)\ \mathcal{S}_{x_4x_5}(y_4 \overline{\oplus} x_4\overline{x}_5)(y_5 \overline{\oplus} \overline{x}_4x_5)$$
$$= (y_2\overline{y}_3 + \overline{y}_1\overline{y}_2y_3)(\overline{y}_4+\overline{y}_5).$$

We use the fact that the components of f can be separated into groups of disjoint support and Lemma 2.2.1 to speed up the range computation.

The range can be also computed by cofactoring output functions with respect to each input. The components $\{f_1, f_2, f_3\}$ are grouped together because they are dependent on $\{x_1, x_2, x_3\}$ and the components $\{f_4, f_5\}$ are grouped together because they are dependent on $\{x_4, x_5\}$.

$$range([f_1,f_2,f_3,f_4,f_5]) = range([f_1,f_2,f_3])\ range([f_4,f_5])$$

We use the ordering $x_1 \succ x_2 \succ x_3$ to find

$$\begin{aligned}
range([f_1,f_2,f_3]) &= range([f_{1x_1}, f_{2x_1}, f_{3x_1}]) + range([f_{1\overline{x}_1}, f_{2\overline{x}_1}, f_{3\overline{x}_1}]) \\
&= range([x_2x_3, 1, 0]) + range([0, x_2+x_3, \overline{x}_2\overline{x}_3]) \\
&= y_2\overline{y}_3 range([x_2x_3]) + \overline{y}_1 range([x_2+x_3, \overline{x}_2\overline{x}_3]) \\
&= y_2\overline{y}_3 + \overline{y}_1(range([x_2+x_3, \overline{x}_2\overline{x}_3]_{x_2}) + range([x_2+x_3, \overline{x}_2\overline{x}_3]_{\overline{x}_2})) \\
&= y_2\overline{y}_3 + \overline{y}_1 range([1,0]) + \overline{y}_1 range([x_3, \overline{x}_3]) \\
&= y_2\overline{y}_3 + \overline{y}_1 range([x_3, \overline{x}_3]) \\
&= y_2\overline{y}_3 + \overline{y}_1(range([x_3, \overline{x}_3]_{x_3}) + range([x_3, \overline{x}_3]_{\overline{x}_3})) \\
&= y_2\overline{y}_3 + \overline{y}_1 range([1,0]) + \overline{y}_1 range([0,1]) \\
&= y_2\overline{y}_3 + \overline{y}_1\overline{y}_2y_3.
\end{aligned}$$

The ordering $x_4 \succ x_5$ is used to find

$$\begin{aligned}
range([f_4,f_5]) &= range([x_4\overline{x}_5, \overline{x}_4x_5]) \\
&= range([x_4\overline{x}_5, \overline{x}_4x_5]_{x_4}) + range([x_4\overline{x}_5, \overline{x}_4x_5]_{\overline{x}_4}) \\
&= range([\overline{x}_5, 0]) + range([0, x_5]) \\
&= \overline{y}_5 + \overline{y}_4
\end{aligned}$$

Finally, we show how this can be done using output cofactoring. The ordering $x_1 \succ x_2 \succ x_3 \succ x_4 \succ x_5$ is used for the generalized cofactor. As before

$$range([f_1,f_2,f_3,f_4,f_5]) = range([f_1,f_2,f_3])\ range([f_4,f_5])$$

For output cofactoring, we use $f_1 \succ f_2 \succ f_3$ to find

$$\begin{aligned} range([f_1, f_2, f_3]) &= range([f_{1f_1}, f_{2f_1}, f_{3f_1}]) + range([f_{1\overline{f}_1}, f_{2\overline{f}_1}, f_{3\overline{f}_1}]) \\ &= range([1,1,0]) + range([0, x_1+x_2+x_3, \overline{x}_1\overline{x}_2\overline{x}_3]) \\ &= y_1y_2\overline{y}_3 + \overline{y}_1 range([x_1+x_2+x_3, \overline{x}_1\overline{x}_2\overline{x}_3]) \\ &= y_1y_2\overline{y}_3 + \overline{y}_1 range([1,0]) + \overline{y}_1 range([0,1]) \\ &= y_2\overline{y}_3 + \overline{y}_1\overline{y}_2y_3 \end{aligned}$$

Here we use $gcofactor(x_1+x_2+x_3, \overline{x}_1+\overline{x}_2+\overline{x}_3) = x_1+x_2+x_3$, $gcofactor(\overline{x}_1\overline{x}_2\overline{x}_3, \overline{x}_1+\overline{x}_2+\overline{x}_3) = \overline{x}_1\overline{x}_2\overline{x}_3$, and $gcofactor(\overline{x}_1\overline{x}_2\overline{x}_3, x_1+x_2+x_3) = 0$ to find $range[f_1, f_2, f_3]$. Also using $f_4 \succ f_5$ and $gcofactor(\overline{x}_4x_5, \overline{x}_4+x_5) = \overline{x}_4x_5$, we get

$$\begin{aligned} range([f_4, f_5]) &= range([f_{4f_4}, f_{5f_4}]) + range([f_{4\overline{f}_4}, f_{5\overline{f}_4}]) \\ &= range([1,0]) + range([0, \overline{x}_4x_5]) \\ &= \overline{y}_4 + \overline{y}_5 \end{aligned}$$

One can mix all three methods to obtain the most efficient implementation for image computation. The choice of one of these techniques over the others depends on particular application. For FSM traversal, the transition relation method in [78] and the input cofactoring method in [19] gave comparable results. Improvements to the choice of variable for input cofactoring were given in [41]. The output cofactoring is preferred for computing local don't cares as discussed later, because the number of output variables is usually much less than input variables.

2.4 Observability Relations

Observability relations or characteristic functions were introduced by Cerny [17]. Later the notion of a general Boolean relation [12] was discussed and derived for a hierarchy of networks.

Definition 2.4.1 *A Boolean relation is a one-to-many multi-output Boolean mapping* $\mathcal{R}$: $B^n \longrightarrow B^l$. *In general* $\mathcal{R}(\mathbf{x}) \subseteq B^l$ *is a set.*

Definition 2.4.1 *A Boolean relation is well-defined if there exists at least one minterm* $\mathbf{z} \in B^l$ *for every* $\mathbf{x} \in B^n$ *such that* $\mathbf{z} \in \mathcal{R}(\mathbf{x})$.

A general way to specify a combinational circuit $\mathcal{N}$ is to use a Boolean relation. This relation gives all the output combinations possible for a particular input.

Definition 2.4.2 *An observability relation [17],* $\mathcal{O} : B^{n+l} \longrightarrow B$ *is the characteristic function of the Boolean relation* $\mathcal{R}(\mathbf{x})$ *which describes the input-output behavior of circuit. The observability relation is defined as* $\mathcal{O}(\mathbf{x},\mathbf{z}) = \{(\mathbf{x},\mathbf{z}) | \mathbf{x} \in B^n, \mathbf{z} \in B^l, \mathbf{z} \in \mathcal{R}(\mathbf{x})\}$.

Example:

Let $z_1 = f_1 = x_1x_2$ and $z_2 = f_2 = x_1 + x_2$ be the output functions of a Boolean network. If the network is completely specified, its observability relation is

$$\mathcal{O}(\mathbf{x},\mathbf{z}) = x_1x_2z_1z_2 + x_1\overline{x}_2\overline{z}_1z_2 + \overline{x}_1x_2\overline{z}_1z_2 + \overline{x}_1\overline{x}_2\overline{z}_1\overline{z}_2.$$

Notice that a particular output combination is generated for each input combination. There is at least one output combination for each input combination; therefore, $\mathcal{O}(\mathbf{x},\mathbf{z})$ is well defined. If the outputs can be either z_1z_2 or $\overline{z}_1\overline{z}_2$ for x_1x_2, the observability relation is

$$\mathcal{O}(\mathbf{x},\mathbf{z}) = x_1x_2(z_1z_2 + \overline{z}_1\overline{z}_2) + x_1\overline{x}_2\overline{z}_1z_2 + \overline{x}_1x_2\overline{z}_1z_2 + \overline{x}_1\overline{x}_2\overline{z}_1\overline{z}_2.$$

If the output z_1 can be either 0 or 1 for the input x_1x_2, the observability relation is

$$\mathcal{O}(\mathbf{x},\mathbf{z}) = x_1x_2z_2 + x_1\overline{x}_2\overline{z}_1z_2 + \overline{x}_1x_2\overline{z}_1z_2 + \overline{x}_1\overline{x}_2\overline{z}_1\overline{z}_2.$$

The term $x_1x_2z_2$ implies that z_1 can be either 1 or 0 for input x_1x_2. We will show later that x_1x_2 is actually the external don't set for output z_1.

Chapter 3

Don't Care Conditions for Single-Output Nodes

The observability relation $\mathcal{O}(\mathbf{x}, \mathbf{z})$ (as defined by Cerny [17]) or Boolean relation (discussed in [12]) provides a description of all the flexibility one has in implementing a Boolean network $\mathcal{N}$. In this chapter, we propose to represent and use this flexibility in a logic synthesis system by adding a single output node to the Boolean network $\mathcal{N}$. The node function for the new node is $\mathcal{O}(\mathbf{x}, \mathbf{z})$. The newly constructed network $\mathcal{N}'$ (called the observability network) has only *one output* and computes 1 for every input $\mathbf{x}$. We show that the observability don't cares (ODC's) for a node y_i in $\mathcal{N}'$ provide the maximum flexibility for implementing y_i and subsume the flexibility obtained for y_i in $\mathcal{N}$ even with don't cares provided at each output. This gives rise to new methods for computing complete ODC's for $\mathcal{N}'$ and hence for $\mathcal{N}$.

In practice, it is not always possible to compute full ODC's for all the nodes in the network. We consider subsets of ODC's which can be computed efficiently. Compatible ODC subsets have the added advantage that functions at the nodes of the network can be optimized using their ODC subsets simultaneously. We develop techniques for computing compatible ODC's for complex nodes of a multi-level network.

3.1 Introduction

An important part of logic synthesis is the node simplification phase where the local function at each node in a Boolean network is minimized using a two-level minimizer

(such as ESPRESSO [11]) and using don't cares derived from the environment of the node. These don't cares arise from various sources; external don't cares (EDC), satisfiability don't cares (SDC), and observability don't cares (ODC). It was shown in [6] that if a node is minimized using all three types of don't cares, then each connection to and inside that node is irredundant (and hence there exists a test for both stuck-at-1 and stuck-at-0 for each connection). In this sense the don't cares so defined are *complete.*

This theory is explicitly for the case where each node in the Boolean network is a single output node. However, one can view a Boolean network itself as a single node with multiple-outputs, for which a complete don't care theory is missing. This lack becomes especially apparent when attempting to specify and use external don't cares for combinational logic minimization. In practice, each separate output of a network is given an external don't care set $d_i(\mathbf{x})$. It has been observed that these must be independent or *compatible* [59]. But such external don't cares can never be complete since they cannot provide *all* the flexibility allowed in simplifying a circuit [12]. To circumvent this, Boolean relations were defined and techniques for finding minimal sum-of-products representations implementing a Boolean relation were given. Previously, Cerny [17] had defined the observability relation for a circuit. These ideas form the basis for a complete theory of don't cares for multiple-output nodes of logic networks. This chapter integrates these ideas and provides techniques for computing the full flexibility allowed for minimizing a logic network by expanding on the work presented in [68].

We define for a given Boolean network $\mathcal{N}$, an observability network $\mathcal{N}'$, by adding a **single** node whose logic function is the observability relation $\mathcal{O}(\mathbf{x},\mathbf{z})$ for $\mathcal{N}$. We then show that the regular treatment of ODC's for $\mathcal{N}'$ includes all the flexibility allowed by the Boolean relation.

The idea of having an extra node on top of network $\mathcal{N}$ to represent its Boolean relation was originally suggested in [33]. The combined network was called the interconnection network. An initial network $\mathcal{N}$ compatible with the Boolean relation is derived and minimized using ATPG redundancy removal techniques. Later, in [31] it was proposed to put the characteristic function (observability relation) of the Boolean relation as an extra network on top of the Boolean network. The MSPF's [59] are then computed from this new interconnected network for each intermediate node of $\mathcal{N}$ in terms of primary inputs. These are used to optimize each node using the techniques of [66]. It was also mentioned that this gives the maximum flexibility at each node.

The approach presented in this chapter generalizes these notions in the sense that the observability relation is defined for all networks, regardless of whether Boolean relations, external don't cares, or completely specified functions are considered. Unlike [31] where MSPF's are considered in terms of primary inputs, we compute ODC's in terms of both intermediate variables and primary inputs. The fact that this leads to more flexibility was used in [67] and also commented on in [24]. Here we make these notions more precise. Using this we give an algorithm for incrementally computing complete ODC's in topological order while visiting each node only once. The algorithm does not require representing ODC's in vector form nor associating variables to each edge as suggested in [23].

The computation of full ODC's at each node of a multi-level network is expensive and impractical for large circuits. In practice, subsets of ODC are computed. We present techniques for computing compatible observability don't care subsets at each node of the network. These are computed for complex nodes and can be used for the optimization of each node independent of sets computed for other nodes.

3.2 Don't Cares in a Boolean Network

The don't care conditions in a Boolean network are divided into three groups, satisfiability don't cares (SDC's) and observability don't cares (ODC's) which are related to the structure of the network, and external don't cares (EDC's) which are usually supplied by the user.

3.2.1 Satisfiability Don't Cares

A multi-level network $\mathcal{N}$ with n primary inputs and m intermediate nodes is given. The n primary inputs result in 2^n *input combinations or minterms* in the space B^n.

Definition 3.2.1 *If y_i is the variable at an intermediate node and f_i its logic function, then $y_i = f_i$; therefore, we don't care if $y_i \neq f_i$ (i.e. $y_i \oplus f_i$ is don't care). The expression $SDC = \sum_i (y_i \oplus f_i)$ is called the satisfiability don't care set.*

Simulation of $\mathcal{N}$ with a particular input minterm forces the value of each intermediate node to either 0 or 1. Some combination of values on the nodes are possible and some are not possible. The SDC of the network contains all the impossible combinations in B^{n+m}. The number of these combinations is exactly $2^{m+n} - 2^n$.

When we simplify an intermediate node y_i of a multi-level Boolean network, we usually use a subset of the SDC derived from a subset of the nodes that can be substituted with "high probability" into the node being optimized. A two-level minimizer effectively substitutes some set of variables, corresponding to the nodes of the network, into f_i.

Example:

Let the following nodes be the intermediate nodes of a multi-level network where a, b, c, d, e, f are primary inputs.

$$\begin{aligned} t &= s\bar{k} + \bar{s}abcd + \bar{s}\bar{a}\bar{b}cd \\ k &= ab + \bar{a}\bar{b} \\ s &= ef + \bar{e}\bar{f} \\ r &= cd \end{aligned}$$

If we simplify t using the SDC's,

$$(k \oplus (ab + \bar{a}\bar{b})) + (s \oplus (ef + \bar{e}\bar{f})) + (r \oplus (cd))$$

we obtain $t = s\bar{k} + \bar{s}kr$. Thus, in addition to s and k, r has also been substituted into the function representing t. The Boolean function at t as a function of the primary input variables (called its *global function*) has not changed since only the satisfiability don't cares were used for the simplification [53]; equivalently, t has not changed in B^n.

3.2.2 Observability Don't Cares

Given a Boolean network $\mathcal{N}$, there are global functions associated with each of the intermediate nodes of the network which give a specified value for each input minterm. At times, it is possible to change the global function at a node y_o without observing any change at any of the outputs of the network. The observability don't cares computed for y_o give all such conditions.

Definition 3.2.2 *The observability don't cares (ODC's) at each intermediate node y_o of a multi-level network are conditions under which y_o can be either 1 or 0 while the functions generated at each primary output remain unchanged. If $\mathbf{z} = (z_1, \ldots, z_l)$, then the complete ODC at node y_o is*

$$ODC_o = \{m \in B^n | \mathbf{z}_{y_o}(m) \equiv \mathbf{z}_{\bar{y}_o}(m)\}.$$

Thus $ODC_o = \prod_{i=1}^{l} \overline{\frac{\partial z_i}{\partial y_o}}$.

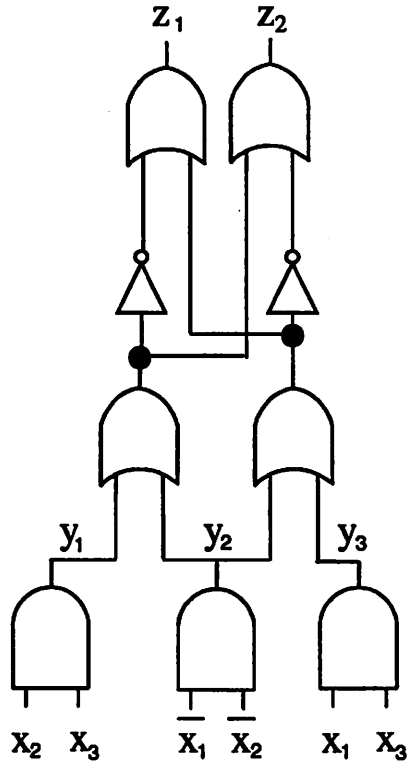

Figure 3.1: Example

The ODC of an intermediate node y_o need not be expressed in terms of primary inputs. It can also be expressed in terms of intermediate nodes of the network.

Example:

As shown in Figure 3.1, $ODC_2 = y_1y_3 + \overline{y}_1\overline{y}_3$. This means that whenever both y_1 and y_3 are 1 or both are 0, the value of y_2 has no effect at any of the outputs. ODC_2 can be reexpressed in terms of the inputs x_1, x_2, and x_3. The ability to express the ODC's in terms of various sets of variables is important to node simplification. Once we have $ODC_2 = \overline{x}_1\overline{x}_2 + x_1x_2 + \overline{x}_3$, we can set y_2 (initially $y_2 = \overline{x}_1\overline{x}_2$) to 0.

Another interesting fact is that the ODC at a node expressed in terms of intermediate variables can intersect the SDC of the network. The cube $y_1y_2y_3$ is in ODC_2 and also in SDC of the network in Figure 3.1 because y_1 and y_2 cannot be equal to 1 at the same time under any input combination.

3.2.3 External Don't Cares

In general, we don't care about the value of every single output for every single input combination.

Definition 3.2.3 *The external don't care set for each output z_i of the network $\mathcal{N}$ is all the input combinations for which the value of z_i is not important.*

Definition 3.2.4 *The external don't care sets for the outputs of the network are compatible if each output function can be changed as allowed by its external don't care set irrespective of the changes made to other outputs as allowed by their external don't care sets.*

The EDC's and ODC's can be used to find flexibilities in implementing each intermediate node of a multi-level network. There can be conditions that are not captured by EDC's and ODC's alone, but this combination is what is used in practice most often and is very effective.

3.2.4 Terminology

We represent the observability plus external don't care set at node y_i by d_i. If this is a global function, it is denoted by d_i^g. If d_i is a local function in terms of the fanins of the node, it is denoted by d_i^l. If d_i is a compatible don't care subset (defined later), it is denoted by d_i^c; if it is a maximal subset, it is denoted by d_i^m. If d_i is computed with respect to a particular output z_l, it is denoted by $d_{i;l}$. The don't care set for an edge connecting nodes y_i and y_j is denoted by d_{ij} [1]. For example, the global, compatible don't care for the edge (i,j) with respect to output k is $d_{ij;k}^{cg}$. If it is clear from the context what kind of don't care we are referring to, some superscripts or subscripts may be deleted.

3.3 Observability Network

The observability relation as defined in 2.4 can express the behavior of networks that are completely specified, those with external don't cares, and those with a given Boolean relation describing the input-output behavior of the circuit.

[1] The function at an edge is the same as the function of its fanin node. The don't care set at an edge is all the conditions under which the value of that edge can be either 1 or 0.

If the network $\mathcal{N}$ is completely specified and $g_1, \ldots, g_l$ are the global functions of $\mathcal{N}$, expressing each primary output in terms of primary inputs, the observability relation of $\mathcal{N}$ is

$$\mathcal{O}(\mathbf{x}, \mathbf{z}) = (z_1 \overline{\oplus} g_1)(z_2 \overline{\oplus} g_2) \ldots (z_l \overline{\oplus} g_l).$$

External don't cares are another special case of Boolean relations; they are specified for each output and are input combinations under which that output can have any value. If the external don't cares $d_1, \ldots, d_l$, are given in terms of primary inputs, the observability relation of $\mathcal{N}$ is

$$\mathcal{O}(\mathbf{x}, \mathbf{z}) = \prod_{i=1}^{l} (d_i + z_i \overline{\oplus} g_i).$$

In the most general case, the user gives an observability relation which expresses the input-output behavior of the network.

To represent and use the flexibility supplied by the observability relation in a synthesis system we propose the following.

Definition 3.3.1 *The observability network $\mathcal{N}'$ of $\mathcal{N}$ is derived by adding one additional node $\mathcal{O}$ to $\mathcal{N}$. The logic function for this node is the observability relation $\mathcal{O}(\mathbf{x}, \mathbf{z})$ of $\mathcal{N}$. $\mathcal{N}'$ has only one output, namely $\mathcal{O}$, and $\mathcal{O}$ has $n + l$ inputs, namely all inputs and outputs of $\mathcal{N}$.*

$\mathcal{N}'$ has many interesting properties that can be used for optimization and verification of $\mathcal{N}$. $\mathcal{N}$ with global output functions $g_1, \ldots, g_l$ is compatible with its observability relation if $\mathcal{O}(\mathbf{x}, g(\mathbf{x})) \equiv 1$ because each input minterm $\mathbf{x}$ produces an output minterm $g(\mathbf{x})$ that is allowed by the observability relation. If $\mathcal{N}$ is compatible, the output value of $\mathcal{N}'$ is always 1 no matter what the input is. Thus, $\mathcal{N}'$ is the tautology. Logic synthesis techniques can be used to optimize $\mathcal{N}$. The optimized network is valid if and only if the output of $\mathcal{N}'$ is always 1 [2].

The observability relation allows defining external don't cares not just in terms of primary inputs, but also primary outputs. It turns out that external don't cares defined this way (using output variables) need not be compatible (see section 3.5).

Lemma 3.3.1 *Let y_i be any node in $\mathcal{N}$ (y_i is also in $\mathcal{N}'$) and $\mathcal{O}$ the output of $\mathcal{N}'$. Then, $\overline{\frac{\delta \mathcal{O}}{\delta y_i}} = \mathcal{O}_{y_i} \mathcal{O}_{\overline{y}_i}$.*

[2] Thus one technique for verifying a combinational circuit $\mathcal{N}$ is to build the BDD for its observability network $\mathcal{N}'$ in terms of the primary inputs and check whether it is the tautology.

Proof Because $\mathcal{N}'$ is a tautology, and for any input combination either $y_i = 1$ or $y_i = 0$, then $\mathcal{O}_{y_i} = 1$ or $\mathcal{O}_{\overline{y}_i} = 1$. As a result, $\overline{\mathcal{O}}_{y_i}\overline{\mathcal{O}}_{\overline{y}_i} = 0$. $\overline{\frac{\partial \mathcal{O}}{\partial y_i}} = \mathcal{O}_{y_i}\mathcal{O}_{\overline{y}_i} + \overline{\mathcal{O}}_{y_i}\overline{\mathcal{O}}_{\overline{y}_i} = \mathcal{O}_{y_i}\mathcal{O}_{\overline{y}_i}$. ∎

Theorem 3.3.2 *Let $\mathcal{O}(\mathbf{x}, \mathbf{z})$ be an observability relation and $\mathcal{N}$ implement compatible global functions $g(\mathbf{x})$, i.e. $\mathcal{O}(\mathbf{x}, g(\mathbf{x})) = 1$. Let y_i be any node in $\mathcal{N}$ with global function f_i^g and let ODC_i be the complete ODC for node i in $\mathcal{N}'$. Then network $\tilde{\mathcal{N}}$, obtained by replacing f_i^g by $\tilde{f}_i^g$, is compatible if and only if $f_i^g\overline{ODC}_i \leq \tilde{f}_i^g \leq f_i^g + ODC_i$.*

Proof If the change in node y_i for some particular input $\mathbf{x}$ is not observable in $\mathcal{N}'$, then because of Lemma 3.3.1 the value of the output function $\mathcal{O}$ must be 1 before and after the change. Therefore if any change occurs at the outputs of $\mathcal{N}$, it is allowed by the observability relation meaning the new network is compatible. On the other hand, if the change at node y_i for some particular input $\mathbf{x}$ is observable at the output of $\mathcal{N}'$, then the output of the observability network for $\mathbf{x}$ must be 0 which means the new network is not compatible. ∎

Theorem 3.3.3 *If there are no Boolean relations or external don't cares, the complete ODC computed for y_i from $\mathcal{N}$ is equal to that computed from $\mathcal{N}'$.*

Proof If the change in node y_i for some particular input $\mathbf{x}$ is not observable in $\mathcal{N}$, it will not be observable in $\mathcal{N}'$, because the input and the output of $\mathcal{N}$ remain the same after the change, and the characteristic function $\mathcal{O}$ evaluates to 1 before and after the change. Therefore ODC_i *in* $\mathcal{N} \subseteq ODC_i$ *in* $\mathcal{N}'$. On the other hand, if the change in y_i for some particular input $\mathbf{x}$ is not observable in $\mathcal{N}'$, then, since there are no Boolean relations or external don't cares, for each input minterm $\mathbf{x}$, there is only one output minterm $\mathbf{z}$ such that $\mathcal{O}(\mathbf{x}, \mathbf{z}) = 1$. As a result, a change not observable in $\mathcal{N}'$ is also not observable in $\mathcal{N}$. ∎

3.4 Computing ODC's

In [23] a method is described for computing complete ODC's at every node of a multi-level network. The ODC at a node is computed recursively in terms of the ODC's at all its fanout edges. The described method uses ODC's in vector form, where each element is the ODC with respect to a single primary output. Each such element is computed separately at each node. We describe another method for computing complete ODC's, and drive the result in [23] using this new formulation. We show that the ODC's can be computed without

using the vector form and the edge ODC's as proposed in [23] by using the observability relation.

We express the ODC of an intermediate node in terms of variables of some separating set of nodes in the network.

Definition 3.4.1 *A separating set of nodes is a minimal set of nodes which separates all primary outputs from primary inputs if all the fanout edges of the nodes in the set are removed.*

Let $\mathcal{Y} = (y_1, y_2, \ldots, y_p)$ be a separating set of nodes in network $\mathcal{N}$ and y_o an intermediate node in $\mathcal{N}$ as shown in Figure 3.3. A fundamental result used throughout this section is the ability to rebuild a function f if the observability don't cares with respect to f are known for each variable corresponding to a node in the separating set.

Theorem 3.4.1 *Given* $\overline{\frac{\partial f}{\partial y_1}}, \overline{\frac{\partial f}{\partial y_2}}, \ldots, \overline{\frac{\partial f}{\partial y_p}}$, *the vertices in* B^p *can be divided into two sets, one the offset and the other the onset of* f.

To prove this, we give an iterative procedure. Let $F^0 = 1$. At the jth step do,

$$F^j = (y_j F^{j-1} + \overline{y_j}\,\overline{F}^{j-1})(\frac{\partial f}{\partial y_j})_{y_{j+1}\ldots y_p} + F^{j-1}(\overline{\frac{\partial f}{\partial y_j}})_{y_{j+1}\ldots y_p}. \tag{3.1}$$

After the pth iteration we have a set F^+. This operation is very similar to integration of continuous functions where a function is built from its partial derivatives.

Lemma 3.4.2 *If* $f_{y_1\ldots y_p} = 1$ *then* $f = F^+$, *otherwise* $\overline{f} = F^+$.

Proof Assume $f_{y_1\ldots y_p} = 1$. The other case is symmetrical. Then $F^0 = f_{y_1\ldots y_p}$. By induction, assume $F^{j-1} = f_{y_j\ldots y_p}$. Since $(\overline{\frac{\partial f}{\partial y_j}})_{y_{j+1}\ldots y_p} = (f_{y_j} f_{\overline{y_j}})_{y_{j+1}\ldots y_p} + (\overline{f}_{y_j}\overline{f}_{\overline{y_j}})_{y_{j+1}\ldots y_p}$ and $F^{j-1}\overline{f}_{y_j\ldots y_p} = \overline{F}^{j-1} f_{y_j\ldots y_p} = 0$, (3.1) gives $F^j = y_j F^{j-1}\overline{f}_{\overline{y}_j y_{j+1}\ldots y_p} + \overline{y}_j\overline{F}^{j-1} f_{\overline{y}_j y_{j+1}\ldots y_p} + F^{j-1} f_{\overline{y}_j y_{j+1}\ldots y_p}$. The first and third terms simplify to $y_j F^{j-1} = y_j f_{y_j\ldots y_p}$. The second and third terms simplify to $\overline{y}_j f_{\overline{y}_j y_{j+1}\ldots y_p}$. Therefore

$$\begin{aligned}
F^j &= y_j F^{j-1}\overline{f}_{\overline{y}_j y_{j+1}\ldots y_p} + \overline{y}_j\overline{F}^{j-1} f_{\overline{y}_j y_{j+1}\ldots y_p} + F^{j-1} f_{\overline{y}_j y_{j+1}\ldots y_p} \\
&= y_j F^{j-1} + \overline{y}_j f_{\overline{y}_j y_{j+1}\ldots y_p} + F^{j-1} f_{\overline{y}_j y_{j+1}\ldots y_p} \\
&= y_j F^{j-1} + \overline{y}_j f_{\overline{y}_j y_{j+1}\ldots y_p} \\
&= y_j f_{y_j\ldots y_p} + \overline{y}_j f_{\overline{y}_j y_{j+1}\ldots y_p} \\
&= f_{y_{j+1}\ldots y_p}.
\end{aligned}$$

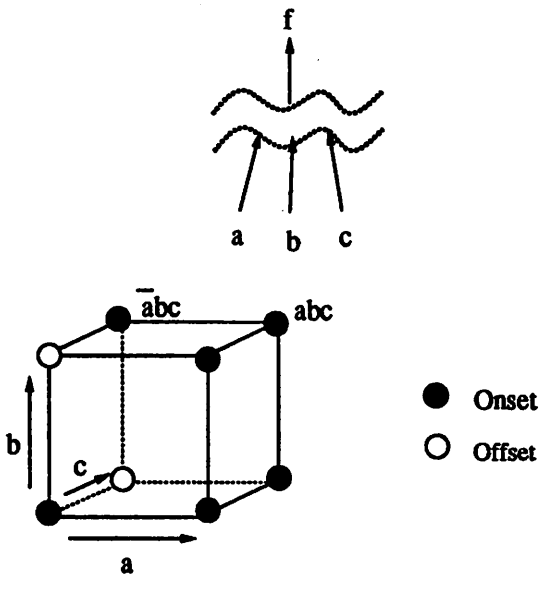

Figure 3.2: Example

Thus $F^p = F^+ = f$. ∎

Corollary 3.4.3 *Given* $(\overline{\frac{\partial f}{\partial y_1}})_{y_2 \ldots y_p}, (\overline{\frac{\partial f}{\partial y_2}})_{y_3 \ldots y_p}, \ldots, \overline{\frac{\partial f}{\partial y_p}}$, *the vertices in* B^p *can be divided into two sets, the offset and the onset of* f.

Corollary 3.4.3 implies that less information is needed to rebuild the function f than $\overline{\frac{\partial f}{\partial y_1}}, \ldots, \overline{\frac{\partial f}{\partial y_p}}$.

Example:

Figure 3.2 shows the onset and the offset for a function $f = a + bc + \bar{b}\bar{c}$. $\overline{\frac{\partial f}{\partial a}} = bc + \bar{b}\bar{c}$, $\overline{\frac{\partial f}{\partial b}} = a$, and $\overline{\frac{\partial f}{\partial c}} = a$. We start with the vertex abc and find all the other vertices that are in the same set as abc. We know that abc is in the onset of f. Because $f_{abc} = 1$, F^+ is the onset of f. The starting point in the iteration is $F^0 = f_{abc} = 1$. The iteration is done using the order $y_1 = a, y_2 = b$, and $y_3 = c$ in formula (3.1). $abc \in \overline{\frac{\partial f}{\partial a}}$, thus $\bar{a}bc$ and abc must be in the same set giving $F^1 = f_{bc} = 1$. The fact that $\bar{a}bc \notin \overline{\frac{\partial f}{\partial b}}$ and $abc \in \overline{\frac{\partial f}{\partial b}}$ gives $F^2 = f_c = a + b$, and finally $F^3 = c(a+b) + \bar{c}(a+b)a + \bar{c}(\bar{a}\bar{b})\bar{a}$ which results in $F^+ = f = a + bc + \bar{b}\bar{c}$.

3.4.1 A New Approach

We show that by using (3.1) the ODC of a node can be computed if the ODC of each of its fanout nodes is known. Let node y_o have fanouts $y_1, y_2, \ldots, y_r$ as shown in Figure

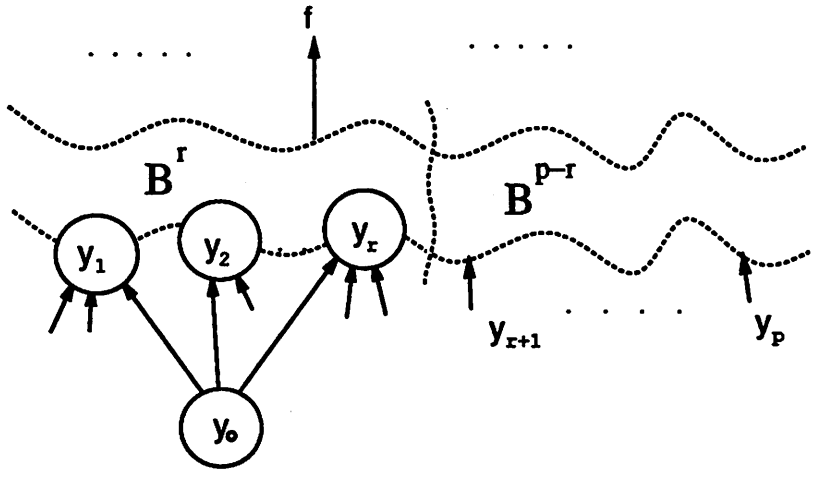

Figure 3.3: A Separating Set of Nodes

3.3 [3]. The conditions under which $y_1, \ldots y_r$ are not observable at f, i.e., $\overline{\frac{\partial f}{\partial y_1}}, \overline{\frac{\partial f}{\partial y_2}}, \ldots, \overline{\frac{\partial f}{\partial y_r}}$, are given in terms of the separating set variables $y_1, \ldots y_p$. Let $\mathcal{O}^0 = 1$ and

$$\mathcal{O}^j = (y_j \mathcal{O}^{j-1} + \overline{y_j}\,\overline{\mathcal{O}}^{j-1})(\frac{\partial f}{\partial y_j})_{y_{j+1} \ldots y_r} + \mathcal{O}^{j-1}(\overline{\frac{\partial f}{\partial y_j}})_{y_{j+1} \ldots y_r}. \tag{3.2}$$

Theorem 3.4.4 *Let $\mathcal{O}^+ = \mathcal{O}^r$ with the fanout variables $y_1, \ldots, y_r$ of y_o eliminated by substituting $y_i = f_i$. Then $ODC_o = \overline{\frac{\partial \mathcal{O}^+}{\partial y_o}}$.*

Proof We prove this by introducing a set of two-partitions in B^r, one for each minterm in B^{p-r} as follows. For each $m_j \in B^{p-r}$, compute $(\overline{\frac{\partial f}{\partial y_1}})_{m_j}, (\overline{\frac{\partial f}{\partial y_2}})_{m_j}, \ldots, (\overline{\frac{\partial f}{\partial y_r}})_{m_j}$. By using (3.1), and $(\overline{\frac{\partial f}{\partial y_1}})_{m_j}, (\overline{\frac{\partial f}{\partial y_2}})_{m_j}, \ldots, (\overline{\frac{\partial f}{\partial y_r}})_{m_j}$, partition B^r into $P_j = (F_j^+, \overline{F}_j^+)$. Find such partitions for all $m_j \in B^{p-r}$.

By Theorem 3.4.1, the onset of f_{m_j} is one set of partition P_j and the offset is the other set. Without loss of generality, assume f is such that $f = m_1 F_1^+ + m_2 F_2^+ + \ldots + m_{2^{p-r}} F_{2^{p-r}}^+$. Consequently, $\overline{f} = m_1 \overline{F}_1^+ + m_2 \overline{F}_2^+ + \ldots + m_{2^{p-r}} \overline{F}_{2^{p-r}}^+$. Now eliminate the fanout variables $y_1, \ldots, y_r$ of y_o, by replacing $y_i = f_i$. Thus all dependencies on y_o are given explicitly and

$$ODC_o = \overline{\frac{\partial f}{\partial y_o}} = C_{y_o} f + C_{y_o} \overline{f} = C_{y_o} \sum (m_j F_j^+) + C_{y_o} \sum (m_j \overline{F}_j^+). \tag{3.3}$$

[3] For simplicity, the fanout nodes of y_o ,$y_1, \ldots, y_r$, are assumed to be the first r nodes in a separating set $\mathcal{Y}$.

F_j^+ is computed as follows using (3.1):

$$\begin{aligned} F_j^1 &= (y_1 F_j^0 + \overline{y_1}\overline{F_j^0})(\frac{\partial f}{\partial y_1})_{y_2 \ldots y_r m_j} + F_j^0(\overline{\frac{\partial f}{\partial y_1}})_{y_2 \ldots y_r m_j} \\ &= [(y_1 \mathcal{O}^0 + \overline{y_1}\overline{\mathcal{O}^0})(\frac{\partial f}{\partial y_1})_{y_2 \ldots y_r} + \mathcal{O}^0(\overline{\frac{\partial f}{\partial y_1}})_{y_2 \ldots y_r}]_{m_j} = (\mathcal{O}^1)_{m_j} \end{aligned}$$

Therefore, by induction it follows easily that

$$F_j^+ = F_j^r = [(y_r \mathcal{O}^{r-1} + \overline{y_r}\overline{\mathcal{O}^{r-1}})\frac{\partial f}{\partial y_r} + \mathcal{O}^{r-1}\overline{\frac{\partial f}{\partial y_r}}]_{m_j} = (\mathcal{O}^+)_{m_j}.$$

By (3.3)

$$\begin{aligned} ODC_o &= C_{y_o}\sum(m_j \mathcal{O}^+_{m_j}) + C_{y_o}\sum(m_j \overline{\mathcal{O}}^+_{m_j}) \\ &= (\sum m_j) C_{y_o}\mathcal{O}^+ + (\sum m_j) C_{y_o}\overline{\mathcal{O}}^+ \\ &= C_{y_o}\mathcal{O}^+ + C_{y_o}\overline{\mathcal{O}}^+ = \overline{\frac{\partial \mathcal{O}^+}{\partial y_o}}. \end{aligned}$$

■

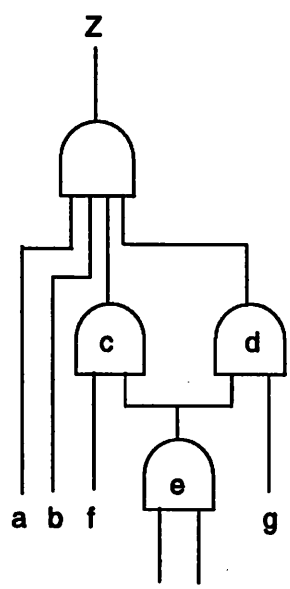

Figure 3.4: Example

Example:

In the example shown in Figure 3.4, the ODC at node e can be computed using $\overline{\frac{\partial z}{\partial c}} = \overline{a}+\overline{b}+\overline{d}$ and $\overline{\frac{\partial z}{\partial d}} = \overline{a}+\overline{b}+\overline{c}$. To find ODC_e, let $y_1 = c$ and $y_2 = d$ in (3.2).

$$\mathcal{O}_e^0 = 1 \qquad \mathcal{O}_e^1 = \overline{a}+\overline{b}+c \qquad \mathcal{O}_e^2 = \overline{a}+\overline{b}+cd = \mathcal{O}_e^+$$

Substituting for $c = ef$ and $d = eg$ we get

$$ODC_e = \overline{\frac{\partial \mathcal{O}_e^+}{\partial e}} = C_e(\overline{a} + \overline{b} + efg) + C_e(ab(\overline{e} + \overline{f} + \overline{g})) = \overline{a} + \overline{b} + \overline{f} + \overline{g},$$

which can be easily verified to be correct.

3.4.2 Deriving Damiani's Formula

The first practical formulation for computing full ODC's was given by Damiani in [23]. The complete ODC for a node is computed using the ODC's of its fanout edges [4]. Unlike the usual practice where a variable is assigned to each node in the network, a variable is assigned to each edge of the network. Here we show that Damiani's formula for computing ODC's can be derived from 3.2 by assigning variables to each edge of the network. In his computation, it is required to compute ODC's with respect to each output separately. A vector ODC is defined at each node. The number of elements in this ODC vector are equal to the number of outputs and *ith* component is ODC at that node with respect to *ith* output. If an output is not in the transitive fanout set of the node, the ODC of the node with respect to that output is set equal to 1. If the vector ODC of a node is known, the vector ODC's of all its fanin edges can be easily computed [5]. Therefore vector ODC's of the nodes in the network can be computed in topological order starting from the primary outputs.

Theorem 3.4.5 *(Damiani [23]) If $y_1, \ldots, y_r$ are the fanout* **edges** *of y_o and the vector ODC's at these edges, $(\overline{\frac{\partial z_k}{\partial y_1}}), \ldots, (\overline{\frac{\partial z_k}{\partial y_r}}), k = 1, \ldots, l$, are known, then the ODC at node y_o is given by*

$$ODC_o = \prod_{k=1}^{l} \overline{\frac{\partial z_k}{\partial y_o}}. \tag{3.4}$$

where

$$\overline{\frac{\partial z_k}{\partial y_o}} = (\overline{\frac{\partial z_k}{\partial y_1}})_{y_2 \ldots y_r} \overline{\oplus} (\overline{\frac{\partial z_k}{\partial y_2}})_{\overline{y}_1 y_3 \ldots y_r} \overline{\oplus} \ldots \overline{\oplus} (\overline{\frac{\partial z_k}{\partial y_r}})_{\overline{y}_1 \ldots \overline{y}_{r-1}} \tag{3.5}$$

[4] The ODC of a fanout edge is all the conditions under which the value of the function at that edge, which equals the function at its fanin node, can be set to either 0 or 1 and this change is not observable at any of the outputs.

[5] We add the inverse of Boolean difference of the function at the node with respect to variable of the edge to each component of vector ODC of the node to get vector ODC for edge.

Proof Because $y_1, \ldots, y_r$ are the fanout edges of y_o, then $y_o = y_1 \ldots = y_r$. Compute $\mathcal{O}^+ = \mathcal{O}^r$ using (3.2) and $(\overline{\frac{\partial z_k}{\partial y_1}}), \ldots, (\overline{\frac{\partial z_k}{\partial y_r}})$, and eliminate variables $y_1, \ldots, y_r$ by substituting $y_1 = y_2 = \ldots y_r = y_o$ in $\mathcal{O}^+$. By Theorem 3.4.4 $(\overline{\frac{\partial z_k}{\partial y_o}}) = \mathcal{O}^+_{y_o} \overline{\oplus} \mathcal{O}^+_{\overline{y_o}}$. If $y_1, \ldots, y_r$ had not been eliminated, this would be equivalent to

$$(\overline{\frac{\partial z_k}{\partial y_o}}) = \mathcal{O}^+{}_{y_1 \ldots y_r} \overline{\oplus} \mathcal{O}^+{}_{\overline{y}_1 \ldots \overline{y}_r}$$

Note from (3.2) $\mathcal{O}^j_{y_j} = \mathcal{O}^{j-1}$. Thus $\mathcal{O}^+_{y_1 \ldots y_r} = 1$ and therefore $(\overline{\frac{\partial z_k}{\partial y_o}}) = \mathcal{O}^+{}_{\overline{y}_1 \ldots \overline{y}_r}$. Cofactoring both sides of (3.2) with respect to $\overline{y}_1 \ldots \overline{y}_r$ we get

$$(\mathcal{O}^j)_{\overline{y}_1 \ldots \overline{y}_r} = (\mathcal{O}^{j-1})_{\overline{y}_1 \ldots \overline{y}_r} \overline{\oplus} (\overline{\frac{\partial z_k}{\partial y_j}})_{\overline{y}_1 \ldots \overline{y}_{j-1} y_{j+1} \ldots y_r}$$

Substituting for $(\mathcal{O}^{j-1})_{\overline{y}_1 \ldots \overline{y}_r}$ we get

$$(\mathcal{O}^j)_{\overline{y}_1 \ldots \overline{y}_r} = (\mathcal{O}^{j-2})_{\overline{y}_1 \ldots \overline{y}_r} \overline{\oplus} (\overline{\frac{\partial z_k}{\partial y_{j-1}}})_{\overline{y}_1 \ldots \overline{y}_{j-2} y_j \ldots y_r} \overline{\oplus} (\overline{\frac{\partial z_k}{\partial y_j}})_{\overline{y}_1 \ldots \overline{y}_{j-1} y_{j+1} \ldots y_r}$$

Therefore, by induction

$$(\overline{\frac{\partial z_k}{\partial y_o}}) = (\mathcal{O}^r)_{\overline{y}_1 \ldots \overline{y}_r} = \overline{\oplus}_{j=1}^r (\overline{\frac{\partial z_k}{\partial y_j}})_{\overline{y}_1 \ldots \overline{y}_{j-1} y_{j+1} \ldots y_r}.$$

■

3.4.3 Using the Observability Relation

Theorem 3.3.2 shows that the ODC's for the nodes of $\mathcal{N}$ can be computed from $\mathcal{N}'$. This gives rise to a new procedure for computing complete ODC's in a network in topological order. This is summarized below.

$\mathcal{N}'$ has only one primary output; therefore, vector ODC's are not needed [6]. The ODC's at each primary output z_k of $\mathcal{N}$ is $ODC_{z_k} = \overline{\frac{\partial \mathcal{O}}{\partial z_k}}$. If no output flexibility exists in implementing the network, the observability relation is

$$\mathcal{O}(\mathbf{x}, \mathbf{z}) = (z_1 \overline{\oplus} g_1)(z_2 \overline{\oplus} g_2) \ldots (z_l \overline{\oplus} g_l)$$

where g_k is the global function at the output z_k. The observability don't care at the output z_k is

$$ODC_{z_k} = (z_1 \oplus g_1) + \ldots + (z_{k-1} \oplus g_{k-1}) + (z_{k+1} \oplus g_{k+1}) + \ldots + (z_l \oplus g_l).$$

[6] We can also apply Damiani's technique on the observability network $\mathcal{N}'$ directly and since there is only one output the technique also results in scalar ODC's.

New ODC Computation:

- Build the observability network $\mathcal{N}'$.
- Order the nodes in the network in topological order from outputs. A node y_o is processed if the ODC's at its fanout nodes $(y_1, \ldots, y_r)$ are known.
- For each node y_o with fanout nodes $(y_1, \ldots, y_r)$:
  - Build the partition for node y_o using the ODC's of the fanout nodes and (3.2) to get $\mathcal{O}_o^+$.
  - Replace variables $y_1, \ldots, y_r$ with their local functions $f_1, \ldots f_r$ in $\mathcal{O}_o^+$.
  - Compute $ODC_o = \overline{\frac{\partial \mathcal{O}_o^+}{\partial y_o}} = C_{y_o}\mathcal{O}_o^+ + C_{y_o}\overline{\mathcal{O}}_o^+$.

ODC_{z_k} is a subset of the SDC for the whole network. Although each term $(z_i \oplus g_i)$ evaluates to 0, it contains useful information about the structure of the network; in effect this is the same information held by the vector ODC's of Damiani for network $\mathcal{N}$. All the components of this vector ODC are equal to 1 except for the *kth* component where it is 0.

Example:

To illustrate this process, we compute ODC's using the observability network for the example given in [23] using our method (denoted by ODC), Damiani's method applied to observability network (denoted by SODC), and Damiani's method in vector form (denoted by VODC). This example is shown in Figure 3.5. A variable is associated with each edge as in [23].

The output functions and their ODC's are

$$
\begin{aligned}
g_1 &= x_1x_4 + (x_2 \oplus x_3) \\
g_2 &= x_1 + x_4 + (x_2 \oplus x_3) \\
\mathcal{O}(x,z) &= (g_1 \overline{\oplus} z_1)(g_2 \overline{\oplus} z_2) \\
ODC_{z_1} &= SODC_{z_1} = z_2 \oplus g_2 \\
ODC_{z_2} &= SODC_{z_1} = z_1 \oplus g_1 \\
VODC_{z_1} &= \begin{pmatrix} 0 \\ 1 \end{pmatrix}
\end{aligned}
$$

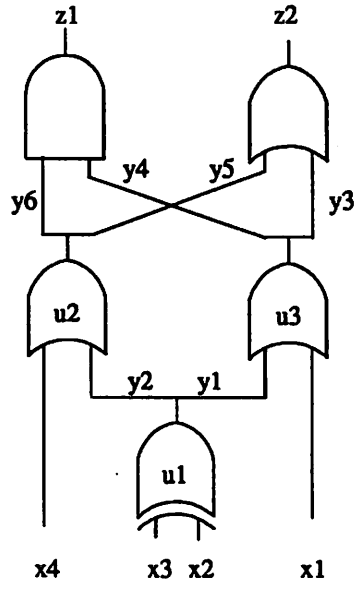

Figure 3.5: Example

$$VODC_{z_2} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

First we compute ODC's at nodes u_2 and u_3:

$$\begin{aligned}
\mathcal{O}(x, u_2, u_3) &= (g_1 \overline{\oplus} (u_2 u_3))(g_2 \overline{\oplus} (u_2 + u_3)) \\
ODC_{u_2} &= u_3 \overline{g}_2 + \overline{u}_3 g_1 \\
ODC_{u_3} &= u_2 \overline{g}_2 + \overline{u}_2 g_1 \\
SODC_{u_2} = (SODC_{y_5})_{\overline{y}_6} \overline{\oplus} (SODC_{y_6})_{y_5} &= (y_3 + g_1) \overline{\oplus} (\overline{y}_4 + \overline{g}_2) = y_3 \overline{g}_2 + \overline{y}_4 g_1 \\
SODC_{u_3} = (SODC_{y_3})_{\overline{y}_4} \overline{\oplus} (SODC_{y_4})_{y_3} &= (y_5 + g_1) \overline{\oplus} (\overline{y}_6 + \overline{g}_2) = y_5 \overline{g}_2 + \overline{y}_6 g_1 \\
VODC_{u_2} &= \begin{pmatrix} \overline{y}_4 \\ y_3 \end{pmatrix} \\
VODC_{u_3} &= \begin{pmatrix} \overline{y}_6 \\ y_5 \end{pmatrix}
\end{aligned}$$

We use the fact that $\overline{g}_2 g_1 = 0, y_3 = y_4$, and $y_5 = y_6$ to obtain the above result.

$$SODC_{y_1} = x_1 + (x_4 + y_2)\overline{g}_2 + \overline{x}_4 \overline{y}_2 g_1$$

$$SODC_{y_2} = x_4 + (x_1 + y_1)\overline{g}_2 + \overline{x}_1\overline{y}_1 g_1$$

$$VODC_{y_1} = \begin{pmatrix} \overline{y}_2\overline{x}_4 + x_1 \\ y_2 + x_4 + x_1 \end{pmatrix}$$

$$VODC_{y_2} = \begin{pmatrix} \overline{y}_1\overline{x}_1 + x_4 \\ y_1 + x_1 + x_4 \end{pmatrix}$$

Substituting for g_1 and g_2, we find

$$\begin{aligned} \mathcal{O}(x, u_1) &= (g_1 \overline{\oplus} (x_1 x_4 + u_1))(g_2 \overline{\oplus} (x_1 + x_4 + u_1)) \\ ODC_{u_1} &= x_1 x_4 \\ SODC_{u_1} &= (ODC_{y_1})_{\overline{y}_2} \overline{\oplus} (ODC_{y_2})_{y_1} = x_1 x_4 \end{aligned}$$

$$VODC_{u_1} = \begin{pmatrix} x_1 x_4 \\ x_1 + x_4 \end{pmatrix} \Rightarrow ODC_{u_1} = (x_1 + x_4)x_1 x_4 = x_1 x_4.$$

Notice that in applying the procedure outlined above we did not have to rebuild any of $\mathcal{O}$ functions because these were already available to us after collapsing. While computing the ODC for a node y_j, we need to keep $z_i \oplus g_i$ for the primary outputs in the network $\mathcal{N}$ that are transitive fanouts of y_j. The term $z_i \oplus g_i$ can be set to zero for any other primary output. This is equivalent to having 1 in the corresponding row of the VODC.

3.5 Observability Don't Care Subsets

The computation of the complete ODC for optimizing nodes of a large network is often too expensive to be used during synthesis. This is because once the function at a node is changed using its computed ODC, the ODC at other nodes in the network may have to be recomputed. In addition, full ODC's computed for some nodes can be extremely large, especially for circuits that cannot be collapsed in two-level form. Subsets of the ODC have been studied by several authors. The first attempt in [36] gives a linear time algorithm for computing ODC subsets. The ODC's are computed for the nodes of the network in depth first search from outputs by using an approximation operator called RESTRICT. This operator removes any cube in the ODC of a node y_i which has a literal corresponding to a node in its transitive fanout. This approximation, although valid for the networks with reconvergent fanouts, is quite restrictive and loses some useful information. Other

techniques for computing ODC subsets are given by [54, 23]. The formulation suggested in [23] gives the largest subset that can be computed for a node using ODC subsets of its fanouts. The problem with all these ODC subsets is that once the function at a node is changed using its ODC subset, then the ODC subsets computed for other nodes in the network may not be correct any more and must be recomputed.

An interesting ODC subset is introduced by Muroga in [59] where subsets can be computed for nodes of the network and used for the optimization of each node simultaneously. These subsets are called compatible sets of permissible functions (CSPF's). CSPF's are expressed in terms of primary input variables and are only defined for a network decomposed into NOR gates. We expand the concept of CSPF's to complex nodes of a general multi-level network and present procedures for computing *compatible* ODC (CODC) subsets. Another contrast with [59] is that CODC's are expressed in terms of both primary input variables and intermediate variables. We shall see that the ability to use intermediate variables is important and powerful.

3.5.1 Compatible Observability and External Don't Cares

At each intermediate node y_o we can compute a set of *permissible functions* [59], that is if the global function f_o^g for y_o is replaced by any function in this set, the network is still correct. The care set of y_o is composed of all the vertices in B^n for which f_o^g must have a fixed value; if the value of f_o^g is changed for such input combinations, the network computes an incorrect output value. The don't care set of y_o is the remaining vertices of B^n for which f_o^g is not required to have a fixed value (it can be 0 or 1). A function which uses the don't cares in a valid way is a permissible function.

A set of permissible functions for node y_o can be represented by two functions; by a function f_o^g which is usually the current implementation at the node and d_o^g which is a combination of observability and external don't cares expressed in terms of primary inputs. The *maximum observability plus external don't care*, d_o^{mg}, (also MSPF) for y_o is the one having the maximum number of input combinations in d_o^g . Generally, d_o^{mg} depends on the global functions of other nodes of the network. If the global function at any node is changed, d_o^{mg} may have to be recomputed.

Sets of observability plus external don't cares (d_o^g's) at a set S of nodes of a network are *compatible*, if each node $y_i \in S$ can be represented by any function from its permissible

function set independent of how any other node is represented from its set. We denote such sets by d^{cg} or CODC. A Compatible Set of Permissible Function (CSPF) is all the functions allowed by d^{cg}.

At each primary output z_i, we can also have a set of permissible functions represented by a function f_i^g for the current implementation and a function d_i^g for the don't care set. These don't cares are called external don't cares and must be compatible. Such external don't cares can be specified by the designer directly. Alternately, if an observability relation $\mathcal{O}$ is given for the network $\mathcal{N}$, the observability network, the observability network $\mathcal{N}'$ can be constructed and the external don't cares can be derived as the compatible don't cares of the fanins of the complex node $\mathcal{O}$ using methods to be described in Section 3.5.3.

The computation of CODC's for the complex nodes of a multi-level network depends on two key operations. One is the computation of CODC's for the fanin edges of a node, given the CODC of the node and an ordering of the fanins. The second key operation is computing CODC's for each node by intersecting the CODC's of its fanout edges. We first concentrate on a directed tree structure where each node has a single fanout except for primary inputs. Then we extend the developed techniques to a general multi-level network where nodes have multiple fanouts.

3.5.2 CODC's for Trees

We discuss CODC computations for the fanin edges of a node by considering a directed tree structure where each intermediate node has a single fanout and primary inputs have multi-fanouts (a so-called leaf DAG). All the nodes in this tree are ordered topologically. The highest order is given to the root node; every other node gets an ordering less than its fanout. Let y_o represent the root of the directed tree with external don't care d_o^{cm} and with fanins $y_1, y_2, \ldots, y_j$. This tree structure has the property that the observability plus external don't cares at each node are equal to that of its fanout edge. Also, assume an ordering ($\succ$) is given to the fanins of y_o such as $y_1 \succ y_2 \ldots \succ y_j$. This ordering implies that node y_1 gets its maximum possible don't care; the don't care set at y_2 must be compatible to that of y_1; the don't care set at y_3 must be compatible to that of y_1 and y_2 and so forth.

Given d_o^{cm} (in terms of some set of intermediate variables and primary inputs), the maximum don't care at each fanin node (also fanin edge) is $d_{io}^m = \overline{\frac{\partial f_o}{\partial y_i}} + d_o^{cm}$. We let y_1 have its maximum don't care set $d_{1o}^{cm} = \overline{\frac{\partial f_o}{\partial y_1}} + d_o^{cm}$ and show how to find d_{2o}^{cm} which is

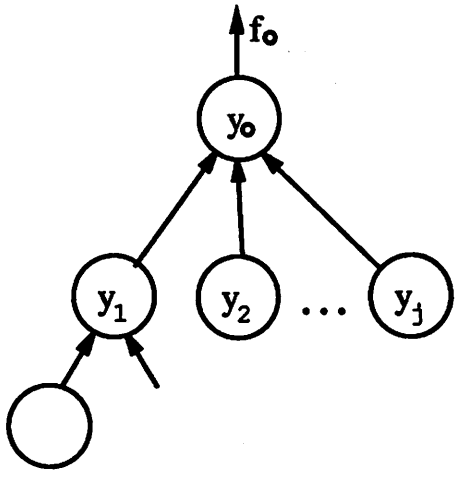

Figure 3.6: A Directed Tree

compatible with d_{1o}^{cm}. This is then generalized for the *ith* fanin of y_o. $d_{2o}^{cm} \subseteq d_{2o}^{m}$ must have the property that any simultaneous changing of the functional representation at y_1 and y_2 to any permissible functions allowed by the corresponding CODC of each node preserves the correct behavior of the network.

Lemma 3.5.1 *Given don't care sets, d_o^{cm}, for a node with function f_o, and d_1^{cm} for y_1, the d_2^{cm} for y_2 is*

$$d_2^{cm} = \overline{d_1^{cm}} \overline{\frac{\partial f_o}{\partial y_2}} + C_{y_1} \overline{\frac{\partial f_o}{\partial y_2}} + d_o^{cm}.$$

Proof: The set $\overline{\frac{\partial f_o}{\partial y_2}}$ contains all possible don't cares for d_2^{cm} besides d_o^{cm}. This can be divided into those that are independent of the value of y_1, $C_{y_1} \overline{\frac{\partial f_o}{\partial y_2}}$, and the rest which require some specific value for y_1, $r = \overline{\frac{\partial f_o}{\partial y_2}} - C_{y_1} \overline{\frac{\partial f_o}{\partial y_2}}$. We seek the maximum set of don't cares which are compatible with d_1^{cm}. Clearly, $C_{y_1} \overline{\frac{\partial f_o}{\partial y_2}}$ is compatible since this says that f_o is insensitive to y_2 independent of the value of y_1. Further $C_{y_1} \overline{\frac{\partial f_o}{\partial y_2}}$ is the maximum such set. The remainder of $\overline{\frac{\partial f_o}{\partial y_2}}$ depends on y_1. Let m be an input minterm of r, $r(m) = 1$. If $d_1^{cm}(m) = 1$, we must have $d_2^{cm}(m_0) = 0$; otherwise, the value of both $f_1(m)$ and $f_2(m)$ can change simultaneously and $f_o(m)$ becomes incorrect. This is because the allowed change in $f_2(m)$ is only valid when $f_1(m)$ has a specific value; this value is not guaranteed when $f_1(m)$ is allowed to change.

Removing such input combinations from $\overline{\frac{\partial f_o}{\partial y_2}} - C_{y_1}\overline{\frac{\partial f_o}{\partial y_2}}$ and making it compatible with d_1^{cm} leads to the term $(\overline{\frac{\partial f_o}{\partial y_2}} - C_{y_1}\overline{\frac{\partial f_o}{\partial y_2}})\overline{d_1^{cm}}$. Thus we keep only those terms for which the value of f_1 will not change. Adding these terms we obtain $d_2^{cm} = \overline{d_1^{cm}}\overline{\frac{\partial f_o}{\partial y_2}} + C_{y_1}\overline{\frac{\partial f_o}{\partial y_2}} + d_o^{cm}$. Clearly, no other input combination contained in $\overline{\frac{\partial f_o}{\partial y_2}}$ can be added to d_2^{cm} without destroying the compatibility of d_2^{cm} to d_1^{cm}; therefore, d_2^{cm} as obtained above is maximal. ■

The don't care set at y_3 compatible with y_1 and y_2 is

$$d_3^{cm} = (\overline{d_1^{cm}}\,\overline{d_2^{cm}} + \overline{d_1^{cm}}C_{y_2} + \overline{d_2^{cm}}C_{y_1} + C_{y_1y_2})\overline{\frac{\partial f_o}{\partial y_3}} + d_o^{cm}.$$

While computing d_{3i}^{cm}, we break $\overline{\frac{\partial f_o}{\partial y_3}}$ into parts that 1) are independent of subsets of the fanins of higher order and 2) are in the intersection of the care sets of the other fanins of higher order. Thus the term $\overline{d_1^{cm}}C_{y_2}$ comes from considering points independent of y_2 that are in the care set of y_1.

The general formula is constructed as follows. Let S_k be the fanins of order greater than y_k, and for some set K,

$$Y_K = \prod_{i \in S_k - K} y_i.$$

Then

$$d_k^{cm} = \left[\sum_{K \subseteq S_k} (\prod_{i \in K} \overline{d_i^{cm}})C_{Y_K}\right]\overline{\frac{\partial f_o}{\partial y_k}} + d_o^{cm}. \tag{3.6}$$

The general term

$$(\prod_{i \in K} \overline{d_i^{cm}})C_{Y_K}\overline{\frac{\partial f_o}{\partial y_k}}$$

is simply the points of $\overline{\frac{\partial f_o}{\partial y_k}}$ which 1) are independent of the variables y_j in $S_k - K$, and 2) are in the care set of all the fanin edges in K.

Lemma 3.5.2 *Given don't care sets, d_o^{cm}, for a node with function f_o, and d_i^{cm} for each of its fanins y_i, with $y_i \succ y_k$, the maximum don't care set for fanin y_k compatible with $\{d_i^{cm}\}$ is given by equation (3.6).*

Proof: The set $\overline{\frac{\partial f_o}{\partial y_k}}$ contains all possible don't cares for d_k^{cm} beside d_o^{cm}. In addition to d_o^{cm}, d_k^{cm} is all the input minterms m for which the value of $f_k(m)$ and any fanin of higher order $f_i(m)$ (as allowed by d_i^{cm}) can change but no change is observed in $f_o(m)$. For any minterm $m \in d_k^{cm} - d_o^{cm}$, the fanins of higher order S_k are divided into two groups, the ones that do not change at all denoted by $\{y_i | i \in K\}$ where $K = \{i | m \notin d_i^{cm}\}$, and the ones

that can change $\{y_i | i \in (S_k - K)\}$. It follows that $m \in (\prod_{i \in K} \overline{d_i^{cm}}) \overline{\frac{\partial f_o}{\partial y_k}}$ and because $f_o(m)$ is insensitive to changes in $\{f_i(m) | i \in (S_k - K)\}$, $m \in C_{Y_K} \overline{\frac{\partial f_o}{\partial y_k}}$ where $Y_K = \prod_{i \in S_k - K} y_i$. Consequently, $m \in (\prod_{i \in K} \overline{d_i^{cm}}) C_{Y_K} \overline{\frac{\partial f_o}{\partial y_k}}$. d_k^{cm} as given by (3.6) contains all such combinations of fanins of higher order into two groups K and $S_k - K$ and each such combination gives a correct compatible subset; therefore d_k^{cm} as in (3.6) is the maximum compatible don't care set for y_k. ■

Example:

Let $f_o = y_1 + y_2 + y_3$, $f_1 = x_1x_2$, $f_2 = x_2x_3$, $f_3 = x_1x_3$, and $d_o^{cm} = 0$; therefore,

$$
\begin{aligned}
d_1^{cm} &= \overline{\frac{\partial f_o}{\partial y_1}} = y_2 + y_3 \\
d_2^{cm} &= \overline{d_1^{cm}} \overline{\frac{\partial f_o}{\partial y_2}} + C_{y_1} \overline{\frac{\partial f_o}{\partial y_2}} = y_1\overline{y}_2 + y_3 \\
d_3^{cm} &= (\overline{d_1^{cm}}\ \overline{d_2^{cm}} + \overline{d_1^{cm}} C_{y_2} + \overline{d_2^{cm}} C_{y_1} + C_{y_1y_2}) \overline{\frac{\partial f_o}{\partial y_3}} = y_1\overline{y}_3 + y_2\overline{y}_3.
\end{aligned}
$$

$d_1^{cm} = x_2x_3 + x_1x_3$ when y_2 and y_3 are substituted with their local functions. Notice that $\overline{\frac{\partial f_o}{\partial y_2}} = y_1 + y_3$, and $y_1y_2\overline{y}_3$ has been removed from the set to make it compatible with d_1^{cm}. Otherwise, if for some input minterm m, $f_1(m) = 1, f_2(m) = 1$, and $f_3(m) = 0$, the value of both $f_1(m)$ and $f_2(m)$ can be set to 0 which gives the incorrect result $f_o(m) = 0$. In this particular case, no such minterm exists because $y_1y_2\overline{y}_3$ is an impossible combination for the given f_1, f_2, and f_3. Consequently, $d_2^{cm} = x_1x_2 + x_1x_3$ which is equal to $\overline{\frac{\partial f_o}{\partial y_2}} = y_1 + y_3$ when y_1 and y_3 are substituted with their local functions.

The set $y_1y_3 + y_2y_3$ has been removed from $\overline{\frac{\partial f_o}{\partial y_3}} = y_1 + y_2$ to get d_3^{cm}. After substituting for y_1, y_2, and y_3, $d_3^{cm} = x_1x_2\overline{x}_3 + \overline{x}_1x_2x_3$. $f_1(x_1x_2\overline{x}_3) = 1$ and $d_1^{cm}(x_1x_2\overline{x}_3) = 0$; therefore f_1 always gives 1 for this minterm and $f_o(x_1x_2\overline{x}_3) = 1$ irrespective of any change in f_3. In the same way, $f_2(\overline{x}_1x_2x_3) = 1$ and $d_2^{cm}(\overline{x}_1x_2x_3) = 0$; therefore f_2 always gives 1 for $\overline{x}_1x_2x_3$.

Once the CODC for a node is found, we can find CODC for each of its fanins and therefore for all the nodes in the directed tree in topological order.

Lemma 3.5.3 *If the intermediate nodes of a network form a directed tree with one output, the computation of (3.6) in topological order leads to* $\{d_i^{cm}\}$ *which are all maximally compatible.*

Proof We first prove that the computed sets are compatible. Let the function at some set of nodes change as allowed by $\{d_i^{cm}\}$. Let y_i be a node whose fanins are all primary inputs and whose fanout is y_j and m any input minterm. Because of (3.6), a change in $f_i(m)$ is either not observable in y_j or is allowed by d_j^{cm} ($d_j^{cm}(m) = 1$); therefore f_j remains correct.

Now, assume y_j is a node whose fanin functions only change as allowed by their corresponding maximal CODC's. Because of Lemma 3.5.2, any change in fanins of y_j as allowed by their CODC's results in a change in f_j as allowed by d_j^{cm}. By induction, it follows that any changes at a set of nodes in the network results in valid changes at the fanouts of those nodes and therefore at all the nodes in the network.

Assume the computed $\{d_i^{cm}\}$ are not maximal. Thus a minterm m can be added to some set d_i^{cm} with fanout y_j. Because of the maximality of (3.6), d_j^{cm} is not maximal and m must be added to d_j^{cm}. This is the case for all the transitive fanouts of y_i especially for the root of the tree y_o. d_o^{cm} is fixed and cannot be increased; therefore $\{d_i^{cm}\}$ are maximal. ∎

The number of terms in equation (3.6) is $2^{|S_k|}$. Thus if a node has many fanins (e.g. the $\mathcal{O}$ node of $\mathcal{N}'$) the CODC computation becomes too time consuming. We consider two different ways to speed up this computation.

The first technique is to apply a limited collapsing on the term $\overline{d}_i^{cm}$ used in the computation of d_k^{cm}. All the variables $y_1, \ldots, y_{i-1} \succ y_i$ are replaced by their corresponding local functions. We represent this new function by E_i where $E_i = \overline{d}_i^{cm}|_{y_1=f_1,\ldots,y_{i-1}=f_{i-1}}$. The following Lemma is essential in obtaining the new formulation for computing compatible don't care sets.

Lemma 3.5.4 *Let $e_1, \ldots, e_n$ be Boolean functions independent of y_o, $Y_1, \ldots Y_n$ be any set of variables excluding y_o, d any Boolean function and*

$$D = C_{y_o}(e_1 C_{Y_1} d + \ldots + e_n C_{Y_n} d).$$

D is also equal to

$$D = e_1 C_{Y_1} C_{y_o} d + \ldots + e_n C_{Y_n} C_{y_o} d.$$

Proof D can be written as

$$D = (e_1 C_{Y_1} d_{y_o} + \ldots + e_n C_{Y_n} d_{y_o})(e_1 C_{Y_1} d_{\overline{y}_o} + \ldots + e_n C_{Y_n} d_{\overline{y}_o}).$$

Notice that $e_i C_{Y_i} d_{y_o} e_j C_{Y_j} d_{\overline{y}_o} \subseteq e_i C_{Y_i} d_{y_o} d_{\overline{y}_o} = e_i C_{Y_i} C_{y_o} d$; therefore we only need to intersect terms with the same indices.

$$D = e_1 C_{Y_1} C_{y_o} d + \ldots + e_n C_{Y_n} C_{y_o} d.$$

■

Lemma 3.5.5 *Let E_i be equal to $\overline{d}_i^{cm}$ where each variable $y_l \succ y_i$ is substituted by its local function f_l in $\overline{d}_i^{cm}$. The maximal compatible don't care set at node y_k is*

$$d_k^{cm} = (E_1 + C_{y_1}) \ldots (E_{k-1} + C_{y_{k-1}}) \overline{\frac{\partial f_o}{\partial y_k}} + d_o^{cm}. \tag{3.7}$$

Proof Because of the introduced collapsing, $C_{y_l} E_i = E_i$ for all the variables $y_l \succ y_i$ since these variables do not appear in E_i. To show that (3.6) and (3.7) are equivalent, we expand (3.7) using Lemma 3.5.4.

$$\begin{aligned}
d_k^{cm} &= (E_1 + C_{y_1}) \ldots (E_{k-1} + C_{y_{k-1}}) \overline{\frac{\partial f_o}{\partial y_k}} + d_o^{cm} \\
&= (E_1 + C_{y_1}) \ldots (E_{k-2} + C_{y_{k-2}}) (E_{k-1} \overline{\frac{\partial f_o}{\partial y_k}} + C_{y_{k-1}} \overline{\frac{\partial f_o}{\partial y_k}}) + d_o^{cm} \\
&= (E_1 + C_{y_1}) \ldots (E_{k-2} E_{k-1} \overline{\frac{\partial f_o}{\partial y_k}} + E_{k-2} C_{y_{k-1}} \overline{\frac{\partial f_o}{\partial y_k}}) + C_{y_{k-2}} (E_{k-1} \overline{\frac{\partial f_o}{\partial y_k}} + C_{y_{k-1}} \overline{\frac{\partial f_o}{\partial y_k}})) \\
&\quad + d_o^{cm} \\
&= (E_1 + C_{y_1}) \ldots (E_{k-2} E_{k-1} \overline{\frac{\partial f_o}{\partial y_k}} + E_{k-2} C_{y_{k-1}} \overline{\frac{\partial f_o}{\partial y_k}} + E_{k-1} C_{y_{k-2}} \overline{\frac{\partial f_o}{\partial y_k}} + C_{y_{k-1}} C_{y_{k-2}} \overline{\frac{\partial f_o}{\partial y_k}}) \\
&\quad + d_o^{cm} \\
&\vdots \\
&= \left[\sum_{K \subseteq S_k} \left(\prod_{i \in K} E_i \right) C_{Y_K} \right] \overline{\frac{\partial f_o}{\partial y_k}} + d_o^{cm}
\end{aligned}$$

■

The number of AND and OR operations required to compute d_k^{cm} from equation 3.7 is linear in k. The second technique to speedup CODC computation is to compute smaller compatible subsets that are more computationally efficient.

Lemma 3.5.6 *Given don't care sets, d_o^c, for a node with function f_o, and fanins $y_1 \succ \ldots \succ y_{k-1} \succ y_k$,*

$$\hat{d}_k^c \equiv \left(\frac{\partial f_o}{\partial y_1} + C_{y_1} \right) \ldots \left(\frac{\partial f_o}{\partial y_{k-1}} + C_{y_{k-1}} \right) \overline{\frac{\partial f_o}{\partial y_k}} + d_o^c \tag{3.8}$$

is a valid compatible observability don't care subset for y_k. Furthermore, $\hat{d}_k^c \subseteq d_k^{cm}$.

Proof: The proof is by induction on k. Since d_o^c enters in all calculations, we can ignore it for the purposes of this proof. Note then that

$$\hat{d}_1^c = d_1^{cm} = \overline{\frac{\partial f_o}{\partial y_1}}$$

and

$$\hat{d}_2^c = d_2^{cm} = (\frac{\partial f_o}{\partial y_1} + C_{y_1})\overline{\frac{\partial f_o}{y_2}}.$$

Thus by Lemma 3.5.2, $\hat{d}_1^{cm}$ and $\hat{d}_2^{cm}$ are compatible.

Now assume that any combination J of up to $k-2$ operations of the form $(\frac{\partial f_o}{\partial y_i}+C_{y_i})$, $i \in J$, and $i \leq k-1$, operating on $\overline{\frac{\partial f_o}{\partial y_k}}$ gives a set mutually compatible with all d_i^{cm}, $i \in J$. Since $\hat{d}_k^c$ is formed by using $k-1$ operations, it is a subset of any of the ones formed by using $k-2$ or less operations. Hence $\hat{d}_k^c$ is mutually compatible with any $k-2$ sets d_i^{cm} for $i \leq k-1$.

We proceed by contradiction. Assume that there is a minterm $m \in \hat{d}_k^c$ that causes non-compatibility. Then m must be in $d_1^{cm} \ldots d_{k-1}^{cm}$. Now $m \notin C_{y_1 \ldots y_{k-1}} \overline{\frac{\partial f_o}{\partial y_k}}$, since otherwise it would be in d_k^{cm} and would be compatible. Thus, by equation (3.8), m must be in at least one of the sets $\frac{\partial f_o}{\partial y_i}$ for $i \leq k-1$. But this contradicts the fact that, using equation (3.6),

$$m \in d_1^{cm} \ldots d_{k-1}^{cm} = \overline{\frac{\partial f_o}{\partial y_1}}(C_{y_1}\overline{\frac{\partial f_o}{\partial y_2}}) \ldots (C_{y_1 \ldots y_{k-2}}\overline{\frac{\partial f_o}{\partial y_{k-1}}})$$

is orthogonal to $\frac{\partial f_o}{\partial y_i}$, $i \leq k-1$, cofactored with respect to any set of y_l's $l \leq i$. Thus no such m exists and hence $\hat{d}_k^{cm}$ is mutually compatible with d_i^{cm}, $i \leq k-1$. Since the d_i^{cm} are maximal by Lemma 3.5.2, $\hat{d}_k^c \subseteq d_k^{cm}$. ∎

3.5.3 CODC's for a General Network

We discuss a technique for computing compatible don't cares for all the nodes of a multi-level network. We first consider the primary outputs. If external don't cares are given in terms of primary inputs, they must be compatible. The d^c at each primary output is set equal to the external don't care at that output. If an observability relation $\mathcal{O}$ is given that has the behavior of the network, compatible external don't cares at the outputs can be computed using Lemmas 3.5.2, 3.5.4, or 3.5.5. An ordering is given to the primary outputs. $\overline{\frac{\partial \mathcal{O}}{\partial z_k}}$ is computed and made compatible to the outputs of higher order. Notice that this external don't care may be a function of both the inputs and outputs of the network.

We now consider how don't cares can be computed for intermediate nodes. A topological ordering is first given to all the nodes in the network and nodes are processed one by one starting from the primary outputs. The fanout edges of y_k inherit the same ordering as their source node. The compatible don't cares for each fanout edge can be computed the same way as done for trees. When y_k is being processed, all the nodes of higher order have been already processed, therefore, their compatible don't care subsets can be used to find CODC for the edge e_{ko} as before:

$$d^c_{ko} = \left[\sum_{K \subseteq S_k} (\prod_{i \in K} \overline{d^c_i}) C_{Y_K}\right] \overline{\frac{\partial f_o}{\partial y_k}} + d^c_o. \tag{3.9}$$

The don't care sets at the fanout edges are then intersected to get the don't care set for the node. The don't care set for each edge has two parts, the one that comes from the fanout node d^c_o, and the one that comes from the Boolean difference $b^c_{io} = \left[\sum_{K \subseteq S_k} (\prod_{i \in K} \overline{d^c_i}) C_{Y_K}\right] \overline{\frac{\partial f_o}{\partial y_k}}$. This notation is used in the proof of the following Lemma.

Lemma 3.5.7 *If the immediate fanout edges (FO_i) of a node y_i have compatible observability don't care subsets then the subset*

$$d^c_i = \prod_{y_k \in FO_i} d^c_{ik} \tag{3.10}$$

is a valid observability subset for y_i which is compatible with CODC's computed for all higher order nodes.

Proof: Let m be an input minterm such that $d^{gc}_i(m) = 1$. $\forall y_k \in FO_i$, we must have $b^{cg}_{ik}(m) = 1$ or $d^{cg}_k(m) = 1$. If $b^{cg}_{ik}(m) = 1$, then $f_i(m)$ can be set to either 0 or 1 and this change is never observable in $f_k(m)$ by Lemma 3.5.2 and the fact that b^{cg}_{ik} is compatible with the other edge fanins of y_k. If $d^{gc}_k(m) = 1$, then $f_i(m)$ can be set to either 0 or 1 and $f_k(m)$ might change value. However, the new function is a permissible function for the node $y_k \in FO_i$. Since the observability don't cares computed for the fanouts of y_i are all compatible, any simultaneous changing of the functions of the fanout nodes as allowed by their respected CODC's is correct. Therefore d^c_i is a valid observability subset.

Any change in f_i for a particular minterm m as allowed by d^c_i results in a change in the function of all the fanout edges as allowed by their respected $\{d^c_{ik}\}$. $\{d^c_{ik}\}$ are compatible with nodes of higher order; therefore d^c_i is also compatible with nodes of higher order. ■

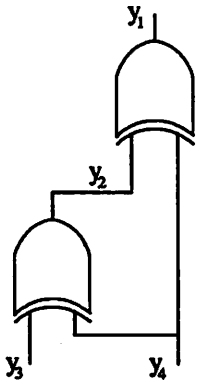

Figure 3.7: Example

The CODC's computed for the nodes of the network as given above are not necessarily maximal. In the above computation, we first compute CODC's for the fanout edges of a node and then intersect them. However, we only require the ODC's for the nodes of the network to be compatible, not the edges. Making the ODC's at the fanout edges compatible, enables us to intersect them and find the CODC for the node but this is not necessarily maximal. We did not notice this fact in our original derivation in [67]. This was pointed out by Damiani in [24] with the example shown in Figure 3.7. Assume any topological ordering. The CODC computation finds $d^c_{41} = 0$ and $d^c_{42} = 0$. Intersecting these we find $d^c_4 = 0$ which is not maximal. It can easily be shown that $d^{cm}_4 = 1$. Although the CODC's computed are not maximally compatible for all the nodes in the network, they are maximally compatible for the nodes that form a tree structure with respect to each primary output where each tree is rooted at one of the primary outputs. As before the primary inputs can have fanout of more than one. As soon as reconvergent fanouts appear with respect to some output, maximality cannot be claimed anymore.

Lemma 3.5.8 *If an intermediate node y_i plus its transitive fanouts form a tree with respect to each primary output, the computation of (3.6) and (3.10) in topological order leads to a set $\{d^{cm}_i\}$ which are maximally compatible for y_i and its transitive fanouts.*

Proof Suppose d^{cm}_i is not maximal. Then there is a minterm m that can be added to d^{cm}_i. If m can be added to d^{cm}_i, then it can be added to any $d^{cm}_{i;k}$ computed with respect to z_k. $d^{cm}_{i;k}$ is what is computed by some fanout edge (i,j) of y_i (there is a path from y_i to z_k through this edge); therefore m can be added to d^c_{ij}. Because of (3.6), m can be also added

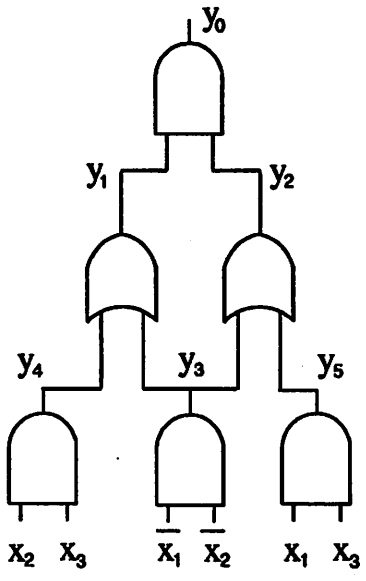

Figure 3.8: Example

to d_j^c. In the same way, one can find a fanout of y_j whose don't care set can be increased by m. This can be repeated until we reach a primary output; however, the don't care sets at primary outputs are fixed. Thus the statement of the Lemma follows. ∎

Example:

We find CODC's for intermediate nodes of the network shown in Figure 3.8. The ordering is $y_o \succ y_1 \ldots \succ y_5$.

$$
\begin{aligned}
d_1^{cm} &= \overline{\frac{\partial f_o}{\partial y_1}} = \overline{y}_2 \\
d_2^{cm} &= (C_{y_1} + \overline{d_1^{cm}})\overline{\frac{\partial f_o}{\partial y_2}} = \overline{y}_1 y_2 \\
d_3^c &= (\overline{\frac{\partial f_1}{\partial y_3}} + d_1^{cm})(\overline{\frac{\partial f_2}{\partial y_3}} + d_2^{cm}) \\
&= (y_4 + \overline{y}_3\overline{y}_5)(y_5 + \overline{y}_3\overline{y}_4 y_5) = y_4 y_5 \\
d_4^{cm} &= (C_{y_3} + \overline{d_3^c})\overline{\frac{\partial f_1}{\partial y_4}} + d_1^{cm} = (\overline{y}_4 + \overline{y}_5)y_3 + \overline{y}_2 \\
&= y_3\overline{y}_4 + \overline{y}_5 \\
d_5^{cm} &= (C_{y_3} + \overline{d_3^c})\overline{\frac{\partial f_2}{\partial y_5}} + d_2^{cm} = (\overline{y}_4 + \overline{y}_5)y_3 + \overline{y}_1 y_2 \\
&= y_3\overline{y}_4 + y_3\overline{y}_4 + \overline{y}_4 y_5
\end{aligned}
$$

Notice that d_3^c is not necessarily maximal because it has reconvergent fanouts. As a result,

$\overline{d}_3^c$ can become larger and therefore d_4^{cm} and d_5^{cm} can become larger. *Although the maximality at y_3 is not gained, this is compensated for by larger CODC sets at other nodes.* CODC's at all other nodes except for y_3 are maximal.

3.6 ODC's and Equivalence Classes of a Cut

Let $\mathcal{Y} = (y_1, y_2, \ldots, y_p)$ be a separating set of nodes in network $\mathcal{N}$ and y_o an intermediate node. Thus $\mathcal{N}$ can be viewed as two networks $\mathcal{N}_1$ and $\mathcal{N}_2$ where $\mathcal{Y}$ is the input of the network $\mathcal{N}_1$ and output of network $\mathcal{N}_2$. The outputs of $\mathcal{N}_1$, $\mathbf{z} = (z_1, \ldots, z_l)$, can be expressed in terms of $y_1, y_2, \ldots, y_p$.

Definition 3.6.1 *We say two minterms $m_i, m_j \in B^p$ are equivalent ($m_i \sim m_j$) if $\mathbf{z}(m_i) = \mathbf{z}(m_j)$.*

This relation divides the space B^p into vertex equivalence classes $[m_1], [m_2], \ldots, [m_q]$ as introduced in [12]. The observability don't cares at y_o can then be expressed in terms of this equivalence relation as follows,

$$ODC_o = \{m \in B^n | \mathcal{Y}_{y_o}(m) \sim \mathcal{Y}_{\overline{y}_o}(m)\}.$$

Furthermore, the ODC of an intermediate node y_o need not be expressed in terms of primary inputs. It can also be expressed in terms of intermediate nodes of the network. In particular, the observability don't cares for each of the nodes in the $\mathcal{Y}$ can be computed in terms of other variables in the separating set of nodes. In what follows we give some properties associated with the ODC's of the nodes in $\mathcal{Y}$.

Starting from the observability don't cares of the nodes in $\mathcal{Y}$ we can find the onset and the offset of each output function; therefore we can find the vertex equivalence classes of all the nodes in the separating set. The equivalence class to which m_i belongs, denoted $[m_i]$, can be computed as follows. Given $\overline{\frac{\partial z_k}{\partial y_i}}, i = 1, \ldots, p, k = 1, \ldots, l$:

- Use (3.1) to find F^{+k} for each output function z_k, $k = 1, \ldots, l$,
- Let $S_i = \{j | m_i \in F^{+j}\}$,
- $[m_i] = (\prod_{k \in S_i} F^{+k})(\prod_{k \in \overline{S}_i} \overline{F}^{+k})$.

The expression $[m_i] = (\prod_{k \in S_i} F^{+k})(\prod_{k \in \overline{S}_i} \overline{F}^{+k})$ is all the minterms for which each output function in S_i or $\overline{S}_i$ has a fixed value of 1 or 0.

Alternately, let $\mathcal{R}(\mathbf{y}, \mathbf{z}) = \prod_{i=1}^{l}(z_i \overline{\oplus} F^{+i})$. Then the equivalence classes are given by

$$E(\mathbf{y}, \mathbf{y}') = \mathcal{S}_z[\mathcal{R}(\mathbf{y}, \mathbf{z})\mathcal{R}(\mathbf{y}', \mathbf{z})]$$

Although this is not an efficient way to compute equivalence classes of $\mathcal{Y}$, it shows the relation between the observability don't cares of the nodes in $\mathcal{Y}$ and equivalence classes of $\mathcal{Y}$. In [23], the authors prove this result using a different approach.

Conversely we can compute the ODC's at each of the nodes in $\mathcal{Y}$ if the vertex equivalence classes of $\mathcal{Y}$ are known.

Theorem 3.6.1 *Let $[m_1], [m_2], \ldots, [m_q]$ be the functions representing the vertex equivalence classes of $\mathcal{Y}$. The observability don't care set of any node y_i in $\mathcal{Y}$ with respect to all the output functions is*

$$ODC_i = \sum_{j=1}^{q} C_{y_i}[m_j]$$

Proof Since ODC_i is independent of y_i, it can be viewed as consisting of pairs of vertices $(m_k, \hat{m}_k)$ in the space of the variables corresponding to nodes in the separating set where $(\hat{m}_k)_{\overline{y}_i} = (m_k)_{y_i}$ and $(\hat{m}_k)_{y_i} = (m_k)_{\overline{y}_i}$. For any such pair $(m_k, \hat{m}_k)$, if both vertices belong to the same equivalence class then both vertices produce the same set of outputs. Otherwise, at least one of the outputs is different. Therefore m_k and $\hat{m}_k$ belong to ODC_i if and only if they are in the same equivalence class. ∎

There are only two equivalence classes for any separating set of nodes $\mathcal{Y}$ in the observability network $\mathcal{N}'$: one is a new observability relation which gives the possible minterms in terms of the variables in $\mathcal{Y}$ for any input minterm; the other is the inverse of first, i.e. all the impossible combinations of variables in $\mathcal{Y}$ for any input minterm. The computation of these observability relations for a network decomposed into multi-output nodes is the subject of the next chapter.

Chapter 4

Observability Relations for Multi-Output Nodes

The observability relation as described in Section 2.4 provides a description of all the flexibility available in implementing a Boolean network $\mathcal{N}$. In this chapter, we develop techniques for finding observability relations for each component of a Boolean network decomposed into a set of multi-output nodes. The multi-output node can be a Boolean network itself. The original decomposition can be obtained in a variety of ways. For example, multi-output nodes can be obtained by clustering a set of single-output nodes in a regular Boolean network. Or, a Boolean network may be partitioned into a hierarchy of smaller networks such that each network satisfies some specific criteria. We show how to obtain maximum flexibility for implementing each element of a partition by computing its observability relation. Compatible observability relations are also computed for a given topological ordering of the nodes. A Boolean relation minimizer (such as [83]) can then be used to find a good two-level implementation from the observability relation computed for the node. Alternately, compatible don't cares can be derived from these observability relations and then used to optimize the multi-level network at the node using a conventional two-level minimizer.

4.1 Previous Work

In [17], an approach is developed for a unified synthesis of combinational and sequential circuits using characteristic functions. Each circuit is composed of a set of multi-

output blocks. The input-output behavior of the whole circuit is described by a Boolean relation called the Output Characteristic Function (OCF). It is assumed that the inputs to the circuit denoted by x are also inputs to all the multi-output blocks. Techniques are developed for finding Boolean relations describing input-output behavior of some multi-output node when the OCF of the circuit and the characteristic functions of all other nodes are known. It is shown that the developed formulas can be computed in a different way if all the blocks in the circuit are combinational implementations. The characteristic function of a combinational implementation of a block allows a unique output for each input. Here we build on the work in [17] and show how to find the compatible and maximal observability relations for a multi-output node n in a combinational circuit, given the observability relation (OCF in [17]) for the circuit and the Boolean relations for all other nodes. We also show that if the Boolean relations given for all other nodes except for node n are the characteristic functions of an implementation at those nodes, the compatible and maximal observability relations obtained for n are equivalent in all cases where a Boolean relation can be computed. As in [17], we first develop techniques for computing Boolean relations for two-way partitioned circuits and then generalize these techniques for any circuit decomposed into multi-output nodes. The techniques used to show the correctness of formulas for computing observability relations of multi-output nodes are different from that of [17] and clarify the distinction between maximal and compatible Boolean relations.

4.2 Two-Way Partitioning of a Boolean Relation

First, techniques for serial and parallel decomposition of an observability relation are discussed. These ideas are then used to find the observability relation for a general multi-output node in the network. Consider the situation depicted in Figure 4.1. An external observability relation or Boolean relation $\mathcal{O}$ is specified for the network. The network is decomposed into two parts (serial or parallel). The objective is to find maximal and compatible Boolean relations for the components of the network.

Definition 4.2.1 *A set of nodes has a set of compatible observability relations if each function at each node can be changed (as allowed by its observability relation) independent of all allowable changes in the functions at other nodes in the set.*

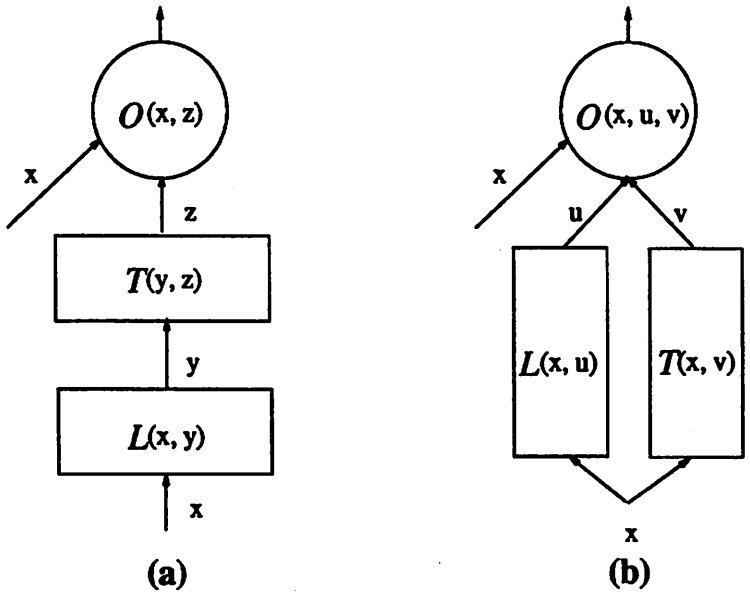

Figure 4.1: Decomposition of Observability Relation

Compatible observability relations computed for the components of the network have the extra advantage that each node can be optimized irrespective of other nodes.

The term *implementation* is often used in this chapter and has the following meaning.

Definition 4.2.2 *An* implementation *of a circuit is an observability relation which allows a single output minterm for each input minterm.*

Definition 4.2.3 *Given the observability relations $\mathcal{T}$ (equivalently $\mathcal{L}$) for one of the two partitions and $\mathcal{O}$ for the whole circuit as shown in Figure 4.1, $\mathcal{T}$ is said to be* consistent *with $\mathcal{O}$ if for any implementation based on $\mathcal{T}$ there is at least one implementation for the other partition in the circuit such that the implementation of the two partitions is allowed by $\mathcal{O}$.*

4.2.1 Serial Decomposition

Let $\mathcal{O}(\mathbf{x},\mathbf{z})$ be the observability relation for the network, and $\mathcal{T}(\mathbf{y},\mathbf{z})$ the observability relation for the top part of the network which is consistent with $\mathcal{O}(\mathbf{x},\mathbf{z})$ as shown in Figure 4.1(a). First, we derive an equation for computing a relation $\mathcal{L}$ for the lower part which is compatible with $\mathcal{T}$ and then find an $\mathcal{L}$ which is maximal with respect to $\mathcal{T}$. In

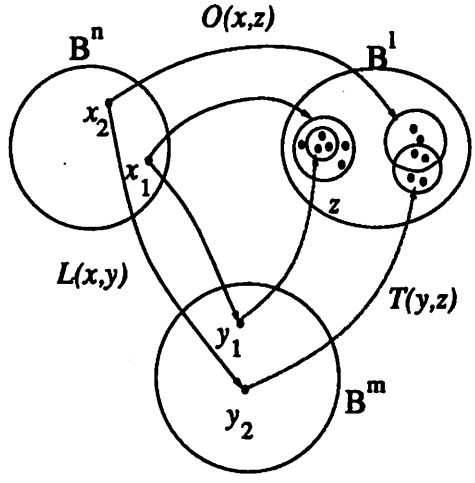

Figure 4.2: Observability Relation for a Cut

either case $\mathcal{L}$ must be consistent with $\mathcal{O}$. The mapping represented by $\mathcal{L}(\mathbf{x},\mathbf{y})$ is shown in Figure 4.2. It gives all the minterms in the separating set $\mathbf{y}$ allowed for some particular minterm $\mathbf{x}_i$. For each $\mathbf{x}_i$, there must be at least one minterm $\mathbf{y}_j$ such that $(\mathbf{x}_i,\mathbf{y}_j) \in \mathcal{L}$. Equivalently $\mathcal{S}_{\mathbf{y}}\mathcal{L}(\mathbf{x},\mathbf{y}) = 1$ or $\mathcal{L}$ must be well-defined. Otherwise, no implementation is possible for the lower part of the circuit. If no $\mathcal{L}$ can be found such that $\mathcal{S}_{\mathbf{y}}\mathcal{L}(\mathbf{x},\mathbf{y}) = 1$, $\mathcal{T}(\mathbf{y},\mathbf{z})$ is not consistent with the Boolean relation $\mathcal{O}(\mathbf{x},\mathbf{z})$.

The set of $\mathbf{z}$'s in relation with $\mathbf{x}_i$ allowed by $\mathcal{O}$ is $\mathcal{O}_{\mathbf{x}_i}$ and the set of $\mathbf{z}$'s allowed by $\mathcal{T}$ for a particular $\mathbf{y}_j$ is $\mathcal{T}_{\mathbf{y}_j}$. We investigate three different possibilities; $\mathcal{T}_{\mathbf{y}_j} \subseteq \mathcal{O}_{\mathbf{x}_i}$ (represented by $\mathbf{x_1}$ and $\mathbf{y_1}$ in Figure 4.2), $\mathcal{T}_{\mathbf{y}_j} \not\subseteq \mathcal{O}_{\mathbf{x}_i}$ but $\mathcal{T}_{\mathbf{y}_j}\mathcal{O}_{\mathbf{x}_i} \neq 0$ (represented by $\mathbf{x}_2$ and $\mathbf{y}_2$ in Figure 4.2), and finally $\mathcal{T}_{\mathbf{y}_j}\mathcal{O}_{\mathbf{x}_i} = 0$.

If $\mathcal{T}_{\mathbf{y}_j} \subseteq \mathcal{O}_{\mathbf{x}_i}$ and $\mathbf{x}_i$ is set to be in relation with $\mathbf{y}_j$ ($(\mathbf{x}_i,\mathbf{y}_j) \in \mathcal{L}$), no constraint is imposed on relation $\mathcal{T}$. This is because $\mathbf{y}_j$ can still accept any $\mathbf{z}_k$ as allowed by $\mathcal{T}$ and $(\mathbf{x}_i,\mathbf{z}_k) \in \mathcal{O}$. The relation $\mathcal{L}$ computed this way is compatible with $\mathcal{T}$ and consistent with $\mathcal{O}$. This means an implementation can be found from $\mathcal{L}$ independent of that found from $\mathcal{T}$ (This is the case for $\mathbf{x}_1$ and $\mathbf{y}_1$ shown in Figure 4.2).

If $\mathcal{T}_{\mathbf{y}_j} \not\subseteq \mathcal{O}_{\mathbf{x}_i}$, $\mathcal{T}_{\mathbf{y}_j}\mathcal{O}_{\mathbf{x}_i} \neq 0$, and we set $(\mathbf{x}_i,\mathbf{y}_j) \in \mathcal{L}$, some constraint is imposed on relation $\mathcal{T}$. If an implementation based on $\mathcal{L}$ gives $\mathbf{y}_j$ for input $\mathbf{x}_i$, any $\mathbf{z}_k$ allowed for $\mathbf{y}_j$ must be in $\mathcal{T}_{\mathbf{y}_j}\mathcal{O}_{\mathbf{x}_i}$. The relations $\mathcal{T}$ and $\mathcal{L}$ are not compatible in this case. $\mathcal{L}$ can be used to find an implementation for the lower part of the circuit. Then $\mathcal{T}$ must be made compatible to this implementation before it is used. It is shown here that $\mathcal{L}$ obtained this way is the

maximal observability relation consistent with $\mathcal{O}$.

Finally, if $\mathcal{T}_{\mathbf{y}_j}\mathcal{O}_{\mathbf{x}_i} = 0$, $\mathbf{x}_i$ cannot be in relation with $\mathbf{y}_j$ without violating $\mathcal{O}$.

Lemma 4.2.1 **(compatible)** *Given the relation $\mathcal{T}(\mathbf{y},\mathbf{z})$ and the observability relation $\mathcal{O}(\mathbf{x},\mathbf{z})$, the relation $\mathcal{L}^{cm}(\mathbf{x},\mathbf{y})$ expressing for each input minterm $\mathbf{x}$ all possible output minterms $\mathbf{y}$ maximally compatible with $\mathcal{T}$ and consistent with $\mathcal{O}$ is*

$$\mathcal{L}^{cm}(\mathbf{x},\mathbf{y}) = C_{\mathbf{z}}(\overline{\mathcal{T}}(\mathbf{y},\mathbf{z}) + \mathcal{O}(\mathbf{x},\mathbf{z})).$$

Proof The relations $\mathcal{T}$ and $\mathcal{L}^{cm}$ must be compatible with each other and consistent with $\mathcal{O}(\mathbf{x},\mathbf{z})$. We first show that $\mathcal{L}^{cm}$ maximally compatible with $\mathcal{T}$ implies that $(\mathbf{x}_i,\mathbf{y}_j) \in \mathcal{L}^{cm}$ if and only if $\mathcal{T}_{\mathbf{y}_j} \subseteq \mathcal{O}_{\mathbf{x}_i}$. This means $\mathcal{L}^{cm}$ consists only of all the pairs $(\mathbf{x}_i,\mathbf{y}_j)$ which satisfy $\mathcal{T}_{\mathbf{y}_j} \subseteq \mathcal{O}_{\mathbf{x}_i}$. If $\mathcal{T}_{\mathbf{y}_j} \subseteq \mathcal{O}_{\mathbf{x}_i}$, then any $\mathbf{z}_k$ in relation with $\mathbf{y}_j$ from $\mathcal{T}$ is also in relation with $\mathbf{x}_i$ allowed by $\mathcal{O}$, therefore, $(\mathbf{x}_i,\mathbf{y}_j) \in \mathcal{L}^{cm}$ is consistent with $\mathcal{O}(\mathbf{x},\mathbf{z})$. On the other hand, if $\{\exists \mathbf{z}_k | \mathbf{z}_k \in \mathcal{T}_{\mathbf{y}_j}, \mathbf{z}_k \notin \mathcal{O}_{\mathbf{x}_i}\}$, then $(\mathbf{x}_i,\mathbf{y}_j) \in \mathcal{L}^{cm}$ puts $\mathbf{z}_k$ in relation with $\mathbf{x}_i$ but $(\mathbf{x}_i,\mathbf{z}_k) \notin \mathcal{O}(\mathbf{x},\mathbf{z})$. As a result a circuit implemented based on $\mathcal{T}$ and $\mathcal{L}^{cm}$ could violate $\mathcal{O}$.

$\mathcal{T}_{\mathbf{y}_j} \subseteq \mathcal{O}_{\mathbf{x}_i}$ if and only if $\mathcal{T}_{\mathbf{y}_j}\overline{\mathcal{O}}_{\mathbf{x}_i} = 0$, or $\overline{\mathcal{T}}_{\mathbf{y}_j} + \mathcal{O}_{\mathbf{x}_i} = 1$. The term $\mathbf{x}_i\mathbf{y}_j C_{\mathbf{z}}(\overline{\mathcal{T}}_{\mathbf{y}_j} + \mathcal{O}_{\mathbf{x}_i})$ is equal to $\mathbf{x}_i\mathbf{y}_j$ if $C_{\mathbf{z}}(\overline{\mathcal{T}}_{\mathbf{y}_j} + \mathcal{O}_{\mathbf{x}_i}) = 1$, and 0 otherwise. $\mathcal{L}^{cm}$ can be written as,

$$\begin{aligned}
\mathcal{L}^{cm}(\mathbf{x},\mathbf{y}) &= \sum_{(\mathbf{x}_i,\mathbf{y}_j)\in B^{n+m}} \mathbf{x}_i\mathbf{y}_j C_{\mathbf{z}}(\overline{\mathcal{T}}_{\mathbf{y}_j} + \mathcal{O}_{\mathbf{x}_i}) \\
&= \sum C_{\mathbf{z}}(\mathbf{x}_i\mathbf{y}_j\overline{\mathcal{T}}_{\mathbf{y}_j} + \mathbf{x}_i\mathbf{y}_j\mathcal{O}_{\mathbf{x}_i}) \\
&= \sum C_{\mathbf{z}}(\mathbf{x}_i\mathbf{y}_j\overline{\mathcal{T}} + \mathbf{x}_i\mathbf{y}_j\mathcal{O}) \\
&= \sum \mathbf{x}_i\mathbf{y}_j C_{\mathbf{z}}(\overline{\mathcal{T}} + \mathcal{O}) \\
&= C_{\mathbf{z}}(\overline{\mathcal{T}} + \mathcal{O}) \sum_{(\mathbf{x}_i,\mathbf{y}_j)\in B^{n+m}} \mathbf{x}_i\mathbf{y}_j \\
&= C_{\mathbf{z}}(\overline{\mathcal{T}}(\mathbf{y},\mathbf{z}) + \mathcal{O}(\mathbf{x},\mathbf{z})).
\end{aligned}$$

■

For $\mathcal{L}^{cm}$ to be a well-defined Boolean relation it is necessary to have $S_{\mathbf{y}}\mathcal{L}^{cm}(\mathbf{x},\mathbf{y}) = 1$ (There exists at least one $\mathbf{y}$ for each $\mathbf{x}$.). $\mathcal{L}^{cm}$ obtained as in Lemma 4.2.1 may not be a well-defined Boolean relation because there can be $\mathbf{x}$'s for which no $\mathbf{y}$ is allowed.

The concept of compatible observability relation is similar to compatible don't cares. As we know, one could also compute maximal don't cares. Likewise, the relation $\mathcal{L}$ can be computed in a way which results in a maximal Boolean relation. The restriction that

$\mathcal{L}$ must be compatible with $\mathcal{T}$ is relaxed. But it is required that for any implementation based on $\mathcal{L}$ there is at least one possible implementation based on $\mathcal{T}$ allowed by $\mathcal{O}$. In this case, we say that $\mathcal{L}$ satisfies $\mathcal{T}$ and is consistent with $\mathcal{O}$.

Lemma 4.2.2 *(maximal) Given the relation $\mathcal{T}(\mathbf{y},\mathbf{z})$ and the observability relation $\mathcal{O}(\mathbf{x},\mathbf{z})$, the maximal relation $\mathcal{L}^m(\mathbf{x},\mathbf{y})$ satisfying $\mathcal{T}$ and consistent with $\mathcal{O}$ is*

$$\mathcal{L}^m(\mathbf{x},\mathbf{y}) = \mathcal{S}_{\mathbf{z}}\mathcal{O}(\mathbf{x},\mathbf{z})\mathcal{T}(\mathbf{y},\mathbf{z}).$$

Proof We first prove that $\mathcal{L}^m$ satisfies $\mathcal{T}$ and is consistent with $\mathcal{O}$ if and only if $\mathcal{T}_{\mathbf{y}_j}\mathcal{O}_{\mathbf{x}_i} \neq 0$ for any $\mathbf{x}_i\mathbf{y}_j \in \mathcal{L}^m$. If $\mathcal{T}_{\mathbf{y}_j}\mathcal{O}_{\mathbf{x}_i} \neq 0$, then there exists at least one $\mathbf{z}_k$ such that $\mathbf{z}_k \in \mathcal{T}_{\mathbf{y}_j}$ and $\mathbf{z}_k \in \mathcal{O}_{\mathbf{x}_i}$. As a result, for any implementation based on $\mathcal{L}^m$ that gives $\mathbf{y}_j$ for $\mathbf{x}_i$, there is at least one implementation satisfying $\mathcal{T}$ that is allowed by $\mathcal{O}$. On the other hand, if $\mathcal{T}_{\mathbf{y}_j}\mathcal{O}_{\mathbf{x}_i} = 0$, then no $\mathbf{z}$ is allowed for any implementation that gives $\mathbf{y}_j$ for $\mathbf{x}_i$.

$\mathcal{T}_{\mathbf{y}_j}\mathcal{O}_{\mathbf{x}_i} \neq 0$ if and only if $\mathcal{S}_{\mathbf{z}}(\mathcal{T}_{\mathbf{y}_j}\mathcal{O}_{\mathbf{x}_i})$ evaluates to 1. The value of $\mathbf{x}_i\mathbf{y}_j\mathcal{S}_{\mathbf{z}}(\mathcal{T}_{\mathbf{y}_j}\mathcal{O}_{\mathbf{x}_i})$ is $\mathbf{x}_i\mathbf{y}_j$ if $\mathcal{S}_{\mathbf{z}}(\mathcal{T}_{\mathbf{y}_j}\mathcal{O}_{\mathbf{x}_i}) = 1$ and 0 otherwise. As a result, we can write

$$\begin{aligned}
\mathcal{L}^m(\mathbf{x},\mathbf{y}) &= \sum_{(\mathbf{x}_i,\mathbf{y}_j)\in B^{n+m}} \mathbf{x}_i\mathbf{y}_j\mathcal{S}_{\mathbf{z}}(\mathcal{O}_{\mathbf{x}_i}\mathcal{T}_{\mathbf{y}_j}) \\
&= \sum \mathcal{S}_{\mathbf{z}}(\mathbf{x}_i\mathbf{y}_j\mathcal{O}_{\mathbf{x}_i}\mathcal{T}_{\mathbf{y}_j}) \\
&= \sum \mathcal{S}_{\mathbf{z}}(\mathbf{x}_i\mathbf{y}_j\mathcal{O}\mathcal{T}) \\
&= (\mathcal{S}_{\mathbf{z}}\mathcal{O}\mathcal{T}) \sum_{(\mathbf{x}_i,\mathbf{y}_j)\in B^{n+m}} \mathbf{x}_i\mathbf{y}_j \\
&= \mathcal{S}_{\mathbf{z}}(\mathcal{O}(\mathbf{x},\mathbf{z})\mathcal{T}(\mathbf{y},\mathbf{z})).
\end{aligned}$$

■

Note the difference between Lemmas 4.2.1 and 4.2.2; Lemma 4.2.1 requires compatibility with respect to $\mathcal{T}$ and consistency with $\mathcal{O}$ while 4.2.2 only requires consistency with $\mathcal{O}$. Clearly, $\mathcal{L}^{cm} \subseteq \mathcal{L}^m$; hence the latter is called maximal. If $\mathcal{S}_{\mathbf{y}}\mathcal{L}(\mathbf{x},\mathbf{y}) \neq 1$ ($\mathcal{L}$ is not a well-defined Boolean relation), $\mathcal{T}$ is not consistent with $\mathcal{O}$, because no implementation of $\mathcal{L}$ can be found for any implementation of $\mathcal{T}$ that is consistent with $\mathcal{O}$.

Lemma 4.2.3 *If $\mathcal{T}(\mathbf{y},\mathbf{z})$ is an implementation (i.e. for each $\mathbf{y}$ there exists a unique $\mathbf{z}$ such that $\mathcal{T}(\mathbf{y},\mathbf{z}) = 1$), then $\mathcal{C}_{\mathbf{z}}(\overline{\mathcal{T}} + \mathcal{O}) = \mathcal{S}_{\mathbf{z}}\mathcal{O}(\mathbf{x},\mathbf{z})\mathcal{T}(\mathbf{y},\mathbf{z})$; therefore $\mathcal{L}^{cm}(\mathbf{x},\mathbf{y}) = \mathcal{L}^m(\mathbf{x},\mathbf{y})$.*

Proof Because $\mathcal{T}$ is an implementation, it allows a unique $\mathbf{z}_k$ for each input $\mathbf{y}_j$; therefore, for any $\mathbf{x}_i$ and $\mathbf{y}_j$, $\mathcal{T}_{\mathbf{y}_j} \subseteq \mathcal{O}_{\mathbf{x}_i}$ if and only if $\mathcal{T}_{\mathbf{y}_j}\mathcal{O}_{\mathbf{x}_i} \neq 0$. $\mathcal{T}_{\mathbf{y}_j} \subseteq \mathcal{O}_{\mathbf{x}_i}$ if and only if $\mathcal{T}_{\mathbf{y}_j}\overline{\mathcal{O}}_{\mathbf{x}_i} = 0$, or $\overline{\mathcal{T}}_{\mathbf{y}_j} + \mathcal{O}_{\mathbf{x}_i} = 1$. $\mathcal{T}_{\mathbf{y}_j}\mathcal{O}_{\mathbf{x}_i} \neq 0$ if and only if $\mathcal{S}_{\mathbf{z}}(\mathcal{T}_{\mathbf{y}_j}\mathcal{O}_{\mathbf{x}_i}) = 1$.

The term $\mathbf{x}_i\mathbf{y}_j\mathcal{C}_{\mathbf{z}}(\overline{\mathcal{T}}_{\mathbf{y}_j}+\mathcal{O}_{\mathbf{x}_i})$ is equal to $\mathbf{x}_i\mathbf{y}_j$ if $\mathcal{C}_{\mathbf{z}}(\overline{\mathcal{T}}_{\mathbf{y}_j}+\mathcal{O}_{\mathbf{x}_i}) = 1$, and 0 otherwise. The term $\mathbf{x}_i\mathbf{y}_j\mathcal{S}_{\mathbf{z}}(\mathcal{T}_{\mathbf{y}_j}\mathcal{O}_{\mathbf{x}_i})$ is equal to $\mathbf{x}_i\mathbf{y}_j$ if $\mathcal{S}_{\mathbf{z}}(\mathcal{T}_{\mathbf{y}_j}\mathcal{O}_{\mathbf{x}_i}) = 1$ and 0 otherwise. Therefore

$$\begin{aligned}
\sum_{(\mathbf{x}_i,\mathbf{y}_j)\in B^{n+m}} \mathbf{x}_i\mathbf{y}_j\mathcal{C}_{\mathbf{z}}(\overline{\mathcal{T}}_{\mathbf{y}_j} + \mathcal{O}_{\mathbf{x}_i}) &= \sum_{(\mathbf{x}_i,\mathbf{y}_j)\in B^{n+m}} \mathbf{x}_i\mathbf{y}_j\mathcal{S}_{\mathbf{z}}(\mathcal{O}_{\mathbf{x}_i}\mathcal{T}_{\mathbf{y}_j}) \\
\sum \mathcal{C}_{\mathbf{z}}(\mathbf{x}_i\mathbf{y}_j\overline{\mathcal{T}}_{\mathbf{y}_j} + \mathbf{x}_i\mathbf{y}_j\mathcal{O}_{\mathbf{x}_i}) &= \sum \mathcal{S}_{\mathbf{z}}(\mathbf{x}_i\mathbf{y}_j\mathcal{O}_{\mathbf{x}_i}\mathcal{T}_{\mathbf{y}_j}) \\
\sum \mathcal{C}_{\mathbf{z}}(\mathbf{x}_i\mathbf{y}_j\overline{\mathcal{T}} + \mathbf{x}_i\mathbf{y}_j\mathcal{O}) &= \sum \mathcal{S}_{\mathbf{z}}(\mathbf{x}_i\mathbf{y}_j\mathcal{O}\mathcal{T}) \\
\mathcal{C}_{\mathbf{z}}(\overline{\mathcal{T}} + \mathcal{O}) \sum_{(\mathbf{x}_i,\mathbf{y}_j)\in B^{n+m}} \mathbf{x}_i\mathbf{y}_j &= (\mathcal{S}_{\mathbf{z}}\mathcal{O}\mathcal{T}) \sum_{(\mathbf{x}_i,\mathbf{y}_j)\in B^{n+m}} \mathbf{x}_i\mathbf{y}_j \\
\mathcal{C}_{\mathbf{z}}(\overline{\mathcal{T}}(\mathbf{y},\mathbf{z}) + \mathcal{O}(\mathbf{x},\mathbf{z})) &= \mathcal{S}_{\mathbf{z}}(\mathcal{O}(\mathbf{x},\mathbf{z})\mathcal{T}(\mathbf{y},\mathbf{z})) \\
\mathcal{L}^{cm}(\mathbf{x},\mathbf{y}) &= \mathcal{L}^{m}(\mathbf{x},\mathbf{y})
\end{aligned}$$

■

Given a relation $\mathcal{L}$ consistent with $\mathcal{O}$, one can also find compatible and maximal relations for $\mathcal{T}$.

Lemma 4.2.4 **(compatible)** *Given the relation $\mathcal{L}(\mathbf{x},\mathbf{y})$ and the observability relation $\mathcal{O}(\mathbf{x},\mathbf{z})$, the relation $\mathcal{T}^{cm}(\mathbf{y},\mathbf{z})$ expressing for each minterm $\mathbf{y}$ all possible output minterms $\mathbf{z}$ maximally compatible with $\mathcal{L}$ and consistent with $\mathcal{O}$ is*

$$\mathcal{T}^{cm}(\mathbf{y},\mathbf{z}) = \mathcal{C}_{\mathbf{x}}(\mathcal{O}(\mathbf{x},\mathbf{z}) + \overline{\mathcal{L}}(\mathbf{x},\mathbf{y}))$$

Proof We first prove that $\mathcal{T}^{cm}(\mathbf{y},\mathbf{z})$ is compatible with $\mathcal{L}(\mathbf{x},\mathbf{y})$ and consistent with $\mathcal{O}(\mathbf{x},\mathbf{z})$ if and only if $\mathcal{L}_{\mathbf{y}_j} \subseteq \mathcal{O}_{\mathbf{z}_i}$ for any $(\mathbf{y}_j,\mathbf{z}_i) \in \mathcal{T}^{cm}$. If $\mathcal{L}_{\mathbf{y}_j} \subseteq \mathcal{O}_{\mathbf{z}_i}$, then any $\mathbf{x}_k$ in relation with $\mathbf{y}_j$ from $\mathcal{L}$ is also in relation with $\mathbf{z}_i$ allowed by $\mathcal{O}$, therefore, $\mathbf{y}_j\mathbf{z}_i \in \mathcal{T}^{cm}$ consistent with $\mathcal{O}(\mathbf{x},\mathbf{z})$. There is no restriction on $\mathcal{L}$. On the other hand, if $\{\exists\mathbf{x}_k|\mathbf{x}_k \in \mathcal{L}_{\mathbf{y}_j}, \mathbf{x}_k \notin \mathcal{O}_{\mathbf{z}_i}, \mathbf{y}_j\mathbf{z}_i \in \mathcal{T}^{cm}\}$, then one can choose an implementation producing output $\mathbf{z}_i$ for $\mathbf{y}_j$ from $\mathcal{T}^{cm}$ and an implementation producing $\mathbf{y}_j$ for input $\mathbf{x}_k$ from $\mathcal{L}$. These two implementations are not allowed by $\mathcal{O}$ because they produce output $\mathbf{z}_i$ for input $\mathbf{x}_k$ and $(\mathbf{x}_k,\mathbf{z}_i) \notin \mathcal{O}$.

$\mathcal{L}_{\mathbf{y}_j} \subseteq \mathcal{O}_{\mathbf{z}_i}$ if and only if $\mathcal{C}_{\mathbf{x}}(\mathcal{O}_{\mathbf{z}_i} + \overline{\mathcal{L}}_{\mathbf{y}_j}) = 1$. The term $\mathbf{y}_j\mathbf{z}_i\mathcal{C}_{\mathbf{x}}(\mathcal{O}_{\mathbf{z}_i} + \overline{\mathcal{L}}_{\mathbf{y}_j})$ is equal to $\mathbf{y}_j\mathbf{z}_i$ if $\mathcal{C}_{\mathbf{x}}(\mathcal{O}_{\mathbf{z}_i} + \overline{\mathcal{L}}_{\mathbf{y}_j}) = 1$ and 0 otherwise. Therefore, we can write $\mathcal{T}^{cm}$ as

$$\mathcal{T}^{cm}(\mathbf{y},\mathbf{z}) = \sum_{(\mathbf{y}_j,\mathbf{z}_i)\in B^{m+l}} \mathbf{y}_j\mathbf{z}_i\mathcal{C}_{\mathbf{x}}(\mathcal{O}_{\mathbf{z}_i} + \overline{\mathcal{L}}_{\mathbf{y}_j})$$

$$
\begin{aligned}
&= \sum \mathbf{y}_j \mathbf{z}_i C_{\mathbf{x}}(\mathbf{z}_i \mathcal{O}_{\mathbf{z}_i} + \mathbf{y}_j \overline{\mathcal{L}}_{\mathbf{y}_j}) \\
&= \sum \mathbf{y}_j \mathbf{z}_i C_{\mathbf{x}}(\mathbf{z}_i \mathcal{O} + \mathbf{y}_j \overline{\mathcal{L}}) \\
&= C_{\mathbf{x}}(\mathcal{O} + \overline{\mathcal{L}}) \sum_{(\mathbf{y}_j, \mathbf{z}_i) \in B^{m+l}} \mathbf{y}_j \mathbf{z}_i \\
&= C_{\mathbf{x}}(\mathcal{O}(\mathbf{x}, \mathbf{z}) + \overline{\mathcal{L}}(\mathbf{x}, \mathbf{y})).
\end{aligned}
$$

■

If $\mathcal{L}$ is an implementation, $\mathcal{T}^{cm}$ is also maximal ($\mathcal{T}^{cm} = \mathcal{T}^m$). It would seem from the previous lemmas that a relation $\mathcal{T}$ which is maximal with respect to a general observability relation $\mathcal{L}$ and consistent with $\mathcal{O}$ is

$$\mathcal{T}^m(\mathbf{y}, \mathbf{z}) = S_{\mathbf{x}}(\mathcal{O}(\mathbf{x}, \mathbf{z}) \mathcal{L}(\mathbf{x}, \mathbf{y})).$$

This formula is obtained from the assumption that $\mathcal{T}^m$ is maximal with respect to $\mathcal{L}$ and consistent with $\mathcal{O}$ if and only if $\mathcal{L}_{\mathbf{y}_j} \mathcal{O}_{\mathbf{z}_i} \neq 0$ for any $\mathbf{y}_j \mathbf{z}_i \in \mathcal{T}^m$. However, $\mathcal{T}^m$ may not be well-defined. Suppose $\mathcal{L}_{\mathbf{y}_j} = 0$ ($\mathbf{y}_j$ cannot be obtained by any $\mathbf{x}$). Then $\mathcal{L}_{\mathbf{y}_j} \mathcal{O}_{\mathbf{z}_i} = 0$ and no $\mathbf{z}$ is allowed for such $\mathbf{y}$. We can add another term to the above equation to take care of all the $\mathbf{y}$'s that cannot be obtained from $\mathcal{L}$

$$\mathcal{T}^m(\mathbf{y}, \mathbf{z}) = S_{\mathbf{x}}(\mathcal{O}(\mathbf{x}, \mathbf{z}) \mathcal{L}(\mathbf{x}, \mathbf{y})) + \overline{S_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \mathbf{y})}. \tag{4.1}$$

$\overline{S_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \mathbf{y})}$ is all the $\mathbf{y}$'s not obtained from any $\mathbf{x}$, therefore all $\mathbf{z}$'s are allowed for any such $\mathbf{y}$.

The above formula by itself is not correct. This is because, if an implementation is obtained from $\mathcal{T}$, there is no guarantee that an implementation can be obtained for the lower part of the circuit. Assume an implementation based on $\mathcal{T}^m$ gives $\mathbf{z}_i$ for $\mathbf{y}_j$. Then any implementation for the lower part of the circuit must choose $\mathbf{x}_k \in \mathcal{L}_{\mathbf{y}_j} \mathcal{O}_{\mathbf{z}_i}$ to give $\mathbf{y}_j$, but there is no guarantee that all the $\mathbf{x}$'s can be covered this way. There could be an $\mathbf{x}$ that is in no $\mathcal{L}_{\mathbf{y}_j} \mathcal{O}_{\mathbf{z}_i}$ where $\mathbf{z}_i$ is the output for $\mathbf{y}_j$ in the implementation based on $\mathcal{T}^m$.

Example:

We give a simple example to demonstrate this. Assume $\mathcal{L}$ is an implementation which gives $\mathbf{y}_1$ for every possible input $\mathbf{x}$. Also assume $\mathcal{O}$ is such that it allows $\mathbf{z}_1$ for every input $\mathbf{x}$ and $\mathbf{z}_2$ only for input $\mathbf{x}_1$ as shown in Figure 4.3. Clearly, there is an implementation for the circuit satisfying $\mathcal{L}$ and $\mathcal{O}$. The implementation for the top part of the circuit can give $\mathbf{z}_1$ for every $\mathbf{y}$; the implementation for the lower part is $\mathcal{L}$ itself; therefore, $\mathcal{L}$ is consistent

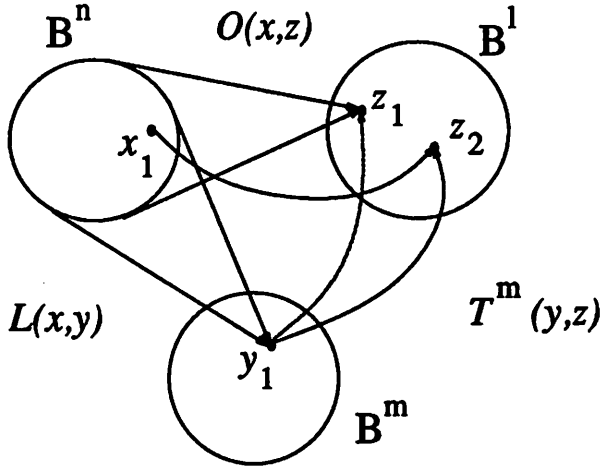

Figure 4.3: Example

with $\mathcal{O}$. If $\mathcal{T}^m$ is computed as in (4.1), then $\mathbf{y_1z_1}, \mathbf{y_1z_2} \in \mathcal{T}^m$. All $\mathbf{z}$'s are allowed for any other $\mathbf{y}$. A possible implementation based on $\mathcal{T}^m$ can give $\mathbf{z_2}$ for all $\mathbf{y}$, but in such case, no implementation from $\mathcal{L}$ can be found which satisfies $\mathcal{O}$, therefore, (4.1) is not correct by itself.

Every valid implementation $t^m \in \mathcal{T}^m$ must allow at least one implementation for the lower part of the circuit satisfying both $\mathcal{L}$ and $\mathcal{O}$. From Lemma 4.2.2

$$l^m(\mathbf{x},\mathbf{y}) = \mathcal{S}_\mathbf{z}\mathcal{O}(\mathbf{x},\mathbf{z})t^m(\mathbf{y},\mathbf{z})$$

is all the implementation allowed satisfying t^m and consistent with $\mathcal{O}$ for the lower part of the circuit. t^m is consistent with $\mathcal{O}$ if l^m is well-defined, i.e. $\mathcal{S}_\mathbf{y}\mathcal{S}_\mathbf{z}\mathcal{O}(\mathbf{x},\mathbf{z})t^m(\mathbf{y},\mathbf{z}) = 1$. It satisfies $\mathcal{L}$ and is consistent with $\mathcal{O}$ if $l^m\mathcal{L}$ is well-defined, i.e. $\mathcal{S}_\mathbf{y}(\mathcal{L}(\mathbf{x},\mathbf{y})\mathcal{S}_\mathbf{z}\mathcal{O}(\mathbf{x},\mathbf{z})t^m(\mathbf{y},\mathbf{z})) = 1$. As a result, the set of all implementations possible for the top part of the circuit is all $t^m \in T^m$ (given by (4.1)) such that $\mathcal{S}_\mathbf{y}(\mathcal{L}(\mathbf{x},\mathbf{y})\mathcal{S}_\mathbf{z}\mathcal{O}(\mathbf{x},\mathbf{z})t^m(\mathbf{y},\mathbf{z})) = 1$.

Unfortunately, we do not know how to express this as a Boolean relation. The above discussion seems to lead to a constrained form of Boolean relation minimization:

$$\min\ t \ \leq\ \mathcal{S}_\mathbf{x}(\mathcal{O}(\mathbf{x},\mathbf{z})\mathcal{L}(\mathbf{x},\mathbf{y})) + \overline{\mathcal{S}_\mathbf{x}\mathcal{L}(\mathbf{x},\mathbf{y})} \tag{4.2}$$

$$s.t. \qquad \mathcal{S}_\mathbf{y}(\mathcal{L}(\mathbf{x},\mathbf{y})\mathcal{S}_\mathbf{z}\mathcal{O}(\mathbf{x},\mathbf{z})t(\mathbf{y},\mathbf{z})) \equiv 1. \tag{4.3}$$

It is unknown if this can be rewritten in a simpler form.

4.2.2 Parallel Decomposition

Given the observability relations $\mathcal{T}$ consistent with the relation $\mathcal{O}$ as shown in Figure 4.1(b), we develop formulas for computing maximal and compatible $\mathcal{L}$.

Lemma 4.2.5 *(compatible) Given the observability relations* $\mathcal{T}(\mathbf{x},\mathbf{v})$ *and* $\mathcal{O}(\mathbf{x},\mathbf{u},\mathbf{v})$*, the relation* $\mathcal{L}^{cm}(\mathbf{x},\mathbf{u})$ *expressing for each input minterm* $\mathbf{x}$ *all possible output minterms* $\mathbf{u}$ *maximally compatible with* $\mathcal{T}$ *and consistent with* $\mathcal{O}$ *is*

$$\mathcal{L}^{cm}(\mathbf{x},\mathbf{u}) = C_{\mathbf{v}}(\overline{\mathcal{T}}(\mathbf{x},\mathbf{v}) + \mathcal{O}(\mathbf{x},\mathbf{u},\mathbf{v})).$$

Proof We first prove that $\mathcal{L}^{cm}$ is compatible with $\mathcal{T}$ and consistent with $\mathcal{O}$ if and only if $\mathcal{T}_{\mathbf{x}_i} \subseteq \mathcal{O}_{\mathbf{x}_i\mathbf{u}_j}$ for each $(\mathbf{x}_i,\mathbf{u}_j) \in \mathcal{L}^{cm}$. If $\mathcal{T}_{\mathbf{x}_i} \subseteq \mathcal{O}_{\mathbf{x}_i\mathbf{u}_j}$, then any $\mathbf{v}_k$ in relation with $\mathbf{x}_i$ from $\mathcal{T}$ also satisfies $(\mathbf{x}_i,\mathbf{u}_j,\mathbf{v}_k) \in \mathcal{O}$; therefore, $(\mathbf{x}_i,\mathbf{u}_j) \in \mathcal{L}^{cm}$ is compatible with $\mathcal{T}$ and consistent with $\mathcal{O}$. On the other hand, if $\{\exists \mathbf{v}_k | \mathbf{v}_k \in \mathcal{T}_{\mathbf{x}_i}, \mathbf{v}_k \notin \mathcal{O}_{\mathbf{x}_i\mathbf{u}_j}\}$, then $(\mathbf{x}_i,\mathbf{u}_j) \in \mathcal{L}^{cm}$ and $\mathbf{x}_i\mathbf{v}_k \in \mathcal{T}$ may put $(\mathbf{v}_k,\mathbf{u}_j)$ in relation with $\mathbf{x}_i$ but $(\mathbf{x}_i,\mathbf{u}_j,\mathbf{v}_k) \notin \mathcal{O}(\mathbf{x},\mathbf{u},\mathbf{v})$. As a result an implementation based on $\mathcal{T}$ and $\mathcal{L}^{cm}$ could result in a circuit that violates $\mathcal{O}$.

$\mathcal{T}_{\mathbf{x}_i} \subseteq \mathcal{O}_{\mathbf{x}_i\mathbf{u}_j}$ if and only if $C_{\mathbf{v}}(\overline{\mathcal{T}}_{\mathbf{x}_i} + \mathcal{O}_{\mathbf{x}_i\mathbf{u}_j}) = 1$. The term $\mathbf{x}_i\mathbf{u}_j C_{\mathbf{v}}(\overline{\mathcal{T}}_{\mathbf{x}_i} + \mathcal{O}_{\mathbf{x}_i\mathbf{u}_j})$ is equal to $\mathbf{x}_i\mathbf{u}_j$ if $C_{\mathbf{v}}(\overline{\mathcal{T}}_{\mathbf{x}_i} + \mathcal{O}_{\mathbf{x}_i\mathbf{u}_j}) = 1$ and 0 otherwise. As a result, $\mathcal{L}^{cm}$ can be written as

$$\begin{aligned}
\mathcal{L}^{cm}(\mathbf{x},\mathbf{u}) &= \sum \mathbf{x}_i\mathbf{u}_j C_{\mathbf{v}}(\overline{\mathcal{T}}_{\mathbf{x}_i} + \mathcal{O}_{\mathbf{x}_i\mathbf{u}_j}) \\
&= \sum C_{\mathbf{v}}(\mathbf{x}_i\mathbf{u}_j\overline{\mathcal{T}}_{\mathbf{x}_i} + \mathbf{x}_i\mathbf{u}_j\mathcal{O}_{\mathbf{x}_i\mathbf{u}_j}) \\
&= \sum C_{\mathbf{v}}(\mathbf{x}_i\mathbf{u}_j\overline{\mathcal{T}} + \mathbf{x}_i\mathbf{u}_j\mathcal{O}) \\
&= \sum \mathbf{x}_i\mathbf{u}_j C_{\mathbf{v}}(\overline{\mathcal{T}} + \mathcal{O}) \\
&= C_{\mathbf{v}}(\overline{\mathcal{T}} + \mathcal{O}) \sum \mathbf{x}_i\mathbf{u}_j \\
&= C_{\mathbf{v}}(\overline{\mathcal{T}}(\mathbf{x},\mathbf{v}) + \mathcal{O}(\mathbf{x},\mathbf{u},\mathbf{v})).
\end{aligned}$$

■

Lemma 4.2.6 *(maximal) Given the observability relations* $\mathcal{T}(\mathbf{x},\mathbf{v})$ *and* $\mathcal{O}(\mathbf{x},\mathbf{u},\mathbf{v})$*, the relation* $\mathcal{L}^{m}(\mathbf{x},\mathbf{u})$ *expressing for each input minterm* $\mathbf{x}$ *the maximal possible minterms* $\mathbf{u}$ *consistent with* $\mathcal{O}$ *is*

$$\mathcal{L}^{m}(\mathbf{x},\mathbf{u}) = S_{\mathbf{v}}\mathcal{O}(\mathbf{x},\mathbf{u},\mathbf{v})\mathcal{T}(\mathbf{x},\mathbf{v}).$$

Proof We prove first that $\mathcal{L}^m$ is consistent with $\mathcal{O}$ and maximal with respect to $\mathcal{T}$ if and only if $\mathcal{T}_{\mathbf{x}_i}\mathcal{O}_{\mathbf{x}_i\mathbf{u}_j} \neq 0$, for each $(\mathbf{x}_i, \mathbf{u}_j) \in \mathcal{L}^m$. If $\mathcal{T}_{\mathbf{x}_i}\mathcal{O}_{\mathbf{x}_i\mathbf{u}_j} \neq 0$, then there exists at least one $\mathbf{v}_k$ such that $\mathbf{v}_k \in \mathcal{T}_{\mathbf{x}_i}$ and $\mathbf{v}_k \in \mathcal{O}_{\mathbf{x}_i\mathbf{u}_j}$. As a result, for any implementation based on $\mathcal{L}^m$ that gives $\mathbf{u}_j$ for $\mathbf{x}_i$, there is at least one implementation allowed by $\mathcal{T}$ that satisfies $\mathcal{O}$. On the other hand, if $\mathcal{T}_{\mathbf{x}_i}\mathcal{O}_{\mathbf{x}_i\mathbf{u}_j} = 0$, then no $\mathbf{v}_k$ is allowed for any implementation that gives $\mathbf{u}_j$ for $\mathbf{x}_i$.

$\mathcal{T}_{\mathbf{x}_i}\mathcal{O}_{\mathbf{x}_i\mathbf{u}_j} \neq 0$ if and only if $\mathcal{S}_{\mathbf{v}}(\mathcal{T}_{\mathbf{x}_i}\mathcal{O}_{\mathbf{x}_i\mathbf{u}_j}) = 1$. $\mathbf{x}_i\mathbf{u}_j\mathcal{S}_{\mathbf{v}}(\mathcal{O}_{\mathbf{x}_i\mathbf{u}_j}\mathcal{T}_{\mathbf{x}_i})$ is equal to $\mathbf{x}_i\mathbf{u}_j$ if $\mathcal{S}_{\mathbf{v}}(\mathcal{T}_{\mathbf{x}_i}\mathcal{O}_{\mathbf{x}_i\mathbf{u}_j}) = 1$ and 0 otherwise. As a result, we can write

$$\begin{aligned}
\mathcal{L}^m(\mathbf{x},\mathbf{u}) &= \sum \mathbf{x}_i\mathbf{u}_j\mathcal{S}_{\mathbf{v}}(\mathcal{O}_{\mathbf{x}_i\mathbf{u}_j}\mathcal{T}_{\mathbf{x}_i}) \\
&= \sum \mathcal{S}_{\mathbf{v}}(\mathbf{x}_i\mathbf{u}_j\mathcal{O}_{\mathbf{x}_i\mathbf{u}_j}\mathcal{T}_{\mathbf{x}_i}) \\
&= \sum \mathcal{S}_{\mathbf{v}}(\mathbf{x}_i\mathbf{u}_j\mathcal{O}\mathcal{T}) \\
&= \mathcal{S}_{\mathbf{v}}(\mathcal{O}\mathcal{T})\sum \mathbf{x}_i\mathbf{u}_j \\
&= \mathcal{S}_{\mathbf{v}}(\mathcal{O}(\mathbf{x},\mathbf{u},\mathbf{v})\mathcal{T}(\mathbf{x},\mathbf{v})).
\end{aligned}$$

■

Lemma 4.2.7 *If $\mathcal{T}(\mathbf{x},\mathbf{v})$ is an implementation (i.e. for each $\mathbf{x}$ there exists a unique $\mathbf{v}$ such that $\mathcal{T}(\mathbf{x},\mathbf{v}) = 1$), then $\mathcal{C}_{\mathbf{v}}(\overline{\mathcal{T}}(\mathbf{x},\mathbf{v}) + \mathcal{O}(\mathbf{x},\mathbf{u},\mathbf{v})) = \mathcal{S}_{\mathbf{v}}\mathcal{O}(\mathbf{x},\mathbf{u},\mathbf{v})\mathcal{T}(\mathbf{x},\mathbf{v})$ and therefore, $\mathcal{L}^{cm}(\mathbf{x},\mathbf{u}) = \mathcal{L}^m(\mathbf{x},\mathbf{u})$.*

Proof Because $\mathcal{T}$ is an implementation, it allows a unique $\mathbf{v}_k$ for each input $\mathbf{x}_i$; therefore, for any $\mathbf{x}_i$ and $\mathbf{u}_j$, $\mathcal{T}_{\mathbf{x}_i} \subseteq \mathcal{O}_{\mathbf{x}_i\mathbf{u}_j}$ if and only if $\mathcal{T}_{\mathbf{x}_i}\mathcal{O}_{\mathbf{x}_i\mathbf{u}_j} \neq 0$. $\mathcal{T}_{\mathbf{x}_i} \subseteq \mathcal{O}_{\mathbf{x}_i\mathbf{u}_j}$ if and only if $\mathcal{C}_{\mathbf{v}}(\overline{\mathcal{T}}_{\mathbf{x}_i} + \mathcal{O}_{\mathbf{x}_i\mathbf{u}_j}) = 1$. $\mathcal{T}_{\mathbf{x}_i}\mathcal{O}_{\mathbf{x}_i\mathbf{u}_j} \neq 0$ if and only if $\mathcal{S}_{\mathbf{v}}(\mathcal{T}_{\mathbf{x}_i}\mathcal{O}_{\mathbf{x}_i\mathbf{u}_j}) = 1$. The term $\mathbf{x}_i\mathbf{u}_j\mathcal{C}_{\mathbf{v}}(\overline{\mathcal{T}}_{\mathbf{x}_i} + \mathcal{O}_{\mathbf{x}_i\mathbf{u}_j})$ is equal to $\mathbf{x}_i\mathbf{u}_j$ if $\mathcal{C}_{\mathbf{v}}(\overline{\mathcal{T}}_{\mathbf{x}_i} + \mathcal{O}_{\mathbf{x}_i\mathbf{u}_j}) = 1$ and 0 otherwise. The term $\mathbf{x}_i\mathbf{u}_j\mathcal{S}_{\mathbf{v}}(\mathcal{O}_{\mathbf{x}_i\mathbf{u}_j}\mathcal{T}_{\mathbf{x}_i})$ is equal to $\mathbf{x}_i\mathbf{u}_j$ if $\mathcal{S}_{\mathbf{v}}(\mathcal{T}_{\mathbf{x}_i}\mathcal{O}_{\mathbf{x}_i\mathbf{u}_j}) = 1$ and 0 otherwise. As a result,

$$\begin{aligned}
\sum \mathbf{x}_i\mathbf{u}_j\mathcal{C}_{\mathbf{v}}(\overline{\mathcal{T}}_{\mathbf{x}_i} + \mathcal{O}_{\mathbf{x}_i\mathbf{u}_j}) &= \sum \mathbf{x}_i\mathbf{u}_j\mathcal{S}_{\mathbf{v}}(\mathcal{O}_{\mathbf{x}_i\mathbf{u}_j}\mathcal{T}_{\mathbf{x}_i}) \\
\sum \mathcal{C}_{\mathbf{v}}(\mathbf{x}_i\mathbf{u}_j\overline{\mathcal{T}}_{\mathbf{x}_i} + \mathbf{x}_i\mathbf{u}_j\mathcal{O}_{\mathbf{x}_i\mathbf{u}_j}) &= \sum \mathcal{S}_{\mathbf{v}}(\mathbf{x}_i\mathbf{u}_j\mathcal{O}_{\mathbf{x}_i\mathbf{u}_j}\mathcal{T}_{\mathbf{x}_i}) \\
\sum \mathcal{C}_{\mathbf{v}}(\mathbf{x}_i\mathbf{u}_j\overline{\mathcal{T}} + \mathbf{x}_i\mathbf{u}_j\mathcal{O}) &= \sum \mathcal{S}_{\mathbf{v}}(\mathbf{x}_i\mathbf{u}_j\mathcal{O}\mathcal{T}) \\
\sum \mathbf{x}_i\mathbf{u}_j\mathcal{C}_{\mathbf{v}}(\overline{\mathcal{T}} + \mathcal{O}) &= \sum \mathbf{x}_i\mathbf{u}_j\mathcal{S}_{\mathbf{v}}(\mathcal{O}\mathcal{T}) \\
\mathcal{C}_{\mathbf{v}}(\overline{\mathcal{T}} + \mathcal{O})\sum \mathbf{x}_i\mathbf{u}_j &= \mathcal{S}_{\mathbf{v}}(\mathcal{O}\mathcal{T})\sum \mathbf{x}_i\mathbf{u}_j \\
\mathcal{C}_{\mathbf{v}}(\overline{\mathcal{T}}(\mathbf{x},\mathbf{v}) + \mathcal{O}(\mathbf{x},\mathbf{u},\mathbf{v})) &= \mathcal{S}_{\mathbf{v}}(\mathcal{O}(\mathbf{x},\mathbf{u},\mathbf{v})\mathcal{T}(\mathbf{x},\mathbf{v})).
\end{aligned}$$

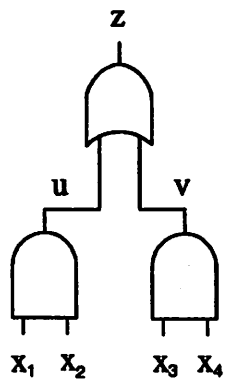

Figure 4.4: Example

Therefore, $\mathcal{L}^{cm}(\mathbf{x},\mathbf{u}) = \mathcal{L}^{m}(\mathbf{x},\mathbf{u})$. ∎

Example:

We find maximum and compatible observability relations for nodes u and v using the observability relation of the separating set u, v shown in Figure 4.4. These nodes have a single output which allows comparison with compatible observability don't cares obtained for the nodes. The observability relation for the circuit is $\mathcal{O}(\mathbf{z},\mathbf{x}) = z_1 \overline{\oplus} (x_1x_2 + x_3x_4)$. The observability relation for the separating set u, v is $\mathcal{O}(u,v,\mathbf{x}) = (u+v)\overline{\oplus}(x_1x_2 + x_3x_4)$. The current implementation at v is $\mathcal{T}(v,\mathbf{x}) = v\overline{\oplus}(x_3 + x_4)$. Thus,

$$\begin{aligned}
\mathcal{L}^{m}(u,\mathbf{x}) &= \mathcal{S}_v\mathcal{O}(u,v,\mathbf{x})\mathcal{T}(v,\mathbf{x}) \\
&= \mathcal{S}_v(u+v)\overline{\oplus}(x_1x_2 + x_3x_4)v\overline{\oplus}(x_3 + x_4) \\
&= x_3x_4 + u\overline{\oplus}(x_1x_2).
\end{aligned}$$

The full ODC at node u from the observability relation at u is

$$\begin{aligned}
ODC_u &= C_u\mathcal{L}^{m}(u,\mathbf{x}) \\
&= C_u(x_3x_4 + u\overline{\oplus}(x_1x_2)) \\
&= x_3x_4
\end{aligned}$$

which is equal to what one gets with a direct computation of ODC's. Having computed $\mathcal{L}^{m}(u,\mathbf{x})$ we know how to find $\mathcal{T}^{c}(v,\mathbf{x})$ which is compatible to it.

$$\begin{aligned}
\mathcal{T}^{c}(v,\mathbf{x}) &= C_u(\mathcal{O}(u,v,\mathbf{x}) + \overline{\mathcal{L}}^{m}(u,\mathbf{x})) \\
&= C_u((u+v)\overline{\oplus}(x_1x_2 + x_3x_4) + (\overline{x}_3\overline{x}_4(u \oplus (x_1x_2))) \\
&= v\overline{\oplus}(x_3x_4) + x_1x_2(\overline{x}_3 + \overline{x}_4).
\end{aligned}$$

The CODC at v is

$$\begin{aligned} CODC_v &= C_v(v\overline{\oplus}(x_3x_4) + x_1x_2(\overline{x}_3 + \overline{x}_4)) \\ &= x_1x_2(\overline{x}_3 + \overline{x}_4) \end{aligned}$$

4.3 Compatible and Maximal Observability Relations

Here we address the problem of finding *complete don't cares* for a multi-output node n and then simplifying it. This complete don't care set is effectively the observability relation for the node defined in the same way as the observability relation of a normal Boolean network. This can be either a local observability relation $\mathcal{O}^l(\mathbf{y}, \mathbf{u})$ where the $\mathbf{y}$ are the local fanins for the node and the $\mathbf{u}$ are the node outputs, or the global observability relation $\mathcal{O}^g(\mathbf{x}, \mathbf{u})$ giving the relation required between the primary inputs $\mathbf{x}$ and the node outputs $\mathbf{u}$.

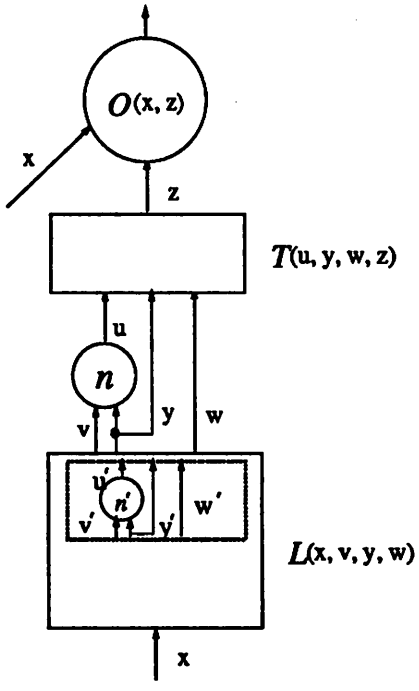

Figure 4.5: Observability Network for a Network of Multi-Output Nodes

Each multi-output node n has a set of inputs v which are unique to that node, a set of inputs y which are shared with other nodes, and a set of outputs u. The rest of the variables in the two separating sets of variables before and after n which divide the

network into two partitions are represented by $\mathbf{w}$ (a typical node is shown in Figure 4.5). The function at n is represented by $F(\mathbf{v},\mathbf{y})$. The variables $(\mathbf{u},\mathbf{y},\mathbf{w})$ and $(\mathbf{v},\mathbf{y},\mathbf{w})$ represent separating sets of variables in the network just after and just before node n (see Figure 4.5). Note that the variables $\mathbf{y}$ which are fanins of n can be inputs to other nodes in the fanout network as shown in Figure 4.5. The characteristic function for the top part of the network $\mathcal{T}(\mathbf{u},\mathbf{y},\mathbf{w},\mathbf{z})$ represents the present implementation after n and the characteristic function for the lower part of the network $\mathcal{L}(\mathbf{x},\mathbf{v},\mathbf{y},\mathbf{w})$, the implementation before n. $\mathcal{L}(\mathbf{x},\mathbf{v},\mathbf{y},\mathbf{w})$ is also called the controllability function [18] for the separating set $(\mathbf{v},\mathbf{y},\mathbf{w})$, and gives the image computation from $\mathbf{x}$ to $(\mathbf{v},\mathbf{y},\mathbf{w})$.

Using the lemmas for parallel and serial decomposition of a Boolean relation, the observability relation for a multi-output node n shown in Figure 4.5 can be computed. Let $\mathcal{L}_1$ be the relation between $\mathbf{y},\mathbf{w}$ and inputs $\mathbf{x}$, and $\mathcal{L}_2$ be the relation between $\mathbf{v}$ and inputs $\mathbf{x}$.

Lemma 4.3.1 **(compatible, global)** *Given the relations $\mathcal{L}_1(\mathbf{x},\mathbf{y},\mathbf{w})$ and $\mathcal{T}(\mathbf{u},\mathbf{y},\mathbf{w},\mathbf{z})$, the compatible global observability relation for n is given by*

$$\mathcal{O}^{cg}(\mathbf{x},\mathbf{u}) = C_{\mathbf{y},\mathbf{w}}(\overline{\mathcal{L}}_1(\mathbf{x},\mathbf{y},\mathbf{w}) + C_{\mathbf{z}}(\overline{\mathcal{T}}(\mathbf{u},\mathbf{y},\mathbf{w},\mathbf{z}) + \mathcal{O}(\mathbf{x},\mathbf{z}))).$$

In particular, if $\mathcal{L}(\mathbf{x},\mathbf{v},\mathbf{y},\mathbf{w}) = \mathcal{L}_1(\mathbf{x},\mathbf{y},\mathbf{w})\mathcal{L}_2(\mathbf{x},\mathbf{v})$,

$$\mathcal{O}^{cg}(\mathbf{x},\mathbf{u}) = C_{\mathbf{y},\mathbf{w},\mathbf{v},\mathbf{z}}(\overline{\mathcal{L}}(\mathbf{x},\mathbf{v},\mathbf{y},\mathbf{w}) + \overline{\mathcal{T}}(\mathbf{u},\mathbf{y},\mathbf{w},\mathbf{z}) + \mathcal{O}(\mathbf{x},\mathbf{z})).$$

Proof $\mathcal{O}'(\mathbf{x},\mathbf{u},\mathbf{y},\mathbf{w}) = C_{\mathbf{z}}(\overline{\mathcal{T}}(\mathbf{u},\mathbf{y},\mathbf{w},\mathbf{z}) + \mathcal{O}(\mathbf{x},\mathbf{z})))$ is the compatible observability relation for the network at the separating set $(\mathbf{u},\mathbf{y},\mathbf{w})$ (serial decomposition, Lemma 4.2.1). $C_{\mathbf{y},\mathbf{w}}(\overline{\mathcal{L}}_1 + \mathcal{O}')$ is the compatible parallel decomposition for node n (Lemma 4.2.5). If $\mathcal{L} = \mathcal{L}_1\mathcal{L}_2$, then $\mathcal{S}_{\mathbf{v}}\mathcal{L}_2(\mathbf{x},\mathbf{v}) = 1$ resulting in $C_{\mathbf{v}}\overline{\mathcal{L}}_2(\mathbf{x},\mathbf{v}) = 0$. The global observability relation of n can be written as

$$\begin{aligned}
\mathcal{O}^{cg}(\mathbf{x},\mathbf{u}) &= C_{\mathbf{y},\mathbf{w}}(\overline{\mathcal{L}}_1(\mathbf{x},\mathbf{y},\mathbf{w}) + C_{\mathbf{z}}(\overline{\mathcal{T}}(\mathbf{u},\mathbf{y},\mathbf{w},\mathbf{z}) + \mathcal{O}(\mathbf{x},\mathbf{z})))\\
&= C_{\mathbf{y},\mathbf{w}}(\overline{\mathcal{L}}_1(\mathbf{x},\mathbf{y},\mathbf{w}) + C_{\mathbf{v}}\overline{\mathcal{L}}_2(\mathbf{x},\mathbf{v}) + C_{\mathbf{z}}(\overline{\mathcal{T}}(\mathbf{u},\mathbf{y},\mathbf{w},\mathbf{z}) + \mathcal{O}(\mathbf{x},\mathbf{z})))\\
&= C_{\mathbf{v},\mathbf{y},\mathbf{w}}(\overline{\mathcal{L}}_1(\mathbf{x},\mathbf{y},\mathbf{w}) + \overline{\mathcal{L}}_2(\mathbf{x},\mathbf{v}) + C_{\mathbf{z}}(\overline{\mathcal{T}}(\mathbf{u},\mathbf{y},\mathbf{w},\mathbf{z}) + \mathcal{O}(\mathbf{x},\mathbf{z})))\\
&= C_{\mathbf{v},\mathbf{y},\mathbf{w}}(\overline{\mathcal{L}}(\mathbf{x},\mathbf{v},\mathbf{y},\mathbf{w}) + C_{\mathbf{z}}(\overline{\mathcal{T}}(\mathbf{u},\mathbf{y},\mathbf{w},\mathbf{z}) + \mathcal{O}(\mathbf{x},\mathbf{z})))\\
&= C_{\mathbf{v},\mathbf{y},\mathbf{w},\mathbf{z}}(\overline{\mathcal{L}}(\mathbf{x},\mathbf{v},\mathbf{y},\mathbf{w}) + \overline{\mathcal{T}}(\mathbf{u},\mathbf{y},\mathbf{w},\mathbf{z}) + \mathcal{O}(\mathbf{x},\mathbf{z})).
\end{aligned}$$

■

We can also compute $\mathcal{O}^{mg}(\mathbf{x},\mathbf{u})$ which is maximal with respect to $\mathcal{L}_1$, $\mathcal{T}$, and consistent with $\mathcal{O}$.

Lemma 4.3.2 **(maximal, global)** *Given the relations $\mathcal{L}_1(\mathbf{x},\mathbf{y},\mathbf{w})$ and $\mathcal{T}(\mathbf{u},\mathbf{y},\mathbf{w},\mathbf{z})$, the maximal global observability relation for n is given by*

$$\mathcal{O}^{mg}(\mathbf{x},\mathbf{u}) = \mathcal{S}_{\mathbf{y},\mathbf{w},\mathbf{z}}\mathcal{L}_1(\mathbf{x},\mathbf{y},\mathbf{w})\mathcal{O}(\mathbf{x},\mathbf{z})\mathcal{T}(\mathbf{u},\mathbf{y},\mathbf{w},\mathbf{z}).$$

In particular, if $\mathcal{L}(\mathbf{x},\mathbf{v},\mathbf{y},\mathbf{w}) = \mathcal{L}_1(\mathbf{x},\mathbf{y},\mathbf{w})\mathcal{L}_2(\mathbf{x},\mathbf{v})$ ($\mathcal{L}$ can be decomposed if it is an implementation), then

$$\mathcal{O}^{mg}(\mathbf{x},\mathbf{u}) = \mathcal{S}_{\mathbf{y},\mathbf{w},\mathbf{z}}\mathcal{L}(\mathbf{x},\mathbf{v},\mathbf{y},\mathbf{w})\mathcal{T}(\mathbf{u},\mathbf{y},\mathbf{w},\mathbf{z})\mathcal{O}(\mathbf{x},\mathbf{z}).$$

Proof $\mathcal{O}'(\mathbf{x},\mathbf{u},\mathbf{y},\mathbf{w}) = \mathcal{S}_{\mathbf{z}}\mathcal{T}(\mathbf{u},\mathbf{y},\mathbf{w},\mathbf{z})\mathcal{O}(\mathbf{x},\mathbf{z})$ is the maximal observability relation for the network at the separating set $(\mathbf{u},\mathbf{y},\mathbf{w})$ (serial decomposition, Lemma 4.2.2). $\mathcal{S}_{\mathbf{y},\mathbf{w}}(\mathcal{L}_1\mathcal{O}')$ is the maximal parallel decomposition for node n (Lemma 4.2.6). If $\mathcal{L} = \mathcal{L}_1\mathcal{L}_2$, then $\mathcal{S}_{\mathbf{v}}\mathcal{L}_2(\mathbf{x},\mathbf{v}) = 1$ because $\mathcal{L}_2$ must be well-defined. The global observability relation of n can be written as

$$\begin{aligned}
\mathcal{O}^{mg}(\mathbf{x},\mathbf{u}) &= \mathcal{S}_{\mathbf{y},\mathbf{w},\mathbf{z}}\mathcal{L}_1(\mathbf{x},\mathbf{y},\mathbf{w})\mathcal{T}(\mathbf{u},\mathbf{y},\mathbf{w},\mathbf{z})\mathcal{O}(\mathbf{x},\mathbf{z}) \\
&= \mathcal{S}_{\mathbf{y},\mathbf{w},\mathbf{z}}\mathcal{L}_1(\mathbf{x},\mathbf{y},\mathbf{w})\mathcal{S}_{\mathbf{v}}(\mathcal{L}_2(\mathbf{x},\mathbf{v}))\mathcal{T}(\mathbf{u},\mathbf{y},\mathbf{w},\mathbf{z})\mathcal{O}(\mathbf{x},\mathbf{z}) \\
&= \mathcal{S}_{\mathbf{v},\mathbf{y},\mathbf{w},\mathbf{z}}\mathcal{L}_1(\mathbf{x},\mathbf{y},\mathbf{w})\mathcal{L}_2(\mathbf{x},\mathbf{v})\mathcal{T}(\mathbf{u},\mathbf{y},\mathbf{w},\mathbf{z})\mathcal{O}(\mathbf{x},\mathbf{z}) \\
&= \mathcal{S}_{\mathbf{v},\mathbf{y},\mathbf{w},\mathbf{z}}\mathcal{L}(\mathbf{x},\mathbf{v},\mathbf{y},\mathbf{w})\mathcal{T}(\mathbf{u},\mathbf{y},\mathbf{w},\mathbf{z})\mathcal{O}(\mathbf{x},\mathbf{z}).
\end{aligned}$$

■

If $\mathcal{L}(\mathbf{x},\mathbf{v},\mathbf{y},\mathbf{w})$ and $\mathcal{T}(\mathbf{u},\mathbf{y},\mathbf{w},\mathbf{z})$ are implementations, $\mathcal{O}^{mg}(\mathbf{x},\mathbf{u}) = \mathcal{O}^{cg}(\mathbf{x},\mathbf{u})$ because of Lemmas 4.2.3 and 4.2.7.

Let $\mathcal{L}_3(\mathbf{x},\mathbf{v},\mathbf{y})$ be the relation between $\mathbf{v},\mathbf{y}$ and inputs $\mathbf{x}$ and $\mathcal{L}_4(\mathbf{x},\mathbf{w})$ be the relation between $\mathbf{w}$ and inputs $\mathbf{x}$.

Lemma 4.3.3 **(compatible, local)** *Given the relations $\mathcal{L}_3(\mathbf{x},\mathbf{v},\mathbf{y})$ and $\mathcal{T}(\mathbf{u},\mathbf{y},\mathbf{w},\mathbf{z})$, the compatible local observability relation for n is given by*

$$\mathcal{O}^{cl}(\mathbf{v},\mathbf{y},\mathbf{u}) = \mathcal{C}_{\mathbf{x}}(\mathcal{O}^{cg}(\mathbf{x},\mathbf{u}) + \overline{\mathcal{L}}_3(\mathbf{x},\mathbf{v},\mathbf{y}))$$

where $\mathcal{O}^{cg}(\mathbf{x},\mathbf{u})$ *is given by 4.3.1. In particular, if* $\mathcal{L}(\mathbf{x},\mathbf{v},\mathbf{y},\mathbf{w}) = \mathcal{L}_3(\mathbf{x},\mathbf{v},\mathbf{y})\mathcal{L}_4(\mathbf{x},\mathbf{w})$ *(*$S_{\mathbf{w}}\mathcal{L}_4(\mathbf{x},\mathbf{w}) = 1$ *if* $\mathcal{L}_4(\mathbf{x},\mathbf{w})$ *is well-defined),*

$$\mathcal{O}^{cl}(\mathbf{v},\mathbf{y},\mathbf{u}) \;=\; C_{\mathbf{x},\mathbf{w}}(\mathcal{O}^{cg}(\mathbf{x},\mathbf{u}) + \overline{\mathcal{L}}(\mathbf{x},\mathbf{v},\mathbf{y},\mathbf{w})).$$

Proof $\mathcal{O}^{cl}(\mathbf{v},\mathbf{y},\mathbf{u}) = C_{\mathbf{x}}(\mathcal{O}^{cg}(\mathbf{x},\mathbf{u}) + \overline{\mathcal{L}}_3(\mathbf{x},\mathbf{v},\mathbf{y}))$ is simply the compatible parallel decomposition made local by quantifying away $\mathbf{x}$. If $\mathcal{L} = \mathcal{L}_3\mathcal{L}_4$ and $C_{\mathbf{w}}\overline{\mathcal{L}}_4(\mathbf{x},\mathbf{w}) = 0$, then the local compatible observability relation for n is

$$\begin{aligned}
\mathcal{O}^{cl}(\mathbf{v},\mathbf{y},\mathbf{u}) &= C_{\mathbf{x}}(\mathcal{O}^{cg}(\mathbf{x},\mathbf{u}) + \overline{\mathcal{L}}_3(\mathbf{x},\mathbf{v},\mathbf{y})) \\
&= C_{\mathbf{x}}(\mathcal{O}^{cg}(\mathbf{x},\mathbf{u}) + \overline{\mathcal{L}}_3(\mathbf{x},\mathbf{v},\mathbf{y}) + C_{\mathbf{w}}\overline{\mathcal{L}}_4(\mathbf{x},\mathbf{w})) \\
&= C_{\mathbf{x},\mathbf{w}}(\mathcal{O}^{cg}(\mathbf{x},\mathbf{u}) + \overline{\mathcal{L}}_3(\mathbf{x},\mathbf{v},\mathbf{y}) + \overline{\mathcal{L}}_4(\mathbf{x},\mathbf{w})) \\
&= C_{\mathbf{x},\mathbf{w}}(\mathcal{O}^{cg}(\mathbf{x},\mathbf{u}) + \overline{\mathcal{L}}(\mathbf{x},\mathbf{v},\mathbf{y},\mathbf{w})).
\end{aligned}$$

■

As is the case in equation 4.2, we do not know of a way to express $\mathcal{O}^{ml}(\mathbf{v},\mathbf{y},\mathbf{u})$ as a Boolean relation. However, if both $\mathcal{T}(\mathbf{u},\mathbf{y},\mathbf{w},\mathbf{z})$ and $\mathcal{L}(\mathbf{x},\mathbf{v},\mathbf{y},\mathbf{w})$ are implementations, $\mathcal{O}^{cl}(\mathbf{v},\mathbf{y},\mathbf{u})$ is also maximal.

$\mathcal{O}^{cl}(\mathbf{v},\mathbf{y},\mathbf{u})$ is a Boolean relation and can be minimized as a two-level function using the program GYOCRO [83]. The result can then be used to replace the present implementation for n, $\mathbf{u} = F(\mathbf{y})$. Alternately, if n is a multi-level network itself, compatible external don't cares can be derived using (3.6), (3.7), or (3.8) for each output of n and *full_simplify* [70] can be used to optimize n.

4.3.1 Node Optimization Using Maximal Observability Relations

The algorithm in Figure 4.6 shows the computation of the maximal observability relation for each multi-output node n_i of a Boolean network and the simplification of the node using GYOCRO. We first order all the nodes in topological order from outputs. $\mathcal{O}_0(\mathbf{x},\mathbf{z})$ is the given specification of the circuit. The inputs of n_i are $\mathbf{v}_i$ and $\mathbf{y}_i$ and its outputs are $\mathbf{u}_i$. $\mathbf{w}_i$ are the rest of variables shared by the separating sets of variables before and after n_i. For each node n_i, the observability relation $\mathcal{O}_i(\mathbf{x},\mathbf{u}_i,\mathbf{y}_i,\mathbf{w}_i)$ is computed from the old relation $\mathcal{O}_{i-1}(\mathbf{x},\mathbf{u}_{i-1},\mathbf{y}_{i-1},\mathbf{w}_{i-1})$ and the new implementation of n_{i-1} after its optimization

$$\tilde{\mathcal{O}}_i(\mathbf{x},\mathbf{v}_{i-1},\mathbf{y}_{i-1},\mathbf{w}_{i-1}) \;=\; S_{\mathbf{u}_{i-1}}\mathcal{O}_{i-1}(\mathbf{x},\mathbf{u}_{i-1},\mathbf{y}_{i-1},\mathbf{w}_{i-1})(\mathbf{u}_{i-1} \equiv F_{i-1}(\mathbf{v}_{i-1},\mathbf{y}_{i-1}))$$

function node_simplify
/* multi-output node simplification using maximal observability relations */
begin
$\mathcal{O}_0(\mathbf{x}, \mathbf{z})$ = given specification for the circuit
NodeArray = nodes ordered in topological order from the outputs
for each node n_i in NodeArray in topological order **begin**
$\mathcal{O}_i(\mathbf{x}, \mathbf{u}_i, \mathbf{y}_i, \mathbf{w}_i) = \mathcal{S}_{\mathbf{u}_{i-1}} \mathcal{O}_{i-1}(\mathbf{x}, \mathbf{u}_{i-1}, \mathbf{y}_{i-1}, \mathbf{w}_{i-1})(\mathbf{u}_{i-1} \equiv F_{i-1}(\mathbf{v}_{i-1}, \mathbf{y}_{i-1}))$
$\mathcal{L}_i(\mathbf{x}, \mathbf{v}_i, \mathbf{y}_i, \mathbf{w}_i) = \prod_{k=1}^{|\mathbf{v}_i|}(g_{v_k} \overline{\oplus} v_k) \prod_{k=1}^{|\mathbf{y}_i|}(g_{y_k} \overline{\oplus} y_k) \prod_{k=1}^{|\mathbf{w}_i|}(g_{w_k} \overline{\oplus} w_k)$
$\mathcal{O}_i^{mg}(\mathbf{x}, \mathbf{u}_i) = \mathcal{S}_{\mathbf{y}_i, \mathbf{w}_i} \mathcal{L}_i(\mathbf{x}, \mathbf{v}_i, \mathbf{y}_i, \mathbf{w}_i) \mathcal{O}_i(\mathbf{x}, \mathbf{u}_i, \mathbf{y}_i, \mathbf{w}_i)$
$\mathcal{O}_i^{ml}(\mathbf{v}_i, \mathbf{y}_i, \mathbf{u}_i) = \mathcal{C}_{\mathbf{x}, \mathbf{w}_i}(\mathcal{O}_i^{mg}(\mathbf{x}, \mathbf{u}_i) + \overline{\mathcal{L}}_i(\mathbf{x}, \mathbf{v}_i, \mathbf{y}_i, \mathbf{w}_i))$
Optimize n_i with the relation $\mathcal{O}_i^{ml}$ using GYOCRO
end
end

Figure 4.6: Maximal Observability Relation Computation and Node Simplification

function node_simplify
/* multi-output node simplification using compatible observability relations */
begin
$\mathcal{O}_0(\mathbf{x}, \mathbf{z})$ = given specification for the circuit
NodeArray = nodes ordered in topological order from the outputs
for each node n_i in NodeArray in topological order **begin**

$$\mathcal{O}_i(\mathbf{x}, \mathbf{u}_i, \mathbf{y}_i, \mathbf{w}_i) = C_{\mathbf{u}_{i-1}}(\mathcal{O}_{i-1}(\mathbf{x}, \mathbf{u}_{i-1}, \mathbf{y}_{i-1}, \mathbf{w}_{i-1}) + \overline{\mathcal{O}}^{cl}_{i-1}(\mathbf{v}_{i-1}, \mathbf{y}_{i-1}, \mathbf{u}_{i-1}))$$

$$\mathcal{L}_i(\mathbf{x}, \mathbf{v}_i, \mathbf{y}_i, \mathbf{w}_i) = \prod_{k=1}^{|\mathbf{v}_i|}(g_{v_k} \overline{\oplus} v_k) \prod_{k=1}^{|\mathbf{y}_i|}(g_{y_k} \overline{\oplus} y_k) \prod_{k=1}^{|\mathbf{w}_i|}(g_{w_k} \overline{\oplus} w_k)$$

$$\mathcal{O}^{cg}_i(\mathbf{x}, \mathbf{u}_i) = C_{\mathbf{y}_i, \mathbf{w}_i}(\overline{\mathcal{L}}_i(\mathbf{x}, \mathbf{v}_i, \mathbf{y}_i, \mathbf{w}_i) + \mathcal{O}_i(\mathbf{x}, \mathbf{u}_i, \mathbf{y}_i, \mathbf{w}_i))$$

$$\mathcal{O}^{cl}_i(\mathbf{v}_i, \mathbf{y}_i, \mathbf{u}_i) = C_{\mathbf{x}, \mathbf{w}_i}(\mathcal{O}^{cg}_i(\mathbf{x}, \mathbf{u}_i) + \overline{\mathcal{L}}_i(\mathbf{x}, \mathbf{v}_i, \mathbf{y}_i, \mathbf{w}_i))$$

end
Nodes can be optimized using GYOCRO independent of each other
end

Figure 4.7: Compatible Observability Relation Computation and Node Simplification

$$\mathcal{O}_i(\mathbf{x}, \mathbf{u}_i, \mathbf{y}_i, \mathbf{w}_i) \;=\; \tilde{\mathcal{O}}_i(\mathbf{x}, \mathbf{v}_{i-1}, \mathbf{y}_{i-1}, \mathbf{w}_{i-1}).$$

The variables $\mathbf{v}_{i-1}, \mathbf{y}_{i-1}, \mathbf{w}_{i-1}$ are regrouped to form $\mathbf{u}_i, \mathbf{y}_i, \mathbf{w}_i$ (the same as $\mathbf{v}, \mathbf{y}, \mathbf{w}$ and $\mathbf{u}', \mathbf{y}', \mathbf{w}'$ in Figure 4.5). $\mathcal{L}_i(\mathbf{x}, \mathbf{v}_i, \mathbf{y}_i, \mathbf{w}_i)$ is computed using the global functions $\{g_{v_k}, g_{w_k}, g_{y_k}$ at each of the inputs $\mathbf{v}, \mathbf{y}$ of n_i as well as those corresponding to $\mathbf{w}_i$. $\mathcal{O}^{mg}_i(\mathbf{x}, \mathbf{u}_i)$ and $\mathcal{O}^{ml}_i(\mathbf{v}_i, \mathbf{y}_i, \mathbf{u}_i)$ are then computed and used to improve the current implementation at n_i. Lemma 4.3.3 is used to compute $\mathcal{O}^{ml}_i(\mathbf{v}_i, \mathbf{y}_i, \mathbf{u}_i)$ from $\mathcal{O}^{mg}_i(\mathbf{x}, \mathbf{u}_i)$. $\mathcal{O}^{ml}_i(\mathbf{v}_i, \mathbf{y}_i, \mathbf{u}_i)$ is maximal because $\mathcal{L}_i(\mathbf{x}, \mathbf{v}_i, \mathbf{y}_i, \mathbf{w}_i)$ is an implemetation.

4.3.2 Node Optimization Using Compatible Observability Relations

The algorithm in Figure 4.7 shows the computation of compatible observability relations for multi-output nodes of a Boolean network and the simplification of the nodes using GYOCRO. The nodes are ordered topologically from outputs as before. $\mathcal{O}_0(\mathbf{x}, \mathbf{z})$ is the given specification of the circuit. For each node n_i, the observability relation $\mathcal{O}_i(\mathbf{x}, \mathbf{u}_i, \mathbf{y}_i, \mathbf{w}_i)$ is computed from the old relation $\mathcal{O}_{i-1}(\mathbf{x}, \mathbf{u}_{i-1}, \mathbf{y}_{i-1}, \mathbf{w}_{i-1})$ and $\mathcal{O}^{cl}_{i-1}(\mathbf{v}_{i-1}, \mathbf{y}_{i-1}, \mathbf{u}_{i-1})$. This is then used to find $\mathcal{O}^{cg}_i(\mathbf{x}, \mathbf{u}_i)$ and $\mathcal{O}^{cl}_i(\mathbf{v}_i, \mathbf{y}_i, \mathbf{u}_i)$ as shown. Each node n_i can be optimized

independent of the others using its observability relation because the observability relations are compatible.

4.4 Conclusion

In this chapter, we have expanded the theory of don't cares which is for single-output nodes to a theory that can be applied to multi-output nodes. The flexibility at each node is represented by a Boolean relation. Techniques are provided for computing both maximal and compatible observability relations for multi-output nodes of a Boolean network as is the case with don't cares for single-output nodes. The maximal observability relation for a multi-output node of a combinational circuit is the maximum flexibility for manipulating that node. The compatible observability relations for a set of nodes ordered topologically allows optimization of each such node independent of the optimization done at other nodes in the set. The practicality of these techniques for optimizing large circuits depends on how efficiently one can represent and manipulate observability relations in BDD or any other form. These techniques are currently being investigated.

Chapter 5

Node Simplification: Practical Issues

We present an algorithm for computing local don't cares at each intermediate node of a Boolean network based on image computation techniques. The local don't care set for each node, expressed in terms of immediate fanins of that node, is a combination of satisfiability don't cares, compatible or maximal observability don't cares, and external don't cares. These don't cares can be directly used for the simplification of each node by a two-level minimizer. The simplification is very fast and the optimized circuits are 100 percent testable in most cases. This is a practical method and much more powerful than previous methods developed for node simplification because it computes almost the full local don't care set at each node using the image computation techniques developed by Coudert et al [21]. The image computation technique allows us to use the external don't cares very effectively. Furthermore, there is no restriction on how the external don't cares are represented, because BDD's corresponding to external don't cares are built for local don't care computation.

5.1 Introduction

The objectives of multi-level logic synthesis are to find networks which are optimum with respect to area, delay, and/or testability of the circuit. The synthesis process is usually divided into a technology dependent, and a technology independent part [10, 7]. In the technology independent part, one tries to simplify the logic equations representing

the Boolean network as much as possible. A common cost function used at this stage is the literal count of the Boolean network in factored form. Experiments show that this cost function correlates well with the final area of the mapped circuits when standard cell libraries are used for the mapping. Thus, transformations are applied on a Boolean network to find a representation with the least number of literals in factored form. The minimal area network is also used as the starting point for delay oriented optimization. Additional transformations are later applied to improve the performance of the circuit.

One important transformation in the technology independent stage is to apply two-level logic minimizers on nodes of the multi-level network to optimize the two-level function associated with each single node of the network. The input to the two-level minimizer is composed of an onset cover and a don't care set. The onset cover is the function at the node in terms of its fanin variables. The don't care set at each node may contain information about the structure of the network and is a combination of external, observability, and satisfiability don't cares. A don't care set of appropriate size for two-level minimizers must be computed.

External don't cares are conditions under which the value of the outputs are not important and are very effective in the simplification of multi-level networks. However, problems arise when external don't cares are used for node simplification along with two-level minimizers. The external don't cares must be represented in a way that is suitable for two-level minimizers.

Originally, external don't cares were not supported in MIS-II. A recent version represents the external don't cares by a separate multi-level network which has the same set of primary inputs as the original network. Corresponding to each primary output in the care network is a primary output in the don't care network representing the external don't care for that output; whenever a don't care output is turned on by a primary input minterm x, the x is a don't care input for the corresponding output.

We describe a new algorithm for computing don't cares in the local space (space of fanin variables) of each intermediate node. This allows the effective use of the external don't cares in the node minimization process. The local don't cares are represented in terms of the immediate fanin variables of each intermediate node, and are a combination of satisfiability, observability and external don't cares. This new technique is faster than the one introduced in [67], produces significantly superior results, and can be applied to a wide range of circuits. The key operations are the computation of compatible observability plus

external don't cares in BDD form and the effective use of the image computation techniques to find the local don't cares at each node.

5.2 Node Simplification

In systems, like MIS [10], which use an algorithmic approach to multi-level logic synthesis, a two-level minimizer, such as ESPRESSO [11], or a modified version of it [50] is used to simplify the nodes of a multi-level network. The objective of a general two-level logic minimizer is to find a logic representation with a minimal number of implicants and literals while preserving the functionality. There are two general approaches. One is based on the offset of the logic function and the other uses tautology. Logic minimizers, such as ESPRESSO or MINI [38], generate the offset to determine whether a given cube is an implicant and to obtain a global view of the expansion to prime process. The input usually contains a cover for the onset and a cover for the don't care set. A cover for the offset is generated from the input using either a complement algorithm based on the Unate Recursive Paradigm [11] or the Disjoint Sharp Process [38]. The number of cubes in the offset can grow exponentially with the number of input variables; hence the offset generation could be quite time consuming. The other approach to this problem is to use tautology. Literals in a cube are raised individually and tautology is used to determine if the new cube is covered by the union of the onset and the don't care set. The major disadvantage of this approach is that there is no global picture for ordering the literals to be raised and hence this approach can give results that are sub-optimal. This approach is usually slower.

Functions with many cubes in the offset and don't care set happen quite often at the nodes of a multi-level logic network. ESPRESSO can easily run out of memory while applying the Unate Recursive Paradigm to generate the offset. Other aspects of this environment are that the initial cover is usually small, and both the initial cover and the don't care cover mainly consist of primes.

Example:

If no don't cares are used, the input to a two-level minimizer to simplify node y_2 shown in Figure 5.1 is

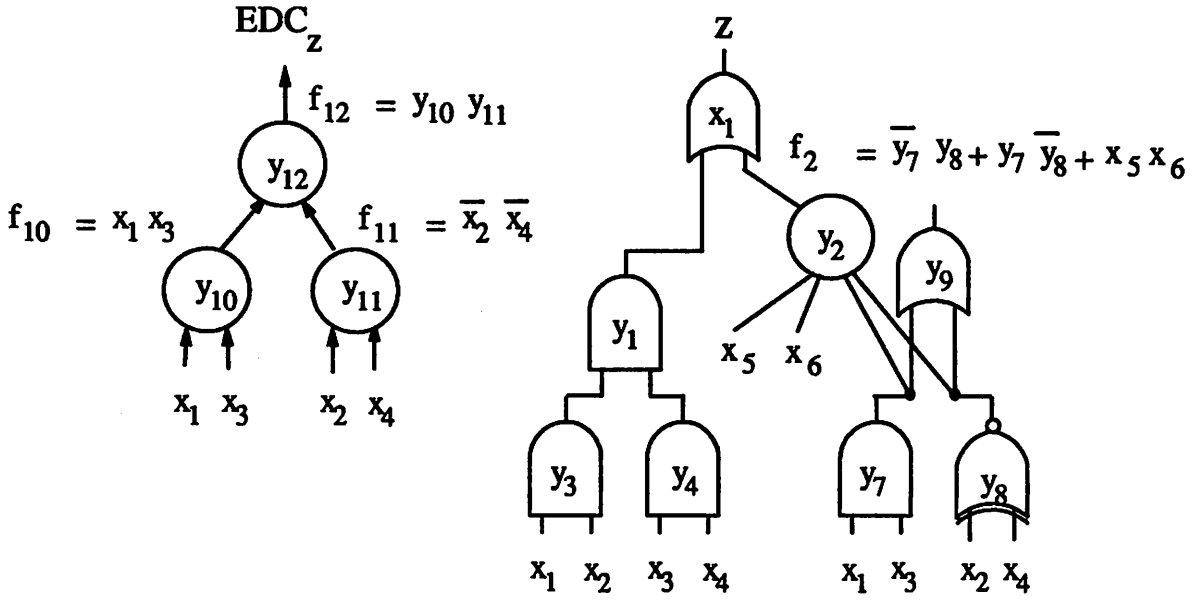

Figure 5.1: Node Simplification

x_5	x_6	y_7	y_8	y_2
2	2	0	1	1
2	2	1	0	1
1	1	2	2	1.

Each row with numbers $0, 1, 2$ in Figure 5.1 represents a cube. A 0 in a column shows a variable in negative form; a 1 shows a variable in positive form; a 2 shows that variable is missing in the cube. If satisfiability don't cares for nodes $y_1, y_3, y_4, y_7, y_8, y_9$ are also generated, the input cover with the above order for SDC's is as shown in Figure 5.2.

As it is clear from this example the size of don't care set grows rapidly as SDC's for more nodes are generated and the offset generation in ESPRESSO becomes impractical. To avoid such problems, new two-level minimization techniques based on reduced offsets were proposed [50, 69]. The reduced offset for a cube is never larger than the entire offset of the function and in practice has been found to be much smaller. The reduced offset can be used in the same way as the full offset for the expansion of a cube and no quality is lost.

The use of reduced offset speeds up the node simplification for nodes in a multi-level network. However, if the size of don't care set is too large, the computation of reduced

x_1	x_2	x_3	x_4	y_1	y_3	y_4	x_5	x_6	y_7	y_8	y_9	y_2
2	2	2	2	2	2	2	2	2	0	1	2	1
2	2	2	2	2	2	2	2	2	1	0	2	1
2	2	2	2	2	2	2	1	1	2	2	2	1
2	2	2	2	0	1	1	2	2	2	2	2	2
2	2	2	2	1	0	2	2	2	2	2	2	2
2	2	2	2	1	2	0	2	2	2	2	2	2
1	1	2	2	2	0	2	2	2	2	2	2	2
0	2	2	2	2	1	2	2	2	2	2	2	2
2	0	2	2	2	1	2	2	2	2	2	2	2
2	2	1	1	2	2	0	2	2	2	2	2	2
2	2	0	2	2	2	1	2	2	2	2	2	2
2	2	2	0	2	2	1	2	2	2	2	2	2
1	2	1	2	2	2	2	2	2	0	2	2	2
2	2	0	2	2	2	1	2	2	1	2	2	2
0	2	2	2	2	2	1	2	2	1	2	2	2
2	1	2	0	2	2	2	2	2	2	0	2	2
2	0	2	1	2	2	2	2	2	2	0	2	2
2	1	2	1	2	2	2	2	2	2	1	2	2
2	1	2	1	2	2	2	2	2	2	1	2	2
2	2	2	2	2	2	2	2	2	2	1	0	2
2	2	2	2	2	2	2	2	2	1	2	0	2
2	2	2	2	2	2	2	2	2	0	0	1	2

Figure 5.2: Input to Two-Level Minimizer

offsets is not possible either; therefore, filters must be introduced to keep the don't care size reasonably small. The filters are chosen heuristically hoping that the quality of the node simplification is not reduced significantly.

5.3 Using Don't Cares

While doing node simplification, a two-level minimizer is applied on each of the nodes of a multi-level network. The structure of the Boolean network is captured by don't cares. In MIS-II [10], a subset of satisfiability don't cares is used for the simplification of each node. This subset known as the support subset [65] is the satisfiability don't care set of all the nodes whose local function is dependent on a subset of the variables in the local function of the node being simplified. By using support subset we can effectively cause a Boolean substitution of the nodes of the network into the node being simplified, but in general we do not get maximum simplification of the node. As an example of the subset filter, while simplifying node y_2 shown in Figure 5.1 the SDC for node y_9 is generated because the support of y_9 is a subset of the support of y_2. Thus substitution of node y_9 in y_2 will happen if such possibility exists and it results in a simpler function at y_2.

Observability don't cares are computed in terms of intermediate variables in the network. The most general technique for expressing external don't cares is to represent them with another network with the same primary inputs and one output for each output in the care network. To fully utilize ODC's plus EDC's for the simplification of each intermediate node one has to find how the current representation of the node is related to these don't cares. The relation between EDC's plus ODC's and the current representation at each node is usually only through primary inputs. To get the most simplification possible for each node, one has to provide this connection, which is the structure of the Boolean network, to the two-level minimizer.

The most straightforward approach is to establish this connection through SDC's. SDC's are generated for all the nodes in the transitive fanin cone of the node being simplified to relate the current representation of the node to the primary inputs. SDC's are also generated to relate EDC plus ODC to primary inputs. These are all the nodes in the transitive fanin cone of the support of EDC plus ODC.

Example:

The EDC plus ODC for y_2 in Figure 5.1 is $d_2^m = y_1 + y_{12}$. SDC's for nodes y_7 and y_8 relate y_2

to primary inputs. SDC's for nodes $y_1, y_3, y_4, y_{10}, y_{11}$, and y_{12} relate d_2^m to primary inputs. We also generate SDC of y_9 because it may be substituted in the representation of y_2. The input to the two-level minimizer has 15 input variables and 33 cubes for this very small example. After node simplification, the representation at y_2 becomes $f_2 = y_9 + x_5x_6$.

It is obvious that this approach is not practical for networks with many levels; the size of satisfiability don't care set grows very large in such cases and node simplification becomes impossible.

The command *cspf_simplify* in the most recent release of MIS-II computes the external and observability don't cares for each node using the techniques in [67]. The external don't cares are only allowed in two-level form expressed directly in terms of primary inputs. CODC's are computed for the simplification of each node. These CODC's are in terms of intermediate variables in the network. A collapsing and filtering procedure is used to find a subset of CODC which is in the transitive fanin cone of the node being simplified. A limited SDC is generated to use CODC plus EDC in two-level form. EDC's cannot be represented in two-level form in many cases because the number of cubes in the sum-of-products representation of EDC's grows very large. Also, because of collapsing and filtering and the limited SDC generated, the quality is reduced considerably compared to what is possible.

Example:

The EDC plus ODC for y_2 in Figure 5.1 in terms of primary inputs is $d_2^{mg} = x_1x_2x_3x_4 + x_1\overline{x}_2x_3\overline{x}_4$. SDC's for nodes y_7, y_8 and y_9 must be generated. The SDC's for nodes y_7 and y_8 relate the EDC plus ODC of y_2 to the current representation at that node. The SDC for y_9 allows the substitution of y_9 in y_2. The input to the two-level minimizer is as shown in Figure 5.3. After node simplification, the representation at y_2 becomes as before $f_2 = y_9 + x_5x_6$.

At each intermediate node, there is a local function $f_i : B^r \longrightarrow B$ which is the function of the node in terms of its immediate fanins. Ideally, one would like to express the external plus observability don't cares of each node in terms of its immediate fanins not primary inputs. This reduces the number of variables in the input given to the two-level minimizer considerably. The local don't cares for y_i are minterms $m_l \in B^r$ for which the value of f_i can be either 1 or 0 and this change does not affect the behavior of the Boolean network. The local don't cares for y_i are related to the EDC, ODC of y_i and SDC's of the network and are as effective in node simplification as the full don't care set. They can be

x_1	x_2	x_3	x_4	x_5	x_6	y_7	y_8	y_9	y_2
2	2	2	2	2	2	0	1	2	1
2	2	2	2	2	2	1	0	2	1
2	2	2	2	1	1	2	2	2	1
1	1	1	1	2	2	2	2	2	2
1	0	1	0	2	2	2	2	2	2
1	2	1	2	2	2	0	2	2	2
0	2	2	2	2	2	1	2	2	2
2	2	0	2	2	2	1	2	2	2
2	1	2	1	2	2	2	0	2	2
2	0	2	0	2	2	2	0	2	2
2	1	2	0	2	2	2	1	2	2
2	0	2	1	2	2	2	1	2	2
2	2	2	2	2	2	1	2	0	2
2	2	2	2	2	2	2	1	0	2
2	2	2	2	2	2	0	0	1	2

Figure 5.3: Input to Two-Level Minimizer

used to remove all the redundancies within a node.

Example:

The local don't care for y_2 is $d_2^{lm} = y_7 y_8$. By examining the subset support, we determine that the SDC of y_9 should be included. The input to the two-level minimizer is

x_5	x_6	y_7	y_8	y_9	y_2
2	2	0	1	2	1
2	2	1	0	2	1
1	1	2	2	2	1
2	2	1	1	2	2
2	2	1	2	0	2
2	2	2	1	0	2
2	2	0	0	1	2

After node simplification, the representation at y_2 becomes as before $f_2 = y_9 + x_5 x_6$.

5.4 Computing Local Don't Cares

We describe a new method for using various kinds of don't cares, i.e. satisfiability don't cares, observability don't cares, and external don't cares, to optimize a multi-level network. At each intermediate node, we find local don't cares in terms of fanins of the node being simplified.

Let y_o be the node being simplified and $f_o : B^r \longrightarrow B$ be the local function at this node in terms of its fanins $y_1, \ldots, y_r$. The local don't care set d_o^l is all the points in B^r for which the value of f_o is not important.

$d_o^l = \prod_{i=1}^{m} d_{o;i}^l$, where $d_{o;i}^l$ is the don't care with respect to primary output z_i in the transitive fanout of node y_o [1]. A minterm $m^l \in d_{o;i}^l$ if either **a)** there is no primary input combination $m^g \in B^n$ of the Boolean network that generates m^l or **b)** all such primary input combinations are in the observability plus external don't care set of the node y_o with respect to z_i ($d_{o;i}^g$).

[1] We include here the product over all outputs even if z_i is not in the transitive fanout cone of y_o, since then $d_{o;i}^l = 1$.

To find d_o^l, we first find the observability plus external don't care set, d_o^g, in terms of primary inputs. Notice that $d_o^g = \prod d_{o;i}^g$. The care set of y_o in terms of primary inputs is $\overline{d_o^g}$. The local care set $\overline{d_o^l}$ is computed by finding all combinations in B^r reachable from $\overline{d_o^g}$. Any combination in B^r that is not reachable from $\overline{d_o^g}$ is in the local don't care set d_o^l.

Theorem 5.4.1 *The procedure outlined above finds all the local don't cares of y_o.*

Proof Assume there is a don't care minterm $m^l \in B^r$ such that $m^l \notin d_o^l$. Clearly, there must be some input minterms $m_1^g, m_2^g, \ldots$ that generate m^l. All such minterms must be in the observability plus external don't care set computed for node y_o; otherwise the local function at node y_o must have a specific value for m^l. Thus $m^l \in d_o^l$ by construction, contradicting $m^l \notin d_o^l$. ∎

Theorem 5.4.2 *If the two-level function associated with each node y_o in the network is prime and irredundant (with respect to d_o^l), then every connection in the Boolean network is single stuck-at-fault testable.*

Proof Each node has a sum-of-products representation. We consider two kinds of faults in particular. First assume the input y_i to some AND term c can be set to 1. This implies the corresponding cube $\tilde{c}$, with y_i replaced with $\overline{y}_i$ is a local don't care and by Theorem 5.4.1 is in d_o^l. Thus c was not a prime. The second kind of fault is the input to some OR term stuck-at-0. This implies that the associated cube is redundant. But this is not possible, because the two-level representation at each node is prime and irredundant with d_o^l as the don't care set and d_o^l contains all the local don't cares. A stuck-at-0 at an AND gate is equivalent to a stuck-at-0 of the OR gate and a stuck-at-1 at an OR gate is equivalent to the node function being 1; thus none of the cubes are prime. ∎

The above theorem implies that repeated simplification of the nodes in order, until no change occurs, using local don't cares leads to a network that is 100% testable.

5.5 Implementation

In practice, it is computationally expensive to compute complete observability don't cares for each of the nodes of the network. Instead, we use compatible observability don't cares. As discussed in Chapter 3, subsets of observability don't cares are compatible if

the function at each node can be changed (as allowed by its observability don't care subset) independent of allowable changes in the functions at other nodes in the network. These compatible subsets can be computed for all the nodes by traversing the Boolean network once. By using compatible observability don't cares we cannot guarantee 100% testability although experimental results show in most cases the optimized networks are 100% testable.

The computation of observability plus external don't cares and the image computation to find reachable points in the local space of each node are done using BDD's. We used the BDD package in SIS [74] which is implemented based on the techniques in [9]. First we find BDD's at each of the nodes of the Boolean network in terms of the primary inputs. The size of the generated BDD's is dependent on the ordering of the input variables in the network. We applied the ordering given in [51]. BDD's are also built for each of the primary outputs in the external don't care network using this same ordering.

The algorithm for node simplification using local don't cares is shown in Figure 5.4. The computation starts from the primary outputs and proceeds towards the primary inputs. First we order all the nodes in the network in topological order from outputs. This ordering is done in depth first manner. The compatible observability don't care set at each primary output is initialized to the external don't care set at that output if the external don't care set exists, otherwise, it is set to 0.

The intermediate nodes are processed one by one in the chosen topological order. We find the compatible global observability plus external don't care set at each intermediate node. This computation is based on equations (3.7) and (3.10). Some filtering is added to speedup the computation. The complement of the global don't care set $\overline{d}_i^{cg}$ computed for a node is used to find the local don't cares. Any vertex in the local space of the node being simplified (y_i) which cannot be generated under any input combination in $\overline{d}_i^{cg}$ is in the local don't care set of y_i. In what follows, we explain the techniques applied in more detail.

5.5.1 External Don't Cares

The external don't care set is represented by a separate multi-output network $\mathcal{N}_{exdc}$. $\mathcal{N}_{exdc}$ has the same number of inputs and outputs as the care network $\mathcal{N}$. Primary inputs are exactly the same as the care network. The function at each primary output of $\mathcal{N}_{exdc}$ represents the external don't care set for the corresponding output in the care network.

```
function full_simplify
begin
   for each primary output z_i begin
      d_i^gc = external don't care for z_i
   end
   NodeArray = nodes ordered in topological order from the outputs
   for each node y_i in NodeArray in topological order begin
      /* find a compatible don't care set for y_i */
      d_i^gc = get_compatible_dc(y_i)
      /* find the local don't care set by range computation */
      Let (g_{k_1}, g_{k_2}, ..., g_{k_r}) be global functions at fanins of y_i
      d_i^l = range([g_{k_1}, g_{k_2}, ..., g_{k_r}]_{d_i^gc})   /*gives d_i^l in sum-of-products form*/
      SDC_i = SDC's of substitutable nodes in y_i
      simplify node y_i using (SDC_i + d_i^l)
   end
end
```

Figure 5.4: don't care computation and node simplification

```
.model  Example
.inputs  x1  x2  x3  x4  x5  x6
.outputs  z
.names  y1  y2      z
        1   2       1
        2   1       1
.names  y3  y4      y1
        1   1       1
.names  x5  x6  y7  y8      y3
        1   1   2   2       1
        2   2   1   0       1
        2   2   0   1       1
.names  x1  x2      y3
        1   1       1
.names  x3  x4      y4
        1   1       1
.names  x1  x3      y7
        1   1       1
.names  x2  x4      y8
        1   0       1
        0   1       1
.names  y7  y8      y9
        1   2       1
        2   1       1
.exdc
.model  Example – DC
.inputs  x1  x2  x3  x4  x5  x6
.outputs  z
.names  y10  y11     z
        1    1      1
.names  x1  x3      y10
        1   1       1
.names  x2  x4      y11
        0   0       1
.end
```

Figure 5.5: Berkeley Logic Interchange Format

A new construct called *.exdc* is added to Berkeley Logic Interchange Format (BLIF) to describe external don't cares. The description of $\mathcal{N}_{exdc}$ using exactly the same format as the care network, comes after *.exdc*. An example is shown in Figure 5.5 which is the same network and don't care network of Figure 5.1.

5.5.2 Inverse of Boolean Difference

While computing ODC's or CODC's, the expression $\overline{\frac{\partial f}{\partial y_i}} = f_{y_i} f_{\overline{y}_i} + \overline{f}_{y_i} \overline{f_{\overline{y}_i}}$ is computed repeatedly. The direct computation of this function in sum-of-products form is inefficient because one has to compute the complement of f_{y_i} and $f_{\overline{y}_i}$ and then perform the necessary AND and OR operations. Let $f = py_i + q\overline{y}_i + r$ be the function in sum-of-products form and y_i be the variable with respect to which the Boolean difference is computed. Then

$$\begin{aligned} \overline{\frac{\partial f}{\partial y_i}} &= (p+r)(q+r) + \overline{p}\,\overline{q}\,\overline{r} \\ &= pq + \overline{p}\,\overline{q} + r. \end{aligned}$$

It is faster to compute the complement of p and q in sum-of-products form because they contain less cubes than f_{y_i} and $f_{\overline{y}_i}$. The AND operations pq and $\overline{p}\,\overline{q}$ can also be computed much faster. This formulation was suggested by Adnan Aziz [5].

5.5.3 Computing Observability and External Don't Cares at Each Node

The compatible observability plus external don't care set at node y_i is found by using the compatible observability don't care set for each fanout edge (i,k) of y_i (see Figure 5.7). The compatible observability don't care set for y_i is then obtained by intersecting the observability don't care subsets computed for its fanout edges.

The algorithm in Figure 5.6 shows how a compatible don't care set is computed for a node. The compatibility operations for the computation of observability don't care subsets at each fanout edge are done in sum-of-products form because intermediate variables are needed for such operations [2]. Furthermore, it is much easier to filter out unwanted cubes if the representation is in sum-of-products form. All other operations are done in BDD form. To distinguish the operations in sum-of-products form from the rest we added the comment */* cube */* at the end of each such operation in Figure 5.6. The computation of CODC

[2]It would be possible to do this computation in BDD form, but this would require finding an ordering for both primary inputs and intermediate variables.

function get_compatible_dc(y_i)
/* find Compatible don't care for each node */
begin
  /* a topological ordering $\succ$ is given for all the nodes in the network*/
  FanoutList= A list of fanouts of y_i
  $d_i^{cg} = 1$
  for each node y_k with function f_k in FanoutList **begin**
    $D = filter(\overline{\frac{\partial f_k}{\partial y_i}})$ /* cubes */
    Let $(y_{j_1}, \ldots, y_{j_p})$ be fanins of y_k such that $y_{j_l} \succ y_i$
    for each fanin y_{j_l} of y_k such that $y_{j_l} \succ y_i$ **begin**
      /* replace all the variables $y_{j_1}, \ldots y_{j_{l-1}} \succ y_{j_l}$ in $\frac{\partial f_k}{\partial y_{j_l}}$
      with their local functions to get E_{j_l} */
      $E_{j_l} = \frac{\partial f_k}{\partial y_{j_l}}|_{y_{j_1}=f_{j_1},\ldots,y_{j_l}=f_{j_l}}$ /* cubes */
    end
    $D^c = (E_{j_1} + C_{y_{j_1}}) \ldots (E_{j_p} + C_{y_{j_p}})D$ /* cubes */
    D^{cg} = transform D^c into BDD form in terms of primary inputs
    $d_{ik}^{cg} = D^{cg} + d_k^{cg}$
    $d_i^{cg} = d_i^{cg} d_{ik}^{cg}$
  end
  return d_i^{cg}
end

Figure 5.6: don't care computation and node simplification

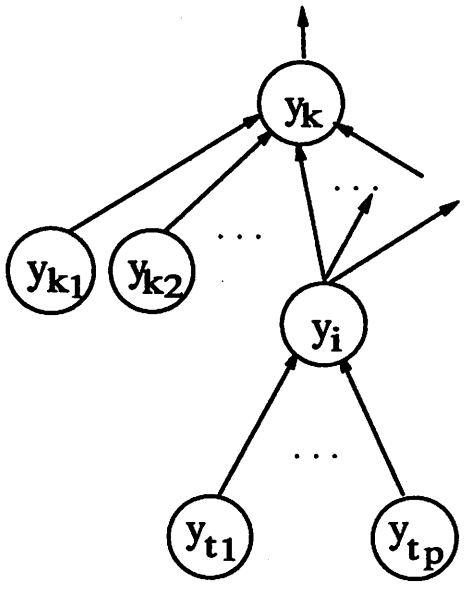

Figure 5.7: An Intermediate Node

for an edge can be done more efficiently as given by Lemma 3.5.5, if a small number of intermediate variables are replaced with their local functions. When $\frac{\partial f_k}{\partial y_{j_l}}$ is computed we replace all the fanins of y_k which have higher order than y_{j_l} with their local functions in $\frac{\partial f_k}{\partial y_{j_l}}$ to get E_{j_l} (as shown in Figure 5.6). These (E_{j_l}'s) are then used to compute D^c which is a part of CODC for the edge (i,k). Once the D^c is computed in sum-of-products form in terms of intermediate and primary input variables, we substitute each variable with the global BDD corresponding to that variable to get D^c in BDD form in terms of only primary inputs. The don't care set computed for node y_k, d_k^{cg}, which is kept in BDD form is then added to D^c to get d_{ik}^{cg}. The CODC's of the fanout edges are ANDed together one by one to get the CODC of the node which is later used for image computation.

5.5.4 Filtering

Let $y_{t_1}, \ldots, y_{t_p}$ be the fanins of y_i and d_i^{cg} the global don't care at y_i as shown in Figure 5.7. The local don't care set for y_i is

$$\begin{aligned} d_i^l(y_{t_1}, \ldots, y_{t_p}) &= \overline{S_{\mathbf{x}} \overline{d_i^{cg}}(\mathbf{x})(g_{t_1}(\mathbf{x}) \overline{\oplus} y_{t_1}) \ldots (g_{t_p}(\mathbf{x}) \overline{\oplus} y_{t_p})} \\ &= C_{\mathbf{x}}(d_i^{cg}(\mathbf{x}) + g_{t_1}(\mathbf{x}) \oplus y_{t_1} + \ldots + g_{t_p}(\mathbf{x}) \oplus y_{t_p}) \end{aligned}$$

In general, d_i^{cg} can depend on primary input variables outside the transitive fanin cone of y_i as well as those in the cone. Let $\mathbf{x}''$ be all the primary inputs outside the cone and $\mathbf{x}'$ the rest of primary inputs. No global function g_{t_i} is dependent on x'', therefore,

$$d_i^l(y_{t_1},\ldots,y_{t_p}) = C_{\mathbf{x}'}(C_{\mathbf{x}''}d_i^{cg}(\mathbf{x}) + g_{t_1}(\mathbf{x}) \oplus y_{t_1} + \ldots + g_{t_p}(\mathbf{x}) \oplus y_{t_p}).$$

The dependency on $\mathbf{x}''$ can be removed by computing $C_{\mathbf{x}''}d_i^{cg}$ first and then doing the image computation.

When we compute d_i^{cg} as shown in Figure 5.6 a heuristic filtering step is introduced which removes cubes that are dependent on variables x''. For each node y_i we generate a list of nodes, *FoutInList*, which is the transitive fanouts of the transitive fanins of y_i. Notice that the global function corresponding to a node y_r not in *FoutInList* is completely dependent on $\mathbf{x}''$; otherwise, y_r is a transitive fanout of some node in $\mathbf{x}'$ which is a transitive fanin of y_i and therefore y_r is in *FoutInList*. The filter in Figure 5.6 removes cubes which have literals corresponding to nodes not in *FoutInList*. If global functions for these cubes are computed, they result in new cubes $c_1,\ldots,c_n$ in terms of primary inputs such that each cube has some variables from $\mathbf{x}''$. Let $c_{n+1},\ldots,c_p$ be all other cubes in d_i^g dependent only on $\mathbf{x}'$, then

$$C_{\mathbf{x}''}d_i^{cg} = C_{\mathbf{x}''}(c_1 + \ldots + c_n) + c_{n+1} + \ldots c_p.$$

$C_{\mathbf{x}''}(c_1 + \ldots + c_n) = 0$ in most cases because there is usually some minterm m in terms of variables in $\mathbf{x}''$ for which $(c_1 + \ldots + c_n)_m = 0$. Our experiments support the effectiveness of this filtering.

Example:

Let $y_p = x_1 + x_2$ be a variable which is not in *FoutInList* of y_i and $c = y_p y_{t_i} \ldots y_{t_j}$ a cube in d_i^c. The global function for c is $(x_1 + x_2)g(\mathbf{x}')$. $C_{x_1x_2}(x_1 + x_2)g(\mathbf{x}') = 0$.

5.5.5 Computing the Image

After computing the compatible observability plus external don't care set at y_i in terms of primary inputs d_i^{cg}, we find all the combinations of variables in the local space of y_i which are possible for some input vector in $\overline{d}_i^{cg}$. This is done by cofactoring each global fanin function of y_i with respect to $\overline{d}_i^{cg}$ and then finding all the reachable points using a range computation algorithm. The cofactor operations are generalized cofactor operations defined in Section 2.3.1. We also introduced three different techniques, transition relation

method, output cofactoring, and input cofactoring for range computation in Section 2.3. We discuss their relative merits for computing local don't cares.

The number of immediate fanins of a node y_i being simplified is usually much less than the number of the primary inputs in the transitive fanin cone of y_i. As a result the size of the support of $\mathbf{g} = [g_1, \ldots, g_m]$ (i.e. n where $\mathbf{x} = [x_1, \ldots, x_n]$ is the support of $\mathbf{g}$ and $\mathbf{g}$ is a vector representing the global functions at the fanins of y_i) is much greater than the number of elements of $\mathbf{g}$ (i.e. m). Therefore, if output cofactoring is chosen, the number of cofactoring operations that needs to be done to compute the range of $\mathbf{g}$ is considerably less than the case where input cofactoring is used. The shortcoming of the transition relation method for this application is that one has to order both the $\mathbf{y}$ and the $\mathbf{x}$ variables to build the characteristic function $G(\mathbf{x},\mathbf{y}) = \prod_{1 \leq i \leq m}(y_i \overline{\oplus} g_i(\mathbf{x}))$ for range computation. The simplification of all the nodes in the network requires the ordering of all the intermediate variables in addition to primary input variables. We do not know of a good ordering for all these variables at the same time. The other disadvantage of the transition relation method is that the range must be computed in BDD form and then transformed into sum-of-products form to be fed to a two-level minimizer (two-level minimizers which manipulate BDD's are not fully developed yet). As a result, we used output cofactoring for range computation.

The range computation algorithm is shown in Figure 5.8. Given a Boolean function $\mathbf{g} = [g_1, \ldots, g_m]$, we compute the characteristic function of the range of $\mathbf{g}$ recursively using the following equation:

$$range(\mathbf{g})(y) \;=\; y_1 range([(g_2)_{g_1}, \ldots, (g_m)_{g_1}]) + \overline{y_1} range([(g_2)_{\overline{g_1}}, \ldots, (g_m)_{\overline{g_1}}])$$

The terms inside the recursive calls to the range computation are cofactored by g_1 or $\overline{g_1}$ to decrease the complexity of the recursive computation. More importantly, at each step of the recursion, whenever the remaining single output functions can be grouped into sets of disjoint support, the range computation proceeds on each group independently. This reduces the worst case complexity from 2^m to $2^{s_1} + \ldots + 2^{s_k}$ where $(s_1, \ldots, s_k)$ are the sizes of the independent groups ($s_1 + \ldots + s_k = m$). If an element of $\mathbf{g}$ has a fixed value of 1 or 0, that element is removed and the corresponding literal is ANDed in sum-of-products form with the result of the range computation for the rest of the elements in $\mathbf{g}$.

The algorithm for partitioning elements of $\mathbf{g}$ into groups of disjoint support is shown in Figure 5.9. We first find the support of each function. There is a single bit associated with each variable in the set representing the support of each g_i. This bit is set

```
function range([g_1, ..., g_m])
/* returns the range in sum-of-products form */
begin
    if (m == 0) return 1
    for (1 ≤ i ≤ m) begin
        if (g_i == 1) return y_i . range([g_1, ..., g_{i-1}, g_{i+1}, ..., g_m])   /* cubes */
        if (g_i == 0) return ȳ_i . range([g_1, ..., g_{i-1}, g_{i+1}, ..., g_m])   /* cubes */
    end
    /* partition [g_1, ..., g_m] into groups G_1, ..., G_k with disjoint supports*/
    [G_1, ..., G_k] = partition [g_1, ..., g_m]
    if (k > 1) return range(G_1) . range(G_2) ...range(G_k)
    select output i with the smallest support
    return y_i . range([(g_1)_{g_i}, ..., (g_{i-1})_{g_i}, (g_{i+1})_{g_i}, ..., (g_m)_{g_i}])
                + ȳ_i . range([(g_1)_{ḡ_i}, ..., (g_{i-1})_{ḡ_i}, (g_{i+1})_{ḡ_i}, ..., (g_m)_{ḡ_i}])
end
```

Figure 5.8: Range Computation Algorithm

to 1 if the variable is present in the support and 0 otherwise. We OR any two support sets with a 1 in the same column, and substitute the two sets with the support set obtained by ORing. After this is done for all the columns, we get sets s_i that represent each partition G_{j_i}. Any g_k whose support is in s_i belongs to the partition G_{j_i}.

The routine *range* is called by passing as argument a multiple output Boolean function. It returns the local care set for a node y_i in sum-of-products form. Once the care set is known, the don't care set can be computed by finding the inverse of the care set. One can find the don't care set directly with slight modifications to the algorithm shown in Figure 5.8.

d_i^l obtained by the range computation routine is in sum-of-products form; therefore, it can be used directly as input to a two-level minimizer. Before simplifying y_i, we add to d_i^l satisfiability don't cares of the nodes that can be substituted in y_i. These nodes are nodes with a support that is a subset of support of y_i.

5.6 Conclusion

We have introduced an algorithm for computing local don't cares for simplification of intermediate nodes of a multi-level network. The external don't care network used by the algorithm can be a general multi-level network, because BDD's are used for representing external don't cares in terms of primary inputs. The compatible observability don't cares can be computed more efficiently because set operations are much faster using BDD's. This technique allows larger circuits to be optimized. The technique is limited to those networks where the BDD's of the network functions can be built, which depends on the quality of BDD ordering methods available.

```
function partition([g_1, ..., g_m])
/* partition into groups with disjoint support */
begin
    /*find the support of each function */
    for (1 ≤ i ≤ m) begin
        s_i = support(g_i)
    end
    /* OR any two support sets which have common parts */
    for (1 ≤ j ≤ n) begin
        fs = NIL
        for (1 ≤ i ≤ m) begin
            if s_i == NIL continue
            if s_i(j) == 0 continue
            if fs == NIL
                fs = s_i
            else begin
                fs = OR(fs, s_i)
                s_i = NIL
            end
        end
    end
    /* Make a group G_j corresponding to each support set*/
    j = 0
    for (1 ≤ i ≤ m) begin
        if s_i == NIL continue
        G_j = all g_k whose support is contained in s_i
        j = j + 1
    end
end
```

Figure 5.9: Partition into Groups with Disjoint Support

Chapter 6

Scripts for Technology Independent Optimization

In this chapter, we investigate the set of operations used for the optimization of a multi-level Boolean network. These operations are run in a prespecified order, known as a *script*, in MIS-II. Scripts embody different kinds of manipulations used for optimizing multi-level networks. These manipulations can be grouped as

1. extraction which is extracting common expressions among the nodes of a Boolean network and creating new nodes representing them,
2. node restructuring which is reducing the amount of logic at each node, and
3. elimination which is removing nodes whose value is below some threshold in the Boolean network.

We examine each category, explain the shortcomings of current approaches and provide modifications that improve the robustness of the methods and the quality of optimized networks. We present a new script which uses *full_simplify* and *fast_extract*, and give results on a large set of benchmark circuits. The results show that the modifications improve the robustness of the scripts and the quality of results significantly.

6.1 Introduction

The key to implementing a set of logic functions in a small area with small delay through the circuit is to find a good Boolean decomposition of those logic functions. The set of logic functions are represented as a Boolean network. Each logic function can be implemented using simple gates in many different ways. The decomposition is equivalent to partitioning logic functions into smaller ones whose interaction generates the desired behavior at the primary outputs of the Boolean network. Each partition is a logic function itself and is represented by an intermediate node in the Boolean network. These intermediate nodes are later mapped into gates available in the library. The best implementation for a set of logic equations is usually obtained by a multi-level network where many levels of logic are used to implement each logic function.

The operation of extracting the right intermediate nodes is computationally expensive because of many different ways in which a Boolean network can be decomposed. Therefore we use an iterative procedure to improve the quality of the multi-level network incrementally. The cost function we use to measure the quality of a Boolean network is the number of literals in factored form. For each intermediate node of the network we find the best factored form representation and count the number of literals in each representation. The sum of literal counts for all the nodes is the literal count for the whole network. Experiments show that the literal count in factored form correlates well with the final area of the mapped circuits [1]. The three sets of transformations used to reduce the cost are extraction, node simplification, and elimination.

Extraction is using algebraic techniques to capture common sub-expressions in the network and creating new intermediate nodes which represent those sub-expressions. Three different algebraic techniques, kernel extraction algorithms, cube extraction algorithms, and a polynomial time two-cube extraction algorithm are discussed in this chapter. It is shown that the two-cube extraction algorithm can give results as good as other approaches while the time spent for extraction is substantially less.

Once new nodes are extracted we use node restructuring techniques to improve the quality of each node locally. This is usually done by introducing new fanins to the node

[1] Of course area minimization is not the only optimization criterion of interest. However, we have found that a minimal area circuit is a good starting point for optimizations whose goals are performance or testability. Furthermore, even after performance optimization, area reclamation is necessary to achieve acceptable results.

and then removing some other fanins while preserving the desired behavior at the primary outputs. Node restructuring can also be done in different ways such as node simplification, algebraic resubstitution, and redundancy removal. The most powerful technique of all is node simplification where a two-level minimizer is applied to each node in the network. Node simplification is a local operation. Don't care sets which contain information about the structure of the network are used to make node simplification more effective. These don't cares are a combination of observability, satisfiability, and external don't cares discussed in Chapter 3.

We associate a value to each node in the multi-level network that measures how good it is to keep that node in the network. Once the node restructuring is done, we remove (or eliminate) all nodes whose value is below some threshold [10]. The process of extraction, node restructuring, and elimination is repeated many times in different orders until no more improvement is possible.

In practice we start from a PLA or a multi-level network and use the extraction, node restructuring, and elimination algorithms in different order to improve the quality of the Boolean network incrementally. The extraction, node simplification, and elimination are all greedy algorithms. While extracting new nodes, we look for the best algebraic divisor at each step although it can affect other divisors extracted later on. In a simplification algorithm we first find some ordering of the nodes in the network and then in sequence try to simplify the local function at each of the nodes as much as possible. The order in which the intermediate nodes of the network are simplified is important because simplification of one node can affect simplification of other nodes in the network. The elimination algorithm is again a greedy algorithm and therefore, the elimination of one node can affect the value of other nodes in the network. Because most operations are done in a greedy way the final result is very dependent on the starting point and the order in which the optimizing operations are done.

Currently, there are two scripts in MIS-II for logic minimization, the *algebraic script* and the *Boolean script*. The main difference is that the algebraic script performs simplification without any don't cares while the Boolean script performs simplification using satisfiability don't cares. Each script contains a sequence of operations that have performed well on most industrial examples. In the following sections, we evaluate each of the commands in these scripts, investigate why they fail on some circuits and show ways to improve the quality of each command. We then propose new scripts which include these modifica-

tions and show results that prove the effectiveness of our modifications. In Chapter 7, we map these circuits into a set of given library gates and show the improvement on mapped circuits.

6.2 Scripts Used for Logic Minimization

Our ultimate goal is to obtain a script which is robust in the sense that it rarely fails (no timeout or space out problems) on a wide variety of circuits and produces results as good as any manually directed script. Three sources of non-robustness in MIS-II are kernel extraction, simplification, and elimination algorithms.

- **Kernel Extract:** Some functions have many kernels (especially symmetrical functions). This either causes a spaceout problem or the time for extraction becomes enormous. This is exacerbated by the present implementation of kernel extract in MIS-II which selects only a few kernels among all kernels generated and then reextracts all the kernels again. One possible fix is the use of two-cube kernels as proposed in [81]. According to our experimental results, the two approaches are comparable in quality in most cases, but two-cube extraction is much faster.

- **Simplify:** The problem here is generating and using don't cares. The filters used for keeping the size of don't cares small are not always effective. As mentioned in Chapter 5, the input to the two-level minimizer has an onset and a don't care set. The offset must be generated to find a good representation for the function. The complementation necessary for generating the offset of a function and using it within a two-level minimizer becomes infeasible when the don't care set is large. On the other hand not using the don't cares degrades the quality of the results. An efficient method for node simplification with don't cares is given in Chapter 5. The timeout and spaceout problems can still occur but more rarely.

- **Eliminate:** This can cause creation of a node whose sum-of-products form has too many cubes. The solution we propose is an intelligent ordering of the nodes in the network and then a controlled elimination.

We now discuss these problems and their proposed solutions in more detail.

6.2.1 Kernel and Cube Extraction

An important step in network optimization is extracting new nodes representing logic functions whose interaction gives the desired behavior at the outputs of the multi-level network. We do not know of any Boolean decomposition technique that performs well and is not computationally expensive; therefore we use algebraic techniques. The basic idea is to look for expressions that are observed many times in the nodes of the network and extract such expressions. Each such expression is implemented only once as a node and the output of this node replaces the occurrence of the expression in any other node in the network. This technique is dependent on the sum-of-products representation at each node in the network and therefore a slight change at a node can cause a large change in the final result, for better or for worse.

The current algebraic techniques in MIS-II are based on kernels [13]. The kernels of a logic expression f are defined as

$$K(f) = \{g \mid g = f/c, g \text{ is cube free}\}$$

where c is a cube, g has at least two cubes and is the result of algebraic division of f by c, and there is no common literal among all the cubes in g (i.e. g is cube free). This set is smaller than the set of all algebraic divisors of the nodes in the network; therefore it can be computed much faster and is almost as effective. One problem encountered with this in practice is that the number of kernels of a logic expression can be exponential in the number of cubes appearing in that expression. Furthermore, after a kernel is extracted from a node, its set of kernels changes. There is no easy way of updating other kernels, thus kernel extraction is usually repeated. Once all kernels are extracted, the largest intersection that divides most nodes in the network is sought. There is no notion of the complement of a kernel being used at this stage. After kernels are extracted, one looks for the best single cube divisors and extracts such cubes. The kernels and cubes are sought only in uncomplemented form (e.g. if $a + b$ is extracted, we do not substitute its complement $\overline{a}\overline{b}$ at this stage.). Later, Boolean or algebraic resubstitution can perform division by the complement as well.

A more recent algebraic technique extracts only two-cube divisors and two-literal single-cube divisors both in normal and complemented forms [81]. This approach has several advantages in terms of computation time while the quality of the final result is as good as

that obtained by kernel-based approaches. This was first observed in [81] and later confirmed by our experimental results. It can be shown that the total number of double-cube divisors and two-literal single-cube divisors is polynomial in the number of cubes appearing in the expression. Also, this set is created once, and can be efficiently updated when a divisor is extracted. Additionally, one looks for both the normal and complemented forms of a divisor in all the nodes in the network so in choosing the best divisor a better evaluation can be made based on the usefulness of the divisor as well as its complement. There is also no need for algebraic resubstitution once divisors are extracted.

The algorithm of [81] works as follows. First all two-cube divisors and two-literal single-cube divisors are recognized and put in a list. A value is associated with each divisor which measures how many literals are saved if that expression is extracted. This value includes the usefulness of the complement in the cases where the complements are single cubes or two-cube divisors. Common cube divisors are also evaluated at the same time so that "kernel" extraction and "cube" extraction are nicely interleaved by this process. The divisor with highest value is extracted greedily. All other divisors and their values are updated and the whole process is repeated until no more divisors can be extracted. This technique has been implemented in MIS-II and is called *fast_extract* or *fx*.

One shortcoming of this approach is that the size of each divisor is limited to no more than two cubes. However, large nodes are effectively extracted by the combined process of *fast_extract* and elimination. Elimination as explained later, is used to increase the size of some divisors and remove others that are not effective.

6.2.2 Simplification

To improve the local function at each intermediate node of a multi-level network, we apply a two-level minimizer to each node. The two-level minimizer finds an optimal representation of the node. The input to the two-level minimizer is composed of an onset cover and a don't care set. The onset cover is the function of the node in terms of its fanins. The don't care set contains information about the structure of the Boolean network. Different don't care subsets are used to optimize a node y_i. One approach uses only satisfiability don't cares of the nodes whose support is included in the support of the node being simplified. This is called the support subset and is used in *Boolean script* in MIS-II. The second approach uses no don't care at all. The input given to the two-level minimizer

contains only an onset for y_i. Algebraic techniques are used to substitute other nodes of the network in the functional representation of y_i. This technique is used in *algebraic script.* The third approach is *full_simplify* where local don't cares and subset support satisfiability don't cares are used for node simplification.

Boolean script produces much better results in general as compared to *algebraic script* but it cannot be applied to some circuits because simplification with support subset fails (e.g. `apex1, apex2, apex3, etc.`). This happens because nodes in the original network are extremely large. There are nodes with more than 100 cubes and 40 fanins. When a node y_i is being simplified, we generate satisfiability don't cares for substitutable nodes (nodes that can be substituted in y_i with high probability). To generate satisfiability don't cares at a node we have to find the complement of the function at that node. The operation of finding the complement of the function might not complete if the function has too many cubes and too many fanins. Even if the satisfiability don't care generation is successful, the two-level minimizer might fail because of the large number of variables and large size of the don't care set. Finally, big nodes are not usually substituted into the other nodes of the network, even if the two-level minimizer completes its job. As a result, much CPU time is wasted and no optimization is obtained if large nodes are allowed to be substituted in other nodes. In the cases where big nodes can be substituted in other nodes, we can postpone this operation until some common expressions of these nodes have been extracted and nodes are somewhat smaller.

To remedy this problem we introduce the following measures. **1)** We do not allow any node with more than 100 cubes to be substituted in any other node in the network. **2)** We do not allow a node y_j which has more than twice the number of cubes of y_i (the node being simplified) to be substituted in y_i even though the support of y_j might be included in the support of y_i. **3)** We limit the size of the don't care set of the substitutable nodes to 6000 literals. We order all the nodes that could be substituted in y_i and generate the satisfiability don't cares for these nodes until the limit of 6000 was reached. We use the size of the nodes as the criterion for the ordering, i.e. the smallest node is the first in the ordering. This is because it is more likely that smaller nodes are substitutable in y_i. The rest of the nodes may be substituted when the two-level minimizer is applied again.

The improved version of simplify discussed in Chapter 5 uses the local don't cares in terms of immediate fanin variables of each node to simplify it. The advantage of this approach is that we can remove most redundancies in the network. The size of these local

don't cares is usually very small, therefore they do not cause any problem with two-level minimizers. To compute the local don't cares, BDD's are built for the nodes in the network, external plus observability don' cares are computed for each node in terms of primary inputs of the network, and image computation techniques are used to map the computed don't care set to the local space of the node being simplified. There are two problems with this approach.

First, BDD's cannot built for some circuits either because no good ordering of primary inputs exists for building such BDD's (e.g. multipliers C6288) or because the current heuristic used for ordering the primary inputs does not perform well (e.g., the computation of local don't cares for C432 can be done much faster if a different ordering is used.) Also, at times we can find local don't cares for the nodes of the network easily before it is optimized. But once it is optimized such operations become very expensive. This is because the used heuristic fails to find a good ordering from the structure of the optimized network. One could partition large circuits for which BDD's cannot be built into smaller ones and then apply *full_simplify* to each subcircuit.

The second problem is that the image computation techniques used for extracting local don't cares for a node might not complete if that node has many fanins. This is because the technique employed for image computation can be exponential in the number of the fanins of the node being simplified. We noticed this problem only a few times on very large nodes in circuits like misex3, apex3, and apex4.

6.2.3 Elimination

To improve the cost of a Boolean network, we eliminate some nodes in the network and then extract new nodes. The elimination of a node is equivalent to replacing the variable associated with that node with the local function at the node everywhere in the Boolean network. First, we associate a value to each node in the network which measures the quality of that node. The value of a node y_i is defined to be

$$area\_value(y_i) = (\sum_{f_k \in FANOUT(y_i)} N(f_k, y_i) - 1).(L(y_i) - 1) - 1$$

where $L(y_i)$ is the number of literals in factored form for y_i and $N(f_k, y_i)$ is the number of times either y_i or $\overline{y}_i$ appears in factored form in f_k as shown in [10]. The elimination is done by removing the nodes whose value is below some threshold. These nodes are found

one by one and then removed from the network.

In the past, we did not take into the consideration the order in which these nodes were removed. If y_i is eliminated the value of its immediate fanins, immediate fanouts, and immediate fanins of the immediate fanouts can change. If y_k is an immediate fanout, then in most cases $L(y_k)$ increases, thus $area\_value(y_k)$ increases. If y_l is an immediate fanin, then in most cases $\sum_{f \in FANOUT(y_l)} N(f, y_l)$ increases, thus $area\_value(y_l)$ increases. If y_l is an immediate fanin of an immediate fanout y_k, then because $L(y_k)$ increases, $\sum_{f \in FANOUT(y_l)} N(f, y_l)$ could increase. Therefore, if a node is removed, the value of its fanins, fanouts, or fanins of its fanouts may go above the threshold. Such nodes will not be eliminated any more. As a result, the ordering in which nodes are eliminated from the network is important.

The most common value used for the threshold is -1. A node is eliminated in that case either if $L(y_i) = 1$ or $\sum_{f \in FANOUT(y_i)} N(f, y_i) = 1$. The nodes with $L(y_i) = 1$ will be eliminated no matter what ordering is used. This is not the case when $\sum_{f \in FANOUT(y_i)} N(f, y_i) = 1$. Because the *fast_extract* algorithm completes on all the benchmarks, we would like to have an ordering which works well when used with *fast_extract*. All the divisors extracted by *fast_extract* have less than or equal to two cubes. We know that to find an optimized network with low cost, we need larger divisors. Therefore, our heuristics are designed to make divisors larger in general. For example, let

$$f = (a+b+c)(t+v+w)(x+y+z).$$

After extracting two-cube kernels, we obtain the network

$$\begin{aligned} f &= 2\,4\,6 \\ 1 &= a+b \\ 2 &= 1+c \\ 3 &= t+v \\ 4 &= 3+w \\ 5 &= x+y \\ 6 &= 5+z \end{aligned}$$

Our experiments show that the best results are obtained if we try to eliminate the nodes that fanin to a node before that node itself is eliminated. To implement this, we first find

the level of each node in the network starting from the primary inputs which are assigned level 0. The level of each node y_i is $level(y_i) = 1 + max(level(y_l), \forall y_l \in FANIN(y_i))$. We order nodes according to their level and remove the ones with lower levels first if their value is below the threshold. Applying `eliminate -1` with this ordering to the above example we get,

$$\begin{aligned} f &= 24x + 24y + 24z \\ 2 &= a + b + c \\ 4 &= t + v + w \end{aligned}$$

which is the desired decomposition.

The other problem with `eliminate` is that some nodes in the network may become too large after elimination is done. For example using the current version of eliminate in MIS-II, if one does eliminate -1 on some circuits (e.g. `C432`, `C2670`, and `C7552`), the number of cubes in some nodes becomes so large that no other transformation can be applied on these circuits. To prevent such node explosions, we set a limit on the number of cubes that each node can have. We set this limit to be equal to twice the number of cubes of the node with the most cubes in the network. We can still remove as many nodes as desired from the network by repeated application of `eliminate`. Furthermore, we prevent sudden size explosions so that subsequent optimizing operations can be applied to the Boolean network. Because of this simple modification to `eliminate`, we can handle most circuits where we had memory problems before.

6.3 Scripts

We ran two different scripts on a large set of benchmarks repeatedly until no more improvement was obtained and compared the final results together. The **rscript** shown in Figure 6.1 uses the *fast_extract* command in MIS-II. This script is known as rugged script and is one of the scripts available within SIS [74]. The **bscript**, which is the same as *Boolean script* in MIS-II with our modifications to `eliminate` and `simplify`, and the addition of `full_simplify`, uses the kernel extraction command *gkx* and the cube extraction command *gcx*. These commands are repeated many times with different threshold values. A kernel or cube is extracted only if its value is above the threshold. *sweep* removes internal nodes with no fanout or no fanin, buffers and inverters. *eliminate* does the elimination of nodes

```
rscript
    sweep; eliminate -1
    simplify -m nocomp
    sweep; eliminate 5
    simplify -m nocomp
    resub -a
    fast_extract
    sweep; eliminate -1
    full_simplify
```

Figure 6.1: Simplification Script Using fast_extract

whose value is below the given threshold. *simplify -m nocomp* is a simplification algorithm implemented in the ESPRESSO [11] environment which uses reduced offsets [50]. *resub -a* is an algebraic technique for substituting each node of the network into other nodes in the network. It considers both the normal and complement form of the node. *decomp -g* successively extracts the best kernels until no more can be extracted. It is used to break big nodes into smaller nodes.

6.4 Experimental Results

We run a large set of experiments to measure the effectiveness of our transformations for area optimization, removing combinational and sequential redundancies, and effective use of external don't cares. These same transformations are also used for performance optimization as shown in [77, 79].

6.4.1 Area Optimization

The results of the experiments for technology independent optimization of a large set of ISCAS and MCNC benchmarks are shown in Tables 6.1 and 6.2. The starting circuits in Table 6.1 are multi-level circuits. The starting circuits in Table 6.2 are PLA's. The **start** columns in the two tables show the number of literals in factored form before any

```
bscript
    sweep; eliminate -1
    simplify -m nocomp
    sweep; eliminate 5
    simplify -m nocomp
    resub -a

    gkx -abt 30
    resub -a; sweep
    gcx -bt 30
    resub -a; sweep

    gkx -abt 10
    resub -a; sweep
    gcx -bt 10
    resub -a; sweep

    gkx -ab
    resub -a; sweep
    gcx -b
    resub -a; sweep

    eliminate 0
    decomp -g
    sweep; eliminate -1
    full_simplify
```

Figure 6.2: Simplification Script Using Kernel Extraction

optimization is done. This number is obtained after *sweep; eliminate -1* in SIS.

Columns **rscript** and **bscript** are the results of running each script repeatedly until no more optimization is possible. This usually happens after 2 or 3 iterations of the scripts. We perform some initial transformations on the circuits before any of the scripts are applied. These transformations are different for the two sets of circuits. **rCPU** and **bCPU** are the times taken by **rscript** and **bscript** respectively.

The column **obest** is the result obtained by MIS-II presented at the International Workshop on Logic Synthesis in 1989. These results were obtained by a variety of scripts and/or human interaction in directing the order of application of MIS-II commands. **nbest** is the best result obtained by us using a few variations on these scripts. **ratio** is the ratio of the literals in factored form in the **nbest** column over the one in the **obest** column. The **average** row shows the average improvement in literals in factored form.

First we discuss Table 6.1, where the starting circuits are multi-level circuits. For multi-level circuits we apply `full_simplify` first and then both scripts repeatedly until no more improvement is obtained. Both scripts run successfully on most circuits. *full_simplify* does not perform any optimization on C6288 and C7552 because BDD's cannot be built for these two circuits. We use both scripts on C6288 and C7552 without *full_simplify*. We do not get any optimization on C6288, because the number of literals increases after we run any of the scripts. On the other hand, we obtain considerable optimization for C7552 without using *full_simplify*. The size of the BDD's for C2670 and C3540 is also very large. Overall, the results shown in columns **rscript**, **bscript**, and **nbest** are considerably better than the previous results on most of these benchmarks. On the average, we obtain 17% improvement over the old results as shown in Table 6.1.

Table 6.2 shows the results of the experiments where the starting point is a PLA. In this case we apply `resub -a; simplify -d` first and then run the scripts repeatedly until no more optimization is possible. `resub -a` allows algebraic substitution of one output into the other outputs. The starting literal count for the PLA's is shown in factored form. It is observed that for circuits where the starting literal count in sum-of-products form is around 10000 or more, the order in which simplification, algebraic resubstitution, and algebraic extraction are done is very important. For example, the simplification of a particular node can make a major difference in the final result. This is because the algebraic techniques are so dependent on the representation of the nodes and nodes in these circuits are usually very large. These circuits are `apex1`, `apex2`, `apex3`, `apex4`, `misex3`, and

circuit	start	rscript	bscript	rCPU	bCPU	obest	nbest	ratio
C432	322	203	196	970	752	321	187	0.58
C499	562	554	550	317	501	552	550	1.00
C880	433	417	417	70	105	416	398	0.96
C1355	562	554	550	317	501	552	550	1.00
C1908	769	512	511	1516	1571	541	511	0.94
C2670	1031	724	737	1007	994	1031	716	0.69
C3540	1633	1221	1248	2501	1543	1633	1200	0.73
C5315	2425	1722	1709	1118	1361	1796	1709	0.95
C6288	3313	nop	nop					
C7552	3022	2159	2209	939	1442	2505	2159	0.86
apex6	835	723	720	163	259	784	720	0.92
apex7	289	239	237	52	44	240	237	0.99
b9	162	125	124	11	21	132	124	0.94
k2	2930	990	996	10628	3119	–	968	–
des	6101	3216	3257	7010	6993	3538	3216	0.91
f51m	169	85	116	20	37	118	85	0.72
rot	764	668	663	333	449	704	663	0.94
z4ml	77	41	36	7	7	43	36	0.84
9symml	237	186	205	271	291	176	161	0.91
average	1349	–	–	–	–	–	–	0.83

Table 6.1: Performance of Scripts Starting from Multi-Level Circuits

start, rscript, bscript, obest, nbest: number of literals in factored form
rCPU, bCPU: in seconds on a IBM Risc System/6000 530
nop: no optimization was obtained

circuit	start	rscript	bscript	rCPU	bCPU	obest	nbest	ratio
5xp1	163	100	96	14	23	104	89	0.86
9sym	283	197	178	344	191	216	178	0.82
alu4	2058	100	104	430	387	263	100	0.38
bw	296	163	162	36	43	163	161	1.00
clip	264	123	131	52	88	119	108	0.91
rd53	71	34	36	5	6	33	34	1.03
rd73	247	70	56	15	38	74	56	0.76
rd84	482	116	112	109	180	124	74	0.60
sao2	288	114	114	67	94	118	114	0.97
seq	3707	1706	–	timeout	timeout	1176	877	0.75
vg2	246	85	84	10	12	86	84	0.98
xor5	28	16	16	2	2	16	16	1.00
apex1	3831	1379	1428	2369	1835	1247	1063	0.85
apex2	663	167	–	983	timeout	246	167	0.68
apex3	3263	1617	1553	2367	573	1401	1426	1.02
apex4	5976	2318	2321	timeout	timeout	2592	2163	0.83
apex5	2848	745	777	420	514	890	745	0.84
e64	2144	253	253	109	135	253	253	1.00
o64	130	130	130	3	3	130	130	1.00
misex1	88	52	51	6	7	49	50	1.02
misex2	164	103	103	10	13	103	101	0.98
misex3	1929	703	676	timeout	timeout	371	547	1.47
misex3c	850	443	439	5175	4317	452	439	0.97
con1	19	19	19	1	2	19	19	1.00
duke2	938	392	382	413	212	393	360	0.92
average	1239	–	–	–	–	–	–	0.90

Table 6.2: Performance of Scripts Starting from PLA's

start, rscript, bscript, obset, nbest: number of literals in factored form
rCPU, bCPU: in seconds on a IBM Risc System/6000 530
timeout: set to 15000 sec. of CPU time

seq. For example, in the case of **apex1** if we run *resub -a; fast_extract* and then **rscript** repeatedly we can decrease the number of literals to 1063 as shown in the column **nbest** in Table 6.2. This shows the need for better network decomposition strategies. We are still able to get considerable improvement over the results shown in column **obest**.

It is not possible to get any comparison of *fast_script* and kernel extraction techniques from Tables 6.1 and 6.2 because most of the CPU time is spent on node simplification. To better measure the merits of *fast_extract*, we first run *simplify -m nocomp; resub -a* on all the PLA circuits. Then we run *fast_extract; eliminate 0* and compare this with the kernel extraction technique in Figure 6.2 (the series of commands starting from *gkx -abt 30* to *eliminate 0*). The result is shown in Table 6.4.

The column **fx** shows the number of literals in factored form when *fast_extract* is used. The column **gkx** shows the number of literals in factored form when *gkx* and *gcx* with different threshold values are used. The number of literals in factored form obtained from *fast_extract* is slightly better than the one obtained by using *gkx* and *gcx* while the CPU time is 21 times less. As a result, no quality is lost by using two cube kernels. The total number of intermediate nodes in the networks optimized by *fast_extract* and *gkx, gcx* after **eliminate** are comparable which shows divisors obtained by fast_extract are as big as the ones obtained by the kernel extraction algorithm after elimination.

6.4.2 Sequential Optimization

A set of benchmarks with external don't cares are generated to see how effectively external don't cares can be used for optimizing these networks. The ISCAS sequential circuit benchmarks are used for this experiment. For each circuit, an external don't care set is generated by finding all the states that are unreachable starting from a given initial state. These operations are done using BDD's [78] and [47]. The external don't care set for each output function is equal to the set of unreachable states. An external don't care network is generated from the BDD representing the set of unreachable states. There is an output in the external don't care network corresponding to each output in the care network. The outputs of the external don't care network compute the same function. We build an internal node in the external don't care network which computes the function of BDD representing unreachable states. This node then fans out to all the outputs in the external don't care network. To build the internal node corresponding to the BDD representing the

circuit	fx	gkx	ratio	CPUfx	CPUgkx	nodefx	nodegkx
5xp1	118	117	1.01	1	2	15	13
9sym	225	234	0.96	11	8	17	28
alu4	399	396	1.01	7	16	46	45
bw	161	161	1.00	0	2	33	33
clip	147	148	0.99	2	4	19	20
rd53	39	41	0.95	0	1	7	5
rd73	73	94	0.78	2	7	14	9
rd84	176	194	0.91	14	129	22	27
sao2	194	181	1.07	2	6	28	16
seq	1613	1687	0.96	105	1745	247	229
vg2	88	87	1.01	0	1	10	11
xor5	16	20	0.80	0	0	4	2
apex1	1535	1521	1.01	44	807	270	252
apex2	313	345	0.91	126	3216	48	53
apex3	1478	1554	0.95	55	280	178	165
apex4	2195	–	–	50	timeout	228	inc
apex5	780	799	0.98	14	43	157	153
e64	253	253	1.00	5	22	95	95
o64	130	130	1.00	0	0	1	1
misex1	52	52	1.00	0	0	9	9
misex2	108	106	1.02	0	1	25	24
misex3	732	676	1.08	151	4116	111	97
misex3c	461	457	1.01	4	9	40	40
con1	20	19	1.05	0	0	3	2
duke2	452	460	0.98	3	15	80	72
–	**total**	**total**	**aver**	**total**	**total**	**total**	**total**
–	9563	9732	0.98	502	10429	1479	1401

Table 6.3: Comparison of Algebraic Extraction Techniques

fx, gkx: number of literals in factored form
ratio: ratio of numbers of literals in factored form from fx over gkx
CPUfx, CPUgkx: in seconds on a IBM Risc System/6000 530
nodefx, nodegkx: number of intermediate nodes in the network
timeout: set to 15000 sec. of CPU time

unreachable states, a two input multiplexor node is built in the external don't care network corresponding to each node of the BDD. To keep the size of the external don't care network small a limited optimization is done (sweep; eliminate -1; simplify) on the external don't care network. The care network and the external don't care network are saved in blif form. The command *extract_seq_dc* in SIS finds unreachable states of a finite state machine and stores them in an external don't care network as explained above.

We simplify these networks using the rugged script shown in Figure 6.1 which uses *full_simplify* and compare it with the same script when *full_simplify* is replaced by *simplify*. The result when *full_simplify* is used is under the column **rscript** and when it is not used is under column **sscript** shown in Table 6.4. Substantial improvement was obtained by using external don't cares in the simplification of each of the intermediate nodes. The reason for such improvement is that the initial encodings of these circuits are not good. *full_simplify* improves the encoding of the circuit in a restricted way. The size of the external don't care networks in sum-of-products form varies in the examples shown. We do not have any problem with the size of these networks as long as we can build BDD's for them.

Our optimization is focused on the combinational part of these circuits. As a result of this optimization, the input to some latches may become constant. Such latches can be removed from the circuit if the constant value is the same as the initial value of the latch. In addition, if as a result of this optimization, a set of latches forms a cycle where the outputs of these latches do not fan out to any of the outputs of finite state machine, these latches can be removed from the circuit. Another sequential transformation currently used within SIS is retiming [45, 25, 52, 73] which can be combined with *full_simplify*. New transformations for improving the structure of a finite state machine and its encoding are currently under investigation.

6.4.3 Testability

full_simplify can be used to remove both combinational and sequential redundancies in a circuit. We do not have the algorithms to measure sequential testability of a circuit at this point but the testability of combinational circuits can be measured. We ran rugged script on the multi-level circuits from the ISCAS and MCNC benchmark sets and measured the testability of these circuits. The results are shown in Table 6.5. The *atpg* command in SIS was used to find the total number of redundant plus undetected faults over the total

circuit	start	exdc net.	sscript	sCPU	rscript	rCPU
s344	168	4844	143	12	136	197
s386	205	33	152	11	112	14
s400	186	3914	160	15	124	147
s526	292	8141	197	34	141	302
s641	216	142	200	13	158	30
s420	201	37	159	12	42	8
s713	204	141	193	16	155	32
s820	523	40	332	137	304	312
s838	407	69	329	492	42	24
sand	818	30	693	463	604	2780

Table 6.4: Performance of the Scripts on Sequential Circuits

start:	number of literals in factored form
exdc net.:	number of literals in don't care network in sum-of-products form
sscript:	literals in factored form of optimized circuit using simplify in script
rscript:	literals in factored form of optimized circuit using full_simplify in script
sCPU, rCPU:	in seconds on a IBM Risc System/6000 530

number of faults in each circuit. The testability of each circuit was measured before and after optimization. The total number of redundant plus undetected faults over the total number of faults in the circuit is shown in columns denoted by **start-test** and **final-test** in Table 6.5. Most redundancies in the optimized circuits are removed because *full_simplify* is used. *full_simplify* is more powerful than redundancy removal algorithms [71, 40] because it restructures the network while removing redundancies.

6.5 Conclusion

We have provided improvement in both robustness and quality for the techniques used for optimizing multi-level networks in SIS and presented a rugged script which embodies a set of these operations in a prespecified order. The obtained results in the tables show the effectiveness of this new script on a large set of benchmarks. The results presented are much better than the ones obtained by the previous version of MIS-II and also most other logic synthesis systems. To improve robustness still further, we need to expand our techniques to handle two classes of circuits; first, circuits for which we cannot build BDD's;

circuit	start	start-test	rscript	CPU	final-test
C432	322	7/542	196	752	2/553
C880	433	0/850	417	105	0/801
C1355	562	0/878	550	501	0/890
C1908	769	7/1239	511	1571	6/846
C2670	1031	37/2057	737	994	8/1726
C3540	1633	38/2779	1248	1543	11/2361
C5315	2425	5/3954	1709	1361	8/3010
C6288	3313	3/8050	spaceout		
C7552	3022	46/5608	spaceout		
apex6	835	0/1946	720	259	0/1467
apex7	289	14/646	237	44	1/446
b9	162	1/360	124	21	0/354
k2	2930	97/3383	996	3119	4/2033
des	6101	113/9111	3257	6993	0/5523
f51m	169	0/102	116	37	0/233
rot	764	30/1620	663	449	0/1490

Table 6.5: Measuring Testability

start: number of literals in factored form
CPU: in seconds on a IBM Risc System/6000 530

second, circuits where the starting sum-of-products representation has more than 10000 literals. In the latter case, the performance of algebraic techniques is somewhat results. Because these circuits are large, we cannot use the full power of simplification techniques to restructure the nodes properly.

Chapter 7

Boolean Matching in Logic Synthesis

In this chapter, we discuss Boolean matching which can be used to map a Boolean network into a set of library gates in a particular technology. The mapping of a circuit is a technology dependent transformation which is different from the Chapters 5 and 6 of this thesis where technology independent transformations are discussed. First, a new formulation for finding the existence of a Boolean match between two functions with don't cares is presented. An algorithm for Boolean matching is then developed based on this new formulation and is used within a technology mapper as a substitute for tree matching algorithms. The new algorithm is fast and uses symmetries of the gates in the library to speed up the matching process. Local don't cares are computed for each sub-function of the network being mapped and used for Boolean matching in terms of its inputs. To reduce the frequency with which Boolean matching is used, the gates in the library are grouped into classes such that it is sufficient to try to match a function with the class representative. Experimental results show significant improvement in the final area of the mapped circuits compared to previous approaches in SIS.

7.1 Introduction

Detection of "equivalence" of Boolean functions, also called *matching*, is a problem arising in logic synthesis when a Boolean network is to be implemented in terms of reusable building blocks. Many solutions have been proposed for this problem almost since the

introduction of packaged logic gates. In [42], a tree matching algorithm is used to implement a network in terms of the gates in a library. This technique had been applied before in programming language compilers for generation of optimal code for expression trees [3]. A pioneering work using Boolean methods as an alternative for technology mapping was given in [49]. Unlike tree matching, the Boolean matching techniques allow the use of don't care information. This can result in better circuits because some matches not detectable by tree matching techniques can be found. Additionally, there is no need to add inverters to the circuit, as proposed in [26], because both input phases of a function being matched are considered at the same time.

In [49], two different algorithms for Boolean matching are proposed, one of which uses don't cares and the other does not. When matching without don't cares, symmetries are used to speed up the matching process. Techniques for finding symmetries of a function are discussed in [27]. For matching with don't cares, an alternative algorithm that does not use symmetries was proposed [49]. In [27], symmetries were also computed in the presence of don't cares and it was shown that symmetry is not a transitive property when don't cares are present. Hence, computation of symmetry sets in this situation is expensive. The algorithm in [49] uses a matching compatibility graph, built during the setup phase, to find the existence of a match between two functions in the presence of don't care conditions. Each node of this graph corresponds to an NPN-equivalent [58] function. The size of this graph grows exponentially with the size of the variable support of the functions, and has limited the use of don't cares realistically to the matching of functions with at most 4 inputs.

We present a new Boolean matching algorithm which uses both symmetries and don't care conditions of the functions being matched at the same time. This Boolean matching is done using BDD's. Disjoint sets of variables are used to build BDD's for the two functions. During matching, variables from the two sets are matched with each other one by one. The number of onset and don't care set minterms of the two functions can be computed easily because BDD's are used to represent these functions. These numbers are used to check for some necessary conditions without which a match cannot exist. Similar techniques, based on the number of onset and don't care set minterms of the two functions, are used to find the corresponding inputs of the two functions when a match exists.

We use this new Boolean matching technique within a technology mapping environment which uses the methods developed in [42, 26, 64]. First, a circuit is decomposed into a set of disjoint trees with two-input nodes. Dynamic programming approaches are

then used to map each tree. Nodes of a tree are visited in depth first order starting from the leaves. At each node, the best match for the subtree at that node is recorded. This best match is obtained by looking at all the sub-functions rooted at that node and matching them with the gates in the library. Don't cares are computed for each sub-function using techniques discussed in Chapter 5.

The second application of Boolean matching is to group all the gates in the library. As demonstrated in [64], the inclusion of complex CMOS gates in the library is useful because it may lead to a significant reduction in the required area for implementing some combinational functions. However, larger cell libraries require more matchings and imply the use of functions with more inputs, making technology mapping with very large libraries computationally expensive. In [49], a technique for speeding-up the matching by grouping gates in the library was proposed. The groups of gates are composed in such a way that, after finding a match with a representative gate, the match with all gates in the group is determined. We use Boolean matching to group gates in the library. The gates in the library are matched with each other and the ones that match with inverted inputs or output are stored in the same data structure.

The designed Boolean matching technique is practical and gives good results, on average about 12% improvement in area over tree matching. Although we have only used this Boolean matching technique to find the best mapped circuit in terms of area and for organizing the given library, the algorithm is general and can be used in other contexts such as delay optimization, or layout driven technology mapping [61].

7.2 Boolean Matching

We address the Boolean matching problem for two functions $f(x_1, \ldots, x_m)$ and $g(y_1, \ldots, y_m)$ with the same number of inputs and with don't care sets $d_f(x_1, \ldots, x_m)$ and $d_g(y_1, \ldots, y_m)$. The objective is to find an assignment of variables x to y such that there exists a function that is a cover of both f and g. If such an assignment exists, the two functions can be matched.

A particular assignment of variables of g to f $(y_{i_1} = x_{j_1}, y_{i_2} = \overline{x}_{j_2}, \ldots, y_{i_m} = x_{j_m})$ can be represented by a new function

$$A_k(x, y) = (y_{i_1} \overline{\oplus} x_{j_1})(y_{i_2} \oplus x_{j_2}) \ldots (y_{i_m} \overline{\oplus} x_{j_m}).$$

In general, both $(y_{i_1}\overline{\oplus}x_{j_1})$ and $(y_{i_1} \oplus x_{j_1})$ are possible assignments; the first sets $y_{i_1} = x_{j_1}$; the second sets $y_{i_1} = \overline{x}_{j_1}$. The function $\hat{g}_k$ under variable assignment $\mathcal{A}_k$ is simply $\hat{g}_k(x) = \mathcal{S}_y\mathcal{A}_k(x,y)g(y)$.

Lemma 7.2.1 *Let $\hat{g}$ and $\hat{d}_g$ represent the new function obtained from g and d_g by switching y's with x's corresponding to a particular assignment. A matching under this assignment exists if and only if $\hat{g} - \hat{d}_g \subseteq f + d_f$ and $f - d_f \subseteq \hat{g} + \hat{d}_g$.*

Proof Assume a matching exists under the given variable assignment and let h represent the function for which the matching exists. Thus $\hat{g}-\hat{d}_g \subseteq h \subseteq \hat{g}+\hat{d}_g$ and $f-d_f \subseteq h \subseteq f+d_f$; therefore, $\hat{g} - \hat{d}_g \subseteq f + d_f$ and $f - d_f \subseteq \hat{g} + \hat{d}_g$. On the other hand, if $\hat{g} - \hat{d}_g \subseteq f + d_f$ and $f - d_f \subseteq \hat{g} + \hat{d}_g$, we let $h = (f - d_f) + (\hat{g} - \hat{d}_g)$. Clearly, $\hat{g} - \hat{d}_g \subseteq h \subseteq \hat{g} + \hat{d}_g$ and $f - d_f \subseteq h \subseteq f + d_f$. ∎

Lemma 7.2.2 *The matching under variable assignment $\mathcal{A}_k$ exists if and only if*

$$\mathcal{M}_k = \mathcal{C}_x(\mathcal{S}_y(\mathcal{A}_k(d_f + d_g + f\overline{\oplus}g))) \equiv 1 \tag{7.1}$$

(The significance of the consensus operation is shown in the next Lemma).

Proof Let $\hat{g} = \mathcal{S}_y\mathcal{A}_k g$ and $\hat{d}_g = \mathcal{S}_y\mathcal{A}_k d_g$ then

$$\begin{aligned}
\mathcal{M}_k &= \mathcal{C}_x(\mathcal{S}_y(\mathcal{A}_k(d_f + d_g + f\overline{\oplus}g))) \\
&= \mathcal{C}_x(\mathcal{S}_y\mathcal{A}_k d_f + \mathcal{S}_y\mathcal{A}_k d_g + \mathcal{S}_y\mathcal{A}_k fg + \mathcal{S}_y\mathcal{A}_k\overline{f}\overline{g}) \\
&= \mathcal{C}_x(d_f\mathcal{S}_y\mathcal{A}_k + \hat{d}_g + f\mathcal{S}_y\mathcal{A}_k g + \overline{f}\mathcal{S}_y\mathcal{A}_k\overline{g}) \\
&= \mathcal{C}_x(d_f + \hat{d}_g + f\overline{\oplus}\hat{g})
\end{aligned}$$

$\mathcal{C}_x(d_f + \hat{d}_g + f\overline{\oplus}\hat{g}) = 1$ if and only if $(d_f + \hat{d}_g + f\overline{\oplus}\hat{g}) = 1$. Assume $(d_f + \hat{d}_g + f\overline{\oplus}\hat{g}) = 1$. Let m be a minterm in $\hat{g} - \hat{d}_g$. Then $m \in f + d_f$, otherwise $d_f + \hat{d}_g + f\overline{\oplus}\hat{g} \neq 1$. As a result, $\hat{g} - \hat{d}_g \subseteq f + d_f$. In the same way, $(d_f + \hat{d}_g + f\overline{\oplus}\hat{g}) = 1$ implies $f - d_f \subseteq \hat{g} + \hat{d}_g$. Therefore, if $\mathcal{C}_x(d_f + \hat{d}_g + f\overline{\oplus}\hat{g}) = 1$, a match exists. If $f - d_f \subseteq \hat{g} + \hat{d}_g$ and $\hat{g} - \hat{d}_g \subseteq d_f + f$, then $\hat{g}f + d_f + \hat{d}_g = \hat{g} + d_f + \hat{d}_g$ and $\overline{f}\,\overline{\hat{g}} + d_f + \hat{d}_g = \overline{\hat{g}} + d_f + \hat{d}_g$. Therefore $(d_f + \hat{d}_g + f\overline{\oplus}\hat{g}) = 1$. ∎

We can organize equation (7.1) in a more computationally efficient way by using the result of the following lemma.

Lemma 7.2.3 *If $i \neq j$, $\mathcal{C}_{x_i}\mathcal{S}_{y_j}(x_j\overline{\oplus}y_j)h(x,y) = \mathcal{S}_{y_j}(x_j\overline{\oplus}y_j)\mathcal{C}_{x_i}h(x,y)$.*

Proof

$$\begin{aligned} C_{x_i}S_{y_j}(x_j\overline{\oplus}y_j)h &= C_{x_i}S_{y_j}(x_jy_jh_{y_j} + \overline{x}_j\overline{y}_jh_{\overline{y}_j}) \\ &= C_{x_i}(x_jh_{y_j} + \overline{x}_jh_{\overline{y}_j}) \\ &= (x_j(C_{x_i}h)_{y_j} + \overline{x}_j(C_{x_i}h)_{\overline{y}_j}) \\ &= S_{y_j}(x_j\overline{\oplus}y_j)C_{x_i}h. \end{aligned}$$

■

Lemma 7.2.4 *Let* $A_k = (y_1\overline{\oplus}x_1)(y_2\overline{\oplus}x_2)\ldots(y_m\overline{\oplus}x_m)$. *Then* $\mathcal{M}_k = C_x(S_y(A_k(d_f + d_g + f\overline{\oplus}g)))$ *can be expressed as*

$$\mathcal{M}_k = (C_{x_m}S_{y_m}(x_m\overline{\oplus}y_m)\ldots C_{x_1}S_{y_1}(x_1\overline{\oplus}y_1)(d_f + d_g + f\overline{\oplus}g))$$

Proof The statement of the lemma follows by induction and lemma 7.2.3. ■

Not all the possible assignments of variables y to x are required to check whether a matching exists. First we express necessary conditions for a matching to exist. Let $|f|$ represent the number of minterms in the function f. Once BDD's are built for functions f and g, then $|f|$ and $|g|$ can be easily found by traversing the corresponding BDD's only once. Given node n in the BDD of f with children nl and nr, the number of minterms in the function represented by n in the ordered BDD of f can be found if this number is known at nl and nr. We represent the difference between the variables n and nl in the variable ordering by l (if n appears right before nl, $l = 1$) and the difference between the variable of n and nr in the variable ordering by r. The number of onset points for the function at n is $|n| = 2^{l-1}|nl| + 2^{r-1}|nr|$. Initially, the number of minterms at *node 1* is set to 1 and *node 0* is set to 0. Also, if the root of the BDD is not the first variable in the ordering, we multiply the count at the root node by 2^k where k is the difference between the root node and the first variable in the ordering.

Theorem 7.2.5 *A matching between* f *and* g *exists under any variable assignment only if* $|f - d_f| \subseteq |g + d_g|$, $|\overline{f} - d_f| \subseteq |\overline{g} + d_g|$, $|g - d_g| \subseteq |f + d_f|$, *and* $|\overline{g} - d_g| \subseteq |\overline{f} + d_f|$. *In particular, if* $d_f = 0$ *and* $d_g = 0$, $|f| = |g|$.

Proof Each onset point of f must be mapped to an onset point or don't care point of g and each offset point of f must be mapped to an offset point or don't care point of g. If

$|f - d_f| > |g + d_g|$ some onset points in f cannot be mapped to any onset or don't care point of g. The proof is similar for other cases. ∎

Lemma 7.2.6 *A matching under the assignment $x_i = y_j$ exists only if $|f_{x_i} - d_{f_{x_i}}| \subseteq |g_{y_j} + d_{g_{y_j}}|$, $|\overline{f}_{x_i} - d_{f_{x_i}}| \subseteq |\overline{g}_{y_j} + d_{g_{y_j}}|$, $|g_{y_j} - d_{g_{y_j}}| \subseteq |f_{x_i} + d_{f_{x_i}}|$, $|\overline{g}_{y_j} - d_{g_{y_j}}| \subseteq |\overline{f}_{x_i} + d_{f_{x_i}}|$, $|f_{\overline{x}_i} - d_{f_{\overline{x}_i}}| \subseteq |g_{\overline{y}_j} + d_{g_{\overline{y}_j}}|$, $|\overline{f}_{\overline{x}_i} - d_{f_{\overline{x}_i}}| \subseteq |\overline{g}_{\overline{y}_j} + d_{g_{\overline{y}_j}}|$, $|g_{\overline{y}_j} - d_{g_{\overline{y}_j}}| \subseteq |f_{\overline{x}_i} + d_{f_{\overline{x}_i}}|$, and $|\overline{g}_{\overline{y}_j} - d_{g_{\overline{y}_j}}| \subseteq |\overline{f}_{\overline{x}_i} + d_{f_{\overline{x}_i}}|$. In particular, if $d_f = 0$ and $d_g = 0$, $|f_{x_i}| = |g_{y_j}|$ and $|f_{\overline{x}_i}| = |g_{\overline{y}_j}|$.*

Proof If $x_i = y_j$, each onset point of $(f_{x_i} - d_{f_{x_i}})$ must be mapped to a point in $(g_{y_j} + d_{g_{y_j}})$, therefore, $|f_{x_i} - d_{f_{x_i}}| \subseteq |g_{y_j} - d_{g_{y_j}}|$. Other cases can be proved in the same way. ∎

Corollary 7.2.7 *A matching under the assignment $\overline{x}_i = y_j$ exists only if $|f_{x_i} - d_{f_{x_i}}| \subseteq |g_{\overline{y}_j} + d_{g_{\overline{y}_j}}|$, $|\overline{f}_{x_i} - d_{f_{x_i}}| \subseteq |\overline{g}_{\overline{y}_j} + d_{g_{\overline{y}_j}}|$, $|g_{\overline{y}_j} - d_{g_{\overline{y}_j}}| \subseteq |f_{x_i} + d_{f_{x_i}}|$, $|\overline{g}_{\overline{y}_j} - d_{g_{\overline{y}_j}}| \subseteq |\overline{f}_{x_i} + d_{f_{x_i}}|$, $|f_{\overline{x}_i} - d_{f_{\overline{x}_i}}| \subseteq |g_{y_j} + d_{g_{y_j}}|$, $|\overline{f}_{\overline{x}_i} - d_{f_{\overline{x}_i}}| \subseteq |\overline{g}_{y_j} + d_{g_{y_j}}|$, $|g_{y_j} - d_{g_{y_j}}| \subseteq |f_{\overline{x}_i} + d_{f_{\overline{x}_i}}|$, and $|\overline{g}_{y_j} - d_{g_{y_j}}| \subseteq |\overline{f}_{\overline{x}_i} + d_{f_{\overline{x}_i}}|$.*

From now on, we concentrate on the use of Boolean matching in technology mapping where we try to match a sub-function with don't cares in the network with a library function which has no don't cares ($d_g = 0$).

7.3 Boolean Matching for Technology Mapping

The objective of a technology mapper is to map a circuit into a set of gates in the library. The given circuit is first decomposed into a set of 2-input gates and then into a set of disjoint trees. As in [42, 26, 64], we use dynamic programming to map each tree into a set of library gates. The trees are mapped in topological order; each tree is mapped after all its fanin trees. Mapping is a two step process. In the first step, called *matching*, we find the minimum cost matching for the root of the tree. In the second step, called *gate assignment*, we implement the logic function of the tree in terms of library gates as determined in the matching phase.

The first phase of technology mapping is to traverse the target tree bottom-up from the inputs. At each node, all possible functions up to a given number of inputs having that node as output are considered. These functions are called *cluster functions*; their corresponding subgraphs are called *clusters* [49]. In our formulation, a cluster is represented

by a root node and a set of leaf nodes (cutset of nodes) separating the root node from the rest of the network. We use an iterative algorithm for cluster generation. It starts with a cluster consisting only of the root node, and generates new clusters by expanding every cluster. Expansion of a cluster is done by removing each node of the cutset one at a time and adding its fanin nodes to it. If some of the clusters generated in this process have been generated before, or contain more nodes than the maximum number of inputs in any gate of the library, they are simply discarded. Each iteration expands the clusters generated in the previous iteration only. Cluster generation is stopped after an iteration that does not produce more clusters.

During gate assignment we build a new network that contains the best map at each tree. At each tree, we need to choose the phase of the root node of the tree. The less costly phase in terms of area is currently chosen unless the root node is a primary output where the positive phase is chosen. The penalty for using the phase that is not implemented is the cost of an inverter. After all the trees in the network are mapped, we traverse these trees in reverse order, and check what phase of the root is used in each tree. If the implemented phase in the new network for a particular tree is always inverted before it is used by its fanout trees, we switch to the other phase of that tree to reduce cost.

The matching problem is to find any library function that can be matched with a cluster function. The correspondence between the inputs of the cluster function and the library gate is sought first, then one checks if the functions are equivalent under such condition. In the presence of local don't cares the matching problem can be formulated as follows. Let $f(x_1, x_2, \ldots, x_n)$ be a cluster function with local don't-care $d(x_1, x_2, \ldots, x_n)$, and $g(y_1, y_2, \ldots, y_m)$ be a library function where $m \leq n$. If $m > n$, some of the inputs of the library gate must be set to 0, 1, or tied together. Such gates can be added to the library in a preprocessing step. For architectures composed of particular types of gates where the case $m > n$ is important, special techniques can be devised to do Boolean matching. If $m < n$, a matching exists only if the support f can be reduced using the given don't care set. This is unlikely in a well-optimized circuit because most redundant connections are already removed. For each cluster function we generate all the possible supports and try to match each one with a library gate of the same number of variables.

7.3.1 Generating all Supports

Let f be a cluster function with don't care d. The objective is to generate a new function $\tilde{f}$ with don't care set $\tilde{d}$ for each possible support of the cluster function f. The circuits given for mapping are usually well optimized and do not have many redundancies; therefore, we expect a few possible supports by which f can be represented. After generating a function $\tilde{f}$ for each possible support, $\tilde{f}$ is compared to all the library gates with the same number of inputs for a possible match.

Let $\mathbf{x} = \{x_1, \ldots, x_m\}$ be the set of variables in f. $\mathbf{x}$ is called the support of f.

Lemma 7.3.1 *A support $\mathbf{x}_i \subseteq \mathbf{x}$ is a possible support for f if and only if $\mathcal{S}_{\overline{\mathbf{x}}_i}(f-d) \subseteq (f+d)$ where $\overline{\mathbf{x}}_i = \mathbf{x} - \mathbf{x}_i$ (This means, $\overline{\mathbf{x}}_i$ is the set of all the variables in $\mathbf{x}$ that are not in $\mathbf{x}_i$.).*

Proof Assume $\mathcal{S}_{\overline{\mathbf{x}}_i}(f-d) \subseteq (f+d)$ and let $\tilde{f}_i = \mathcal{S}_{\overline{\mathbf{x}}_i}(f-d)$. Because $f - d \subseteq \mathcal{S}_{\overline{\mathbf{x}}_i}(f-d)$, it follows that $f - d \subseteq \tilde{f}_i \subseteq f + d$. Therefore $\mathbf{x}_i$ is a possible support for f. On the other hand if $f - d \subseteq \tilde{f}_i \subseteq f + d$ and $\mathbf{x}_i$ is the support of $\tilde{f}_i$, it follows that $(f-d)_{x_k} \subseteq \tilde{f}_i$ and $(f-d)_{\overline{x}_k} \subseteq \tilde{f}_i$ for every variable $x_k \in \overline{\mathbf{x}}_i$. As a result $\mathcal{S}_{x_k}(f-d) \subseteq \tilde{f}_i$ and because this is true for each $x_k \in \overline{\mathbf{x}}_i$

$$\mathcal{S}_{\overline{\mathbf{x}}_i}(f-d) \subseteq \tilde{f}_i.$$

Therefore $\mathcal{S}_{\overline{\mathbf{x}}_i}(f-d) \subseteq f + d$. ∎

Lemma 7.3.2 *In a similar way, a support $\mathbf{x}_i \subseteq \mathbf{x}$ is a possible support for f if and only if $(f-d) \subseteq \mathcal{C}_{\overline{\mathbf{x}}_i}(f+d)$.*

Proof Assume $(f-d) \subseteq \mathcal{C}_{\overline{\mathbf{x}}_i}(f+d)$ and let $\tilde{f}_i = \mathcal{C}_{\overline{\mathbf{x}}_i}(f+d)$. It follows that

$$f - d \subseteq \tilde{f}_i \subseteq f + d$$

and $\mathbf{x}_i$ is a possible support for f. On the other hand if $f - d \subseteq \tilde{f}_i \subseteq f + d$ and $\mathbf{x}_i$ is the support of $\tilde{f}_i$, it follows that $\tilde{f}_i \subseteq \mathcal{C}_{x_k}(f+d)$ for each $x_k \in \overline{\mathbf{x}}_i$ and thus

$$\tilde{f}_i \subseteq \mathcal{C}_{\overline{\mathbf{x}}_i}(f+d).$$

Therefore $(f-d) \subseteq \mathcal{C}_{\overline{\mathbf{x}}_i}(f+d)$. ∎

Lemma 7.3.3 *If $\tilde{f}_i$ with support $\mathbf{x}_i \subseteq \mathbf{x}$ satisfies $f - d \subseteq \tilde{f}_i \subseteq f + d$, it also satisfies $\mathcal{S}_{\overline{\mathbf{x}}_i}(f-d) \subseteq \tilde{f}_i \subseteq \mathcal{C}_{\overline{\mathbf{x}}_i}(f+d)$.*

Proof Applying $C_{\overline{\mathbf{x}}_i}$ to $\tilde{f}_i \subseteq (f+d)$ and considering the fact that $\tilde{f}_i$ is independent of variables in $\overline{\mathbf{x}}_i$, it gives

$$\begin{aligned} C_{\overline{\mathbf{x}}_i}\tilde{f}_i &\subseteq C_{\overline{\mathbf{x}}_i}(f+d) \\ \tilde{f}_i &\subseteq C_{\overline{\mathbf{x}}_i}(f+d). \end{aligned}$$

In the same way, if we apply $S_{\overline{\mathbf{x}}_i}$ to $(f-d) \subseteq \tilde{f}_i$, it gives

$$\begin{aligned} S_{\overline{\mathbf{x}}_i}(f-d) &\subseteq S_{\overline{\mathbf{x}}_i}\tilde{f}_i \\ S_{\overline{\mathbf{x}}_i}(f-d) &\subseteq \tilde{f}_i. \end{aligned}$$

■

The maximum don't care set for any function $\tilde{f}$, $f-d \subseteq \tilde{f} \subseteq f+d$, with support $\mathbf{x}_i$ is $\tilde{d} = C_{\overline{\mathbf{x}}_i}(f+d) - S_{\overline{\mathbf{x}}_i}(f-d)$. Because we do Boolean matching, it is enough to generate one function with its maximum don't care set for each possible support.

The algorithm shown in Figure 7.1 is used to generate all the possible supports for a cluster function f. The original arguments given to *generate_support* are $f_l = f-d$, $f_h = f+d$, *vars* which is all the variables in f and d (this is also saved as a possible support for f), and *start* = 0. To check whether the size of support can be reduced by removing variable x_i, the condition of Lemma 7.3.1 is used. If $S_{x_i} f_l \subseteq f_h$, the new support is saved and also used to generate other supports which exclude x_i.

Other techniques have been recently suggested for generating all possible supports of a function [75, 30]. We are investigating these.

7.3.2 Boolean Matching Algorithm

The algorithm for finding the existence of a match between a library gate $g(y_1, \ldots, y_m)$ and a cluster function $f(x_1, \ldots, x_m)$ with don't care set $d(x_1, \ldots, x_m)$ is shown in Figure 7.2. f and g have the same number of inputs. The argument M is originally set to $M = d + f \overline{\oplus} g$. The argument i shows the variable in f for which a match is sought. i is set to 0 originally. Before calling *boolean_match*, we check the necessary condition given by Theorem 7.2.5. If that condition is not satisfied, f and g cannot be matched. Each input x_i of f must be matched with an input y_j of g.

x_i can be equal to y_j if the necessary conditions as given by Lemma 7.2.6 are satisfied. If they are not satisfied, $\overline{x}_i = y_j$ is tried. If that is not possible either, y_j is not a

```
function generate_support(f_l, f_h, vars, start)
begin
    for (i = start; i < number(vars); i++) begin
        x_i = vars(i)
        if (S_{x_i} f_l ⊆ f_h) begin
            newvars(k) = vars(k) for k < i
            newvars(k) = vars(k + 1) for k ≥ i
            save newvars as a possible support
            newf_l = S_{x_i} f_l
            newf_h = C_{x_i} f_h
            generate_support(newf_l, newf_h, newvars, i)
        end
    end
end
```

Figure 7.1: Generating Supports

```
function boolean_match(f, d, g, i, M)
begin
  if (M = 0) return match_not_found
  if (M = 1) return match_found

  x_i = ith variable in f
  for each variable y_j of g not matched yet begin
    if y_j is symmetric to a y_k already tested
      continue
    /* check the necessary conditions for x_i = y_j */
    if ((|f_{x_i} - d_{x_i}| ≤ |g_{y_j}|) and (|f̄_{x_i} - d_{x_i}| ≤ |ḡ_{y_j}|) and
      (|f_{x̄_i} - d_{x̄_i}| ≤ |g_{ȳ_j}|) and (|f̄_{x̄_i} - d_{x̄_i}| ≤ |ḡ_{ȳ_j}|)) begin
      newM = C_{x_i} S_{y_j}(x_i ⊕̄ y_j) M
      (newf, newd, newg) = choose (f_{x_i}, d_{x_i}, g_{y_j}) or (f_{x̄_i}, d_{x̄_i}, g_{ȳ_j})
      if (boolean_match(newf, newd, newg, i + 1, newM) == match_found)
        return match_found
    end
    /* check the necessary conditions for x̄_i = y_j */
    if ((|f_{x_i} - d_{x_i}| ≤ |g_{ȳ_j}|) and (|f̄_{x_i} - d_{x_i}| ≤ |ḡ_{ȳ_j}|) and
      (|f_{x̄_i} - d_{x̄_i}| ≤ |g_{y_j}|) and (|f̄_{x̄_i} - d_{x̄_i}| ≤ |ḡ_{y_j}|)) begin
      newM = C_{x_i} S_{y_j}(x_i ⊕ y_j) M
      (newf, newd, newg) = choose (f_{x_i}, d_{x_i}, g_{ȳ_j}) or (f_{x̄_i}, d_{x̄_i}, g_{y_j})
      if (boolean_match(newf, newd, newg, i + 1, newM) == match_found)
        return match_found
    end
  end
  return match_not_found
end
```

Figure 7.2: Boolean Matching

possible match for x_i and is skipped. If no input of g can be set equal to x_i, f and g cannot be matched.

7.3.3 Symmetries

If the switching of two inputs of a gate has no effect on the function of that gate, those two inputs are symmetric. Most gates in the library have many symmetries. We find all such symmetries for all the gates in the library in a preprocessing step. This step is made easier by the fact that we have no don't cares for the gates in a library. For example, gate g might have two inputs y_k and y_j which are symmetric. If $x_i = y_k$ is not possible, then clearly x_i cannot be set equal to y_j either and is skipped. Two inputs y_i and y_j of a function g are symmetric if $g_{y_i\overline{y}_j} \equiv g_{\overline{y}_i y_j}$ as shown in [27]. Furthermore if y_i is symmetric with y_j and y_j is symmetric with y_k, y_i is also symmetric with y_k, since symmetry is transitive when there are no don't cares.

There is another kind of symmetry which can be used to speed up Boolean matching. Given a library gate $g = y_1y_2 + y_3y_4 + y_5y_6$, y_1 is symmetric with y_2, y_3 is symmetric with y_4, and y_5 is symmetric with y_6. If we switch the variables y_3 and y_4 with y_1 and y_2, we get exactly the same function. In this example, y_1y_2 are *group symmetric* with y_3y_4 and y_5y_6. Therefore if a variable x_i cannot be matched with y_1, it cannot be matched with any other variable in g and no matching exists. On the other hand, if y_1 has been matched with some other variable x_j and x_i cannot be matched with y_2, we still need to try $x_i = y_3$. To find group symmetries for a function g, we first put all the inputs that are symmetric in one group. Every two groups with the same number of inputs (greater than 1) are compared. For example, the comparison is done as follows for the function $g = y_1y_2 + y_3y_4 + y_5y_6$. We build a BDD for g with the following ordering order(y_1) = 1, order(y_2) = 2, order(y_3) = 3, order(y_4) = 4, order(y_5) = 5, order(y_6) = 6. To check the group symmetry of y_1, y_2 with y_3, y_4, we build a new BDD with the ordering order(y_1) = 3, order(y_2) = 4, order(y_3) = 1, order(y_4) = 2, order(y_5) = 5, order(y_6) = 6 and check if it is equal to the original BDD for g. Group symmetry is a transitive property like symmetry. Symmetries and group symmetries are found for each gate in the library and stored in a preprocessing step.

7.3.4 Heuristic for Assigning Inputs

Once we find a variable y_j that can be set equal to x_i, we reduce the size of the matching problem at hand by one variable and try to match the rest of the variables in f and g. Using the result of lemma 7.2.4, we compute $newM = C_{x_i} S_{y_j}(x_i \overline{\oplus} y_j)M$. A necessary and sufficient condition for the matching to exist is that $newM = 1$ after all the variables are matched as in lemma 7.2.2.

The necessary condition given by lemma 7.2.6 to match x_i and y_j requires computing both f_{x_i} and $f_{\overline{x}_i}$ and comparing them with g_{y_j} and $g_{\overline{y}_j}$ respectively. When we match a second pair of variables $x_l = y_k$, we need to compute $f_{x_i x_l}, f_{x_i \overline{x}_l}, f_{\overline{x}_i x_l}$, and $f_{\overline{x}_i \overline{x}_l}$ and compare it with $g_{y_j y_k}, g_{y_j \overline{y}_k}, g_{\overline{y}_j y_k}$, and $g_{\overline{y}_j \overline{y}_k}$. This number grows exponentially as we match more variables.

When we set $x_i = y_j$, the pairs (f_{x_i}, g_{y_j}) and $(f_{\overline{x}_i}, g_{\overline{y}_j})$ must be matched respectively. We only choose one of the pairs (f_{x_i}, g_{y_j}) and $(f_{\overline{x}_i}, g_{\overline{y}_j})$ to be passed to the next step of the algorithm to be used for checking necessary conditions as given by lemma 7.2.6.

For example, let $f = x_1 x_2 x_3$ and $g = y_1 y_2 y_3$. First we try to find a match for variable x_1. $x_1 = y_1$ satisfies the necessary condition ($f_{x_1} = x_2 x_3, f_{\overline{x}_1} = 0, g_{y_1} = y_2 y_3$, and $g_{\overline{y}_1} = 0$). The pair $(f_{\overline{x}_1} = 0, g_{\overline{y}_1} = 0)$ cannot give us any further information because the necessary conditions are always satisfied for this pair irrespective of what variables are matched. On the other hand, the pair (f_{x_1}, g_{y_1}) contains all the information that we need. The following heuristic is used to choose one of the two pairs. If $(f_{x_i} - d_{x_i} = 0)$, or $(g_{y_j} = 1)$, the necessary conditions as given in lemma 7.2.6 are always satisfied. Therefore the other pair $(f_{\overline{x}_i}, g_{\overline{y}_j})$ is used to guide the matching. This same principle is used to check the other pair. If the above check is not enough, we choose either of f_{x_i} or $f_{\overline{x}_i}$ which has the larger difference between the number of onset points and offset points. The difference between the onset and offset points is computed as follows, $absolute\_value(|f_{x_i} - d_{x_i}| - |\overline{f}_{x_i} - d_{x_i}|)$ and $absolute\_value(|f_{\overline{x}_i} - d_{\overline{x}_i}| - |\overline{f}_{\overline{x}_i} - d_{\overline{x}_i}|)$.

This algorithm runs in linear time in the number of input variables for a library gate with one minterm in the onset or offset (AND, OR, NAND, NOR).

7.4 Don't Care Computation

The network is first decomposed into a set of trees. We compute compatible external plus observability don't cares at each of the nodes of the network as explained in Chapter 5. These trees are sorted in topological order. Each tree is mapped after all its fanin trees have been already mapped. Image computation techniques are used to find local don't cares at the leaves of the tree that is being mapped. The leaves of the tree correspond to primary inputs or roots of other trees that have been already mapped; therefore, the functions at these roots are fixed. Given the external plus observability don't care set $D_t^g(\mathbf{x})$ for the root of the tree in terms of primary inputs of the original circuit and global functions $g_1(\mathbf{x}),\ldots g_m(\mathbf{x})$ at each leaf of the tree in terms of the primary inputs, the local don't care set for the tree in terms of its leaf variables $t_1, t_2, \ldots, t_m$ is

$$D_t^l(\mathbf{t}) = \overline{S_{\mathbf{x}}[\overline{D}_t^g(\mathbf{x})(g_1(\mathbf{x})\overline{\oplus}t_1)\ldots(g_m(\mathbf{x})\overline{\oplus}t_m)]}.$$

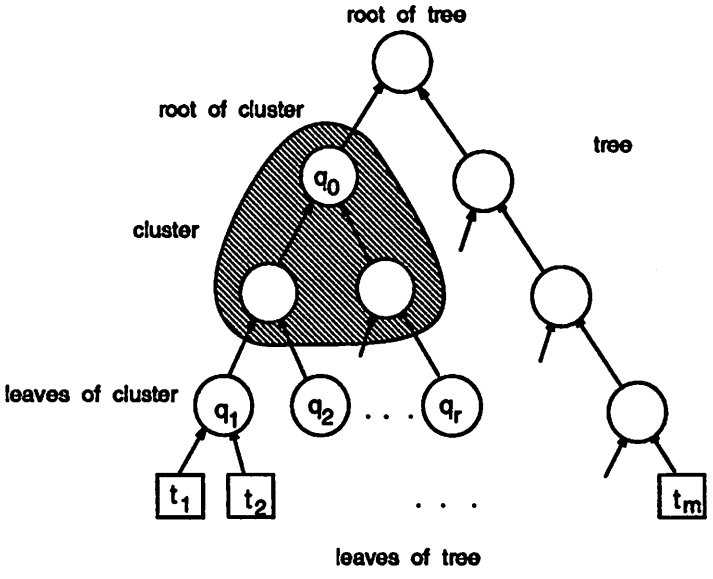

Figure 7.3: Cluster Functions

To compute D_t^l, we first build BDD's corresponding to global functions (functions in terms of primary inputs) at each leaf of a tree and build a BDD representing D_t^g. The functions $g_1,\ldots,g_m$ are cofactored (generalized cofactor) with respect to $\overline{D}_t^g$ and the recursive image computation method in Figure 5.8 is then used to find all the reachable points

in the space of the leaves of the tree. The complement of the reachable set of points gives the local don't care set for the tree in terms of its leaf variables.

The tree itself is considered a network where its local don't cares are external don't cares or input combinations that never occur. A dynamic programming approach is used to find the best match for the tree. Nodes of the tree are visited in depth first order starting from the leaves of the tree. At each node q_o we find the best positive and negative matches and record them. This is done by looking at all possible clusters with less than a fixed number of inputs (the maximum number of inputs of a gate in the library) rooted at q_o and matching them with the gates in the library. The cost at the node for a cluster is the cost to implement the cluster itself plus the cost to implement the functions at the leaves of the cluster. Both positive and negative phase costs at each of the leaves are available. The clusters which give the best cost for positive and negative phase matches at q_o are recorded.

The best positive and negative phase matches at each node of the network are sought in topological order from the inputs. Figure 7.3 shows an example where we compute a local don't care set for a cluster with root q_o and leaves $q_1, \ldots, q_r$ and use the don't care set to find a match for the cluster. To compute the local don't cares for a cluster function within a tree as shown in Figure 7.3, we use image computation techniques again. We build BDD's for each leaf q_i of a cluster in terms of the leaves of the tree $\mathbf{t} = (t_1, \ldots, t_m)$. Before matching the cluster rooted at q_o, the best matches for both positive and negative phases of each leaf q_i of the cluster have been found. Because don't cares are used, the positive and negative phase functions at the leaves of a cluster are not necessarily complements of each other. While matching to a particular gate in the library, we could choose either of the two phases for a particular leaf q_i. We need a local don't care set for the cluster which is valid irrespective of the phase chosen for a particular q_i; otherwise separate don't cares must be computed for each possible phase assignment for the q_i's.

Let $f_1^p, \ldots, f_r^p$ be the functions corresponding to positive phase and $f_1^n, \ldots, f_r^n$ be the functions corresponding to negative phase at the leaves of a cluster. We know that f_i^p is not necessarily equal to $\overline{f_i^n}$. The set under which the two phase functions are different is $d_i = f_i^p \overline{f_i^n} + \overline{f_i^p} f_i^n$. We compute a local don't care set which is valid for any function allowed by (f_i^p, d_i).

Lemma 7.4.1 *A local don't care set valid for both phases of the functions at the leaves of*

a cluster is

$$D_q^l(\mathbf{q}) = \overline{S_t(\overline{D}_t^l(\mathbf{t})(q_1 \overline{\oplus} f_1^p(\mathbf{t}) + d_1(\mathbf{t})) \ldots (q_r \overline{\oplus} f_r^p(\mathbf{t}) + d_r(\mathbf{t})))}. \quad (7.2)$$

Proof The term $(q_1 \overline{\oplus} f_1^p(\mathbf{t}) + d_1(\mathbf{t})) \ldots (q_r \overline{\oplus} f_r^p(\mathbf{t}) + d_r(\mathbf{t}))$ is a Boolean relation which gives all the $\mathbf{q}$ combinations for a particular input $\mathbf{t}_j$ The given flexibility d_i at node q_i is captured by this Boolean relation. The term $\overline{D}_t^l(\mathbf{t})$ is all $\mathbf{t}$ combinations that are care points. Therefore, the term

$$S_t(\overline{D}_t^l(\mathbf{t})(q_1 \overline{\oplus} f_1^p(\mathbf{t}) + d_1(\mathbf{t})) \ldots (q_r \overline{\oplus} f_r^p(\mathbf{t}) + d_r(\mathbf{t})))$$

gives all the $\mathbf{q}$ combinations that are possible under the care points and the complement of this set is a valid local don't care set. ∎

Notice that the above computation gives a local don't care set which is valid for any function allowed by (f_i^p, d_i) at node q_i. However, we only need a local don't care set which is valid for f_i^p and f_i^n. The condition which gives either function f_i^p or f_i^n but no other function at q_i cannot be represented by a Boolean relation. We do not know of an efficient way to compute a don't care set which is only valid for the two functions f_i^p and f_i^n, although this don't care set will be larger than that computed from equation 7.2.

Lemma 7.4.2 *If only the external don't care set computed for the tree D_t^l is used, but not the observability don't care set within the tree,*

$$D_q^l(\mathbf{q}) = \overline{S_t(\overline{D}_t^l(\mathbf{t})(q_1 \overline{\oplus} f_1^p(\mathbf{t})) \ldots (q_r \overline{\oplus} f_r^p(\mathbf{t}))} \quad (7.3)$$

is valid for both phases of functions at the leaves of the cluster.

Proof The difference between $\overline{f}_i^n$ and f_i^p is contained in D_t^l ($f_i^p \overline{f}_i^n + \overline{f}_i^p f_i^n \subseteq D_t^l$). Therefore, $d_i \overline{D}_t^l = 0$ and Equation 7.2 reduces to Equation 7.3. ∎

If no observability don't cares are used within a tree, the local don't cares for a cluster can be computed as before. The functions $f_1^p, \ldots, f_r^p$ are cofactored (generalized cofactor) with respect to $\overline{D}_t^l$ and the recursive image computation method in Figure 5.8 is then used to find all the reachable points in the space of the leaves of the cluster. The complement of the reachable set of points gives the local don't care set for the cluster in terms of its leaf variables.

It must be also mentioned that the choice of the functions at the leaves of a cluster affects the local don't cares of that cluster. Hence, dynamic programming might not give the best result for the mapping of a tree when observability don't cares are used within a tree. The choice of the best function at the leaves of a cluster may shrink the local don't care set for the cluster and thus the final result may not be the best match for the circuit. We believe this is not very likely in practice.

In a circuit with large trees, there are usually many clusters. Computing local don't cares for all such clusters is a costly operation.

7.5 Library Organization

Before technology mapping, a setup phase is used to process gates in the library and generate particular data structures called *NUTS*. The term *NUT* is the abbreviation for Negative Unate Transform introduced in [49]. All the gates in a NUT are equivalent to a NUT representative in the sense that the function of each gate can be obtained by inverting some of the inputs of the NUT representative. The NUT structure reduces the number of calls to the Boolean matching algorithm. Finding the best match between a cluster function and the set of gates in the library is therefore reduced to the use of the matching algorithm on the cluster function and all the NUT representatives with the same number of inputs. The matching with the remaining gates in a NUT is directly derived from the assignment information computed during the setup phase.

In the groups we build, we also consider the inversion of the outputs of the gates. This grouping is possible because our matching algorithm considers the matching with both phases of the input nodes at the same time. Instead of computing the negative unate transforms of the input variables as in [49], we use the Boolean matching algorithm to place each gate in its corresponding NUT structure. The setup phase parses the library, reading one gate at a time. A gate is added to a NUT if it or its complement matches the NUT representative. If a gate does not match any of the existing NUT's, then a new NUT is created with that gate as its representative. Symmetries and group symmetries are also computed for each representative at this time to speed up matching with class representative.

Example:

The 2-input functions NOR, NAND, AND and OR are in the same NUT and any of them

can be the representative. Let OR function be the class representative and let f be a cluster function for which a match is sought. If f matches the OR gate, either an OR gate, or a NAND gate can be used to implement f. But the inputs to NAND gate have the opposite phase of the inputs to NOR gate. The cost at the root of the cluster is equal to the cost of the gate plus the cost of inputs to the gate which can be different for the negative and positive phase inputs. The gate (OR or NAND) which gives a lower cost is chosen to represent the positive phase at the root of the cluster. If a match for f exists, we know already that AND and NOR gates are matches for the complement of f. If a match for f does not exist we compare the complement of f with the NUT representative. If the complement of f matches the NUT representative (OR gate here), one of NAND or OR gates is chosen to represent the complement of f and one of the NOR or AND gates is chosen to represent f. If the complement of f does not match the class representative either, f cannot ne matched with this NUT. Therefore, matching both phases of the function at a node with a NUT structure requires at most two calls to the Boolean matching algorithm.

7.6 Results

We ran the new technology mapping algorithm on a set of benchmarks chosen from MCNC and ISCAS combinational circuits and compared the results with technology mapping for area in SIS. Table 7.1 shows the results for combinational circuits without any external don't cares. These circuits are well optimized before technology mapping, using the *rugged script* discussed in Chapter 6. The MCNC library *lib2* is used for the mapping. The column **start** shows the literal count in factored form for each unmapped but optimized circuits. The columns **SIS**, **no_dc**, **tree_dc**, and **full_dc** show the area of mapped circuits. We divide numbers given by the mapper by 464 (half the area of the smallest inverter) to get round small numbers.

As shown in the table, considerable improvements are obtained for some circuits by just using Boolean matching without any don't cares (**no_dc**). For these circuits, we get on average 8 percent improvement in area compared to technology mapping in SIS while spending 3.9 times as much time. The best improvement is obtained (25%) for *C6288*. The column **tree_dc** shows the area obtained when don't cares are computed only for the leaves of each tree. The CPU times and the circuit areas are almost the same as the case with no don't cares. The column **full_dc** shows the result obtained by computing don't

circuit	start	SIS	CPU	no_dc	CPU	tree_dc	CPU	full_dc	CPU
C432	218	437	5	398	31	397	157	381	574
C880	414	783	10	734	49	734	59	734	343
C1355	552	914	11	738	8	738	87	724	1184
C1908	535	933	11	810	27	810	118	793	1774
C2670	748	1339	20	1236	103	–	–	–	–
C3540	1283	2269	36	2176	213	–	–	–	–
C5315	1763	3055	45	3025	173	3025	229	3003	2285
C6288	3367	5453	68	4070	111	–	–	–	–
C7552	3022	4076	58	3690	190	–	–	–	–
z4ml	43	86	1	69	3	69	6	68	69
f51m	80	150	2	148	6	148	9	112	53
apex5	768	1473	19	1362	91	1361	127	1355	1180
apex6	732	1390	19	1345	97	1341	120	1336	882
alu4	102	200	2	196	10	196	11	180	103
rot	664	1283	16	1270	62	1267	74	1255	466
des	4214	5947	137	5698	596	5501	789	5498	11926

Table 7.1: Boolean Matching for Technology Mapping

start: number of literals in factored form for the optimized circuits
SIS: mapped using map -s in SIS
no_dc: mapped using boolean matching in SIS without don't cares
tree_dc: mapped using boolean matching in SIS with DC computed at the leaves of each tree.
full_dc: mapped using boolean matching in SIS with DC computed for each cluster
CPU: in seconds on a IBM Risc System/6000 530

cares for each cluster in the trees. We do not use observability don't cares within a tree for this computation. The local don't cares computed for each cluster are based on equation 7.3. The times spent for mapping are an order of magnitude more than SIS but there is on average 12 percent improvement in the final area of the mapped circuits. Although the time spent is substantially more than the time for tree matching, it is comparable to the time spent for circuit optimization. The entries in the table that do not have results correspond to circuits for which BDD's could not be built or the size of BDD's were too large to be used for don't care computation.

If the benchmark circuits are not optimized first, the improvement over the technology mapping in SIS is very substantial. This is because using local don't cares and Boolean matching removes redundancies so a much stronger optimization on each circuit is

obtained. Even though the results on unoptimized circuits are better, they are inferior to the results obtained after running rugged script on each circuit. When a circuit is optimized using the rugged script, the structure of the circuit is already well established. In most cases, each intermediate node of the circuit becomes a separate tree after tree decomposition and is mapped separately. Because of this, don't cares do not give substantial improvement as it is clear from the results. However, **full_dc** does have an effect. We believe this is due to the don't cares arising internally in the tree from the partially mapped structure. These were not available when the rugged script was applied.

7.7 Conclusion

We have presented a new Boolean matching algorithm that can use don't cares and symmetries efficiently. We have applied this algorithm to technology mapping and have shown that the results of the mapper can be improved compared to tree matching techniques. The computation of local don't cares for each cluster function are discussed and techniques for such computations are presented. We have also organized the library of gates in an efficient way that reduces the number of times the Boolean matching algorithm is used. We developed ways to reduce the number of clusters generated in each tree and also more efficient don't care computation techniques to speed up the Boolean mapper. The same techniques can be used for delay optimization and layout driven technology mapping.

Chapter 8

Conclusions

We have shown that a Boolean relation is a general form for expressing the input-output behavior of a combinational Boolean network. This relation is called an observability relation and can be represented by a node attached to the outputs and inputs of the network, where the function at the node is the characteristic function of the relation. The modified network is called the observability network. In practice, external don't cares which are a subset of the observability relation for the circuit, are usually used to express input-output behavior of a Boolean network because they are computationally less expensive to use for network optimization.

A Boolean network may be decomposed into both single and multiple output nodes. We showed that don't care conditions computed from the observability network at each single output node of a Boolean network give maximum flexibility for implementing that node. Don't cares are not sufficient for multi-output nodes. The full flexibility for implementing multi-output nodes is captured by Boolean relations. It was shown that the concept of compatibility applies to both single and multiple output nodes of a Boolean network, and that compatible don't cares and compatible Boolean relations can be computed.

We have developed algorithms for computing the flexibility for implementing single output nodes, called local don't cares. The flexibility at each output of the network is given by external don't cares. We have given techniques to compute local don't cares at each node from observability and external don't care sets and to use these local don't cares to simplify the local function at each node. Our experimental results show the effectiveness of this approach in terms of reducing logic and removing redundancies in the network.

Techniques are also presented for computing maximal and compatible observability

relations for multi-output nodes of a Boolean network. The practicality of these techniques depends on how efficiently one can manipulate Boolean relations. More work is needed for decomposing a circuit into multi-output nodes. In addition, algorithms are required to map multi-output nodes of a Boolean network into multi-output gates in the library. In the single output case, one decomposes the circuit into NAND or NOR trees and maps each tree individually. We do not know of any approach for the multi-output case, and this needs to be investigated.

One needs to find a way of describing sequential flexibilities. Sequential flexibility of a circuit must allow any possible encoding of that circuit. It is known that in general, the maximum flexibility for implementing a sequential network cannot be represented with a Boolean relation. It might be possible to represent such flexibilities with a set of Boolean relations. If a circuit is decomposed into a set of sequential multi-output circuits, techniques must be devised to find maximum and compatible flexibility for implementing such sequential multi-output nodes. In addition, more work is needed to find a set of transformations that can use sequential flexibilities efficiently.

A new algorithm is presented for Boolean matching, and it is applied to technology mapping for a network of single output nodes.. This algorithm allows the use of don't cares. A circuit is first decomposed into a set of trees and each tree is mapped into a set of gates in the library. Local don't cares are computed for each cluster within a tree and used to find the best match for that cluster in the library. The Boolean matching algorithm is general and can be used for delay optimization and mapping to most technologies.

The final result of technology mapping is dependent on the tree decomposition of the circuit. More work needs to be done to get a good tree decomposition. If external don't cares, or observability relations are present, any optimization which uses these conditions may improve the quality of the final circuit considerably.

Bibliography

[1] P. Abouzeid, K. Sakouti, G. Saucier, and F. Poirot. Multilevel synthesis minimizing the routing factor. In *27th ACM/IEEE Design Automation Conference*, pages 365–368, Orlando, June 1990.

[2] V. D. Agrawal, K-T. Cheng, and P. Agrawal. CONTEST: A Concurrent Test Generator for Sequential Circuits. In *Proceedings of the 25th Design Automation Conference*, pages 84–89, June 1988.

[3] A. V. Aho, M. Ganapathi, and S. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[4] Act 1 Family Gate Arrays. Design reference manual.

[5] A. Aziz. private communication, 1991.

[6] K. Bartlett, R. K. Brayton, G. D. Hachtel, R. M. Jacoby, C. R. Morrison a nd R. L. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. Multi-level Logic Minimization Using Implicit Don't Cares. In *IEEE Transactions on CAD*, pages 723–740, June 1988.

[7] K. A. Bartlett, G. D. Bostick, G. D. Hachtel, R. M. Jacoby, P. H. Lightner, P. H. Moceyunas, C. R. Morrison, and Ravenscroft D. BOLD: A Multiple-Level Logic Optimization System. In *IEEE International Conference on Computer-Aided Design*, November 1987.

[8] M. Beardslee, C. Kring, R. Murgai, H. Savoj, R.K. Brayton, and A. Sangiovanni-Vinventelli. SLIP: A Software Environment for System Level Interactive Partitioning.

In *IEEE International Conference on Computer-Aided Design*, pages 280–283, November 1989.

[9] K. L. Brace, R. E. Bryant, and R. L. Rudell. Efficient Implementation of a BDD Package. In *27th ACM/IEEE Design Automation Conference*, June 1990.

[10] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: Multiple-Level Logic Optimization System. In *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, pages 1062–1081, November 1987.

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

[12] R. K. Brayton and F. Somenzi. Boolean Relations and the Incomplete Specification of Logic Networks. In *VLSI'89*, August 1989.

[13] R.K. Brayton and C. McMullen. The decomposition and factorization of Boolean expressions. In *The International Symposium on Circuits and Systems*, pages 49–54, May 1982.

[14] R. E. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[15] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential Circuit Verification Using Symbolic Model Checking. In *27th ACM/IEEE Design Automation Conference*, Orlando, June 1990.

[16] J. Burns, A. Casotto, M. Igusa, F. Marron, F. Romeo, A. Sangiovanni-Vincentelli, C. Sechen, H. Shin, G. Srinath, and H. Yaghutiel. MOSAICO: An integrated Macrocell Layout System. In *Proceedings of the VLSI-87 Conference*, Vancouver, Canada, August 1987.

[17] E. Cerny. An approach to unified methodology of combinational switching circuits. *IEEE Transactions on Computers*, 27(8), 1977.

[18] E. Cerny and C. Mauras. Tautology Checking Using Cross-Controllability and Cross-Observability Relations. In *IEEE International Conference on Computer-Aided Design*, pages 34–37, November 1990.

[19] H. Cho, G. Hachtel, S. Jeong, B. Plessier, E. Schwarz, and F. Somenzi. ATPG Aspects of FSM Verification. In *IEEE International Conference on Computer-Aided Design*, November 1990.

[20] E. M. Clarke, E. A. Emerson, and P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Language Systems*, 8(2):244–263, April 1986.

[21] O. Coudert, C. Berthet, and J. C. Madre. Verification of Sequential Machines Based on Symbolic Execution. In *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, 1989.

[22] O. Coudert, J. C. Madre, and C. Berthet. Verifying Temporal Properties of Sequential Machines Without Building their State Diagrams. In *Worshop on Computer-Aided Verification*, Rutgers, June 1990.

[23] M. Damiani and G. De Micheli. Observability Don't Care Sets and Boolean Relations. In *IEEE International Conference on Computer-Aided Design*, pages 502–505, November 1990.

[24] M. Damiani and G. De Micheli. Derivation of Don't Care Conditions by Perturbation Analysis of Combinational Multiple-Level Logic Circuits. In *International Workshop on Logic Synthesis*, May 1991.

[25] G. De Micheli. Logic Transformations for Synchronous Logic Synthesis. In *Hawaii International Conference on System Sciences*, pages 407–416, January 1990.

[26] Ewald Detjens, Gary Gannot, Richard Rudell, Alberto Sangiovanni-Vincentelli, and Albert Wang. Technology Mapping in MIS. In *International Conference on Computer Aided Design*, pages 116–119. IEEE, November 1987.

[27] Donald L. Dietmeyer and Peter Schneider. Identification of Symmetry, Redundancy and Equivalence of Boolean Functions. *IEEE Transactions on Electronic Computers*, EC-16(6):804–807, December 1967.

[28] W. E. Donath. *Physical Design Automation of VLSI Systems, Chapter Logic Partitioning.* Benjamin/Cummings Publishing Company Inc., 1988.

[29] C.M. Fiduccia and R. Mattheyses. A linear-time heuristic for improving network partitions. In *19th ACM/IEEE Design Automation Conference*, pages 241–247, July 1982.

[30] M. Fujita and Y Matsunaga. Multi-level Logic Minimization based on Minimal Support and its Application to the Minimization of Look-up Table Type FPGAs. In *IEEE International Conference on Computer-Aided Design*, November 1991.

[31] M. Fujita, Y. Tamiya, Y. Matsunaga, and K.C. Chen. Multi-Level Logic Synthesis for Boolean Relations. In *submitted to VLSI*, 1991.

[32] H. Fujiwara and T. Shimono. On the Acceleration of Test Generation Algorithms. In *IEEE Transactions on Computers*, pages 1137–1144, December 1983.

[33] A. Ghosh, S. Devadas, and A. R. Newton. Heuristic Minimization of Boolean Relations Using Testing Techniques. In *IEEE international Conference on Computer Design*, Cambridge, September 1990.

[34] A. Ghosh, S. Devadas, and A. R. Newton. Test Generation and Verification for Highly Sequential Circuits. In *IEEE Transactions on Computer-Aided Design*, pages 652–667, May 1991.

[35] P. Goel. An Implicit Enumeration Algorithm to generate tests for combinational logic circuits. In *IEEE Transactions on Computers*, volume C30, pages 215–222, March 1981.

[36] G. D. Hachtel, R. M. Jacoby, and P. H. Moceyunas. On Computing and Approximating the Observability Don't Care Set. In *MCNC Workshop in Logic Synthesis*, 1989.

[37] L. J. Hafer and A. Parker. Register-Transfer Level Digital Design Automation: The Allocation Process. In *15th ACM/IEEE Design Automation Conference*, pages 213–219, June 1978.

[38] S. Hong, R. Cain, and D. Ostapko. MINI: A Heuristic Approach for Logic Minimization. *IBM Journal of Research and Development*, 18:443–458, September 1974.

[39] Xilinx Inc. The programmable gate array data book.

[40] R. Jacoby, P. Moceyunas, H. Cho, and Hachtel G. New ATPG Techniques for Logic Optimization. In *IEEE International Conference on Computer-Aided Design*, pages 548–551, November 1989.

[41] S.-W. Jeong, B. Plessier, G.D. Hachtel, and F. Somenzi. Variable Ordering and Selection for FSM Traversal. In *IEEE International Conference on Computer-Aided Design*, pages 476–479, November 1991.

[42] K. Keutzer. Dagon: Technology Binding and Local Optimization by DAG Matching. In *24th ACM/IEEE Design Automation Conference*, pages 341–347, June 1987.

[43] R. P. Kurshan. *Analysis of Discrete Event Coordination.* Springer Verlog, 1990.

[44] T. Larabee. Efficient Generation of Test Patterns Using Boolean Difference. In *Proceedings of the International Test Conference*, pages 795–801, August 1989.

[45] C. E. Leiserson, F. M. Rose, and J. B. Saxe. Optimizing synchronous circuitry by retiming. In R. Bryant, editor, *3rd Caltech Conference on Very Large Scale Integration*, pages 87–116, 1983.

[46] M. Lightner and W. Wolf. Experiments in Logic Optimization. In *IEEE International Conference on Computer-Aided Design*, November 1988.

[47] B. Lin, H. Touati, and R. Newton. Don't Care Minimization of Multi-Level Sequential Logic Networks. In *IEEE International Conference on Computer-Aided Design*, November 1990.

[48] H-K. T. Ma, S. Devadas, A. R. Newton, and A. Sangiovanni-Vincentelli. Test Generation for Sequential Circuits. In *IEEE Transactions on Computer-Aided Design*, pages 1081–1093, October 1988.

[49] F. Mailhot and G. D. Micheli. Technology Mapping Using Boolean Matching. In *European Design Automation Conference*, pages 180–185, March 1990.

[50] A. Malik, R. K. Brayton, and A. Sangiovanni-Vincentelli. A Modified Approach to two-level Logic Minimization. In *IEEE International Conference on Computer-Aided Design*, pages 106–109, November 1988.

[51] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environments. In *IEEE International Conference on Computer-Aided Design*, pages 6–9, November 1988.

[52] Sharad Malik. *Combinational Logic Optimization Techniques in Sequential Logic Synthesis.* PhD thesis, U. C. Berkeley, 1990.

[53] P. McGeer and R. K. Brayton. Consistency and Observability Invariance in Multi-Level Logic Synthesis. In *IEEE International Conference on Computer-Aided Design*, 1989.

[54] P. McGeer and R. K. Brayton. The Observability Don't Care Set and Its Approximations. In *IEEE International Conference on Computer Design*, pages 45,48, September 1990.

[55] J. D. Morison, N. E. Peeling, and T. L. Thorp. ELLA: Hardware Description or Specification? In *IEEE International Conference on Computer-Aided Design*, pages 54–56, November 1984.

[56] R. Murgai, Y. Nishizaki, N. Shenoy, R. Brayton, and Sangiovanni-Vincentelli A. Logic Synthesis for Programmable Gate Arrays. In *27th ACM/IEEE Design Automation Conference*, pages 620–625, Orlando, June 1990.

[57] R. Murgai, N. Shenoy, R. Brayton, and Sangiovanni-Vincentelli A. Improved Logic Synthesis Algorithms for Table Look Up Architectures. In *IEEE International Conference on Computer-Aided Design*, pages 564–567, November 1991.

[58] S. Muroga. *Threshold Logic and its Applications.* John Wiley, 1971.

[59] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney. The Transduction Method - Design of Logic Networks Based on Permissible Functions. In *IEEE Transactions on Computers*, October 1989.

[60] A. Parker, D. Thomas, D. Siewiorek, M. Barbacci, L. Hafer, G. Leive, and J. Kim. The CMU Design Automation System. In *16th ACM/IEEE Design Automation Conference*, pages 73–79, June 1979.

[61] M. Pedram and N. Bhat. Layout Driven Technology Mapping. In *28th ACM/IEEE Design Automation Conference*, pages 99–105, San Francisco, June 1991.

[62] IEEE Press. Ieee standard vhdl language reference manual.

[63] J. Reed, A. Sangiovanni-Vincentelli, and M. Santamauro. A New Symbolic Channel Router: YACR2. In *IEEE Transactions on Computer-Aided Design*, pages 208–219, July 1985.

[64] Rick Rudell. *Logic Synthesis for VLSI Design.* PhD thesis, U. C. Berkeley, April 1989. Memorandum UCB/ERL M89/49.

[65] A. Saldanha, A. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Multi-Level Logic Simplification using Don't Cares and Filters. In *Design Automation Conference*, 1989.

[66] H. Sato, Y. Yasue, F. Matsunaga, and M. Fujita. Boolean Resubstitution with Permissible Functions and Binary Decision Diagrams. In *27th ACM/IEEE Design Automation Conference*, pages 284–289, Orlando, June 1990.

[67] H. Savoj and R. Brayton. The Use of Observability and External Don't Cares for the Simplification of Multi-Level Networks. In *27th ACM/IEEE Design Automation Conference*, pages 297–301, Orlando, June 1990.

[68] H. Savoj and R. K. Brayton. Observability Relations and Observability Don't Cares. In *IEEE International Conference on Computer-Aided Design*, pages 518–521, November 1991.

[69] H. Savoj, A.A. Malik, and R.K. Brayton. Fast Two-Level Logic Minimizers for Two-Level Logic Synthesis. In *IEEE International Conference on Computer-Aided Design*, pages 544–547, November 1989.

[70] H. Savoj, H. Touati, and R. K. Brayton. Extracting Local Don't Cares for Network Optimization. In *IEEE International Conference on Computer-Aided Design*, pages 514–517, November 1991.

[71] M. Schulz and E. Auth. Advanced automatic test pattern generation and redundancy identification techniques. In *ftcs*, pages 30–35, June 1988.

[72] C. Sechen and A. Sangiovanni-Vincentelli. The TimberWolf Placement and Routing Package. In *Proceedings of the 1984 Custom Integrated Circuit Conference*, pages 522–527, Rochester, NY, May 1984.

[73] E. Sentovich and R. K. Brayton. Preserving Don't Care Conditions During Retiming. In *International Conference on VLSI*, August 1991.

[74] E. Sentovich, K.J. Singh, C. Moon, H. Savoj, R. Brayton, and A. Sangiovanni-Vincentelli. Sequential circuit design using synthesis and optimization, 1992.

[75] H. Touati. private communication, 1990.

[76] H. Touati, R. Brayton, and R. Kurshan. Testing language containment for ω-automata using BDD's. In *International Workshop on Formal Methods in VLSI Design*, 1991.

[77] H. Touati, H. Savoj, and R.K. Brayton. Delay Optimization of Combinational Circuits by Clustering and Partial Collapsing. In *IEEE International Conference on Computer-Aided Design*, pages 188–191, November 1991.

[78] H. Touati, H. Savoj, B. Lin, R. Brayton, and A. Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines using BDD's. In *IEEE International Conference on Computer-Aided Design*, November 1990.

[79] Herve Touati. *Performance Driven Technology Mapping.* PhD thesis, U. C. Berkeley, 1990.

[80] C-J. Tseng and D. P. Siewiorek. Automated Synthesis of Data Paths in Digital Systems. In *IEEE Transactions on Computer-Aided Design*, pages 379–395, July 1986.

[81] J. Vasudevamurthy and J. Rajski. A Method for Concurrent Decomposition and Factorization of Boolean Expressions. In *IEEE International Conference on Computer-Aided Design*, pages 510–513, November 1990.

[82] R. A. Walker and D. E. Thomas. Behavioral Transformation for Algorithmic Level IC Design. *IEEE Transactions on Computer-Aided Design*, 8(10):1115–1128, October 1989.

[83] Y. Watanabe and R.K. Brayton. Heuristic Minimization of Boolean Relations. In *International Workshop on Logic Synthesis*, May 1991.

[84] G. Whitcomb and A. R. Newton. Abstract Data Types and High-Level Synthesis. In *27th ACM/IEEE Design Automation Conference*, pages 680–685, Orlando, June 1990.

[85] T. Yoshimura and E. S. Kuh. Efficient algorithms for channel routing. In *IEEE Transactions on Computer-Aided Design*, pages 25–35, January 1982.