SPFDs: A NEW APPROACH TO FLEXIBILITY IN LOGIC SYNTHESIS

by

Subarnarekha Sinha

Memorandum No. UCB/ERL M02/17

24 May 2002

SPFDs: A NEW APPROACH TO FLEXIBILITY IN LOGIC SYNTHESIS

by

Subarnarekha Sinha

Memorandum No. UCB/ERL M02/17

24 May 2002

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

SPFDs : A New Approach to Flexibility in Logic Synthesis

by

Subarnarekha Sinha

B. Tech. (Indian Institute of Technology, Kharagpur, India) 1996
M. S. (University of California, Berkeley) 1998

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Robert K. Brayton, Chair
Professor A. Richard Newton
Professor Theodore A. Slaman

Spring 2002

The dissertation of Subarnarekha Sinha is approved:

Chair Date

Date

Date

University of California, Berkeley

Spring 2002

SPFDs : A New Approach to Flexibility in Logic Synthesis

Abstract

SPFDs : A New Approach to Flexibility in Logic Synthesis

by

Subarnarekha Sinha

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Robert K. Brayton, Chair

Logic synthesis algorithms convert one representation of a Boolean network into another one that is more desirable from the point of view of area, delay, power, testability, wireability or other criteria. The main requirement on such transformations is that it preserves functionality across all required operating conditions. The quality of the final implementation is strongly influenced by the ability of a transformation to suitably express and utilize the flexibility inherent in the original implementation of a Boolean network. Depending on the particular transformation, several formalisms have been developed over the past decade for suitably expressing flexibility. In this dissertation, a new formalism for expressing flexibility called Sets of Pairs of Functions to be Distinguished (SPFDs) is presented. SPFDs provide a powerful mechanism for expressing flexibility during certain logic synthesis transformations. The expressive power of SPFDs is compared with previous formalisms for expressing flexibility. It is proved that the flexibility expressed by them completely contains some previous formalisms (like Incompletely Specified Functions) and extends (but does not completely contain) other formalisms (such as Boolean Relations).

An in-depth exposition is provided for various applications of SPFDs. It is argued that SPFDs provide a more powerful and intuitive mechanism for expressing flexibility in logic and for rewiring of a network. Any such improved formalism usually comes at the cost of increased computational expense in using that flexibility. Efficient algorithms are provided for harnessing this extra flexibility without incurring too much additional overhead. Other interesting applications of SPFDs to some classical logic synthesis problems like functional decomposition and sequential

synthesis are also presented.

Professor Robert K. Brayton
Dissertation Committee Chair

To my family : Ma, Bapi, Dilip and Babu.

Contents

List of Figures

List of Tables

Acknowledgments

I have been really fortunate to have interacted with some wonderful individuals during my stay in Berkeley. I would like to take this opportunity to thank them.

Professor Robert Brayton has been my advisor for the last six years. I would like to thank him for his constant encouragement and guidance during this time. He has been incredibly patient with me through the years, teaching me everything about the process of doing research. His enthusiasm and love for learning is a great inspiration to me. During my weekly meetings with him, he would discuss aspects of my research in great depth. He never dismissed any of my concerns, no matter how small they were, as trivial. Hopefully, some of it has rubbed off on me.

I would like to thank Dean Richard Newton for his insightful comments during my qualifying examination, that have greatly helped me during the course of my research work. He also took time off his busy schedule to provide valuable suggestions for improving this dissertation. I am very grateful to him for that. I am also very thankful to Professor Ted Slaman for introducing me to the beautiful world of mathematical logic. In addition, I greatly appreciate his interest in my research work.

I had the opportunity to work quite closely with Dr Andreas Kuehlmann. Working with him was a great learning experience. His suggestions have greatly helped me in my research. I hope I have learned something from his thoroughness and meticulousness. I am also grateful to him for flying in from Austin for attending my qualifying examination.

During my stay here at Berkeley, I have also had the opportunity to work with some current and former members of the CAD group. I have been lucky to interact with Sunil Khatri, Philip Chong, Fan Mo and William Jiang during the course of my research. It was a wonderful experience working with such a talented group of people. A lot of people - Wilsin Gosti, Desmond Kirkpatrick, Abdallah Tabbara, Bassam Tabbara, Luca Carloni, Sunil Khatri, Philip Chong, Fan Mo, William Jiang - gave me invaluable suggestions when I was preparing for my qualifying examination. I am very grateful to them for all their help during this stressful period of my graduate life. Special thanks are due to Yuji Kukimoto for his great advice during my job hunting process. He was always available to answer my endless stream of questions. Last summer, I had the opportunity to work with Tiziano Villa. I thank him for his interest in my work. His clarity of thought greatly helped me during the last phases of my research.

I have spent many a wonderful afternoon in Berkeley chatting with Freddy Mang, talking about everything under the sun. We went through all the stressful phases of graduate school together

and it greatly helped to have him around. I have had great fun making party plans with Niraj Shah and never really executing them. Sometimes it is more fun to talk about things than actually do them!

I cannot thank my parents and my brother enough for all their love and support over the years. They have been a continual source of strength for me. Their belief in my abilities has always helped me in all my endeavors.

The person I owe the most to is Dilip. He has fulfilled a variety of roles over the past few years: friend, parent, brother,counselor and philosopher. I couldn't have gone through the stresses of graduate school without his love and support. His presence in my life has been a great source of joy and happiness, helping me enjoy even the stress-filled graduate life. Thank you for the wonderful four years we spent together! I eagerly look forward to more great times in the years to come.

Chapter 1

Introduction

1.1 Design Flow

The traditional static CMOS standard cell based design methodology aimed at minimizing the overall gate area and delay. The design process was usually carried out in a top-down fashion with several distinct, relatively decoupled phases like high level synthesis, logic synthesis and physical design (Figure 1.1).

During high level synthesis, a Register Transfer Level (RTL) structure was generated which realized the given behavioral description. Temporal scheduling, and allocation and binding of hardware were the issues considered at this stage.

The input to the logic synthesis phase was the RTL description of the circuit, and a cell library. The circuit was typically represented as a multi-level logic network, that was then optimized for various design objectives like area and delay for generating a gate level netlist implemented with the elements from the given cell library. The optimization phase itself consisted of two sub-phases: technology-independent and technology-dependent optimization. The objective of the technology-independent phase was to simplify the logic level netlist without making any assumptions about the underlying technology to be used for the actual implementation of the circuit. Each node in the multi-level logic network at this stage represented an arbitrarily complex function. The network was then optimized using Boolean and algebraic operations on nodes like node factoring, substitution, elimination, node simplification using *don't cares*, etc. The technology-dependent phase took this netlist as input and transformed it for implementing and optimizing in a particular technology. The mapped netlist was then input to the physical design tools which placed and routed the netlist thereby realizing the physical layout of the circuit which had been optimized for area and delay. This flow

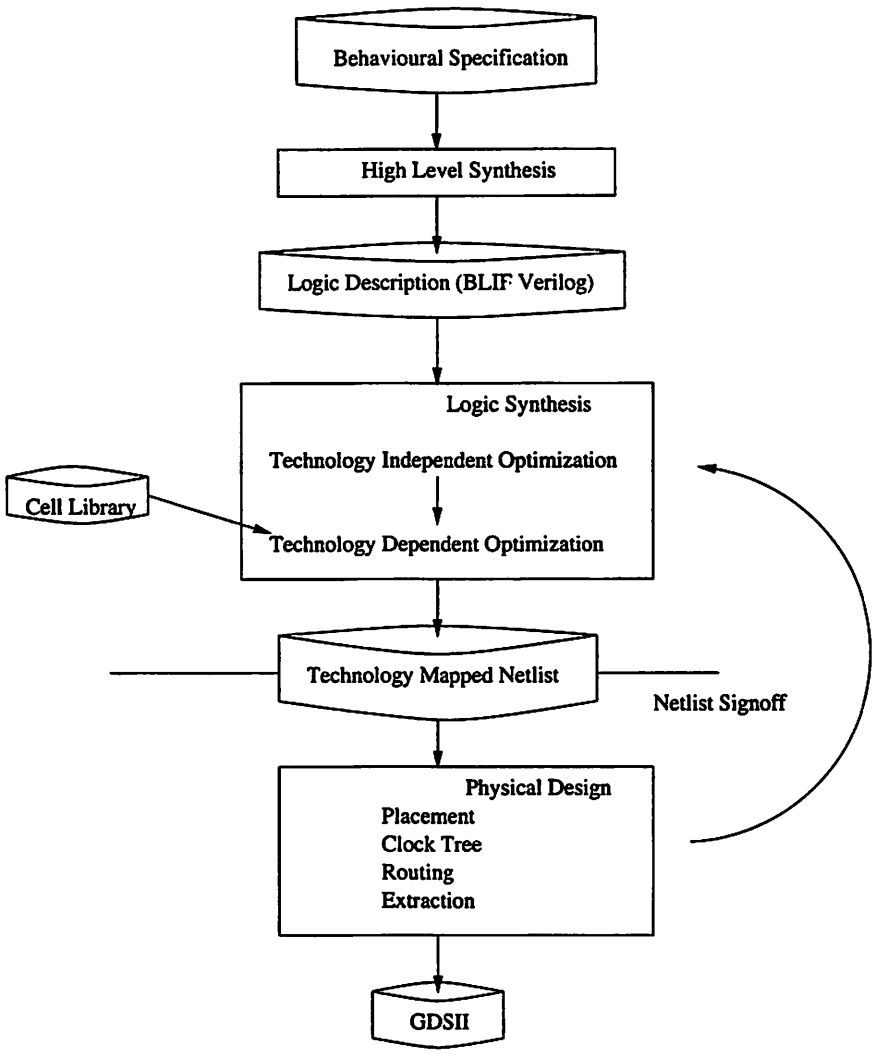


Figure 1.1: Traditional design flow.

was the de-facto standard until the mid-90s when most of the delay was in the gates. It made sense to de-couple logic synthesis from physical design and to focus more on area and delay minimization of gates during the logic synthesis phase.

As process geometries scale down, interconnect becomes an important factor in determining the delay. This is mainly due to the following two reasons. First, the gate delay depends mostly on the output capacitance it drives, of which the net capacitance becomes the largest contributor. Second, the delay of the long nets, which depends on their capacitance, becomes larger than gate delays. This trend has resulted in a revision of the standard design flow. It has necessitated a much closer integration between logic synthesis and physical design so that more accurate estima-

tion of the optimization parameters can be obtained. Another consequence of this trend has been a modification of the focus of traditional logic synthesis transformations to include more interconnect specific optimizations. Some recent work has already started in this area, for instance wireplanning for logic decomposition [1].

The main focus of this dissertation is improving and enhancing some of the transformations of logic synthesis, specially in the light of the changing scope of the area. At the heart of any logic synthesis transformation is the flexibility of changing the given network into a different network for improving some criteria, while still maintaining required input-output functionality. The input-output functionality specifies what the output(s) of the network should be for each input pattern. Depending of the transformation, this flexibility can be modeled and used in different ways. In the next section, some of the commonly used formalisms for modeling flexibility in certain fundamental logic synthesis transformations are described in some detail.

1.2 Flexibility in Logic Synthesis

Logic Synthesis is the process of transforming a set of Boolean functions, obtained from the RTL structure, into a network of gates in a particular technology. The task of logic synthesis is to transform one representation of a network into another , which is more desirable from the point of view of area, delay, power, testability, wireability and/or other criteria. Some common transformations include changes in the local functionality of a group of nodes (*don't care* optimization), logic restructuring during timing optimization, gate resizing for meeting the area-delay constraints, modifying the wiring pattern between the nodes in the network, etc. Each of these transformations exploit the inherent flexibility of the network. Depending on the transformation at hand, this flexibility can be modeled in a particular fashion, thereby making it more suitable for manipulation by the synthesis algorithms.

In the following two sections, the manner in which the inherent flexibility in a Boolean network is modeled in two important transformations in logic synthesis is presented.

1.2.1 Flexibility in Logic

The transformations that exploit implementation flexibility of a node in a multi-level network are described here. This transformation is possible due to the fact that while it is absolutely necessary to maintain certain required input-output functionality of the network, it not always nec-

essary to maintain the identical local functionality at every node in the network. This relaxation of criteria provides the flexibility to transform some nodes in the network and the environment of the node provides the information needed for exploiting this additional flexibility. The basic task for the logic synthesis transformation is to look at a node in a network and try to find different functions, that are more desirable from the point of view of the optimization criteria and can be used instead of the current one. A naive approach would try to replace the original function with all possible functions and see which one gives the best solution, while still satisfying the input-output functionality. However, this is computationally too extensive as the number of possible Boolean functions is very large. Furthermore, some Boolean functions cannot be used as the functionality of the network can change if these functions are used at the node. Over the past decade, a lot of research has focussed on trying to mathematically characterize the flexibility at a node in order to eliminate the *ad hoc* nature of the search process. Incompletely Specified Functions (ISFs) [2] and Boolean Relations [3] are the most common formalisms used for representing the flexibility of a single-output node and a multiple-output node, respectively.

An ISF consists of the *onset*, the *offset* and the *don't care* set. The minterms in the *onset* and *offset* have to produce 1 and 0, respectively. On the other hand, minterms in the *don't care* set can produce either a 0 or a 1. For each assignment of a minterm in the *don't care* set, a new function is obtained. This choice can be exercised to obtain several different functions at the node. A Boolean relation specifies several output values for each input minterm. For each input minterm, any output in the specified set can be chosen. As in the case of ISFs, depending on the choice of the output value for each input minterm, several functions can be derived.

The best function in both cases is chosen depending on the optimization criteria. The most common criteria used is the minimization of area, typically modeled as the number of the literals in the factored form of the function at the node. These transformations are present to different extents in all commercial logic synthesis tools. In Chapter 3, an in-depth exposition of the different formalisms used for expressing this flexibility is provided.

It is necessary to realize that optimization is often limited by the expressive power of the formalisms chosen to represent the flexibility. This has resulted in the sustained effort for improving the power of the formalism used for representing the implementation flexibility of a node in a network. For instance, Boolean relations were introduced to represent the implementation flexibility of a multi-output node since ISFs (which were used to represent the flexibility of a single-output node) were shown to be inadequate [3].

1.2.2 Flexibility in Wiring

Just as the functionality of some nodes in the network can be changed while keeping the overall network functionality unchanged, the wires between the nodes can also be changed without altering input-output behavior of the circuit. The basic task of this synthesis transformation is to replace one wire with another, in order to optimize the circuit for certain criteria. The typical criterion used in the past to select such a change was routability (i.e. whether the new wire is predicted to be easier to implement in the final layout than the one it is replacing). A lot of work has been done in the past decade for characterizing the set of wires that can replace a given wire in the network, without affecting its functionality. Most of the previous work in this area involved adding redundant wires and thereby rendering some of the original wires in the network redundant [4, 5, 6] and hence candidates for removal. This approach is commonly called redundancy addition and removal. Like the formalisms for expressing the implementation flexibility of a node, the quality of the results depend on the power of the formalisms. To improve the quality, this basic idea was extended in a few other papers [5, 7, 8] to allow simple functionality changes of the nodes, thereby allowing more wiring changes to be accepted.

Another set of techniques [9, 10] performed rewiring by modeling the problem of wire reconnections by a flow graph and then solving the problem using maxflow-mincut algorithm on the flow graph.

These techniques do not affect the functionality of the nodes in the network and are suitable for use during the later stages of the design flow when it may be undesirable to perturb the network substantially.

1.3 Focus of this Work

In this dissertation, a new formalism, Sets of Pairs of Functions to be Distinguished (SPFDs)[1], for expressing flexibility during some logic synthesis transformations is presented. The expressive power of SPFDs is compared with previous schemes. It is proved that in some cases the flexibility expressed by SPFDs completely contains the flexibility expressed by previous approaches and in other cases it extends (but does not completely contain) the flexibility expressed by previous approaches.

The problems mentioned in the previous section are revisited. It is illustrated how SPFDs

[1] introduced by Yamashita et. al. in the limited context of FPGA synthesis [11]

provide a very powerful and intuitive mechanism for expressing the flexibility in logic and for rewiring of a network. As mentioned earlier, any improved formalism for expressing flexibility usually comes at the cost of an increase in computational expense in using that flexibility. For instance, while Boolean Relations are more expressive than ISFs, they are also more computationally expensive to manipulate. Efficient algorithms are provided for harnessing this extra flexibility without incurring too much additional overhead.

Finally, other interesting applications of SPFDs to some classical logic synthesis problems like functional decomposition and sequential synthesis are also considered.

1.3.1 Dissertation Outline

Chapter 2 contains all preliminaries, including the definitions and terminology that will be used in the rest of the dissertation. As mentioned before, flexibility in logic is a well-researched problem. In Chapter 3, the various schemes are presented and compared.

SPFDs are formally introduced in Chapter 4, where they are defined and their ability for representing flexibility in logic is compared to previous approaches. How an SPFD attached to a node/wire can be used to represent its information content is also described. This provides an intuitive explanation of what a node contributes to its surrounding network. There are some interesting implications and applications of this connection, some of which are presented in later chapters.

In Chapter 5, the flexibility expressed by SPFDs is used for optimizing a network with the goal of reducing the overall literal count. One major problem associated with the increased flexibility of SPFDs is addressed and techniques are provided for solving them. The results obtained are compared to previous schemes. Image Computation-the process of expressing minterms in one variable space in terms of another variable space- is an important step in the node simplification process. Most previous image computation approaches used Binary Decision Diagrams (BDDs) [12], which are very efficient for set manipulation but often suffer from memory explosion. On the other hand, SAT solvers (e.g. [13, 14]) suffer from the reverse problem. SAT solvers are robust and can handle large circuits but are inefficient for set manipulation operations. A hybrid approach, combining the efficiency of BDDs and the robustness of SAT solvers, for image computation is also presented.

Rewiring a given network using the increased flexibility expressed by SPFDs is described in Chapter 6. Some preliminary theoretical work is presented to compare the power of SPFD-based rewiring schemes relative to previous rewiring schemes like redundancy addition and removal,

described earlier. Two rewiring scenarios are subsequently presented. In one approach, rewiring a Boolean network in an attempt to reduce the wire count without increasing the literal count is described. Another approach for performing SPFD-based rewiring in a combined logic synthesis-physical design environment is presented. Possible interesting extensions of this approach are also described.

An interesting application of SPFDs to functional decomposition is presented in Chapter 7. Given a network topology (i.e. the interconnectivity of the nodes) and its required input-output functionality, an algorithm is provided for synthesizing the nodes in the network. An interesting metaphor describes a network as a channel that transfers information from the inputs to the outputs. Some possible applications of this algorithm are also presented here.

The concept for using SPFDs for sequential circuits is presented as an extension in Chapter 8. Theoretical results are provided for illustrating that SPFDs can be extended very easily for expressing the classical incompatibility graph of a Finite-State Machine(FSM). An algorithm for partitioning the state bits in a sequential circuit is provided, thereby possibly increasing the size of machines that can be handled by sequential synthesis tools. The actual implementation of such techniques is beyond the scope of this dissertation.

Chapter 2

Preliminaries

In this chapter, some basic definitions and concepts that are essential for describing the work presented in this dissertation are presented.

2.1 Boolean Functions and Relations

Definition 2.1 *A* **completely specified Boolean function** f *with* n *inputs and* l *outputs is a mapping:*

$$f : B^n \rightarrow B^l,$$

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

$$\begin{aligned} \textit{onset} &= \{m \in B^n | f(m) = 1\} \\ \textit{offset} &= \{m \in B^n | f(m) = 0\}. \end{aligned}$$

Definition 2.2 *Any vertex in* B^n *is also called a* **minterm**. *A minterm of a function* f *is a vertex* m *such that* $f(m) = 1$.

Definition 2.3 *An* **incompletely specified function** *(ISF)* $\mathcal{F}$ *with* n *inputs and* l *outputs is a mapping:*

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

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

$$\begin{aligned} \textit{onset} &= \{m \in B^n | \mathcal{F}(m) = 1\} \\ \textit{offset} &= \{m \in B^n | \mathcal{F}(m) = 0\} \\ \textit{dcset} &= \{m \in B^n | \mathcal{F}(m) = -\}. \end{aligned}$$

The symbol - means that the function can either be a 0 or 1.

Definition 2.4 *A* **Boolean relation** *is a one-to-many Boolean mapping* $\mathcal{R} : B^n \rightarrow B^l$. *In general* $\mathcal{R}(x) \subseteq B^l$ *is a set.*

Definition 2.5 *A function* $f : B^n \rightarrow B^m$ *is* **compatible** *with* $\mathcal{R}$ *iff for every input minterm* x, $f(x)$ *is a member of* $R(x)$.

Let $x_1, x_2, \cdots, x_n$ be the variables of the space B^n. A vertex or a vector of variables in B^n is represented as $\mathbf{x}$.

Definition 2.6 *Let* $A \subseteq B^n$. *The* **characteristic function** *of* A *is the function* $f : B^n \rightarrow B$ *defined by* $f(x) = 1$ *if* $x \in A$, $f(x) = 0$ *otherwise.*

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

Definition 2.7 *A* **literal** *is a variable in its true or complemented form (e.g.* x_i *or* $\overline{x_i}$*). A* **product term or cube** *is the conjunction of some set of literals (e.g.* $x_1\overline{x_2}x_3$*).*

Definition 2.8 *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 2.9 *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 $f = x_i f_{x_i} + \overline{x_i} f_{\overline{x_i}}$.

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 variables. The leaves are either 0 or 1. Each path of the tree represents a minterm of a function.

Definition 2.10 *A* **cover** *for an ISF* $|\mathcal{F}| : B^n \rightarrow Y$ *is any completely specified Boolean 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* $\mathcal{F} = -$.

There are two common ways for representing the cover of a Boolean function.

Definition 2.11 *A* **sum-of-products** *representation of the cover of a Boolean function is a sum of cubes.*

Definition 2.12 *A* **factored form** *of a cover is either a product or a sum such that:*

- *A product is either a single literal or a product of factored forms.*
- *A sum is either a single literal or a sum of factored forms.*

Both representations have their own advantages and disadvantages and are used depending on the problem at hand.

2.2 Boolean Networks

Definition 2.13 *A* **Boolean network,** $\mathcal{N}$*, is a directed acyclic graph (DAG) such that for each node,* η_i*, 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* (η_i, η_j) *from* η_i *to* η_j *if* f_j *depends explicitly on* y_i *or* $\overline{y_i}$*. A node* η_i *is a fanin of a node* η_j *if there is a directed edge* (η_i, η_j) *and a fanout if there is a directed edge* (η_j, η_i)*. A node* η_i *is a transitive fanin of a node* η_j *if there is a direct path from* η_i *to* η_j *and a transitive fanout if there is a directed path from* η_j *to* η_i*. Primary inputs* $X = (x_1, \cdots, x_n)$ *are inputs of the Boolean network and primary outputs* $Z = (z_1, \cdots, z_m)$ *are its outputs. Intermediate nodes of the Boolean network have at least one fanin and one fanout. The global function* f_i^g *at* η_i *is the function at the node expressed in terms of primary inputs.*

Definition 2.14 *The* **support** *of a function* f *is the set of variables that* f *explicitly depends upon.*

Definition 2.15 *A* **topological ordering of** $\mathcal{N}$ **from the primary outputs** *is a linear ordering such that for nodes* $\eta_i, \eta_j \in \mathcal{N}$*, if* η_i *is a fanout of* η_j*, then* η_i *appears before* η_j *in the linear order. A* **topological ordering of** $\mathcal{N}$ **from the primary inputs** *is a linear ordering such that for any two nodes* $\eta_i, \eta_j \in \mathcal{N}$*, if* η_i *is a fanin of* η_j*, then* η_i *appears before* η_j *in the linear order.*

2.3 Boolean Operations

Definition 2.16 *Given two Boolean functions,* $f : B^n \rightarrow B$ *and* $g : B^n \rightarrow B$*, the* **AND** *operation* $h = f.g$ *is defined as:*

$$h = \{x | f(x) = 1 \wedge g(x) = 1\}.$$

Definition 2.17 *Given two Boolean functions, $f : B^n \rightarrow B$ and $g : B^n \rightarrow B$, the* **OR** *operation $h = f + g$ is defined as:*

$$h = \{x | f(x) = 1 \vee g(x) = 1\}.$$

Definition 2.18 *Given a Boolean function, $f : B^n \rightarrow B$, the* **NOT** *operation $h = \overline{f}$ is defined as:*

$$h = \{x | f(x) = 0\}.$$

Since sets can be represented using Boolean functions, hence set operations like union, intersection and complement can be computed using the above operations.

Definition 2.19 *Let $f : B^n \rightarrow B$ be a Boolean function, and $x = (x_{i_1}, \cdots, x_{i_n})$ a set of input variables of f. The* **smoothing** *of f by x is:*

$$\begin{aligned} S_x f &= S_{x_{i_1}} \cdots S_{x_{i_n}} f \\ S_{x_{i_j}} f &= f_{x_{i_j}} + f_{\overline{x_{i_j}}}. \end{aligned}$$

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 x variables. This is the smallest Boolean function independent of $x_{i_1}, \cdots, x_{i_n}$ which contains f.

Lemma 2.2 *Let $f : B^n \times B^m \rightarrow B$ and $g : B^m \rightarrow B$ be two Boolean functions. Then:*

$$S_x(f(x,y)g(y)) = S_x(f(x,y))g(y).$$

Definition 2.20 *Let $f : B^n \rightarrow B$ be a Boolean function, and $x = (x_{i_1}, \cdots, x_{i_n})$ a set of input variables of f. The* **consensus** *of f by x is:*

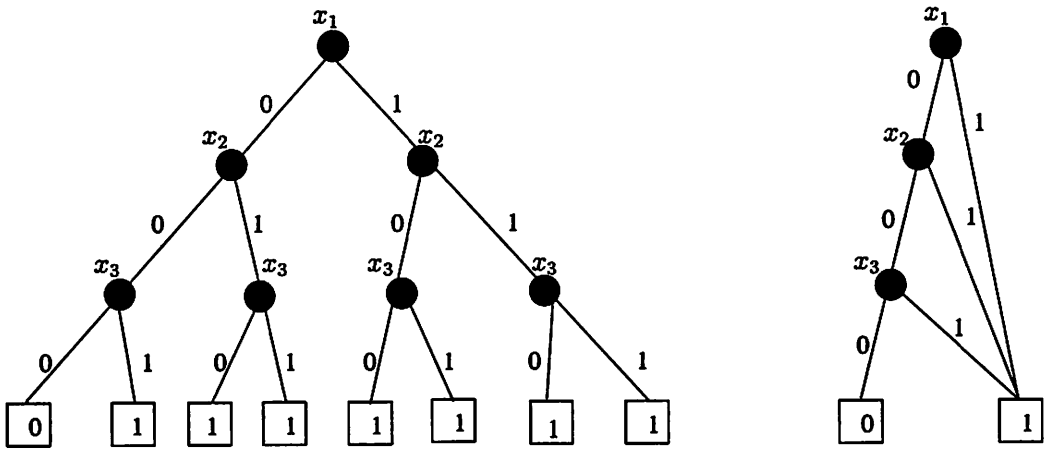
$$\begin{aligned} C_x f &= S_{x_{i_1}} \cdots S_{x_{i_n}} f \\ C_{x_{i_j}} f &= f_{x_{i_j}} f_{\overline{x_{i_j}}}. \end{aligned}$$

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

2.4 Image and Inverse Image Computations

Definition 2.21 *Let $f : B^n \rightarrow B^m$ be a Boolean function and A be a subset of B^n. The* **image** *of A by f is the set $f(A) = \{y \in B^m | y = f(x), x \in A\}$. If $A = B^n$, the image of A by f is also called the range of f.*

Definition 2.22 *Let $f : B^n \rightarrow B^m$ be a Boolean function and A be a subset of B^m. The* **inverse image** *of A by f is the set $f^{-1}(A) = \{x \in B^n | f(x) = y, y \in A\}$.*



$$f = x_1 + x_2 + x_3$$

Figure 2.1: Shannon Decomposition and Binary Decision Diagram of a simple function.

2.5 Binary Decision Diagrams

Binary Decision Diagrams (BDDs) [12] are compact representations of recursive Shannon decompositions. The decomposition is done with the same order along every path from the root to the leaves. BDDs 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) [12], which shall henceforth be referred to as a BDD. BDDs can be used for representing and efficiently manipulating sets. The Shannon decomposition and the BDD of a simple function is shown in Figure 2.1.

2.6 Conjunctive Normal Form and Satisfiability

A Conjunctive Normal Form (CNF) [15] is a conjunction (product) of clauses, where a clause is a disjunction (sum) of literals. For example, $\phi = (x_1 + \overline{x_2})(\overline{x_1} + x_3)$ denotes a CNF formula with two clauses and three variables.

The Boolean satisfiability problem (SAT) [15] for a CNF formula is formulated as follows: Given a CNF formula ϕ, representing a Boolean function $f(x_1, \cdots, x_n)$, the satisfiability problem consists of identifying a set of assignments to the formula variables, $\{x_1 = v_1, \cdots, x_n = v_n\}$, such

that all clauses are satisfied, i.e. $f(v_1, \cdots, v_n) = 1$, or proving no such assignment exists.

A number of logic synthesis transformations like Automatic Test Pattern Generation (ATPG) and redundancy addition and removal can be modeled as SAT problems [16].

2.7 Combinational and Sequential Circuits

Definition 2.23 *A circuit is* **combinational** *if it computes a function which depends only on the values of the inputs applied to the circuit; for each input value, there is a unique output value.*

All circuits with an underlying acyclic topology are considered combinational and can be modeled as a Boolean network. There are circuits containing cycles that are combinational also [17] but these are unusual and are not considered in the rest of the dissertation.

Definition 2.24 *A circuit is* **sequential** *if it computes a function that depends both on the present values of its inputs and the values applied to the inputs of the circuit at some previous time.*

Definition 2.25 *A* **Finite-State Machine(FSM)** *is a quintuple:*

$$M = (S, I, O, \delta, \lambda),$$

S : finite non-empty set of states, I : finite non-empty set of inputs, O : finite non-empty set of outputs, δ: $S \times I \rightarrow S$ transition (or next state function), and $\lambda : S \rightarrow O$ output function.

FSMs provide a behavioral view of sequential circuits. They can be used to describe the transitional behavior of these circuits. They can be used to distinguish among a finite number of classes of input histories: these classes are referred to as the internal states of the machine. FSMs are often represented graphically as a State Transition Graph (STG).

2.8 Notation

A Boolean network is represented by $\mathcal{N}$ and its primary inputs and primary outputs as $PI(\mathcal{N})$ and $PO(\mathcal{N})$, respectively. A node η_j in network $\mathcal{N}$ is associated with two variables, y_j and y_j'. The variables associated with the primary inputs of $\mathcal{N}$ are collectively denoted as either X or X', depending on whether the unprimed or primed variables are used. Both X and X' are referred to as the primary space. Similarly, the variables associated with the fanins of a node η_j are collectively denoted as Y_j or Y_j'. In the sequel, both Y_j and Y_j' are often referred to as the fanin

space of η_j. The fanin and fanout nodes of η_j are collectively denoted as $FI(\eta_j)$ and $FO(\eta_j)$, respectively. Similarly, the transitive fanins and transitive fanouts of η_j are collectively denoted as $TFI(\eta_j)$ and $TFO(\eta_j)$, respectively. The primary inputs in the transitive fanin of η_j are denoted as $PI(\eta_j)$. Similarly, the primary outputs in the transitive fanout of η_j are denoted as $PO(\eta_j)$. The local and global functions of η_j are denoted as f_j and f_j^g, respectively. A directed edge between η_i and η_j, also called a wire, is denoted as $w_{\eta_i \rightarrow \eta_j}$. The expression $\eta_i <_f \eta_j$ denotes that η_i has less flexibility than η_j. Similarly, $\eta_i >_f \eta_j$ denotes that η_i has more flexibility than η_j.

Chapter 3

Flexibility in Node Functionality

In this chapter, some formalisms used for specifying and exploiting the flexibility in logic during combinational logic synthesis are reviewed. In particular, two related problems are considered:

- Deriving and representing a set of permissible functions at a node or a set of nodes in the network.
- Using an appropriate representation of these functions with an associated minimizer to search for one that best fits the optimization criteria.

These two issues are examined for a node embedded in a network of single-output functions and for a node in a network of multiple-output functions as well. One of the most common optimization criteria used for this set of logic transformations is minimization of the literal count of the covers of the functions at each node. It is worthwhile to note that for combinational logic the classical use of flexibility is in two-level sum-of-products minimization, where the objective is to find a cover of a function with the least number of product terms. Usually, this problem is formulated with the input or output *don't cares* [18] given for the function. A *don't care* is an input vector for which a function's value can be either 0 or 1. Thus in minimizing f, one has the option of choosing 1 or 0 for this value and in that sense, additional flexibility is given to the minimizer for making the choice which is best in meeting the minimization criteria. ESPRESSO, developed by Rudell et. al., is the most commonly used two-level minimizer. For a more detailed explanation of ESPRESSO, please refer to [18]. Most logic is implemented in multi-level form and so for the rest of this chapter, the main focus will be on deriving the flexibility of a node in a multi-level network and using that flexibility to simplify the node.

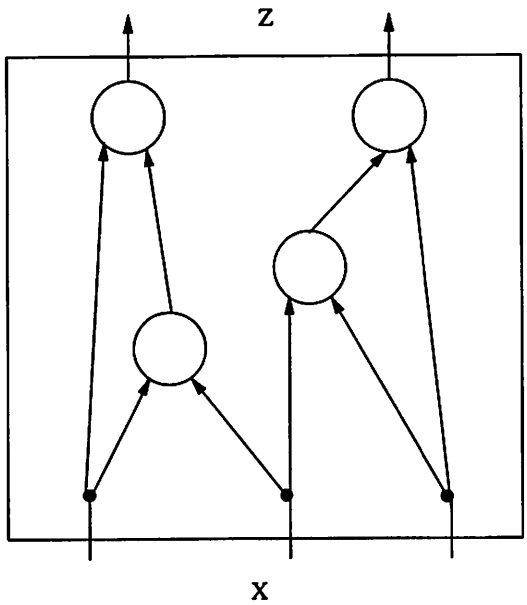


Figure 3.1: Network with single-output nodes.

3.1 Flexibility of a node

3.1.1 Networks of Single-Output Functions

The outputs of the Boolean network (shown in Figure 3.1) are associated with a set of don't cares $d(X) = (d_1(X), \cdots, d_m(X))$, which may be empty in some cases. These are called the *external don't cares*. It is implicitly assumed that these *don't cares* are independent, i.e. the *don't care* set of one primary output can be used independently of the *don't care* sets of other primary outputs.

3.1.1.1 Derived Don't Cares

One of the strategies used for multilevel minimization is to use a two-level minimizer as a subroutine. The algorithm proceeds as follows. The focus is on a single node, say a node η_i, of the Boolean network with the node's current representation given by a cover of its single-output function f_i. Unlike in the two-level case, no *don't cares* are given *a priori* for this node, but it is possible to derive some from the information about the surrounding network. Three classes of *don't cares* are derived: *External Don't Cares* (EXDC), *Satisfiability Don't Cares* (SDC) and *Observability Don't Cares* (ODC) [19].

The EXDC is just the given external *don't cares* for the network. The EXDC is used to restrict the computations performed for SDC and ODC, so that only the care inputs are used in their computation.

The SDC are obtained from the part of the network in the transitive fanin of a node. The constraints of the fanin network ensure that certain input patterns of η_i can never appear. The SDC of a node η_i is given as:

$$SDC(\eta_i) = \sum_{\eta_j;\eta_j \in TFI(\eta_i)} (y_j \oplus f_j).$$

Since they never occur, the output for f_i for these input patterns can be either 0 or 1, i.e. it is a *don't care.*

Example 3.1 *Consider the following multi-level circuit [20]:*

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

If t is simplified using the SDCs,

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

$t = s\bar{k} + \bar{s}kr$. *Here, in addition to s and k, r has been substituted into t. However, the global function of t remains unchanged. This technique is used in some commands in SIS for performing Boolean resubstitution.*

The ODC occurs because of the part of the network separating the node from the primary outputs. The ODC consists of primary input patterns for which toggling the value of the output of η_i does not affect the functionality of the primary outputs, i.e. no primary output is also seen toggling as a result of the changes at the output at η_i. The ODC of a node η_i is given by:

$$ODC(\eta_i) = \prod_{k=1}^{m} \overline{\partial z_k / \partial y_i}.$$

Example 3.2 *Consider the following circuit [20]:*

$$\begin{aligned} f_0 &= y_1 + y_2 + y_3 \\ y_1 &= x_1 x_2 \\ y_2 &= x_2 x_3 \\ y_3 &= x_1 x_3 \end{aligned}$$

The ODC of y_1 *is equal to* $\partial f_0/\partial y_1 = y_2 + y_3$. *Similarly, the ODCs of* y_2 *and* y_3 *are* $(y_1 + y_3)$ *and* $(y_1 + y_2)$ *respectively.*

If the function at each z_k does not depend on the variable y_i, then the computation is a lot more complicated. For details, please refer to [20]. In general, deriving the complete ODC for a node in a network is a very computationally intensive problem. This is because modifying a node using its ODC requires the ODCs of the remaining nodes to be recomputed. Hence, in practice, only subsets of ODC are used.

One such subset that is commonly used are the *Compatible Observability Don't Cares* (CODCs) proposed by Savoj [20]. The CODCs are compatible i.e. if a node is modified using its CODC, the CODCs of the remaining nodes are not affected. The computation of CODCs for intermediate nodes in the network depends on two key operations. One is the computation of CODCs for the fanin edges of a node, given the CODC of a node and an ordering of the fanins. Suppose a node η_i has k fanins, $\eta_{i_1} <_f \cdots <_f \eta_{i_k}$, where η_{i_1} will be assigned least flexibility and η_{i_k} most flexibility. Then, the CODC of the jth fanin η_{i_j} is given as:

$$CODC(\eta_{i_j}) = (\partial f_i/\partial y_{i_{j+1}} + \forall_{y_{i_{j+1}}}) \cdots (\partial f_i/\partial y_k + \forall_{y_{i_k}})\overline{(\partial f_i/\partial y_j)} + CODC(\eta_i).$$

In the first term, $\overline{\partial f_i/\partial y_j}$ denotes the ODC of the fanin η_{i_j} and the rest of the terms produce the compatibility. The second key operation is computing CODCs of each node by intersecting the CODCs of its fanout edges.

Consider again the network in the previous example. Let $y_1 >_f y_2 >_f y_3$. Thus y_1 has the most flexibility and y_3 has the least flexibility. Then, their CODCs are computed as follows:

$$\begin{aligned} d_1 &= \overline{\partial f_0/\partial y_1} = y_2 + y_3 \\ d_2 &= (\partial f_0/\partial y_1 + \forall_{y_1})(\overline{\partial f_0/\partial y_2}) = y_1\overline{y_2} + y_3 \\ d_3 &= (\partial f_0/\partial y_1 + \forall_{y_1})(\partial f_0/\partial y_2 + \forall_{y_2})(\overline{\partial f_0/\partial y_3}) = y_1\overline{y_3} + y_2\overline{y_3} \end{aligned}$$

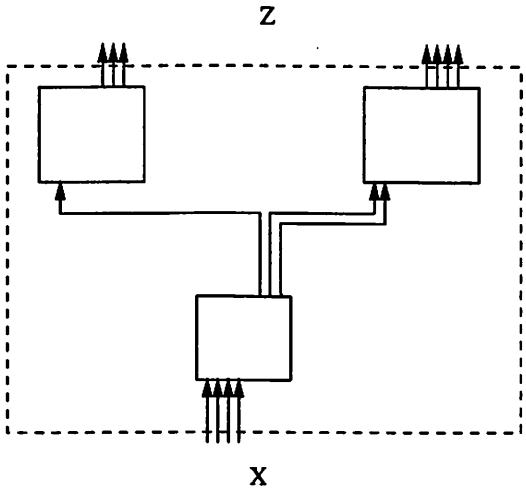


Figure 3.2: Network with multi-output nodes.

Maintaining compatibility can reduce the *don't care* sets of some nodes. For example, y_2 and y_3 have fewer minterms in their *don't care* set than their ODCs.

Another related technique for deriving the implementation flexibility of a node is transduction [21]. In transduction, *don't cares* are represented as permissible functions. Permissible functions are defined at each node and represent the sets of allowable functions for those nodes. As in the case of ODCs, permissible functions can be incompatible (*Maximum Set of Permissible Functions*) or compatible (*Compatible Set of Permissible Functions*). The *Maximum Set of Permissible Functions* (MSPF) is computed for a node η_j in a NOR-gate network $\mathcal{N}$ as follows:

1. Derive a new network $\mathcal{N}'$ by replacing η_j with an OR-gate.

2. Given that $\tilde{z}^k$ is the kth primary output in $\mathcal{N}'$,

$$f_j^M(x) = \begin{cases} f_j(x) & \text{if } \exists_k x \notin EXDC_k \text{ and } z_k(x) \neq \tilde{z}_k(x) \\ - & \text{otherwise} \end{cases}$$

It can be shown that the *don't cares* in the MSPF of a node constitute the ODCs of a node. Thus, ODCs simply are a generalization of the flexibility provided by MSPFs.

3.1.2 Networks of Multiple-Output Nodes

When the nodes in the network have multiple outputs, as in Figure 3.2, the notion of don't cares can be generalized so that each node η_i with l outputs has a set of *don't cares* $d(X) =$

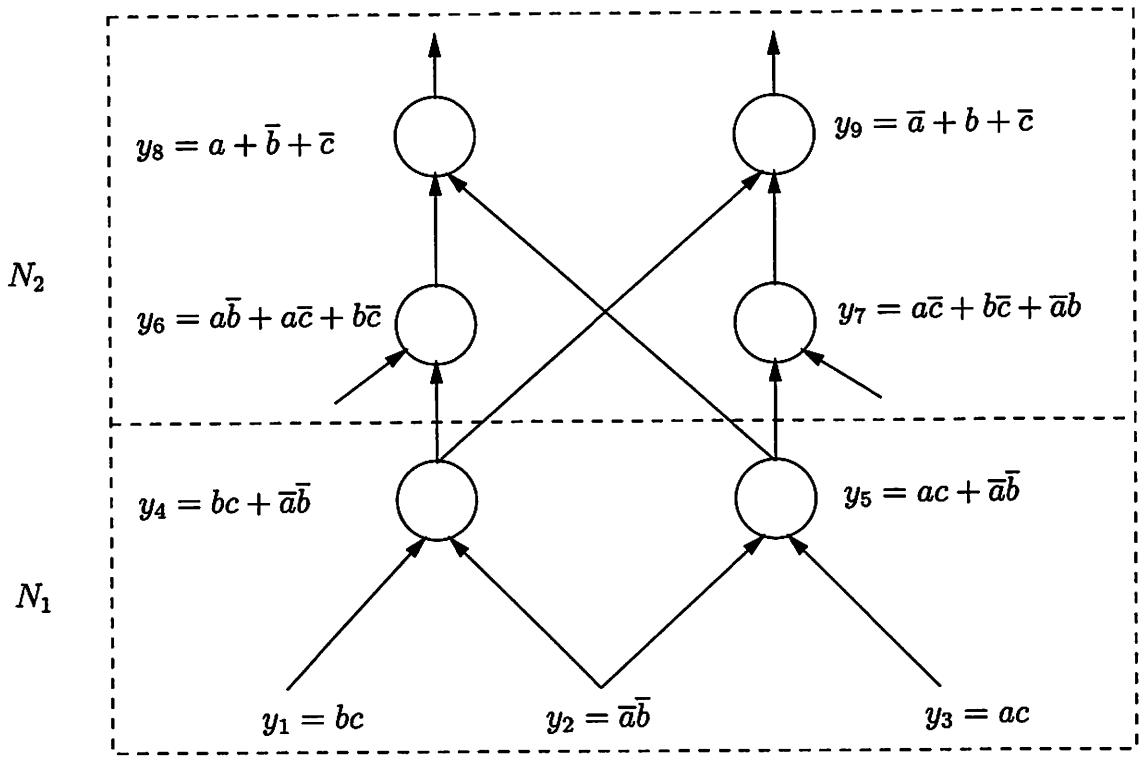


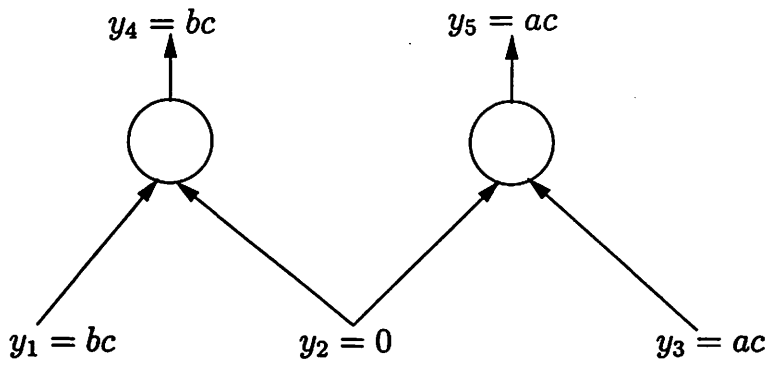
Figure 3.3: Example circuit for Boolean relation.

$(d_1(X), d_2(X), \cdots, d_l(X))$ which provides the flexibility that can be used for deriving a minimum cover of η_i. However, it was shown [3] that *don't cares* can no longer completely specify the flexibility of implementation of a multi-output node in a network. The same paper introduced the concept of Boolean relations.

3.1.2.1 Boolean Relation

A Boolean relation is a relation between inputs and outputs. Suppose x is a vector of the inputs of node η_i and y is a vector of its outputs. Then, a Boolean relation $B(x, y)$ gives a set of allowable outputs y for each input vector x, i.e., the set $\{y|B(x, y) = 1\}$ gives the set of allowable outputs for input x. The reason why Boolean relations can express more flexibility than *don't cares* is that with a Boolean relation, it is possible to express the fact that for a particular input x, either 01 or 10 are valid outputs. But this fact cannot be represented using *don't cares*, since *don't cares* can only be used to represent sets of the form $f(x) = -1$, $f(x) = -0$, $f(x) = 1-$, $f(x) = 0-$, none of which produce the set $f(x) = \{01, 10\}$.

It is also possible to simplify a cluster of nodes in a network of single output nodes by



(a)

y_5 \ y_4	0	1
0	11	01
1	10	11

(b)

Figure 3.4: Network N_1.

treating the cluster as a single multi-output node.

When using Boolean relations, the input-output flexibility of the network is represented as a relation $O(x, z)$ called the *output observability relation.* This relation specifies the set of allowable primary output patterns for each primary input pattern. The Boolean relation representing the flexibility of implementing a node η_i with inputs u and outputs v is given as:

$$\mathcal{O}_i(u, v) = \forall_x[\forall_{\tilde{u},z}[\overline{L_1(x, \tilde{u})} + \overline{L_2(x, v, z)} + \mathcal{O}(x, z)] + \overline{L_1(x, u)}],$$

where $L_1(x, u)$ denotes the mapping from the primary inputs x to the inputs u of η_i and $L_2(x, v, z)$ expresses the mapping from (x, v) to the primary outputs z.

Consider the network shown in Figure 3.3([3]). Suppose the task is to simplify the network N_1 so that the external behavior of the circuit remains unaltered. It is possible to think of N_1 as a multi-output node and derive Boolean relations for this node. Using these Boolean relations, the network N_1 can be modified to the one shown in Figure 3.4(a). This is because the network

N_2 behaves identically for $y_4y_5 = 00, 11$. Hence, it is possible to modify network N_1 to produce 00, when the original network produced 11, as shown in Figure 3.4 (b). This simplification is not possible with *don't cares* as there is no way to express the mutual constraint between the outputs.

A Boolean relation can not be minimized using a minimizer for minimizing functions with *don't cares*. A special Boolean relation minimizer is required. One such minimizer is GYOCRO [22], a heuristic minimizer patterned on the ESPRESSO paradigm.

3.1.3 Multiple Boolean Relations

The most general set of functions is just an arbitrary subset of functions. Such sets can be compacted into Multiple Boolean Relations (MBRs) [23] such that the function is in the subset if and only if it is compatible with one of the relations. It was shown [23] that certain permissible functions cannot be expressed using Boolean relations. This is because a Boolean relation is a set of functions of a special type. For example, a Boolean relation represents a set of functions that are "output correlated" but input uncorrelated. This means that the choice made for one of the outputs for a input affects what the other outputs will be of that particular input, but does not affect any of the choices allowed for the other inputs. On the other hand, a set of functions are "input correlated" if the choice of the output for one input minterm determines the outputs for other input minterms. An example with input correlation is the set of two 2-output functions $\{f_1, f_2\}$ where $f_1(0) = 01, f_1(1) = 10, f_2(0) = 10, f_2(1) = 01$. This set cannot be represented with a single Boolean relation since the choice of 01 for input 0 forces the choice of 10 for input 1. Multifunctions are used for representing a set of functions that are input correlated but output uncorrelated.

A multiple Boolean relation can express both input correlation and output correlation. Each Boolean relation in the MBR expresses output correlation but the choice of the Boolean relation expresses input correlation.

FPGA rectification [24] refers to the problem of modifying the functions of some LUTs in a network of LUTs so that a given network specification can be satisfied. Furthermore, the supports of the LUTs (whose functions can be changed) must remain unchanged. In [23], a procedure is given for solving this problem using MBRs. This problem cannot be framed using either Boolean relations or multifunctions.

In [23], an algorithm for finding all the functions that are contained in a MBR is provided. It is based on finding the smallest cover of primes of a suitable function derived from the given MBR. Then, the function that best satisfies the optimization criteria can be selected.

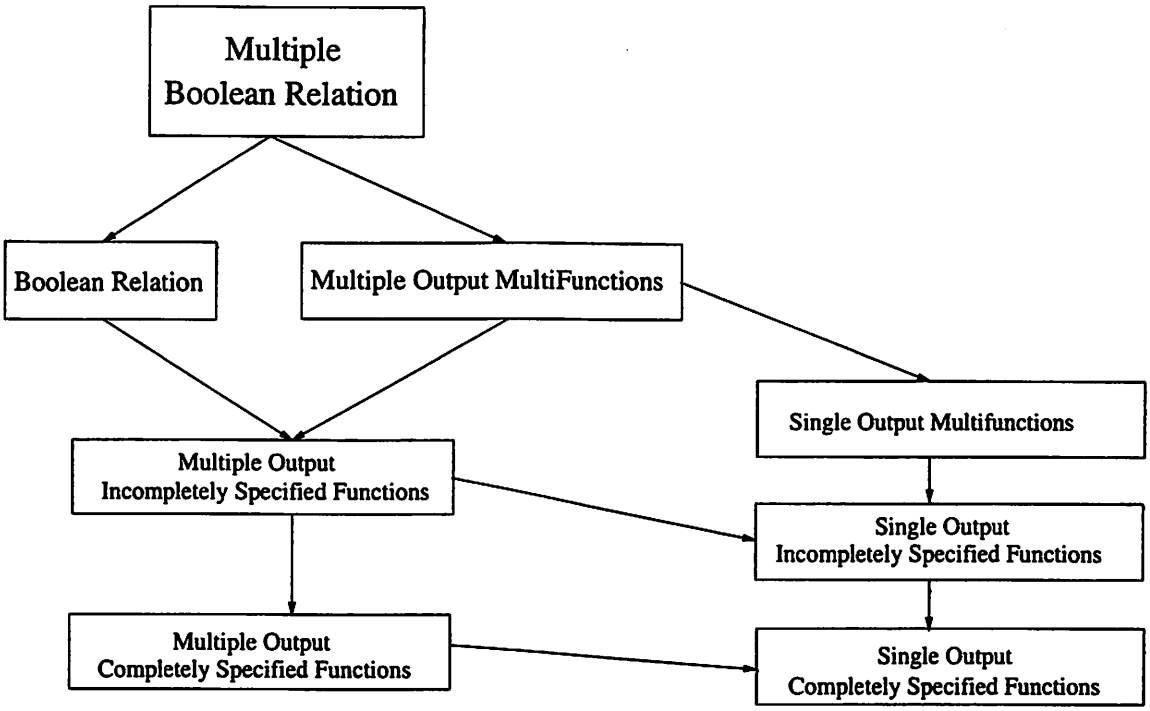


Figure 3.5: Flexibility hierarchy.

3.2 Flexibility Hierarchy

In the previous sections, some of the commonly used techniques for extracting the flexibility of altering the functionality of a node or group of nodes was reviewed. Here, the relative power of the different techniques are compared. This is achieved by ordering them according to the number of functions that can be expressed using the corresponding formalism. Note, that in some cases, the formalism for expressing flexibility may be slightly different from the technique used to extract that flexibility. This difference will be pointed out when relevant.

3.2.1 Completely Specified Functions

Completely specified functions lie at the bottom of the hierarchy. A single output completely specified function f is a many-to-one mapping $f : B^n \rightarrow B$; there are 2^{2^n} possible functions f. A multiple output completely specified function F is a many-to-one mapping $F : B^n \rightarrow B^m$; there are $(2^{2^n})^m = 2^{m.2^n}$ possible functions in F. In both cases, there is no choice in the function to be implemented; each minterm maps to exactly one output minterm.

3.2.2 Incompletely Specified Functions

This formalism is used to represent the flexibility extracted using don't cares. A single output incompletely specified function $\mathcal{F}$ is a many-to-one mapping $\mathcal{F} : B^n \rightarrow Y$, where $Y = \{0, 1, -\}$ and $-$ indicates that the corresponding output is unspecified (it is allowed to be either 0 or 1). There are 3^{2^n} possible single output incompletely specified functions. A multiple output incompletely specified function $\mathcal{F}$ is a many-to-one mapping $\mathcal{F} : B^n \rightarrow Y^m$; there are $3^{m.2^n}$ possible functions.

3.2.3 Multifunctions

A single output multifunction f is a set of many-to-one mappings $\mathbf{f} : B^n \rightarrow B$. Each mapping represents a valid function for the single output. The number of specifications is the combination (power set) of all the possible single output completely specified functions : $2^{2^{2^n}}$. A multiple output multifunction F is m sets of mappings $\mathbf{F} : B^n \rightarrow B$. Each set of mappings represents a set of valid functions for one of the outputs. Since the outputs are chosen independently, the number of possible specifications is the product of the number of specifications for each output : $(2^{2^{2^n}})^m = 2^{m.2^{2^n}}$ Multifunctions allow a choice among several given completely specified functions. Multifunctions can express input correlation.

This formalism was not used previously for expressing flexibility of nodes in combinational networks. In the next chapter, it is shown that the flexibility of single-output nodes expressed using SPFDs is a multifunction.

3.2.4 Boolean Relations

As mentioned before, a Boolean relation $\mathcal{R}$ is a one-to-many multiple output mapping $\mathcal{R} : B^n \rightarrow 2^{B^m}$. Each of the 2^n input minterms maps to a subset of the 2^m possible output minterms, so there are $2^{2^n.2^m}$ possible Boolean relations; $R \subseteq B^n \times B^m$. Boolean relations allow a choice for each input minterm among several output minterms. A value chosen for a particular output may force the values on some remaining outputs; i.e. the output values are correlated.

3.2.5 Multiple Boolean Relations

A Multiple Boolean Relation $\mathcal{M}$ is a set of Boolean relations $\mathcal{M} = \{\mathcal{R}_1, \mathcal{R}_2, \cdots, \mathcal{R}_l\}$, where $\mathcal{R}_i \subseteq B^n \times B^m$. $\mathcal{M}$ represents a collection of multiple output functions. There are $2^{m.2^n}$

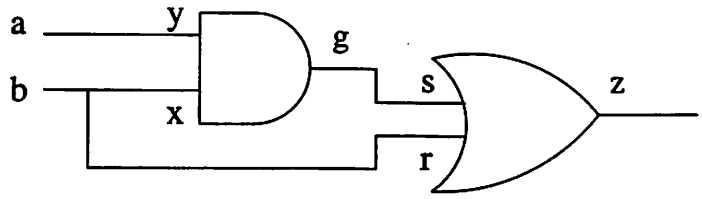


Figure 3.6: Network representation of flexibility.

such functions and $\mathcal{M}$ represents a subset of these, so there are $2^{2^m \cdot 2^n}$ possible specifications.

These different formalisms can be organized according to their expressive power as shown in Figure 3.5. An arrow from representation A to representation B indicates that A is strictly more expressive than B: A can represent a larger set of specifications, but A is usually more difficult to represent and minimize than B. This flexibility hierarchy was first proposed in [23].

3.3 Network Representation of Flexibility

Another set of logic synthesis tools operate directly on a network representation of flexibility, and therefore do not need other representations described earlier i.e. they do not need to derive separate equations for representing *don't cares* or other types of flexibility. These methods are based on determining satisfiability of certain conditions; in particular, whether a node in "testable for stuck-at-0" (or stuck-at-1).

A node is **testable for stuck-at-0** if the functionality of the network would change upon replacing the node with constant 0. Similarly, for a node **testable for stuck-at-1**. A node that is not testable for stuck-at-0 or 1 is called redundant. Redundant nodes can be replaced by a constant, leading to further simplifications. For example, input x of the AND-gate in Figure 3.6 is not testable for stuck-at-0. After replacing it with a constant 0, the network can be further simplified.

The connection between redundancy removal and implementation flexibility was explored by Bartlett et. al [19]. It was proved that if each node in the network is minimized so that it is prime and irredundant using the *don't care* set DC = SDC+ODC+EXDC, then each wire of the network is irredundant. i.e. the network is 100% single stuck-at-1 and stuck-at-0 testable. In general, one may have to iterate this minimization process over all nodes in the network, until no further changes occur, since after minimizing node η_i and then node η_j, it may be possible to further minimize η_j.

Since, these two methods are equivalent, the network representation of the flexibility for expressing the implementation flexibility of a node is not used in the rest of this dissertation.

Chapter 4

Sets of Pairs of Functions to be Distinguished

In this chapter, the concept of Sets of Pairs of Functions to be Distinguished (SPFDs) is introduced. SPFDs were first introduced in the context of FPGA synthesis by Yamashita et. al. [11]. Here, the notion of SPFDs is generalized to general, Boolean networks and it is shown how they can be used to represent and extract the implementation flexibility of a node in a multi-level Boolean network. Later, SPFDs are placed in the flexibility hierarchy described in the previous chapter. The ideas in this chapter are developed for networks of single-output nodes. However, the same ideas can be easily extended to multi-output Boolean networks.

4.1 SPFDs

Definition 4.1 *A function f is said to* **distinguish** *a pair of functions g_1 and g_2 if either one of the following two conditions is satisfied:*

$$g_1 \leq f \leq \overline{g}_2 \tag{4.1}$$

$$g_2 \leq f \leq \overline{g}_1. \tag{4.2}$$

Note that this definition is symmetrical between g_1 and g_2. It is possible to think of g_1 as the onset for f and g_2 as the offset in Condition 4.1 or vice-versa for Condition 4.2.

Example 4.1 *Let $g_1 = ab$ and $g_2 = a\overline{b}$. $f_1 = b$ distinguishes g_1 and g_2 but $f_2 = a$ does not distinguish g_1 and g_2.*

Definition 4.2 *An* **SPFD**

$$\{(g_{1a}, g_{1b}), \ldots, (g_{na}, g_{nb})\}$$

represents a Set of Pairs of Functions to be Distinguished.

Example 4.2 $\{(ab, \overline{a}b), (\overline{a}\overline{b}, a\overline{b})\}$ *is an example of an SPFD.*

A minterm is a special case of a function. So, a set of pairs of minterms that have to be distinguished can also be represented as an SPFD.

Definition 4.3 *A function* f **satisfies** *an SPFD, if* f *distinguishes each pair of the set, i.e.*

$$[((g_{1a} \leq f \leq \overline{g}_{1b}) + (g_{1b} \leq f \leq \overline{g}_{1a})] \wedge \ldots \wedge$$
$$[(g_{na} \leq f \leq \overline{g}_{nb}) + (g_{nb} \leq f \leq \overline{g}_{na})].$$

Example 4.3 *The function* $f_1 = a$ *satisfies the SPFD* $\{(ab, \overline{a}b), (\overline{a}\overline{b}, a\overline{b})\}$ *since it distinguishes each pair in the set. However, the function,* $f_2 = b$*, does not distinguish the functions in the second pair in the set and hence does not satisfy the SPFD.*

The choice of which of the two conditions to satisfy provides the additional flexibility of SPFDs. In a later section, it is shown that SPFDs represent increased flexibility over *don't cares* - the only condition required is that the function implemented at the node satisfy its node SPFD.

4.1.1 Derivation of the SPFD of a node from its function

The SPFD of a node can be derived from its function very easily. The SPFD states that all minterms in the onset of the function have to be distinguished from all minterms in its offset. For example, the SPFD of an OR-gate is $\{(00, 01), (00, 10), (00, 11)\}$, i.e. the offset minterm (00) has to be distinguished from all the onset minterms ($\{01, 10, 11\}$). Note that this SPFD can be satisfied by the NOR function also.

If a node has a *don't care* set associated with it, the SPFD derived from its function specifies that the minterms in the care onset (i.e. onset minterms that are not in the *don't care* set) have to be distinguished from the minterms in the care offset.

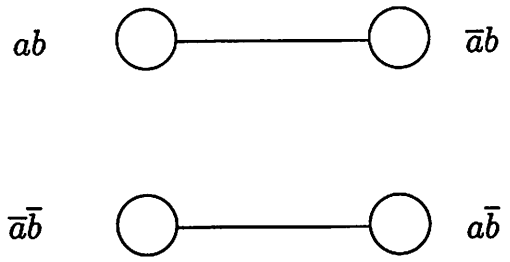


Figure 4.1: SPFD as a graph.

4.1.2 Graphical Representation of SPFDs

An SPFD , $R = \{(g_{1a}, g_{1b}), \ldots, (g_{na}, g_{nb})\}$, can also be represented as a graph, $G = (V, E)$, where

$$\begin{aligned} V &= \{m_k | m_k \in g_{ij}, 1 \leq i \leq n, j = \{a, b\}\}, \\ E &= \{(m_i, m_j) | ((m_i \in g_{pa}) \wedge (m_j \in g_{pb})) \vee \\ &\quad ((m_i \in g_{pb}) \wedge (m_j \in g_{pa})), 1 \leq p \leq n\}. \end{aligned}$$

Every $e \in E$ is referred to as an SPFD edge. For instance, the SPFD in Example 4.2 can be represented by the graph shown in Figure 4.1.

Definition 4.4 *A function f* **satisfies** *an SPFD $R = (V, E)$, if for each edge $(m_i, m_j) \in E$,*

$$f(m_i) \neq f(m_j).$$

Thus, the problem of finding a function that satisfies an SPFD can be reduced to a graph coloring problem. If the SPFD is bipartite i.e. only two colors are required to color the SPFD, then all functions that satisfy the SPFD can be enumerated easily. For nodes in a network of single-output nodes, the SPFDs are mostly bipartite (how they could become non-bipartite is described in Chapter 5). Hence, for single-output nodes, it is possible to explore a lot more functions than allowed by previous methods.

4.1.3 SPFDs and Information

An SPFD can also be thought of as a graph that encapsulates information. For combinational networks, information is the ability to distinguish one primary input minterm from another. Each pair of such minterms is an atomic unit. Each pair in an SPFD associated with a node can

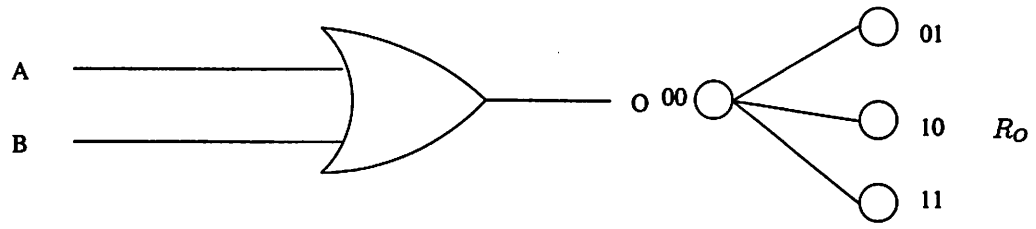


Figure 4.2: OR gate.

distinguish some of the primary input minterms. Since an SPFD can distinguish some of the primary input minterm pairs, hence it provides information. This connection between SPFDs and information can be exploited in a number of interesting applications of SPFD.

4.1.4 Notational Representation of SPFDs

Given an SPFD $\{(g_{1a}, g_{1b}), \cdots, (g_{na}, g_{nb})\}$, it can be denoted as a relation $R(X, X')$, where X and X' denote two sets of variables representing the same input space. For each pair in the SPFD (g_{ia}, g_{ib}), $R(g_{ia}(X), g_{ib}(X')) = 1$, where g_{ia} is expressed in terms of the X variables and g_{ib} is expressed in terms of the X' variables. In some computations, the SPFD is represented as a symmetric relation. Thus, $R(g_{ia}(X), g_{ib}(X')) = 1$ iff $R(g_{ib}(X), g_{ia}(X')) = 1$.

Let $X = (x_1, x_2, \cdots, x_n)$. The computation,

$$R_i(X, X') = R(X, X') \wedge (x_i \neq x_i'),$$

denotes the SPFD edges that can be distinguished by the x_ith variable i.e. an edge $e = (m_1, m_2) \in R_i(X, X')$ if m_1 and m_2 differ in the value of the x_ith variable. Similarly, $R(X, X') \wedge (x_i = x_i')$ denotes the edges that cannot be distinguished by the x_ith variable.

Example 4.4 *Consider the simple OR-gate shown in Figure 4.2. Suppose the input A is associated with two variables, y_a and y_a'. Similarly, let B be associated with variables y_b and y_b'. Then, $R_o \wedge (y_a \neq y_a') = \{(00, 10), (00, 11)\}$ and $R_o \wedge (y_a = y_a') = \{(00, 01)\}$.*

In the rest of this chapter, the concept of the minimum SPFD of a node is introduced. The flexibility expressed by a minimum SPFD is compared to that represented by an ODC. Then, a brief sketch is provided for how compatible SPFDs can be computed for a network. It is also proved that CODCs can be generated using a version of the SPFD generation algorithm. Thus, compatible SPFDs also represent more flexibility than CODCs.

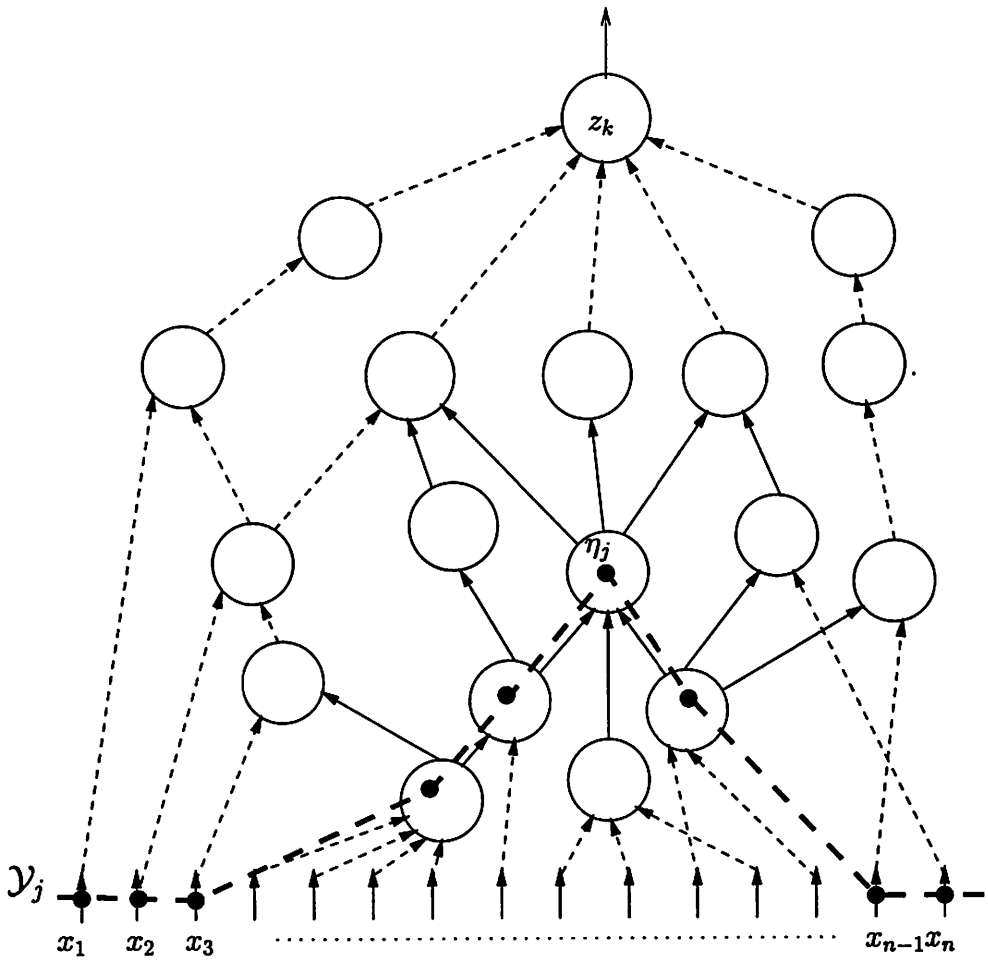


Figure 4.3: The set of nodes marked by dots denotes the separator $\mathcal{Y}_j$.

4.2 Minimum SPFD of a node

Definition 4.5 *Given the SPFDs of its primary outputs, the* **minimum SPFD** *of a node η_j is the minimum set of edges that have to be distinguished by the node. Once the node function is modified using its minimum SPFD, the functionalities of some nodes in the network may have to be changed to ensure correct functionality of the network. However, the functions of the nodes in the transitive fanin of the node should remain unchanged. Also, the topology of the network should remain unchanged, except possibly the removal of η_j.*

So, the **minimum SPFD** of a node denotes the unique information that is provided by the node to the outputs of the network for ensuring correct functionality of the network, given the network topology and the information provided by its fanins.

Definition 4.6 *A* **separator** *containing η_j is a set of nodes, $S = S' \cup \eta_j$, that satisfies the following conditions:*

1. *The primary inputs of the network are completely disconnected from the primary outputs when all the nodes in S are removed.*
2. *A node $\eta_k \in S$ that is a transitive fanin of η_j has to fanout to a node not in the transitive fanin of η_j.*

In the next section, an algorithm is provided for computing the minimum SPFD of η_j using the notion of the separator. The second condition in the definition of the separator ensures that the functionalities of the nodes in the transitive fanin of η_j will remain unchanged after η_j is modified to satisfy its minimum SPFD.

4.2.1 Algorithm for computing the minimum SPFD of a node

Here, the algorithm for computing the minimum SPFD of the node η_j is described. Consider the separator set $\mathcal{Y}_j$ (shown in Figure 4.3) containing η_j.

It includes:

1. The node η_j.
2. All the primary inputs of $\mathcal{N}$ that are not in the transitive fanin of η_j.
3. All nodes (including primary inputs) in the transitive fanin of η_j that fanout to at least one node that is not in the transitive fanin of η_j.

It is easy to see that removing these nodes will disconnect the primary outputs from the primary inputs. Also, each node in $\mathcal{Y}_j$ that is a transitive fanin of η_j has a fanout to at least one node not in the transitive fanin of η_j. Hence, $\mathcal{Y}_j$ is a separator containing η_j.

Algorithm **com_minspfd_for_sep**($\mathcal{N}$, $\mathcal{Y}_j$):

1. *For each $z_k \in PO(\mathcal{N})$*
   (a) *Compute $U_k = \{\eta_i | \eta_i \in \mathcal{Y}_j$ and $\{\eta_i \in FI(\eta_p)$ such that $(\eta_p \notin \mathcal{Y}_j) \wedge (\eta_p \in TFI(z_k))\}\}$. U_k contains all the nodes in the separator $\mathcal{Y}_j$ that provide all the information required by z_k.*

(b) *Derive the SPFD of z_k in terms of the primary inputs. This SPFD specifies that all the primary input minterms in the onset of z_k have to be distinguished from all the primary input minterms in the offset of z_k. Let it be denoted by S_k.*

(c) *Compute the SPFDs of all the nodes in $U_k \setminus \{\eta_j\}$. Given a node $\eta_i \in U_k \setminus \{\eta_j\}$, its SPFD S_i specifies that the primary input minterms in the onset of η_i (f_i^g) have to be distinguished from the primary input minterms in the offset of η_i ($\overline{f_i^g}$).*

(d) *Compute*

$$C_k = \cup_{\eta_i \in U_k, \eta_i \neq \eta_j} S_i.$$

C_k denotes the set of edges that can be distinguished by all the nodes in $U_k \setminus \{\eta_j\}$.

(e) *Compute $R_{jk} = S_k \wedge \overline{C_k}$. Thus, R_{jk} denotes the edges in S_k, the SPFD of z_k, that cannot be distinguished by the other nodes in $\mathcal{Y}_j$ and hence have to be distinguished by η_j.*

2. *$R_j = \cup_{i=1}^{m} R_{jk}$, where m is the number of primary outputs of $\mathcal{N}$.*

Once the function at η_j is simplified using a function that satisfies its minimum SPFD, the SPFDs of the nodes between $\mathcal{Y}_j$ and the primary outputs of $\mathcal{N}$ may have to be modified. This is due to the fact that the information content of these nodes may have to be modified to account for the reduced information available at η_j, due to its new simplified function. These nodes will then have to be resynthesized using their new SPFDs. In a later chapter, the algorithm for computing the new functionalities of the affected nodes is provided.

The same algorithm can be used with different separators of η_j (say $\mathcal{Y}_j^1$ and $\mathcal{Y}_j^2$ illustrated in Figure 4.4). In order to satisfy Condition 2 of the definition of a separator, the other separators of η_j should be lie between the nodes in $\mathcal{Y}_j$ and the primary outputs of $\mathcal{N}$. Thus, $\mathcal{Y}_j$ is the separator closest to the primary inputs. In the sequel, $\mathcal{Y}_j$ is denoted as $\mathcal{Y}_j^0$ to differentiate it from other separators. The SPFD computed in Step 2 of algorithm **com_minspfd_for_sep** using separator $\mathcal{Y}_j^k$ is denoted as R_j^k.

Next, it is argued that R_j^0 computed using separator $\mathcal{Y}_j^0$ has the least number of edges, when compared to any other R_j^k.

Lemma 4.1 *Let the SPFD of node η_i be denoted as $R_i(X, X')$. Then,*

$$R_i(X, X') \subseteq \cup_{\eta_k \in FI(\eta_i)} R_k(X, X'),$$

where $R_k(X, X')$ is the SPFD of η_k.

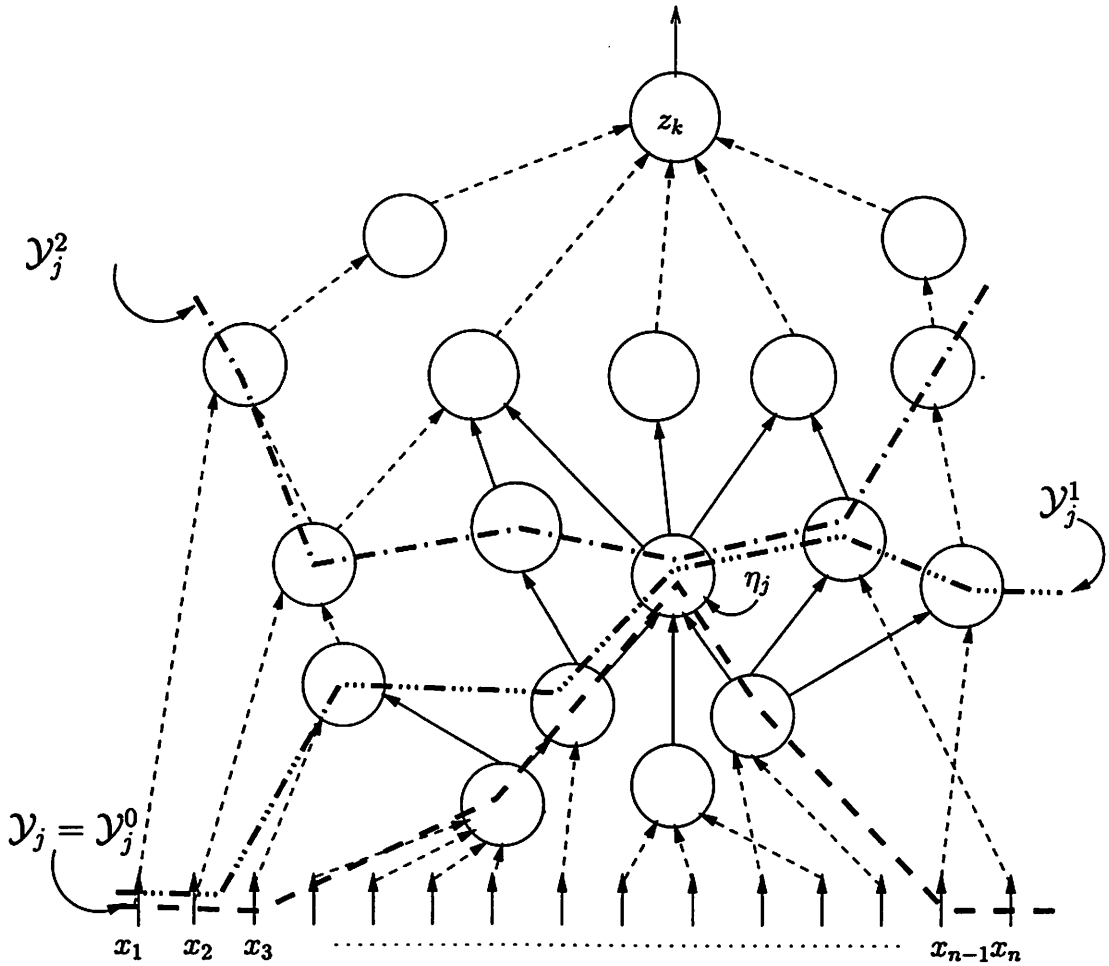


Figure 4.4: Separators: $\mathcal{Y}_j^0$, $\mathcal{Y}_j^1$ and $\mathcal{Y}_j^2$; the nodes connected by a dashed line indicate a separator. Each of these separators can be used in the algorithm **com_minspfd_for_sep** for obtaining an SPFD of η_j. The SPFD computed using $\mathcal{Y}_j^0$ is the minimum SPFD of η_j.

Proof Assume there exists an edge $(m, m') \in R_i(X, X')$ such that $(m, m') \notin \cup_{\eta_k \in FI(\eta_i)} R_k(X, X')$. This implies that $(m, m') \notin R_k(X, X')$ for any fanin η_k of η_i. Hence, m and m' can produce the same outputs for each of the fanins of η_i, i.e. the image of m and m' onto the fanin space Y_i of η_i, will both produce the same minterm y. But, m and m' have to be distinguished at the output of η_i. This implies that the same minterm $y \in Y_i$ has to produce two different values at the output of η_i. This is impossible since all the nodes in $\mathcal{N}$ are deterministic. □

Theorem 4.1 *R_j^0 computed using the separator $\mathcal{Y}_j^0$ in* **com_minspfd_for_sep** *has the least number of edges.*

Proof Suppose there exists another separator which can be used in the above algorithm to produce

an SPFD with fewer edges (Step 2 of **com_minspfd_for_sep**). Let this new separator be $\mathcal{Y}_j^k$ and the SPFD computed for η_j using $\mathcal{Y}_j^k$ in algorithm **com_minspfd_for_sep** be R_j^k.

It is known that $\mathcal{Y}_j^0$ is the separator closest to the primary inputs. Thus $\mathcal{Y}_j^k$ can only contain nodes that are either in $\mathcal{Y}_j^0$ or are in the transitive fanout of nodes in $\mathcal{Y}_j^0$. By Lemma 4.1, the SPFD of a node is always a subset of the union of the SPFDs of its fanins. Hence C_k computed in Step 1(d) using separator $\mathcal{Y}_j^k$ is a subset of C_k computed using $\mathcal{Y}_j^0$. Hence R_j^k has to be a superset of R_j since S_k is the same in both computations in Step 1(e). □

Theorem 4.2 *If any two minterms that are connected by an edge in R_j^0 evaluate to the same value at the output of η_j, the network specification cannot be satisfied if the functionality of the nodes in the transitive fanin of η_j and the topology of the network is unchanged.*

Proof It is proved below that given the functionality of the nodes in the transitive fanin of η_j and the topology of $\mathcal{N}$, if any two minterms have an edge between them in R_j^0, then they cannot be assigned the same value without affecting network functionality. Assume there exists an edge $e = (m, m') \in R_j^0$ such that m and m' can be assigned the same value at the output of η_j. By Step 1(e) of **com_minspfd_for_sep**, e has to belong to the SPFD of some primary output. Let that primary output be z_k. Furthermore, $e \notin C_k$.

Thus e is not distinguished by any of the other nodes in U_k. Since the functionalities of the nodes in the transitive fanin of η_j and the topology of $\mathcal{N}$ remains unchanged (i.e. no new wires are added between previously unconnected nodes), m and m' evaluate to the same value for all the nodes in U_k (computed in Step 1(a) of **com_minspfd_for_sep**) other than η_j. By the above assumption, m and m' also evaluate to the same value at the output of η_j. Hence, m and m' evaluate to the same value for all the nodes in U_k. Let y denote the minterm obtained by computing the image of both m and m' from the primary inputs to the nodes in U_k. Since $e = (m, m')$ belongs to the SPFD of z_k, m and m' have to evaluate to different values at the output of z_k. This implies that the same minterm y at U_k can produce different values at z_k (this is because the nodes in U_k uniquely determine the value at z_k). This is impossible since $\mathcal{N}$ is deterministic. □

In Chapter 7, it is argued that no edges have to be added to R_j^0 for ensuring that the network functionality can be maintained after suitable modification of the functions of the nodes lying between $\mathcal{Y}_j^0$ and the primary outputs. Thus, R_j^0 is the **minimum SPFD** of η_j.

While the separator $\mathcal{Y}_j^0$ (shown in Figure 4.3) provides the minimum SPFD, it may still be beneficial to use the same algorithm with different separators (say $\mathcal{Y}_1$ and $\mathcal{Y}_2$ illustrated in Figure 4.4). Each separator provides an SPFD that is a superset of the minimum SPFD. However, it may be

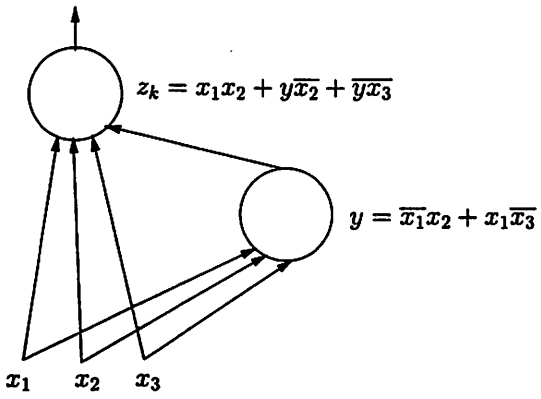


Figure 4.5: Example for Minimum SPFD computation.

more efficient to use a separator closer to the primary outputs since the number of nodes that have to resynthesized can be reduced.

It may also be useful to compute the minimum information that η_j provides to another node η_p in the network, where η_p is such that all paths from η_j to the primary outputs have to pass through η_p. This could be useful because it may be undesirable to change the network beyond η_p for efficiency.

Definition 4.7 *A node η_p is a* **dominator** *of another node η_j, if all paths from η_j to the primary outputs have to pass through η_p.*

Definition 4.8 *Let η_p be dominator of η_j. The* **minimum SPFD of η_j wrt to η_p** *is the unique information that η_j provides to η_p such that the functionalities of the nodes in the transitive fanout of η_p can remain unchanged after η_j is simplified with its minimum SPFD. The functionalities of nodes in the transitive fanin of η_j and the topology of the network must remain also unchanged.*

For computing the minimum SPFD of η_j wrt to its dominator η_p, the sub-network consisting of η_p and all the nodes in its transitive fanin is taken as the input network $\mathcal{N}$ for **com_minspfd_for_sep**. The node η_p is treated as the primary output. Its SPFD is derived from its original function and its ODC. With these modifications, the same algorithm can be used.

Definition 4.9 *The* **minimum SPFD of a wire** $w_{\eta_k \to \eta_j}$ *is the set of edges in the minimum SPFD of η_j that can only be distinguished by the function at η_k.*

The concept of the minimum SPFD of a node wrt to one of its dominators and the minimum SPFD of a wire has applications in rewiring (Chapter 6). Obviously, the notion of a dominator can be extended to a set of nodes dominating η_j.

Unless otherwise specified, the minimum SPFD of a node η_j refers to R_j^0 computed wrt to the primary outputs using the separator $\mathcal{Y}_j^0$. To differentiate this special separator from the other separators, it is referred to as the **minimum separator**. For all other minimum SPFD computations, the parameters will be explicitly mentioned.

Example 4.5 *Consider the circuit shown in Figure 4.5. The separator is* $\{x_1, x_2, x_3, y\}$. *Since all the primary inputs are included in the separator, all the information to* z_k *can be directly provided by them. Hence, the minimum SPFD is empty. So, the node* y *can be replaced with a constant node. But, the functionality of* z_k *has to be modified to reflect the changes in* y.

4.2.2 Connections to Previous Work

Theorem 4.3 *If a minterm is an ODC of* η_j, *then it does not appear in the minimum SPFD of* η_j.

Proof Let m be an ODC of η_j. Assume it appears in the minimum SPFD of η_j and let m belong to the onset of z_k.

Since m appears in the minimum SPFD of η_j, there exists a minterm $\tilde{m}$ in the offset of z_k such that $(m, \tilde{m})$ is not distinguished by any other node in the separator i.e. for all the other nodes in the minimum separator m and $\tilde{m}$ produce the same output value. Also, m can be set to have the same output value as $\tilde{m}$ without affecting network behavior since m is an ODC of η_j. Then all the nodes in the separator set will have identical values for m and $\tilde{m}$ and hence cannot produce different values at z_k (as z_k is a deterministic function). But m and $\tilde{m}$ belong to the onset and offset of z_k, respectively and need to be distinguished. Hence $(m, \tilde{m})$ has to be distinguished by some other node in the separator. Thus the assumption that m belongs to the minimum SPFD of η_j is incorrect. □

Note however that it is not true that if a minterm is missing in the minimum SPFD of η_j, it is also an ODC. In the previous example, the minimum SPFD of y was zero but its ODC is given as $x_1x_2 + \overline{x_2x_3} + x_2x_3$. So, the minterm $\overline{x_1x_2}x_3$ does not belong to the minimum SPFD of y but it is not an ODC of y either. This is due to the fact that the minimum SPFD algorithm requires the re-computation of the functionalities of the nodes between $\mathcal{Y}_j^0$ and the primary outputs. But, the ODC computation does not require that. So, minimum SPFDs allow more changes to the functionality of a node compared to ODCs.

Theorem 4.4 *If an output observability relation* $O(x, z)$ *can be represented as an SPFD, any func-*

tion that is compatible with the Boolean relation

$$\mathcal{O}_j(u,v) = \forall_x[\forall_{\tilde{u},z}[\overline{L_1(x,\tilde{u})} + \overline{L_2(x,v,z)} + \mathcal{O}(x,z)] + \overline{L_1(x,u)}]$$

of η_j also satisfies its minimum SPFD.

Proof Since the observability relation can be represented as an SPFD, each primary output has an SPFD that specifies what pairs of minterms it has to distinguish. Consider a function f that is compatible with the Boolean relation $\mathcal{O}_j$. Assume f does not satisfy the minimum SPFD. Hence there exists an edge $e = (m, m')$ in the minimum SPFD of η_j such that $f(m) = f(m')$. By Theorem 4.2, if both m and m' are assigned the same value, the network functionality is affected if the functionalities of the fanins and the network topology are left unchanged. But in the above computation of the Boolean relation, neither of them are changed. Hence, m and m' cannot be assigned the same value in the function f. Thus the assumption that f does not satisfy the minimum SPFD is incorrect. □

In Section 4.4, a Boolean relation is provided that cannot be expressed using SPFDs. If the *output observability relation* of the network is one such Boolean relation, then the above theorem is no longer valid.

4.3 Compatible SPFDs

As in the case of ODCs, it is not practical to simplify each node in $\mathcal{N}$ using its minimum SPFD, recompute the functions of all the nodes in the transitive fanout of $\mathcal{Y}$ and then move on to the next node. Hence, the notion of compatibility is particularly important for SPFDs. In this section, a brief sketch of a procedure that computes compatible SPFDs for all the nodes in the network is provided. Thus, two nodes can be changed using their SPFDs without having to re-compute their SPFDs from scratch. Or, in information-theoretic terms, the information provided by the nodes are compatible. For convenience, compatible SPFDs are simply referred to as SPFDs in the rest of this dissertation. In the next chapter, a detailed explanation of an efficient algorithm used for computing SPFDs of all the nodes in the network is provided. Here, an intuitive explanation of the algorithm is presented.

SPFDs can be computed for an entire network by starting at the primary outputs. The SPFD of a primary output specifies that the care onset of its function has to be distinguished from

the care offset of its function[1]. Keeping in mind that only one path is necessary, the algorithm starts at a primary output, and assigns to each of its immediate inputs, a subset of the information that the input is responsible for. This assignment is not necessarily unique. Each piece of information (a pair of primary input minterms that the output must distinguish) can be assigned to exactly one of the fanin wires of that output node. The algorithm then considers a node η_j in the next level away from the primary outputs. After all the primary outputs have distributed information requirements to their fanins, the information assigned to each fanout wire of η_j is summed for obtaining the information η_j must propagate to its fanouts. This required information is then distributed to the fanins of η_j. As the network is traversed backward, each node is assigned a set of information requirements; this is an SPFD for the node.

The SPFDs are compatible in the sense that the information content of each SPFD is made compatible to the information content of the SPFD of other nodes that can possibly affect it. Of course, depending on the manner of distributing the edges of the SPFD of a node to its fanins, different results can be obtained. In the next section, a particular scheme for distributing the SPFD edges of a node to its fanins is considered, that can be used for deriving CODCs using SPFDs. Other interesting schemes for distributing the SPFD edges of a node to its fanins are also presented in other chapters of this dissertation.

4.3.1 Emulating CODCs using SPFDs

Consider a node η_p with n fanins, $\{\eta_1, \eta_2, \cdots, \eta_n\}$. The ordering between the fanins, denoted as $\mathcal{O}$, is given as:

$$\eta_1 >_f \eta_2 >_f \cdots >_f \eta_n.$$

Thus, η_1 has the greatest flexibility and η_n has the least flexibility. The SPFDs of η_p and its fanins are expressed in terms of Y_p and Y_p' (the fanin spaces of η_p). In the rest of this section, η_i is often referred to as the ith fanin of η_p.

Algorithm **compute_codc_with_spfd**(η_p, $\mathcal{O}$):

1. *Compute the SPFD of η_p. Denote this as $R_p(Y_p, Y_p')$.*

$$R_p(Y_p, Y_p') = f_p(Y_p)\overline{f_p}(Y_p') + \overline{f_p}(Y_p)f_p(Y_p').$$

Thus, $R_p(Y_p, Y_p')$ specifies that the onset of η_p has to be distinguished from the offset of η_p.

[1]The care onset and care offset are derived from the onset and offset by intersecting them with the complement of the *external don't care* set, respectively.

2. *Process the fanins in order starting from η_1 to η_n.*

3. *For each fanin η_i, repeat the following steps:*

   (a) *Compute*

   $$R_i^*(Y_p, Y_p') = R_p(Y_p, Y_p')(y_i \neq y_i') \prod_{k=(i+1)}^{n} (y_k = y_k').$$

   Thus, $R_i^*(Y_p, Y_p')$ denotes the edges in $R_p(Y_p, Y_p')$ that can be distinguished by η_i but cannot be distinguished by the nodes with lesser flexibility.

   (b) *For k = 1, (i-1)*

      i. *Compute*

      $$R_i^*(Y_p, Y_p') = R_i^*(Y_p, Y_p') \wedge (\overline{R_k(Y_p, Y_p')} \wedge (y_k \neq y_k')).$$

      Thus, $R_i^*(Y_p, Y_p')$ is modified by removing the edges in it that are already in $R_k(Y_p, Y_p')$ and can be distinguished by η_k.

   (c) *Remove the minterms in $R_i^*(Y_p, Y_p')$ that no longer have any edges connected to them and make the remaining $R_i^*(Y_p, Y_p')$ completely connected. Let this SPFD be denoted as $R_i(Y_p, Y_p')$. This is the SPFD of η_i.*

In the next few sections, it is proved that the algorithm **compute_codc_with_spfd** can be used for computing the CODCs of the fanins of a node.

4.3.1.1 Additional Notation

$C_i(Y_p)$ denotes the CODC of η_i obtained by using Savoj's algorithm and the ordering given in Section 4.3.1. Thus,

$$C_i(Y_p) = (\partial f_p/\partial y_1 + \forall_{y_1}) \cdots (\partial f_p/\partial y_{i-1} + \forall_{y_{i-1}})(\overline{\partial f_p/\partial y_i}).$$

Given the SPFD $R_i(Y_p, Y_p')$ of a fanin η_i obtained from the algorithm **compute_codc_with_spfd**, let

$$V_i(Y_p) = \exists_{Y_p'} R_i(Y_p, Y_p').$$

$V_i(Y_p)$ denotes the set of all minterms that have at least one edge attached to them in $R_i(Y_p, Y_p')$.

Consider a minterm m of the Y_p space. The notation $m(j)$ is used to refer to the value of the jth fanin in the minterm m. Also, $m(j = a)$ denotes the minterm that is obtained by setting the value of the jth fanin to a in minterm m. The minterm obtained by toggling the value of the jth fanin in minterm m is denoted as $m(j \downarrow)$.

4.3.1.2 Formal Proof

Theorem 4.5 *For each fanin, a minterm $m \in C_i(Y_p)$ iff it is not present in $R_i(Y_p, Y_p')$. Or in other words,*

$$C_i(Y_p) = \overline{V_i(Y_p)}.$$

Proof Proof by induction on i.

Base Case :

Consider the case of the first fanin i.e. $i = 1$.

($\leftarrow$): Consider a minterm m in $C_1(Y_p)$. Assume that $m \in f_p$. It is easy to see that for all $\tilde{m} \in \overline{f_p}$, m and $\tilde{m}$ differ in some other position besides the value of the first fanin. Thus, m will not be included in $R_1^*(Y_p, Y_p')$ and hence will not be included in $R_1(Y_p, Y_p')$. Thus, $C_1(Y_p) \subseteq \overline{V_1(Y_p)}$.

($\rightarrow$) : Similarly, consider a minterm $m \notin C_1(Y_p)$. Then, $m \in (\partial f_p / \partial y_1)$. Assume $m \in f_p$. So, this implies that there must exist a minterm $\tilde{m} \in \overline{f_p}$ such that $\tilde{m} = m(1 \downarrow)$. Now, the edge $e = (m, \tilde{m})$ can only be distinguished by η_1 and thus has to be included in $R_1(Y_p, Y_p')$. Thus, $m \in V_1(Y_p)$. Hence, $\overline{C_1(Y_p)} \subseteq V_1(Y_p)$ or $\overline{V_1(Y_p)} \subseteq C_1(Y_p)$. Hence, $C_1(Y_p) = \overline{V_1(Y_p)}$.

Inductive Step:

Assume it is true for all fanins less than $(i-1)$. Now, it has to be proved for the ith fanin.

($\leftarrow$): It has to be shown that if a minterm m is an element of $C_i(Y_p)$, then it is not an element of $V_i(Y_p)$. Let $m \in C_i(Y_p)$. Let $m \in f_p$. Also assume that $m \in V_i(Y_p)$. Thus, there must exist a minterm $\tilde{m}$ such that the edge $e = (m, \tilde{m}) \in R_i(Y_p, Y_p')$ and the following two conditions are satisfied:

1. m and $\tilde{m}$ are same in the values of the fanins greater than i. (Step 3(a) of the algorithm.)
2. $e = (m, \tilde{m})$ is not contained in the SPFD $R_k(Y_p, Y_p')$, where $k < i$ and $m(k) \neq \tilde{m}(k)$.

Since $m \in C_i(Y_p)$, hence m is not sensitive to the value of the ith fanin. So, it is easy to see that m and $\tilde{m}$ differ in more fanins besides the ith fanin. Let this subset of fanins be denoted as S. Now, both m and $\tilde{m}$ cannot simultaneously appear in the SPFD $R_k(Y_p, Y_p')$ of any $\eta_k \in S$. Otherwise, condition (2) would not be satisfied by the edge $e = (m, \tilde{m})$.

Let $J \subseteq S$ such that m is a CODC for the fanins in J. Consider the minterm,

$$m' = m(l = \tilde{m}(l)); \{(\eta_l \in J) \wedge (\eta_l = \eta_i)\}.$$

Thus m' is obtained from m by setting the values of the fanins in J and the value of η_i equal to their corresponding values in $\tilde{m}$. Since, by assumption m is a CODC of the fanins in J and of η_i, hence

$m' \in f_p$. Thus, the edge $e' = (m', \tilde{m})$ needs to be distinguished. It is easy to see that this edge can only be distinguished by one or more fanins in $S \setminus J$ [2]. So $\tilde{m}$ has to belong to the SPFD $R_k(Y_p, Y_p')$ of a fanin η_k, such that $\eta_k \in S \setminus J$. Also, since m is not a CODC of any fanin $\eta_k \in S \setminus J$, hence by IH, m has to belong to the corresponding SPFD $R_k(Y_p, Y_p')$.

Hence, there exists at least one fanin η_k, where $k < i$, such that both m and $\tilde{m}$ appear in its SPFD $R_k(Y_p, Y_p')$. Hence, by Step 3(c) of the algorithm, $e \in R_k(Y_p, Y_p')$. Thus, the algorithm will not include e in $R_i(Y_p, Y_p')$. Hence, a contradiction occurs. Thus, $C_i(Y_p) \subseteq \overline{V_i(Y_p)}$.

($\rightarrow$): It has to be shown that if a minterm m does not appear in $V_i(Y_p)$, then it is a CODC. Consider a minterm, $m \notin V_i(Y_p)$. Let $m \in f_p$. This happens in either one of the two cases:

1. m does not appear in Step 3(a) of the algorithm. This means that for every $\tilde{m} \in \overline{f_p}$, m and $\tilde{m}$ always differ in some η_j, where $j > i$. Thus, toggling the values of all the fanins $\leq i$ (i.e $\eta_1 \cdots \eta_i$) in m will necessarily yield a minterm m' such that $m' \in f_p$. Thus,

$$m \in \forall_{y_1, y_2, \cdots, y_{i-1}} \overline{\partial f_p / \partial y_i}.$$

   Thus, m is a CODC of fanin η_i.

2. There exists edges in $R_i^*(Y_p, Y_p')$ in Step 3(a) but are not included in $R_i(Y_p, Y_p')$. This implies that for each such edge $e = (m, \tilde{m})$ (i) m and $\tilde{m}$ are identical in all the fanins $> i$ and (ii) m and $\tilde{m}$ differ in the ith fanins and some other fanin j, where $j < i$. Hence, just by toggling the value of the ith bit in m, it cannot become an offset minterm. Thus,

$$m \in \overline{\partial f_p / \partial y_i}.$$

   So, m is definitely an ODC. In order to show that it is a CODC, it has to be shown that it is also compatible with the CODCs of the fanins from $\{\eta_1, \cdots, \eta_{i-1}\}$. Proof by contradiction. Assume that m is not compatible with the CODCs of the previous fanins. Let m be a CODC of a subset J of the fanins $\{\eta_1, \cdots, \eta_{i-1}\}$[3]. Construct a minterm m' as follows:

$$m' = m(k \downarrow), \{(k = i) \cup (\eta_k \in J)\}.$$

   It is obtained from m by toggling the values of the fanins in J and the value of η_i. Since, m is not compatible with the CODCs of the previous fanins, hence m' has to belong to $\overline{f_p}$. Now, the edge $e' = (m, m')$ has to be distinguished. But, e' can only be distinguished only by the

[2] By construction, m' and $\tilde{m}$ differ only in these fanins.

[3] Note that if m is not a CODC of any fanin $< \eta_i$, then it has to be compatible with their CODCs.

fanins in J or by fanin η_i. Since, m is a CODC for the fanins in J, hence by IH, m does not appear in the SPFDs of the fanins in J. Thus, e can be distinguished only by fanin η_i. But, this contradicts the initial assumption that m does not appear in $V_i(Y_p)$. Hence, m is both an ODC of η_i and it is also compatible with the CODCs of the fanins from $\eta_1, \cdots, \eta_{i-1}$. Hence, m is a CODC.

Thus, $\overline{V_i(Y_p)} \subseteq C_i(Y_p)$. Hence,

$$C_i(Y_p) = \overline{V_i(Y_p)}.$$

□

Hence, the above theorem shows that CODCs can be computed using **compute_spfd_with_codc**. Thus, SPFDs can represent all the flexibility that CODCs represent.

4.4 SPFDs in the Flexibility Hierarchy

In this section, SPFDs are placed in the flexibility hierarchy shown in Figure 3.5. First, it is argued that the flexibility expressed by SPFDs cannot be expressed by either multi-output multifunctions or Boolean relations. Each of these formalisms are considered in turn.

The following example illustrates why the flexibility of SPFDs cannot be completely covered by multifunctions. Consider the following problem: Given a set of single-output or multi-output nodes in a network, suppose it is necessary that some pairs of minterms have to be distinguished by the group of nodes. Also, suppose it is sufficient, if for each pair of minterms, the output of a single node in the set is different. This problem can be very easily formulated using SPFDs. The set of minterm pairs that have to be distinguished forms the SPFD for the group of nodes. Any valid coloring of this SPFD can provide functions for all the nodes in the set that satisfies the given condition. The set of functions represented by this SPFD cannot be captured by any multi-output multifunction since these are incapable of expressing any kind of output correlation. Hence, the requirement that only one output needs to differ for a particular pair of minterms cannot be expressed by them.

Similarly, a Boolean relation cannot express all the flexibility that an SPFD can represent. Consider the following SPFD $\{(00,01),(00,10),(00,11),(01,10),(01,11),(10,11)\}$. Any function that satisfies the SPFD has to assign different values to all the minterms in the SPFD. The functions shown in Table 4.1 satisfy the SPFD.

input	f_1	f_2	f_3	f_4
00	00	01	10	11
01	01	10	11	00
10	10	11	00	01
11	11	00	01	10

Table 4.1: Functions that satisfy SPFD $\{(00,01),(00,10),(00,11),(01,10),(01,11),(10,11)\}$.

But all these functions cannot be captured by a single Boolean relation. This is because Boolean relations cannot capture input correlation. Hence, the requirement that the output value of one input minterm has to be distinguished from the output values of all the other input minterms cannot be captured using a Boolean relation.

Thus, the flexibility expressed by SPFDs cannot be captured using either multi-output multifunctions or Boolean relations. The functions that satisfy an SPFD can be captured by a special type of MBR. This MBR consists of a single Boolean relation and set of constraints:

1. The Boolean relation specifies that each input minterm can take any value from the set $\{0,\cdots,n\}$, where n is the number of input minterms.

2. The constraints are derived from the edges in the SPFD graph. Each edge $e=(m,m')$ in the SPFD graph is translated into a constraint that states that the output produced by m cannot be equal to the output produced by m'.

All functions that are compatible with this MBR definitely satisfy the SPFD. This is because any function f that satisfies this MBR ensures that for any two minterms m, m' that have an edge between them in the SPFD graph, $f(m) \neq f(m')$. Similarly, all functions that are captured by the SPFD are also functions that are contained in the above MBR.

SPFDs can completely capture all the flexibility expressed by multi-output multifunctions. This is because multi-output multifunctions express only input correlations, which can be easily expressed using SPFDs. On the other hand, the flexibility of Boolean relations cannot be completely captured using SPFDs. As an example, consider the following Boolean relation:

$$\begin{aligned} 00 &\rightarrow 0 \\ 01 &\rightarrow 1 \\ 10 &\rightarrow 0,1 \\ 11 &\rightarrow 1,2 \end{aligned}$$

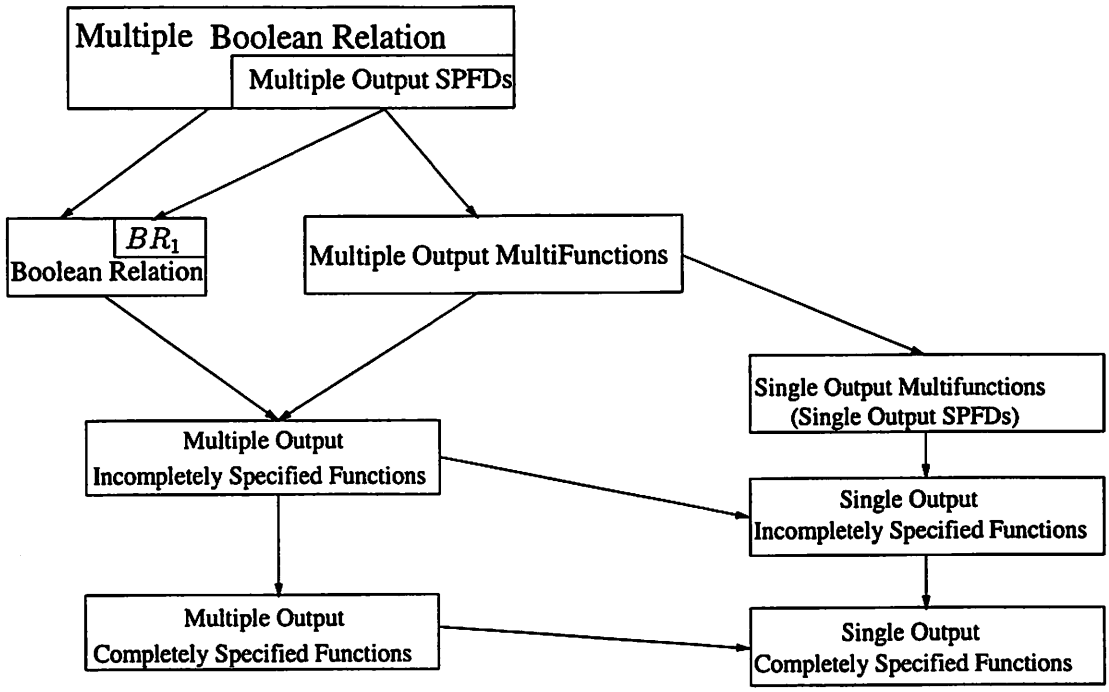


Figure 4.6: Flexibility hierarchy revisited: BR_1 denotes the set of Boolean relations that have a unique input minterm for each output value.

All the functions that are compatible with this Boolean relation cannot be captured using an SPFD because 10 has to be distinguished from 00 or 01 but not both, and 01 can't have output value 2. Only a subset or superset of the functions can be represented using an SPFD. However, there are some specific kinds of Boolean Relations that can be represented using SPFDs. One such type is given below.

Lemma 4.2 *Suppose a Boolean Relation has n output values. If a Boolean Relation has a unique minterm for each output value, then an SPFD can be constructed such that all colorings of the SPFD with n colors can provide a function that is compatible with the Boolean Relation, modulo renaming.*

The SPFD can be constructed by adding an edge between any two minterms that have non-overlapping output parts. Since the Boolean relation has n output values, the unique minterms of each output value form a clique of size n in the SPFD. Thus, any coloring of the SPFD with n colors will uniquely determine the output values of these minterms. Any other minterm will have an edge to all the unique minterms that don't produce the same output value. Hence, the output values of these minterms can also be uniquely determined. Hence, this SPFD can capture all the functions captured

by the Boolean Relation.

Example 4.6 *Consider the following:*

$$\begin{aligned} 00 &\rightarrow 0 \\ 01 &\rightarrow 1 \\ 10 &\rightarrow 0,1 \\ 11 &\rightarrow 2 \end{aligned}$$

The SPFD that can capture all the functions that are compatible with this Boolean relations is :{(00, 11), (00, 01), (01, 11), (11, 10)}. The edge $(11, 10)$ *implies that* 10 *cannot output a 2, but can output any other value.*

It is unclear if there are other types of Boolean relations that can also be completely captured using SPFDs.

Thus, SPFDs can capture all the flexibility expressed by multi-output multifunctions and a part of the flexibility expressed by Boolean relations. Hence, the flexibility hierarchy shown in Figure 3.5 can be re-drawn as shown in Figure 4.6.

4.5 Summary

This chapter provided a detailed exposition of the concept of SPFDs. The relationship between the SPFD of a node and its information content was described. The minimum SPFD of a node was defined and an algorithm was provided for computing it. To avoid the computational expense of minimizing each node in a network using its minimum SPFD, the notion of compatible SPFDs (similar to CODCs) was proposed. The core idea of the algorithm that can be used for computing these SPFDs was also described. It was proved that CODCs can be emulated using compatible SPFDs. Hence compatible SPFDs are strictly more powerful than CODCs. Finally, SPFDs were placed in the flexibility hierarchy described in the previous chapter, by comparing their ability to express flexibility to that of previously proposed approaches.

Chapter 5

SPFDs for Network Optimization

In this chapter, the details of computing compatible SPFDs for nodes in a network are provided. Then, the algorithms that use these SPFDs for resynthesizing the nodes is described. Some problems of these algorithms are outlined and alternative solutions are proposed. The chapter ends with the results of using SPFDs for network optimization. The results are compared to the optimization results of CODCs.

5.1 SPFD Computation Algorithm

In this section, a new scheme for computing compatible SPFDs for the nodes in a network is presented. The algorithm starts by ordering the nodes in the network. This ordering is then used for the distribution of SPFD edges during the SPFD computation phase.

5.1.1 Ordering Schemes

The ordering scheme works as follows: The level of a node η_j is computed recursively as:

$$Level(\eta_j) = max(\{Level(\eta_k) : \eta_k \text{ is a fanout of } \eta_j\}) + 1,$$

where nodes with zero fanout have $Level = 0$ i.e. *Level* is the maximum distance to any Primary Output. Given the levels, the computation order of a node is obtained as follows:

- *A node with a lower level (nearer the primary outputs) occurs earlier in the ordering.*
- *Given two nodes at the same level, the node with the most fanouts is earlier in the ordering.*

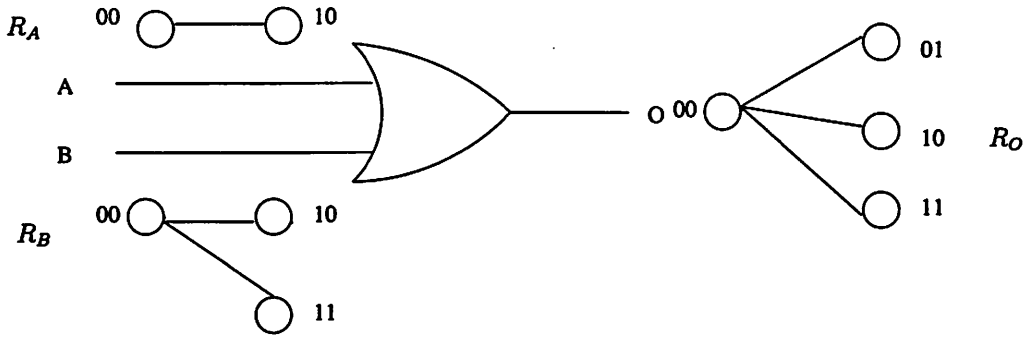


Figure 5.1: SPFDs for the fanins of an OR gate, $O = A + B$, given $A >_f B$.

The nodes are visited, according to their computation order (from lowest to highest), and their SPFDs are computed. Hence, the SPFD of a node is always computed before the SPFDs of its fanins.

5.1.2 Computing the SPFD of an node

The SPFD computation starts at the primary outputs. The SPFD of a primary output node is computed from its function. Thus, its SPFD specifies that all minterms in the onset of the function have to be distinguished from the minterms in the offset of the function. In case, an EXDC set is specified for each primary output, only the care onset has to be distinguished for the care offset.

At a internal node η_j, the following two steps are performed:

1. *The SPFD of each of its fanout wires is computed.*

2. *The SPFD of a node is computed from the SPFDs of its fanout wires.*

The SPFD of each fanout wire $w_{\eta_j \rightarrow \eta_k}$ is computed as follows:

Given R_k, the SPFD of a fanout node η_k, the edges of R_k that can be distinguished by η_j but not by the fanins of η_k later in the ordering are computed (if η_m is later in the ordering than η_j, then η_m is given less flexibility than η_j i.e. $\eta_m <_f \eta_j$). This is denoted as R_{jk}. Thus,

$$R_{jk} = R_k \wedge \{ \prod_{y_i \in Y_k; \eta_i <_f \eta_j} (y_i = y_i')\}(y_j \neq y_j').$$

Example 5.1 *Consider the simple OR-gate shown in Figure 5.1. Its SPFD is given as* $R_o = \{(00,01),(00,10),(00,11)\}$. *Now, let* $A >_f B$, *i.e. A will only distinguish the edges that B cannot distinguish. Of all the edges in* R_o, *the edge* $\{(00,10)\}$ *can only be distinguished by A since*

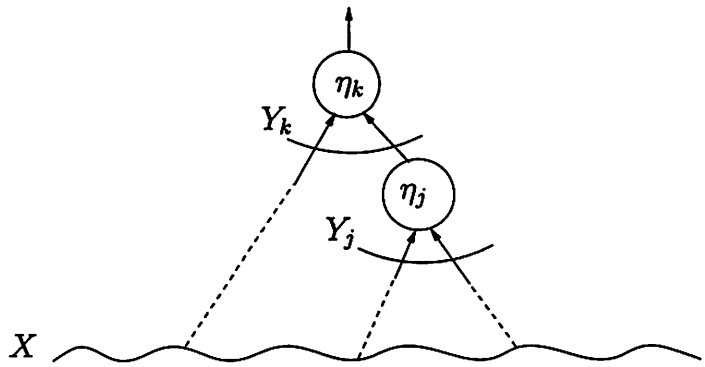


Figure 5.2: Y_j and Y_k spaces.

B is equal to zero for both the minterms in the pair. Thus the SPFD of A is $\{(00, 10)\}$. *The SPFD of B contains all the remaining edges of* R_o *i.e.* $\{(00, 01), (00, 11)\}$. *If* $A <_f B$, *then the SPFD of A would be* $\{(00, 10), (00, 11)\}$.

The SPFD at a node is obtained by first mapping the SPFD of each fanout wire to its local input space and then computing the union of these mapped SPFDs. The SPFD of each fanout wire, $R_{jk}(Y_k, Y_k')$, is mapped to the local space of η_j using the following equation:

$$R_{jk}(Y_j, Y_j') = \exists_{Y_k, Y_k'} R_{jk}(Y_k, Y_k') En(Y_j, Y_k) En(Y_j', Y_k').$$

The encoding relation $En(Y_j, Y_k)$ provides the mapping between the Y_j and the Y_k spaces, shown in Figure 5.2, and is given by:

$$En(Y_j, Y_k) = \exists_X \mathcal{G}(X, Y_j) \mathcal{G}(X, Y_k).$$

$\mathcal{G}(X, Y_j)$ denotes the characteristic relation between the primary inputs of $\mathcal{N}$ and the fanins of η_j.

Thus $(m_1, m_2) \in En(Y_j, Y_k)$ if there exists a primary input minterm x that produces m_1 in the Y_j space and m_2 in the Y_k space.

For the circuit shown in the Figure 5.3, the encoding relation between the Y_3 and Y_1 spaces is:

$$\begin{aligned} 00 &\rightarrow 00, 01 \\ 01 &\rightarrow 11 \\ 10 &\rightarrow 10, 11 \\ 11 &\rightarrow 11 \end{aligned}$$

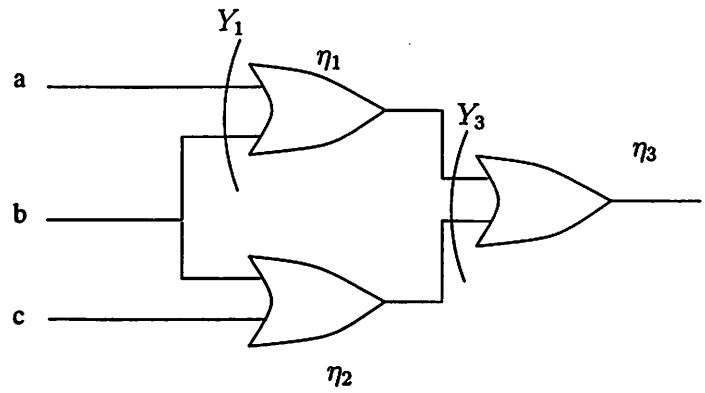


Figure 5.3: Example circuit.

5.1.3 Improvements

Some additional improvements can be built into the SPFD computation algorithm. During the computation of the SPFD R_{jk} of the fanout wire $w_{\eta_j \to \eta_k}$, the edges that are distinguished by the SPFDs of the fanins of η_k earlier in the ordering [1] are removed to get R'_{jk}. Thus,

$$R'_{jk} = R_{jk} \wedge \cap_{\eta_i >_f \eta_j} \exists_{Y_i, Y'_i} (En(Y_i, Y_k) En(Y'_i, Y'_k) \overline{R_i(Y_i, Y'_i)}).$$

This process has the effect of eliminating some edges from R_{jk} that have already been distinguished

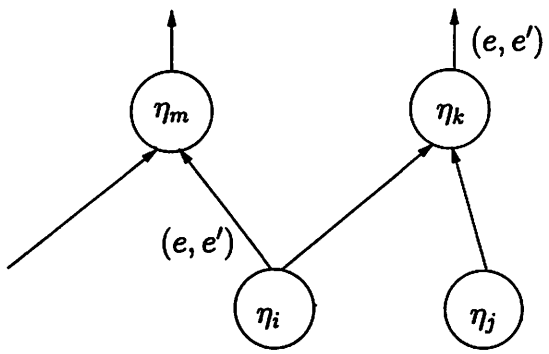


Figure 5.4: Example illustrating the advantages of the improvements in Section 5.1.3.

by other fanins of η_k. The advantage with this scheme is illustrated in Figure 5.4. Assume that node η_i is earlier in the ordering than η_j. Suppose (e, e') belongs to R_k and is distinguished by both η_i and η_j. In both the schemes, let $\eta_j <_f \eta_i$. In the approach explained in the previous section, (e, e')

[1] note that the fanins earlier in the order than η_j already have SPFDs associated with them since the SPFDs of the nodes are computed in that order

is assigned to the SPFD R_{jk} of $w_{\eta_j \to \eta_k}$. Further suppose that (e, e') is required by the fanout wire $w_{\eta_i \to \eta_m}$ [2] and hence is already included in the SPFD of η_i. In the new SPFD computation scheme, (e, e') will not be included in the SPFD of the wire $w_{\eta_j \to \eta_k}$. The previous scheme would add (e, e') to η_j and thus duplicate some information in η_i and η_j.

5.2 Resynthesis Algorithm

Here, the algorithm for resynthesizing the nodes in the network using their SPFDs is presented.

The nodes are resynthesized in a topological order from primary inputs to primary outputs. Thus, when a particular node is being resynthesized, the new implementations of its fanins are available. The SPFD of a node is given in terms of its original inputs. Due to the re-implementation of the fanins, the mapping to these inputs might change. For instance, suppose in the circuit in Figure 5.3, η_j is converted from an OR-gate to an inverter, as shown in Figure 5.5. Then the SPFD of η_3 computed in terms of the Y_3 space now has to be converted to an SPFD in terms of the new fanin space $\hat{Y}_3$.

The modified SPFD of the node η_j under the new encoding of the inputs is obtained by the following:

1. *The mapping between the old fanin space Y_j and the new fanin space, denoted as $\hat{Y}_j$, is computed.*

2. *The original SPFD in terms of the Y_j is translated to a modified SPFD in terms of the $\hat{Y}_j$ space.*

The relation between the old fanin space Y_j and the new fanin space $\hat{Y}_j$, shown in Figure 5.6, is given by:

$$En(Y_j, \hat{Y}_j) = \exists_X \mathcal{G}(X, Y_j)\hat{\mathcal{G}}(X, \hat{Y}_j).$$

$\mathcal{G}(X, Y_j)$ is the characteristic relation between the primary inputs of $\mathcal{N}$ and the original fanins of η_j. $\hat{\mathcal{G}}(X, \hat{Y}_j)$ is the characteristic relation between the primary inputs of $\mathcal{N}$ and the resynthesized fanins of η_j.

[2] This has already been computed since η_m is earlier in the ordering.

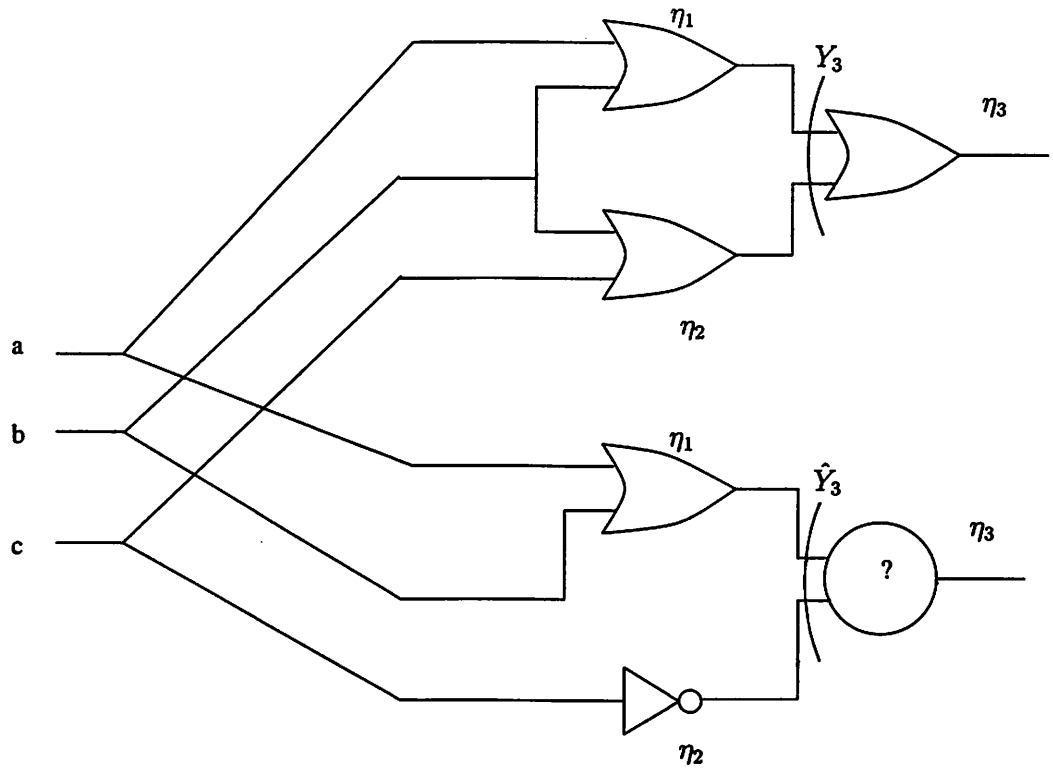


Figure 5.5: Example circuit (Contd).

Thus the encoding relation, $En(Y_3, \hat{Y}_3)$, between the Y_3 and $\hat{Y}_3$ space in Figure 5.5 is given as

$$\begin{aligned} 00 &\rightarrow 01 \\ 01 &\rightarrow 00 \\ 10 &\rightarrow 11 \\ 11 &\rightarrow 11, 10. \end{aligned}$$

The modified SPFD is then computed as:

$$R_j(\hat{Y}_j, \hat{Y}_j') = \exists_{Y_j, Y_j'} R_j(Y_j, Y_j') En(Y_j, \hat{Y}_j) En(Y_j', \hat{Y}_j').$$

Note that these steps are very similar to the mapping and translation phase in the SPFD generation phase.

Figure 5.7 illustrates how the new encoding of the inputs changes the original SPFD of η_3 in Figure 5.3. Note that the edge $(00, 11)$ translates into two edges $\{(01, 11), (01, 10)\}$.

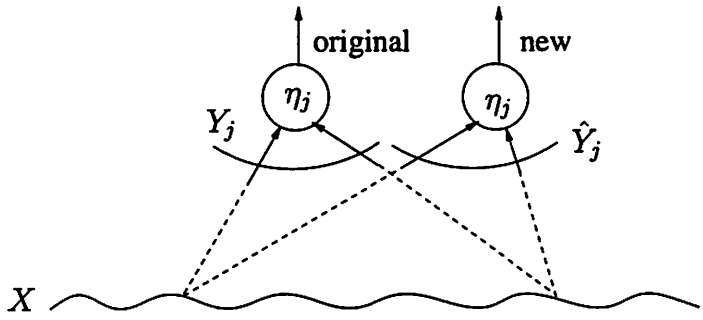


Figure 5.6: Y_j and $\hat{Y}_j$ spaces.

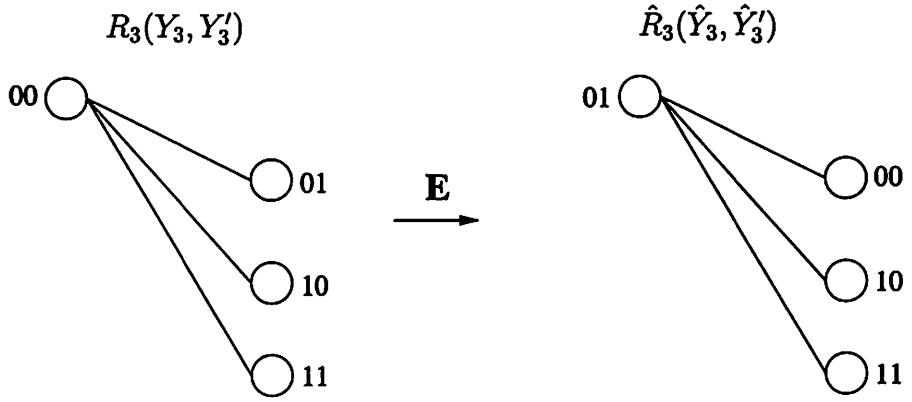


Figure 5.7: Modified SPFD of η_3 under the encoding E.

Any function that satisfies the modified SPFD is a valid new function at η_j. The new functions are derived by coloring the modified SPFD graph such that no two minterms that are connected by an edge have the same color.

With single-output nodes, the modified SPFD is mostly bipartite[3]. Since most of the experiments described later involve networks of single-output nodes, the coloring algorithm used for bipartite SPFDs is explained in a little more detail below.

The main source of flexibility in bipartite SPFDs is the presence of Strongly Connected Components (SCCs). The advantage of SCCs is that the minterms in one SCC do not have to be distinguished from those in another. The coloring algorithm for bipartite SPFDs first enumerates all the SCCs. Then, for each SCC, one set of minterms is placed in the onset and the other set of minterms is placed in the offset. But the choice for one SCC is completely independent of the choice for another SCC. Hence, if there are k strongly connected components in $R_j(\hat{Y}_j, \hat{Y}_j')$, then

[3]In a later section, it is described how the modified SPFD can be non-bipartite.

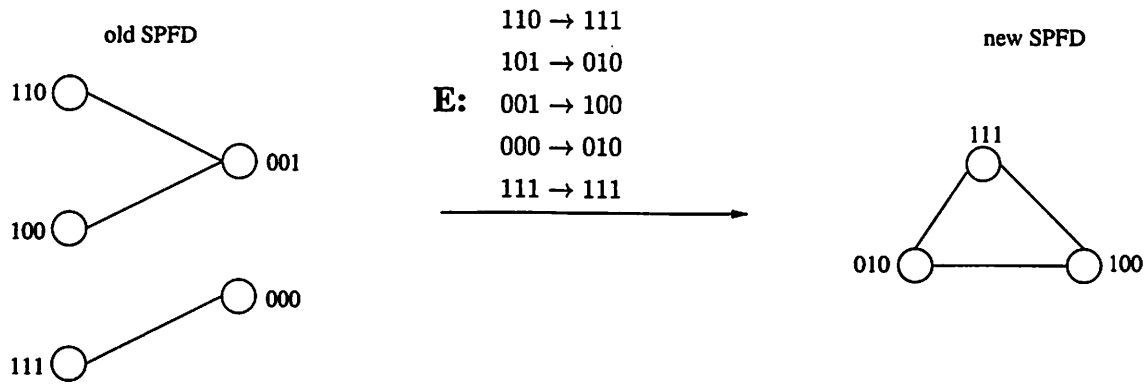


Figure 5.8: Non-bipartition of the modified SPFD after encoding.

there are 2^k functionally different ISFs that can be implemented at η_j. The new implementation at a node is chosen to be the minimum of the minimum covers of all the 2^k ISFs.

Finding the SCCs:

In this section, an implicit algorithm for enumerating all the SCCs is proposed. Given an SPFD $R(z, z')$, the individual SCCs can be obtained as follows. Initially, the two step graph $R_2(z, z') = \exists_y R(z, y)R(y, z')$ and the set of all nodes $N(z) = \exists_y R(z, y)$ in the bipartite graph are obtained. Then the following steps are performed:

1. *Pick* $z_0 \in N(z)$.
2. *Compute the fix point* $E_1(z)$*, which is all the nodes that can be reached from* z_0 *using* R_2*. Compute* $E_0(z) = \exists_y R(y, z)E_1(y)$*, the set of nodes that are connected by an edge to a node in* $E_1(z)$*. Store* (E_1, E_0) *as an SCC pair.*
3. *Let* $N(z) = N(z)\overline{E_1(z) + E_0(z)}$*. If* $N \neq \emptyset$*, go to 1.*

Note that this algorithm assumes the SPFD $R(z, z')$ is a symmetric relation. Since these SPFDs are expressed in terms of the local fanin space, the SPFDs are not too large. Hence, it is not very expensive to express it as a symmetric relation.

Non-bipartition:

There could be situations where $R_j(\hat{Y}_j, \hat{Y}_j')$ is not bipartite, even though $R_j(Y_j, Y_j')$ is. Figure 5.8 illustrates one such example. In such a situation, the result is a general graph. If the

graph can be colored using k colors, the new function can be encoded using $\log k$ bits. Thus the original node is replaced by $\log k$ nodes, all of whose fanouts are the same as the original node. This situation is undesirable since the number of fanins of the fanout nodes may increase. Techniques are being explored for constraining the SPFD propagation through the network so that under any encoding, the graph $R_j(\hat{Y}_j, \hat{Y}_j')$ remains bipartite.

In the next section, it is proved that the SPFD of a primary output node can never be bipartite after translation from the Y_j space to the $\hat{Y}_j$ space. Hence, the primary output nodes in a network of single-output nodes can never be split into multiple nodes. Experiments indicate that non-bipartite structures occur rarely even for internal nodes.

In the sequel, this algorithm of SPFD computation followed by resynthesis is referred to as **compute_global_spfds**.

5.3 Proof of Correctness

In this section, it is proved that the algorithm **compute_global_spfds** always produces an equivalent network.

Definition 5.1 *The* **original SPFD** *of η_j is the SPFD $R_j(Y_j, Y_j')$ attached to it after the SPFD computation phase of* **compute_global_spfds**.

Definition 5.2 *The* **modified SPFD** *of η_j is the SPFD $R_j(\hat{Y}_j, \hat{Y}_j')$ obtained by mapping the original SPFD from the old fanin space Y_j to the new fanin space $\hat{Y}_j$.*

Definition 5.3 *The* **global SPFD** *of η_j is obtained from its original SPFD $R_j(Y_j, Y_j')$ by composing each $y_k \in Y_j$ by $f_k^g(X)$ and each $y_k' \in Y_j'$ by $f_k^g(X')$. It is denoted as $R_j(X, X')$.*

Theorem 5.1 *If the new function at every node η_j in the network satisfies its modified SPFD $R_j(\hat{Y}_j, \hat{Y}_j')$, then it also satisfies its global SPFD $R_j(X, X')$.*

Proof The proof is by induction on the level of each node in the circuit. The first-level nodes are functions of the primary inputs only. For any such node η_j,

$$R_j(\hat{Y}_j, \hat{Y}_j') = R_j(X, X') = R_j(Y_j, Y_j'),$$

since the primary inputs are not changed and hence any function that satisfies the modified SPFD of the node also satisfies its global SPFD.

Now suppose that all level n nodes implement their respective modified SPFDs. Let η_j be a level $n+1$ node. It is proved that if the new function at η_j satisfies its modified SPFD, $R_j(\hat{Y}_j, \hat{Y}_j')$, then it also satisfies the global SPFD $R_j(X, X')$. Proof by contradiction.

Assume η_j satisfies $R_j(\hat{Y}_j, \hat{Y}_j')$ but it does not satisfy $R_j(X, X')$, where $R_j(X, X')$ is obtained from $R_j(Y_j, Y_j')$ by composing each $y_k \in Y_j$ by $f_k^g(X)$ and each $y_k' \in Y_j'$ by $f_k^g(X')$. This means that there exists an edge $(x, x') \in R_j(X, X')$ that is not distinguished by the new function at η_j. Now, the edge $(x, x') \in R_j(X, X')$ corresponds to an edge $(y, y') \in R_j(Y_j, Y_j')$, where y is the image of x in the Y_j space and y' is the image of x' in the Y_j' space. Since, while distributing the SPFD edges, it is ensured that all the edges of the SPFD of a node are assigned to at least one of its fanins, the edge (y, y') has to be assigned to a fanin of η_j and is included in the SPFD of that fanin. Assume that the fanin is η_k. Since the level of $\eta_k \leq n$, then by the induction hypothesis, η_k satisfies its global SPFD $R_k(X, X')$. Since $(y, y') \in R_k(Y_k, Y_k')$, $(x, x') \in R_k(X, X')$ and hence (x, x') is distinguished by η_k. Hence $(x, x') \in R_j(X, X')$ produces an edge $(\hat{y}, \hat{y}') \in R_j(\hat{Y}_j, \hat{Y}_j')$, where $\hat{y}$ and $\hat{y}'$ differ in the bit corresponding to η_k. But it is given that the new function at η_j satisfies $R_j(\hat{Y}_j, \hat{Y}_j')$ and hence $(\hat{y}, \hat{y}')$ has to be distinguished by it. Thus (x, x') also has to be distinguished by the new function at η_j, leading to a contradiction. Thus the new function at η_j satisfies $R_j(X, X')$. □

From the above, it can be concluded that if a node η_j satisfies its modified SPFD $R_j(\hat{Y}_j, \hat{Y}_j')$, then it also satisfies its global SPFD $R_j(X, X')$. By the inductive proof given above, it can be claimed that each node in the circuit implements its global SPFD. In particular, each primary output does so, thereby satisfying the original input-output functionality of the network.

Theorem 5.2 *The modified SPFD of a primary output η_j is always bipartite in a network of single-output nodes.*

Proof It is shown that for every edge $\hat{e} = (\hat{m}_1, \hat{m}_2) \in R_j(\hat{Y}_j, \hat{Y}_j')$, all the minterms of $M_1 = \exists_{\hat{Y}_j} \hat{m}_1 En(Y_j, \hat{Y}_j)$ and $M_2 = \exists_{\hat{Y}_j} \hat{m}_2 En(Y_j, \hat{Y}_j)$ belong to the original onset and offset, respectively (or vice versa). Thus all the edges in the modified SPFD are only between onset minterms and offset minterms. Hence the modified SPFD of a primary output is always bipartite. Proof by contradiction.

Suppose there exists an edge $\hat{e} \in R_j(\hat{Y}_j, \hat{Y}_j')$ such that there is a minterm $m_1 \in M_1$ and a minterm $m_2 \in M_2$ and both m_1 and m_2 belong to the onset. For the edge $\hat{e}$ to occur in the modified SPFD, there has to exist at least one edge in the original SPFD that maps to $\hat{e}$. Assume that edge $\tilde{e} = (\tilde{m}_1, \tilde{m}_2)$ in the original SPFD satisfies the above condition. The original SPFD of the primary output is constructed by adding edges between all onset minterms and all offset minterms. Thus,

$\tilde{m}_1$ has to belong to the onset(offset) and $\tilde{m}_2$ has to belong to the offset(onset). Let $\tilde{m}_1$ belong to the onset and $\tilde{m}_2$ belong to the offset. Then the minterms m_1 and m_2 belong to M_1 and M_2, respectively, only because both m_1 and $\tilde{m}_1$ map to $\hat{m}_1$ and both m_2 and $\tilde{m}_2$ map to $\hat{m}_2$. But the edge $(m_2, \tilde{m}_2)$ has to belong to the original SPFD (since m_2 is in the onset and $\tilde{m}_2$ is in the offset). Hence they have to be distinguished and cannot map to the same minterm $\hat{m}_2$ (by Theorem 5.1). Thus the assumption that both m_1 and m_2 belong to the onset leads to a contradiction. □

5.4 Example

A simple example is provided for illustrating the execution of **compute_global_spfds**. Consider the following network configuration:

$$
\begin{aligned}
f &= g_3(g_1 g_2 + \overline{g_1}\,\overline{g_2}) \text{ where} \\
g_1 &= x_3 + x_1 x_2 \\
g_2 &= \overline{x_1} x_3 + x_1 \overline{x_3} \\
g_3 &= x_1 + x_2 \overline{x_3} + \overline{x_2} x_3
\end{aligned}
$$

Let the ordering algorithm return the ordering : $f < g_3 < g_2 < g_1$. Thus, in terms of flexibility, $g_1 <_f g_2 <_f g_3$.

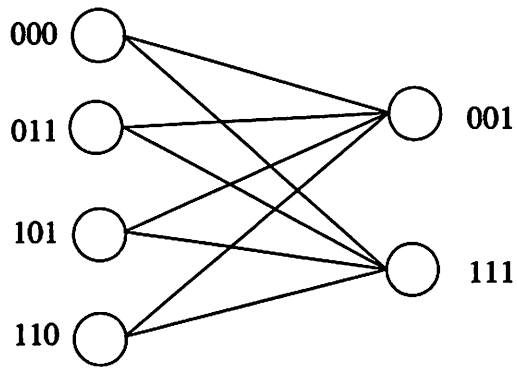


Figure 5.9: SPFD of f in terms of its local inputs.

The SPFD of f in terms of it local inputs, g_1, g_2 and g_3, is

$$\{(000, 111), (110, 111), (011, 111), (101, 111), (000, 001), (110, 001), (011, 001), (101, 001)\}.$$

This can also be represented by a bipartite graph as shown in Figure 5.9. Now, the edges of the SPFD of f have to be distributed to its inputs. An edge is assigned to g_2 only if it is not distinguished

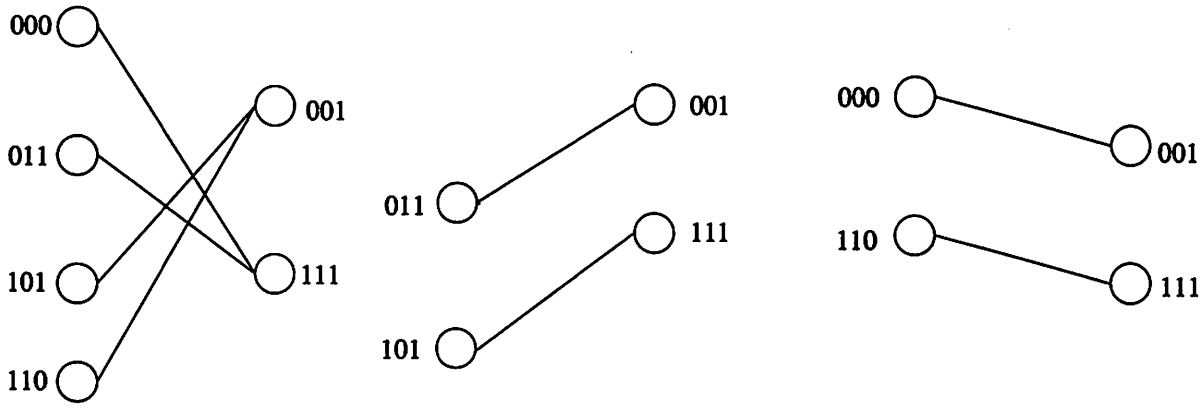


Figure 5.10: SPFDs of (g_1, f), (g_2, f) and (g_3, f) in terms of the local inputs of f.

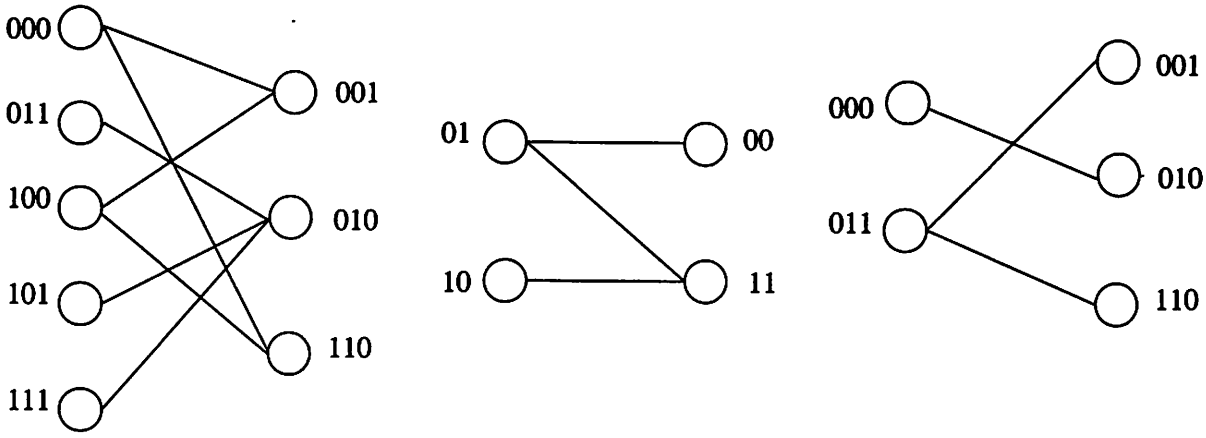


Figure 5.11: SPFDs of g_1, g_2 and g_3 in terms of their respective local inputs.

by g_1 and similarly an edge is assigned to g_3 only if it is not distinguished by both g_1 and g_2. For example, the edge $(011, 001)$ was put on g_2 because $g_1 = 0$ for both minterms. The SPFDs of the wires (g_1, f), (g_2, f) and (g_3, f) in terms of the fanins of f (i.e. g_1, g_2 and g_3) are shown in Figure 5.10. The SPFDs of the nodes g_1, g_2 and g_3 (which are in terms of vertices in the x space) are derived by mapping the SPFDs of the wires (g_1, f), (g_2, f) and (g_3, f) respectively to their input spaces and are shown in Figure 5.11.

Then the new functions at g_1, g_2 and g_3 are derived (as functions of x) using their respective SPFDs. The SPFD of g_1 has two strongly connected components. For the component $\{(000, 001), (100, 001), (000, 110), (100, 110)\}$, let $\{000, 100\}$ be in the onset and $\{001, 110\}$ be in the offset. Similarly for the component $\{(011, 010), (101, 010), (111, 010)\}$, let $\{011, 101, 111\}$ be included in the onset and $\{010\}$ be in the offset. For deriving the new function at g_2, let $\{01, 10\}$

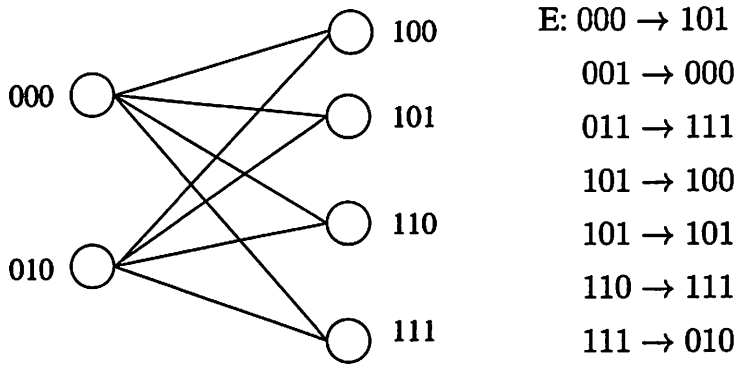


Figure 5.12: Modified SPFD of f under encoding E.

and $\{00, 11\}$ be in the onset and offset, respectively. Similarly for the new function at g_3, let $\{000, 011\}$ be the onset and $\{010, 110, 001\}$ be the offset. Thus, the new functions at g_1, g_2 and g_3 denoted as $\tilde{g_1}$, $\tilde{g_2}$ and $\tilde{g_3}$ respectively are given by:

$$\begin{aligned} \tilde{g_1} &= x_1x_3 + x_2x_3 + \overline{x_2}\,\overline{x_3} \\ \tilde{g_2} &= \overline{x_1}x_3 + x_1\overline{x_3} \\ \tilde{g_3} &= x_2x_3 + \overline{x_2}\,\overline{x_3} \end{aligned}$$

The new functions, $\tilde{g_1}$, $\tilde{g_2}$ and $\tilde{g_3}$, provide a new encoding at the inputs of f and thus the SPFD of f under this encoding is shown in Figure 5.12.

If $\{000, 010\}$ is included in the onset of f and the rest in the offset, the new function at f is given as $f^{new} = \overline{g_1}$. Of course, an inverter has to be inserted at the output to get back the original function. Even then, the savings in the number of literals is considerable.

In contrast, this optimization cannot be obtained using CODCs. This simple example illustrates that SPFDs can perform optimizations on circuits when CODCs cannot. Thus SPFDs can be used to get better optimized circuits.

In the following sections, two main problems associated with SPFD algorithms are addressed: non-robustness and unpredictability. Non-robustness issues arise due to the memory problems of BDD engines. Alternative schemes are provided in the next section for solving this problem. The increased flexibility of SPFDs can produce some uncontrolled changes in the network and thereby adversely affect the predictability of SPFD-based optimization. In a later section, the causes of this unpredictability are provided and some solutions are proposed for countering this problem.

5.5 Robust Computations

The first implementation used BDDs for all computations. This worked well for medium sized circuits. However for larger circuits, computing the relation $En(Y_j, Y_k)$ during the SPFD computation phase and the relation $En(Y_j, \hat{Y}_j)$ during the resynthesis phase caused BDD memory explosion problems.

SAT solvers are known to be more robust than BDD engines. Hence they can handle much larger circuits without significant memory problems. In the next section, an algorithm is presented for computing the above relations using a SAT solver.

5.5.1 SAT-based scheme

The algorithm **com_enc_reln** given below computes $En(Y_j, Y_k)$ for any (η_j, η_k) pair in the network, where η_k is a fanout of η_j, using a constructive approach. $En(Y_j, Y_k)$ is initialized to the empty set. Given η_j and its fanout η_k, a SAT instance C is created which contains the clauses of all nodes in the transitive fanin of η_k. Thus C contains all the information about the relationship between the fanins of η_k and η_j. Each solution S returned by the SAT solver is modified by projecting out the variables not in Y_j or Y_k to yield S'. S' is a cube of $En(Y_j, Y_k)$ since there exists a setting of the primary input variables that gives S'. S' is added to $En(Y_j, Y_k)$. Then C is modified by adding the clause $\overline{S'}$ to C. The SAT solver is invoked again and the process is repeated. The added clause $\overline{S'}$ guides the SAT solver and prevents it from returning another solution S'' which yields S' after projecting out all variables not in Y_j or Y_k. Thus this process guarantees that after each call to the SAT solver, the cube S' is unique. The algorithm stops when the SAT instance is not satisfiable. At this point, all cubes of $En(Y_j, Y_k)$ have been enumerated.

Algorithm **com_enc_reln** ($\mathcal{N}$, η_j, η_k):

1. $En(Y_j, Y_k) = \phi$.
2. *Associate a SAT variable y_i with each node η_i in the network.*
3. *Write down SAT clauses for each node in the transitive fanin of η_k to get a SAT instance C.*
4. *Let $V = Y_j \cup Y_k$.*
5. *Call a SAT solver on C. If the instance is unsatisfiable, go to step 7. Given a solution S, project out variables that are not present in V to get a cube S' of $En(Y_j, Y_k)$. Add S' to*

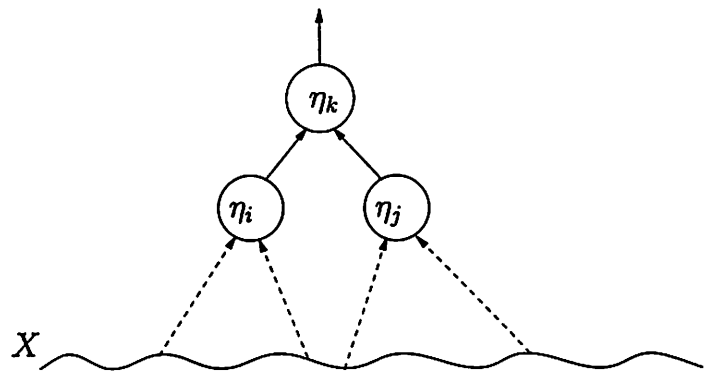


Figure 5.13: Relation between η_i, η_j and η_k.

$En(Y_j, Y_k)$. Thus,

$$En(Y_j, Y_k) = En(Y_j, Y_k) \cup S'$$

6. *Given S', add the complement of S' as a clause to C to get a new SAT instance. Thus, $C = C \wedge \overline{S'}$. Go to step 5.*

7. *Stop. Output $En(Y_j, Y_k)$.*

This algorithm can be implemented using any complete SAT solver, i.e. one that can find a solution if one exists. In the actual implementation, CHAFF [14], developed by Moskewicz et al. at Princeton, was used. CHAFF has an option for enumerating all solutions of a SAT instance over a subset of the variables. So Steps 5 and 6 are implemented in one call to CHAFF.

During the resynthesis process, the relation $En(Y_j, \hat{Y}_j)$ for the node η_j has to be computed. The procedure is similar except for some minor changes. In Step 2, two variables y_i and y_i' are associated with each node η_i in the network. The variable y_i is used for expressing the original function of η_i and the variable y_i' is used for expressing the new function of η_i. In Step 3, clauses are added for both the original and the new functions of all nodes in the transitive fanin of η_j. V, in Step 4, is equal to the union of Y_j and $\hat{Y}_j$.

Some additional efficiencies were built in to reduce the run-times:

- *While computing the relation $En(Y_j, Y_k)$ for (η_j, η_k), if the primary inputs in the transitive fanin of η_i (a fanin of η_k) and η_j are disjoint (as shown in Figure 5.13), then any clauses pertaining to either η_i or any of its transitive fanins are not added to the SAT instance C in Step 3.*

- *After resynthesis, if a node η_i is not changed, a new variable y_i' is not associated with it. Note that ideally this is desirable, whenever the global function of a node is unchanged. However, it may be too computationally expensive to perform the global check. Instead, a node is tagged as unchanged, if the local function of the resynthesized node is the same as the original and all its fanins are also tagged unchanged. This is especially efficient since, in almost all circuits, there are nodes that remain unchanged after resynthesis. By re-using the same variable for the original and the new node, the SAT solver can establish relations between the Y_j and the $\hat{Y}_j$ variables more quickly.*

- *During resynthesis, the relations $R_i(Y_i, \hat{Y}_i)$ for each fanin η_i of the node η_j are also added. The idea is to provide as much learned information to the SAT solver as possible. The encoding relations of the fanins of η_j can help eliminate some unnecessary combinations early on.*

These have helped to reduce the time taken by the SAT solver to compute the encoding relations.

5.5.2 Combined Strategy

SAT solvers suffer from efficiency issues, particularly for set manipulation problems. BDDs, on the other hand, are very suitable for set manipulation. Hence, a hybrid approach combining the efficiency of the BDD engine and the robustness of SAT is used. In this approach, BDDs are used for performing the image computations until the number of BDD nodes increases beyond a certain user defined limit. After that, a SAT solver is used for the remaining image computations.

5.6 Making the Results more Predictable

In this section, the unpredictability that can arise during resynthesis using SPFDs is described. This happens because when a node is changed in the network using SPFDs during the resynthesis procedure, all the nodes in its immediate fanout have to be changed since the mapping of the fanin spaces of these fanouts changes. This is turn causes the mapping of the fanin spaces of their fanouts to change. This domino effect as illustrated in Figure 5.14. The problem with this uncontrolled change is that a choice made early during the network optimization process can adversely affect the nodes in the transitive fanout. This effect could manifest itself as an undesirable increase in the literal count in the factored form after SPFD optimization.

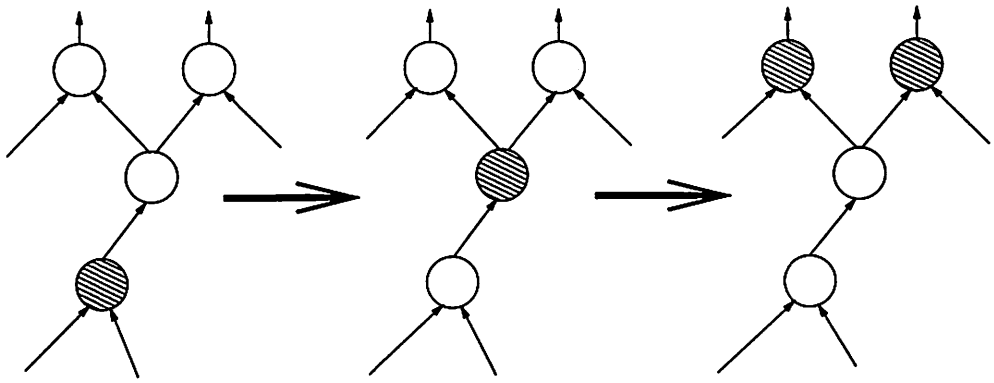


Figure 5.14: Resynthesizing a node using its SPFD can potentially change all the nodes in its transitive fanout.

In the following section, the blocking techniques used for avoiding uncontrolled change and making the results more predictable are explained.

5.6.1 Window-based computation

5.6.1.1 Region of Change

A "region of change" (ROC) denotes the set of nodes that can possibly be affected during the resynthesis step. In the interest of predictability, it is good to have a small ROC. However, a very small ROC can affect the amount of the flexibility of SPFDs that can be used. In this work, a ROC contains a node and its fanouts up to some level as shown in Figure 5.15. A ROC is parameterized by l : all nodes within l fanout levels of a node η_j are in the ROC of η_j.

The algorithm **spfd_simplify** proceeds from inputs to outputs in topological order and at each node η_j performs the following steps:

1. *Compute the ROC at η_j and determine the nodes at the outer boundary of this region. These are the nodes that have at least one fanout to a node not present in the ROC. Denote this set as U_j.*

2. *Derive an SPFD for each node in U_j from the functionality of the current network. Basically, the SPFD of each node in U_j specifies that its onset derived from its current function has to be distinguished from its offset. This ensures that the changes in U_j are not propagated to their fanouts.*

3. *Compute the SPFDs of all the nodes in the ROC in reverse topological order as described in*

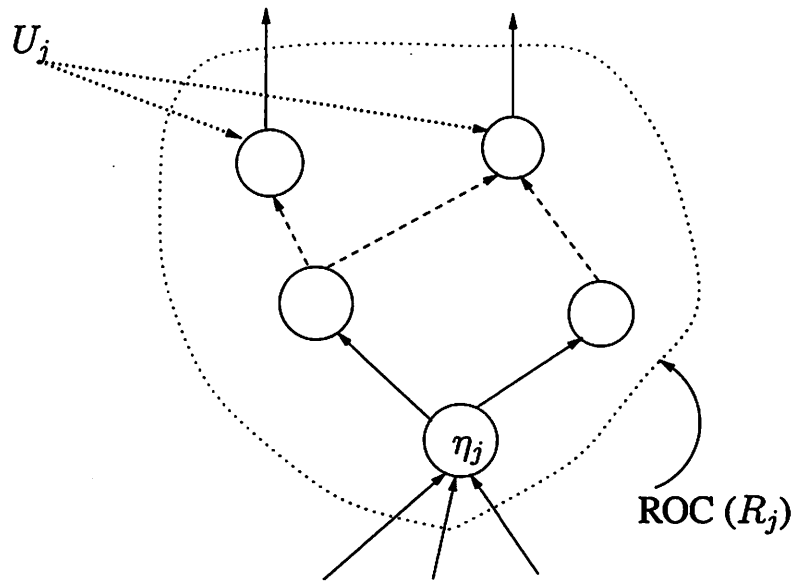


Figure 5.15: Region of Change of η_j.

Section 5.1.

4. *Resynthesize all the nodes in the ROC in forward topological order as described in Section 5.2.*

5. *If the sum of the literal counts (or some other cost function) of resynthesized nodes in the ROC is less than that of the original, replace all the nodes by their new functions.*

Since the nodes in the ROC are modified only if the net effect is positive, it is guaranteed that the final result will be better than or equal to the original result. Also note that as l increases, there are more computations in Steps 3 and 4. So, typically $l \leq 2$.

One problem with this scheme is that the non-observability of the boundary nodes of a ROC is not exploited during the optimization process. In Section 5.6.2, a novel method for combining SPFDs and CODCs for circumventing this problem is described.

5.6.1.2 Parameterized Image Computation

In Section 5.5, a new scheme for image computation combining a SAT solver and a BDD engine was proposed. Here, another scheme for image computations using BDDs is proposed. This scheme can be used with the notion of the ROC since only a few nodes are changed at each step.

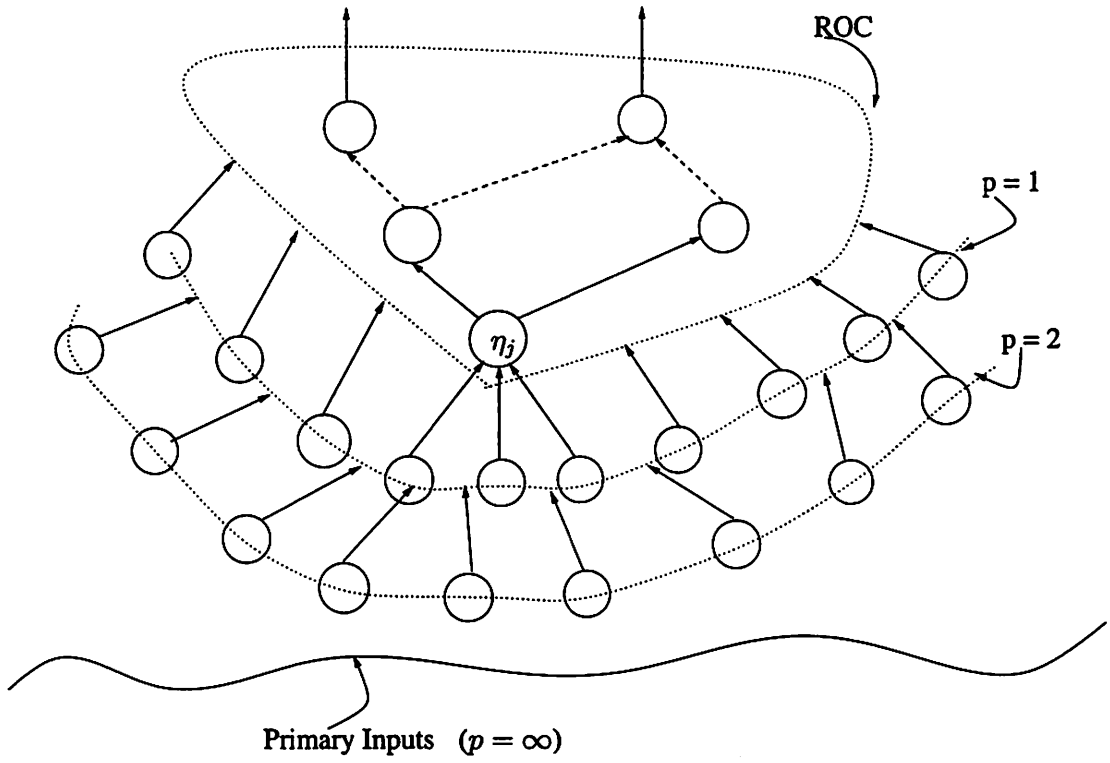


Figure 5.16: Parameterized BDD-based image computation.

The basic idea is illustrated in Fig 5.16. Instead of computing images all the way down to the primary inputs, only a few levels below the ROC are used for image computation. Parameter p is used for specifying the cutset that is used for the operation. For $p = 1$, only the immediate fanins of the nodes in the ROC are used as a cutset for doing the image computations. For $p = 2$, nodes that are fanins of the nodes in the $p = 1$ cutset are used. As the cutset is moved closer to the primary inputs, more of the SDCs are taken into account during the image computation. The cutset consisting solely of primary inputs is denoted as $p = \infty$.

5.6.2 SPFDs and CODCs combined

Here an algorithm for using both CODCs and SPFDs for performing controlled optimizations in the network is sketched.

The algorithm is almost identical to **spfd_simplify** except that in Step 2, the LDCs (CODCs expressed in the fanin space of a node) are used to derive the SPFDs of the boundary nodes. The rest of the algorithm proceeds exactly as before. Once the nodes in the ROC have been resynthe-

sized, the CODCs/LDCs of some of the nodes in the network may no longer be valid. For a detailed explanation of why this happens, please refer to [25].

In the following two sections, an algorithm is provided for computing the LDCs of the boundary nodes (if they become invalid). A naive approach would invalidate the CODCs/LDCs of all the nodes in the network after the resynthesis step. However, in reality, only a few nodes have invalid CODCs/LDCs after the resynthesis step. These nodes are identified in Section 5.6.2.2.

5.6.2.1 Computing the LDCs of a node on demand

The LDC of a node η_i is only computed on demand i.e. it is only computed if η_i is a boundary node of the current ROC being processed:

1. *If LDC of η_i is valid, then return.*
2. *If the CODC of η_i is valid, then compute the LDC from it using an image computation step and return. (Note that either SAT or BDDs can be used for this step.)*
3. *Let TFO_i be the nodes in the transitive fanout of η_i in reverse topological order.*
4. *For each $\eta_k \in TFO_i$, do the following:*
   (a) *If CODC of η_k is valid, then return that. Else, go to the next step.*
   (b) *Determine the CODC of each fanout wire $w_{\eta_k \rightarrow \eta_l}$ of η_k. In order to obtain this, first determine the fanin minterms of η_l which are insensitive to the value of η_k. Then, make them compatible with the fanins of η_l that already have CODCs associated with them. This gives the CODC of $w_{\eta_k \rightarrow \eta_l}$.*
   (c) *Intersect the CODCs of all the fanout wires of η_k to get the CODC of η_k.*
5. *Obtain the LDC of η_i from its CODC by expressing it in terms of the fanin space Y_i.*

5.6.2.2 Updating/Invalidating the CODCs and LDCs of the nodes

If the nodes in a ROC are changed, then it is necessary to update or invalidate the CODCs and/or LDCs of some of the nodes in the network.

Given a node η_j with ROC R_j, the nodes in the transitive fanin of the nodes in U_j have invalid CODCs and hence invalid LDCs. This is because the observability of these nodes may have

changed, due to the changes in the functionalities of the nodes in their transitive fanout (due to the resynthesis of the nodes in R_j).

The CODCs of the nodes in U_j are valid. However, their LDCs are invalid and need to be updated. The LDC of each node η_i in U_j can be easily updated according to the following formula:

$$\hat{L}_i(\hat{Y}_i) = \overline{\exists Y_i En(Y_i, \hat{Y}_i)\overline{L_i(Y_i)}}.$$

The CODCs of the nodes in the transitive fanout of U_j are valid but their LDCs may no longer be valid because of possible changes in the SDCs of the network. Hence, the LDCs of these nodes are invalidated (note that the LDCs are directly updated from the old LDCs whenever possible to avoid unnecessary image computations).

This scheme of using both SPFDs and CODCs for optimizing the nodes in a ROC is called CODC bounding. Only the primary inputs ($p = \infty$) are used for image computation in the CODC bounding algorithm. It is implemented as an option "**-codc**" of **spfd_simplify**.

5.7 Results

Table 5.1 compares the efficiency and robustness of the different image computation schemes used in **spfd_simplify** for $p = \infty$. Column 2 gives the runtimes for **spfd_simplify** ($l = 2$) when it uses the BDD-based image computation scheme. Columns 3 and 4 give similar results for the SAT-based and the BDD+SAT-based image computation schemes, respectively. The improved robustness of the SAT-based schemes (both the pure and the integrated approach) over the pure BDD-based scheme is evident in the last three examples. For these large examples, the BDD-based method ran out of memory but both the SAT-based scheme and the integrated BDD+SAT-based scheme completed. The advantage in terms of efficiency of the BDD+SAT-based scheme over a pure SAT-based image computation is evident in the runtimes. The improvement is because BDDs (which are typically efficient for set manipulations) were used at the beginning and the more time consuming SAT-based scheme was used only when the number of BDD nodes in the BDD manager was large thereby causing memory related problems for BDDs. In these experiments, the BDD node limit was set to 480000. Thus, the integrated BDD+SAT-based scheme exploits the efficiency of the BDDs for the smaller circuits and the robustness of the SAT solver for the larger circuits, thereby making the integrated approach both robust and efficient.

The command **spfd_simplify** used the scheme described in Section 5.6 to optimize the network. Table 5.2 gives the results of **spfd_simplify** for various window sizes (of the ROC) and

circuit	BDD Runtime(s)	SAT Runtime(s)	BDD+SAT Runtime(s)
9symml	149.2	275.4	149.2
b9	8.1	36.2	8.1
cmb	0.5	2.4	0.5
cordic	6.4	17.85	6.4
frg2	232.6	5981.6	232.6
i9	463	1444.0	463
lal	4.4	13.785	4.4
k2	288.6	376.5	288.6
term1	28.0	211.7	28.0
ttt2	6.0	43.5	6.0
x2	1.2	3.7	1.2
x3	76.5	1202.5	76.5
x4	20.2	306.3	20.2
C2670	-	1852.3	1760.2
C3540	-	2784.2	2756.0
C6288	-	4499.9	3294.3

Table 5.1: Comparison of runtimes for different image computation schemes.

circuits	original	full_simplify	l=2; $p = \infty$	l=1; $p = \infty$
9symml	277	270	270	268
b9	236	188	190	190
cmb	62	59	59	59
cordic	194	155	155	155
frg2	2010	1454	1438	1438
i9	1453	1132	1078	1078
lal	223	184	150	194
k2	2928	2889	2664	2664
ttt2	341	268	230	242
term1	625	336	233	251
x2	71	60	58	58
x3	1345	1200	1166	1169
x4	672	568	534	534
C2670	2043	-	1710	1710
C3540	2934	-	2654	2654
C6288	4800	-	4699	4699
C7522	6098	-	4239	4239
% imprv	0	12.30	20.53	19.00

Table 5.2: Comparison of **spfd_simplify** for different values of l and $p = \infty$ vs **full_simplify**.

circuits	original	full_simplify	l=2;p=1	l=2;p=2	l=2; p = ∞
9symml	277	270	268	269	270
b9	236	188	190	190	190
cmb	62	59	59	59	59
cordic	194	155	155	155	155
frg2	2010	1454	1792	1728	1438
lal	223	184	194	184	150
k2	2928	2889	2664	2664	2664
term1	625	336	468	422	233
ttt2	341	268	277	246	230
x2	71	60	59	57	58
x4	672	584	592	536	534
C2670	2043	-	2043	2043	1710
% imprv	0	16.07	12.77	15.64	22.23

Table 5.3: Comparison of **spfd_simplify** for different values of p and $l = 2$ vs **full_simplify**.

how they compare to **full_simplify** (Column 3). Columns 4 and 5 give the results obtained by using a ROC that includes all fanouts within two levels ($l = 2$) and one level ($l = 1$) of the node respectively.

Table 5.3 provides a comparison of the results of **spfd_simplify** for different values of p. Column 3 gives the results for **full_simplify**. Columns 4, 5 and 6 provide results for $p = 1$, $p = 2$ and $p = \infty$, respectively. The ROC window was set to $l = 2$. For $p = 1$ and $p = 2$, BDDs were used for image computation and for $p = \infty$, SAT was used. The results for C2670 indicate that while the BDD-based **full_simplify** ran out of memory, **spfd_simplify** (that also used BDDs for $p = 1, 2$) completed without any memory problems. Also, the quality of the results improved as the cut was moved closer to the primary inputs. This is expected as more of the SDCs were taken into account when the image computation used a cutset closer to the primary inputs. However, it is interesting to note that even for a cutset close to the ROC ($p = 1$), some optimizations were obtained. This factor can be exploited in medium to large circuits for achieving some preliminary optimizations.

Table 5.4 compares **spfd_simplify** (only SPFDs) with **spfd_simplify -codc** (SPFDs with CODC bounding) for $l = 1$ and $l = 2$. Columns 3 and 4 give the results with and without CODC bounding for $l = 2$, respectively. Columns 5 and 6 give the corresponding results for $l = 1$. The results indicate that CODC bounding did not achieve much improvement except for the last example. This is probably because for these examples, CODCs did not provide much additional improvement. This is supported by Table 5.5 which gives the effect of using **full_simplify** with and

circuit	original	ss -codc(l = 2)	ss (l = 2)	ss -codc(l = 1)	ss(l = 1)
cmb	62	59	59	59	59
term1	645	233	233	253	253
ttt2	341	233	235	235	246
x3	1345	1166	1166	1169	1169
x4	672	533	533	533	533
apex6	904	842	844	849	854
apex7	289	262	262	261	261
9symml	277	271	271	265	268
frg2	2010	1407	1436	1315	1436
% imprv	0	20.18	19.93	20.43	19.22

Table 5.4: Comparison of **spfd_simplify** with and without CODC bounding for different values of l and $p = \infty$.

circuit	original	full_simplify	full_simplify -d
cmb	62	59	60
term1	625	369	348
ttt2	341	276	271
x3	1345	1209	1209
x4	672	569	570
apex6	904	885	886
apex7	289	256	256
9symml	277	270	270
frg2	2010	1522	1645
% imprv	0	14.68	14.16

Table 5.5: Effect of CODC optimization on the examples in Table 5.4.

without CODCs[4]. Except for the last example, the effect of CODCs on network optimization was very limited. It was found that only for a few large examples in the MCNC benchmark circuits was the effect of CODCs on network optimization pronounced. However, the runtimes of **spfd_simplify -codc** for these circuits were excessive and hence for them the effectiveness of **spfd_simplify -codc** over **spfd_simplify** could not be tested. The blowup in runtimes of **spfd_simplify -codc** was mainly because the CODCs of many of the nodes in the network had to be recomputed quite often.

[4]The -d option of full_simplify does not use CODCs.

5.8 Summary

This chapter provided an in-depth exposition to the various algorithms for SPFD-based network optimization. SPFDs can be unreliable for network optimization since modifying a node can affect the nodes in its transitive fanout adversely. To counter this, the notion of a ROC was introduced. The results on benchmark circuits were favorable. In general, the quality of results was found to improve as the value of l was increased. However, the runtimes can be quite large for $l > 2$.

Alternative image computation schemes were also proposed. The SAT-based image computation algorithm was implemented using CHAFF. This enabled SPFD optimization on much larger examples than BDD-based methods. However, the SAT-based method can be relatively slow for smaller circuits. To deal with this problem, an integrated scheme that initially used BDDs for image computation and automatically switched to a SAT-based computation when the number of nodes in the BDD manager exceeded a certain user specified limit was also proposed. The results indicated that the integrated approach nicely combines the efficiency of BDD engines with the robustness of SAT techniques. This scheme is general and can be used in other applications where BDD-based computations blow up. For instance, it can be used to make **full_simplify** work for larger circuits. A scheme for parameterized image computation was also proposed that used a parameter p for controlling the cutset through which the image computation was performed. This scheme enabled the use of BDD-based SPFD optimization for large circuits. It will be interesting to explore other schemes such as partitioned BDDs and BDD subsetting/supersetting to increase the portion of the circuit for which BDDs can still be used.

Chapter 6

Wire Manipulation Techniques

In this chapter, first a brief overview of rewiring is provided and it is argued that SPFDs provide a more general framework for rewiring a given network. Then a few different flavors of rewiring are explored.

6.1 Previous Work

The basic idea of rewiring is to replace one wire with another without changing the functionality of the network. Rewiring can have a number of interesting application. For instance, a wire on the critical path can be replaced with another wire that is not on the critical path or a wire in a heavily congested routing area can be replaced with another in a less congested area. Most of the previous work in this area used ATPG-based methods [4, 5, 6]. The common idea in all these algorithms is the ability of adding a redundant wire and in the process, making some of the other wires redundant, which can then be removed. These techniques are often referred to as redundancy addition and removal (RAR). The algorithm proceeds as follows: Given a wire w_t, first the set of mandatory assignments for testing the fault at w_t are identified. A wire is redundant only if the set of mandatory assignments are inconsistent. Then, a new wire w_r is added so that the set of mandatory assignments for w_t become inconsistent. In addition, it is necessary to check that w_r is redundant, so that the functionality of the network is unchanged after addition of w_r. This basic idea of making some wires redundant by adding other redundant wires was first proposed in [4] and was further extended in [5] by allowing changes in functionality of certain nodes for rendering a particular target wire redundant. However, the changes allowed were small since only the assignment of the *don't care* minterms of the function could be altered. This rewiring scheme has been applied

to post-layout logic restructuring for improving the routability of the circuit. More recently, there has been work on rewiring based on functional symmetries [7]. Here, an implication supergate is constructed for detecting the functional symmetries. It is shown that wires of the supergate can be swapped without changing the network functionality. In some recent work [8, 26], the requirement that the added wire had to be redundant was dropped. In [26], some simple functionality changes were allowed for eliminating the error introduced by the irredundant wire. The main problem with the approach was it required the use of a formal verification tool for guaranteeing the correctness of the modified circuit. The work in [8] attempted to eliminate the use of formal verification by finding some necessary and sufficient conditions under which a new irredundant wire could replace an original wire without affecting network functionality. However, the nature of the functionality changes allowed to the nodes, if any, is unclear.

Some related work has been done using the concept of global flow analysis [9]. This technique performs rewiring by modeling the problem of rewiring using a flow graph and then solving it using the maxflow-mincut algorithm on the corresponding flow graph. The advantage of this approach over ATPG methods was that it could simultaneously add and remove many redundant wires at the same time. One drawback of this method was that it allowed only fanout reconnections. This work was extended by [10] for allowing both fanin and fanout reconnections. However, these methods are similar to ATPG-based methods in the sense that they still try to make the wires redundant by making them untestable. Hence they don't allow any functionality changes of the remaining nodes in the network.

6.2 SPFDs and Rewiring

SPFDs provide a powerful tool for rewiring. As mentioned, SPFDs can be used for representing the information content of a wire in the network. This notion can be exploited for rewiring the circuit. Suppose the SPFDs assigned to a wire from node η_k to η_n, denoted as $w_{\eta_k \to \eta_n}$, is a subset of the SPFD assigned to node η_m. Then a fanout from η_m is a candidate replacement for $w_{\eta_k \to \eta_n}$, since $w_{\eta_m \to \eta_n}$ supplies no less information than $w_{\eta_k \to \eta_n}$. The node η_m can also be called an alternate source for the wire $w_{\eta_k \to \eta_n}$. Of course if the replacement is made, the logic function at node η_n may have to change (because the information being supplied is in a different form), but such a function always exists. When a wire $w_{\eta_k \to \eta_n}$ does not provide any unique information to node η_n, it can be removed. Again, the function at η_n has to be changed to account for the different flow of information.

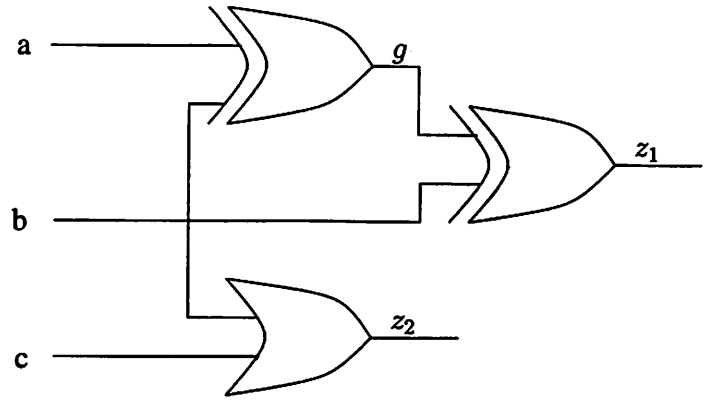


Figure 6.1: Rewiring example.

Consider the circuit shown in Figure 6.1:

$$\begin{aligned} z_1 &= \bar{g}b + g\bar{b} \\ g &= \bar{a}b + a\bar{b} \\ z_2 &= b + c \end{aligned}$$

For wire (g, z), the SPFD is $A = \{(00, 10), (01, 11)\}$ (in the set A, each minterm is of the form gb). If the minterms of A is expressed in terms of the primary inputs, a and b, then a new SPFD $A' = \{(00, 10), (11, 01)\}$ is obtained (the minterms in A' are of the form ab). The primary input a can distinguish both pairs in A'. Hence a fanout from a is a candidate wire for replacing (g, z_1). Simplifying z_1 gives $z_1 = a$. In contrast, redundancy removal based rewiring cannot simplify the circuit. This is because there are no mandatory assignments for propagating a stuck-at-fault on g. This is due to the presence of an XOR gate along the path because the output of an XOR is sensitive to the both its inputs. SPFDs, on the other hand, look for the actual information content and are not affected by the kinds of gates. Moreover SPFDs provide more flexibility for implementing the function. Any function that distinguishes all the edges of the SPFD is a suitable implementation. Using SPFDs, the onset and offset minterms in the original function can be swapped, thereby deriving many different functions, many of which cannot be obtained by ATPG-based techniques (which can work mainly with the *don't care* set).

A systematic exploration of the link between SPFDs and other tools for rewiring would be of interest. It is believed that SPFDs provide more rewiring opportunities than the RAR-based rewiring approaches and the global flow techniques proposed by Berman et. al [9]. In the rest of the section, some initial arguments are presented for illustrating that the simple RAR method proposed

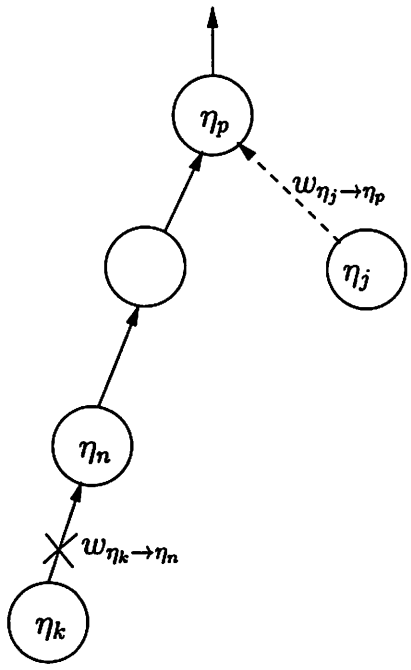


Figure 6.2: Illustration for the proof of Lemma 6.2.

by Cheng et. al. can be emulated using SPFDs. Thus the SPFD rewiring is at least as powerful as RAR.

As mentioned, the basic idea in RAR techniques is to make a wire redundant by adding another redundant wire. It is proved that the same result can be obtained using SPFDs.

Lemma 6.1 *If a wire is redundant, then its minimum SPFD wrt the primary outputs is empty.*

Proof Assume that the wire is s-a-0 redundant. Thus, it can be set to zero and an equivalent circuit will be obtained. Therefore, this wire provides no information to the circuit and hence its minimum SPFD (wrt to the primary outputs) is empty. □

If the minimum SPFD is empty, it is not necessary that the wire is s-a-0 or s-a-1 redundant. This is illustrated by the example in Figure 6.1. The minimum SPFD of the wire (b, g) is empty, but the wire can be tested for both s-a-0 and s-a-1 faults. This is because even though the minimum SPFD of a wire is empty, the current function implementation at the node ensures that some of the necessary information comes in from that wire. Hence it may not be enough to set it to zero; new functions have to be derived at all nodes in the transitive fanout of the wire. However since the minimum SPFD is empty, after setting the wire to a constant zero, the functions of its destination

node and the transitive fanouts of its destination node can always be modified to obtain an equivalent circuit.

Lemma 6.2 *If the RAR techniques say that the wire $w_a = w_{\eta_j \to \eta_p}$ is an alternative wire for wire $w_t = w_{\eta_k \to \eta_n}$ (Figure 6.2), where η_p is a dominator of η_n, then the minimum SPFD of w_t wrt to η_p is contained in the SPFD of η_j derived from its function. The minimum SPFD of w_t wrt to η_p is not empty in the original network $\mathcal{N}$.*

Proof According to RAR, when the new wire w_a is added the functionalities of the nodes beyond η_p are not changed. Hence the new function of η_p after the addition of w_a must be contained within its ODC in $\mathcal{N}$. Consider two networks, $\mathcal{N}_1$ and $\mathcal{N}_2$. $\mathcal{N}_1$ consists of η_p and all its transitive fanins. $\mathcal{N}_2$ consists of η_p and all its transitive fanins and the newly added wire w_a and the nodes in the transitive fanin of η_j. It is given that the minimum SPFD of w_t wrt to η_p is not empty in $\mathcal{N}$. This is the same as the minimum SPFD of w_t wrt to η_p in $\mathcal{N}_1$ (since the computation of the minimum SPFD of w_t wrt to η_p in $\mathcal{N}$ only looks at $\mathcal{N}_1$). Hence the minimum SPFD of w_t wrt to η_p is not empty in $\mathcal{N}_1$. On the other hand, in $\mathcal{N}_2$, w_t becomes redundant after adding w_a. Hence its minimum SPFD (wrt to the primary output of $\mathcal{N}_2$ i.e. η_p) is empty (Lemma 6.1). Since the information required at the output of η_p is the same in $\mathcal{N}_1$ and $\mathcal{N}_2$, the edges in the minimum SPFD of w_t wrt to η_p now have to be contained in the SPFD of η_j derived from its function. □

In the rest of the chapter, two different SPFD-based rewiring schemes are proposed and some preliminary results are provided. The basic idea is the same as that described earlier. But some changes are made for ease of computation and for incorporating different metrics for rewiring.

6.3 Wire Replacement in Boolean Networks

The objective here is replacing a wire $w_{\eta_k \to \eta_j}$ to node η_j with a wire from another node η_s, originally not a fanin of node η_j, such that a new function $\hat{f}_j$ can be found, which depends on η_s but not on η_k. Moreover, the change in the functionality of η_j must not affect the rest of the network. The basic idea is illustrated in Figure 6.3. The function $\hat{f}_j$ must still satisfy the SPFD at η_j and some gain should be obtained by this replacement.

In the next few paragraphs, the algorithm used for identifying alternate wires and a resynthesis process using a chosen alternate wire is described.

The algorithm proceeds in topological order from primary inputs to primary outputs and attempts to change the wiring of each node in the network. In order to avoid a change in the wiring

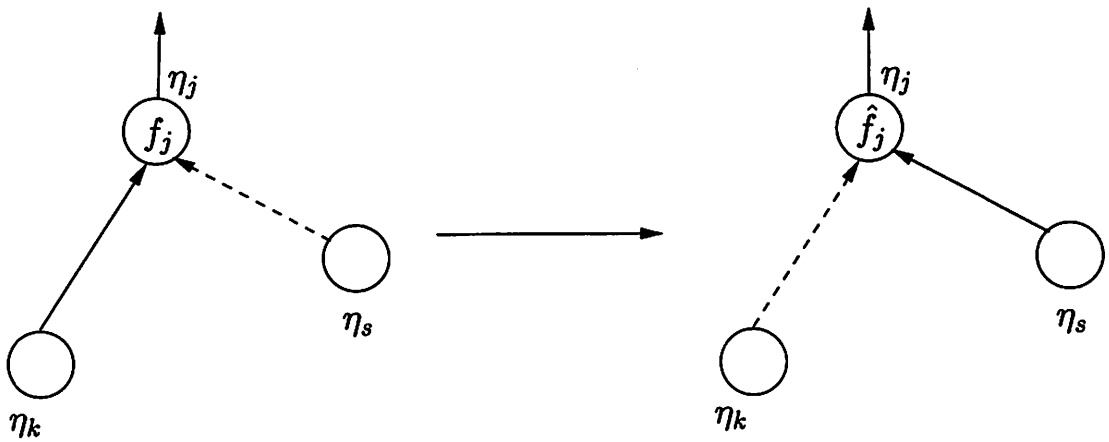


Figure 6.3: Rewiring: the solid lines indicate wires and the dotted lines indicate non-existing wires.

at a node from affecting the remainder of the circuit, the SPFD of each node is derived from its CODC. The CODC is used instead of the ODC since the ODC computation is more expensive, and once a node is modified, the ODCs of the remaining nodes will have to be re-computed. This idea is similar to the ROC idea used in Chapter 5 for blocking the changes made to a single node in a network.

Consider a node η_j in the network. Let $R_j(Y_j, Y_j')$ denote its SPFD derived from its original function and its CODC. The unique set of edges in $R_j(Y_j, Y_j')$ that can only be distinguished by η_k is computed. Let this be denoted as $R_{kj}^{min}(Y_j, Y_j')$. Thus,

$$R_{kj}^{min}(Y_j, Y_j') = R_j(Y_j, Y_j') \wedge \overline{\cup_{y_i \in Y_j; y_i \neq y_j}(y_i \neq y_i')}.$$

Then candidate nodes $\{\eta_s\}$ are sought that can distinguish all the minterms in $R_{kj}^{min}(Y_j, Y_j')$. A necessary and sufficient condition is that $H(y_s) = \phi$, where $H(y_s)$ is derived by the following steps:

1. *Substitute* $y_i = f_i^g(X)$ *in* $R_{kj}^{min}(Y_j, Y_j')$ *for each* $y_i \in Y_j$ *to obtain* $R_{kj}^{min}(X, Y_j')$.
2. *Compute* $R_{kj}^{min}(y_s, Y_j') = \{\exists_X (y_s = f_s^g(X)) R_{kj}^{min}(X, Y_j')\}$.
3. *Substitute* $y_i' = f_i^g(X)$ *in* $R_{kj}^{min}(y_s, Y_j')$ *for each* $y_i' \in Y_j'$ *to obtain* $R_{kj}^{min}(y_s, X)$.
4. *Compute* $H(y_s) = \{\exists_X (y_s = f_s(x)) R_{kj}^{min}(y_s, X)\}$.

$H(y_s)$ has the property that if $H(y_s) \neq \emptyset$, then there exists at least one pair of minterms in $R_{kj}^{min}(Y_j, Y_j')$ that cannot be distinguished by η_s and hence η_s cannot be a candidate.

Since it is not practical to consider all the nodes in the network, only a subset is considered; only the fanins of η_k and the nodes in their transitive fanout are considered. Of course, nodes in the transitive fanout of η_j cannot be considered.

After the set of candidate nodes S is obtained, a new function at η_j is derived for each $\eta_s \in S$. The procedure is similar in spirit to the resynthesis algorithm presented in Chapter 5. The main difference is in the derivation of the modified SPFD.

The modified SPFD of η_j is obtained in the new space:

$$\hat{Y_j} = \{y_i \in Y_j, i \neq k\} \cup \{y_s\}.$$

Then a new minimized function at node η_j is obtained from this modified SPFD as described in Chapter 5. If the number of literals in the factored form of the new function is less than the number in the factored form of f_j, the replacement is done. In case of a tie in the number of literals, the replacement is also done if the level of η_s is less than the level of η_k. Otherwise, the next node in the candidate set is selected and the same procedure repeated.

This procedure is implemented SIS as a command, **wire_replace**.

6.3.1 Results

The results for **wire_replace** are shown in the Table 6.1. The initial circuits were obtained by optimizing the original blif circuits using **script.rugged**. These were then subjected to an iteration of **wire_replace** until no gain was obtained. The number of wires, the number of literals in the factored form of the network and the ratio of these results to the output of **script.rugged** are under the heading **(wire_replace)***. For *i*8, the values of the previous iteration are used, since the program ran out of memory before the iterations could converge. The third set of columns was obtained by taking the result of **(wire_replace)*** and repeating **script.rugged** followed by **(wire_replace)*** until no gain was recorded. For *k*2 and *too_large*, the program ran out of memory even before the first iteration was over. At the bottom of the table the average ratios for both experiments and for both wires and for literals is computed. On average a 11% reduction in wires and 6% in literals after **(wire_replace)*** is obtained. Better results are obtained for the repetition of **script.rugged** and **(wire_replace)***, a 19% reduction in wires and 12% in literals. Note that all the computations are done using BDDs.

	script.rugged		*(wire_replace)**				*(script.rugged, (wire_replace)*)**			
NAMES	wires	literals	wires	ratio	literals	ratio	wires	ratio	literals	ratio
apex6	650	741	625	0.962	724	0.977	621	0.955	715	0.965
apex7	222	245	203	0.914	235	0.959	198	0.892	226	0.922
b9	115	122	111	0.965	119	0.975	112	0.974	122	1
bbara	51	63	46	0.902	60	0.952	49	0.961	61	0.968
bbsse	140	140	119	0.85	126	0.9	102	0.729	110	0.786
c1908	378	540	352	0.931	525	0.972	352	0.931	525	0.972
c432	205	205	186	0.907	222	1.083	186	0.907	222	1.083
c499	344	552	300	0.872	552	1	300	0.872	552	1
c8	128	139	127	0.992	138	0.993	125	0.977	136	0.978
cht	165	165	164	0.994	165	1	163	0.988	164	0.994
cse	213	215	201	0.944	204	0.949	162	0.761	183	0.851
dk16	348	348	316	0.908	321	0.922	187	0.537	245	0.704
dk17	88	89	40	0.455	53	0.596	37	0.42	51	0.573
ex1	279	280	255	0.914	258	0.921	219	0.785	229	0.818
ex2	172	172	142	0.826	160	0.93	134	0.779	151	0.878
ex3	84	86	82	0.976	85	0.988	45	0.536	62	0.721
ex4	91	91	82	0.901	85	0.934	71	0.78	78	0.857
ex5	71	71	60	0.845	67	0.944	26	0.366	51	0.718
ex6	108	109	92	0.852	103	0.945	89	0.824	96	0.881
f51m	60	91	39	0.65	83	0.912	45	0.75	70	0.769
frg1	79	136	44	0.557	127	0.934	51	0.646	127	0.934
frg2	833	886	696	0.836	792	0.894	690	0.828	735	0.83
i6	391	457	391	1	457	1	391	1	457	1
i7	518	584	517	0.998	583	0.998	517	0.998	583	0.998
i8	1012	1015	980	0.968	988*	0.973*				
i9	587	596	584	0.995	596	1	580	0.988	592	0.993
k2	1112	1120	1067	0.96	1082	0.966	*		*	
kirkman	300	308	137	0.457	198	0.643	85	0.283	126	0.409
lal	89	105	82	0.921	101	0.962	79	0.888	102	0.971
planet	614	617	586	0.954	593	0.961	555	0.904	589	0.955
s1	429	430	349	0.814	381	0.886	275	0.641	298	0.693
sand	612	613	566	0.925	574	0.936	521	0.851	550	0.897
scf	983	985	970	0.987	974	0.989	870	0.885	917	0.931
sct	63	79	57	0.905	78	0.987	55	0.873	75	0.949
sse	140	140	119	0.85	126	0.9	102	0.729	110	0.786
styr	596	596	550	0.923	555	0.931	431	0.723	482	0.809
term1	130	179	97	0.746	152	0.849	93	0.715	103	0.575
too_large	266	347	253	0.951	234	0.674	*		*	
ttt2	184	219	160	0.87	206	0.941	122	0.663	163	0.744
vda	611	615	607	0.993	612	0.995	571	0.935	579	0.941
x1	285	298	279	0.979	295	0.99	279	0.979	295	0.99
x2	44	48	43	0.977	48	1	39	0.886	46	0.958
x3	720	787	650	0.903	753	0.957	628	0.872	705	0.896
x4	367	386	347	0.946	381	0.987	332	0.905	367	0.951
z4ml	29	41	28	0.966	38	0.927	28	0.966	38	0.927
AVERAGE				0.888		0.936		0.807		0.871

Table 6.1: Results for **wire_replace**.

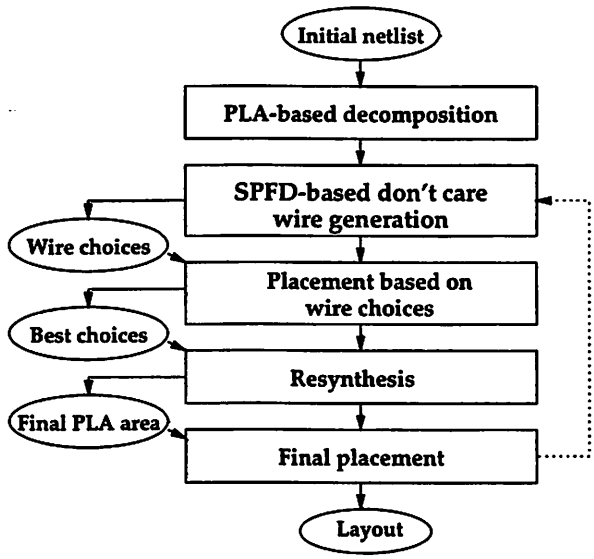


Figure 6.4: *Don't care* wire-based logic/physical design flow.

6.4 Don't Care Wires

The results in the previous section illustrate the rewiring ability of SPFDs. In this section, rewiring is performed in an integrated synthesis- placement engine so that a more sophisticated metric can be used for rewiring.

First a brief description of the flow of the algorithm is provided. Then some of the steps in the flow are examined in greater detail

6.4.1 Flow

The basic flow is shown in Figure 6.4. An initial network is first minimized using the usual logic synthesis methods. Then it is decomposed and clustered into PLAs with the target of absorbing as many wires as possible internally in each PLA, with the constraint that the resulting PLA network has no cycles. During this clustering no placement information is known, so a heuristic is used that the smaller the number of wires, the better the clustering. This usually leads to a smaller number of PLAs. During clustering, the logic is minimized and the PLA folded. Then the result is assessed for being within given bounds on the number of rows and columns (dictated mainly by delay and noise constraints within the PLA) of the resulting PLA structures.

After clustering, a set of compatible alternates is generated for each input wire. SPFDs are used for generating alternate wires. These alternates are used in a floorplanning algorithm where during each move the best choice of alternate wires for each local input is used to evaluate the move. Once the final placement and final netlist are chosen, the logic inside a PLA may change and may no longer fit within the row and column bounds; however, the experiments indicate that the PLA areas are usually well controlled in this process. Note that the number of inputs and outputs does not change for a given PLA.

6.4.2 Network of PLAs

Recent work on noiseless fabrics [27] led to a re-examination of the use of multi-level networks where each logic node is implemented as a PLA. This is a general logic synthesis technique, and has been shown to have advantages even for implementations where noise is not a concern. In some sense the PLAs are similar to the initial application of SPFDs to FPGAs; each node contains a significant logic function, and if that logic function changes, the area requirement for implementing the function does not change much. For FPGAs, the area does not change at all if the number of inputs does not change. For PLAs, the area may change but typically if the number of inputs does not change, the area change can be controlled(it is possible to use bit pairing, folding, etc to keep the area within bounds). PLAs offer some additional advantages in that the layout for each PLA is regular and can be accurately characterized in terms of its electrical characteristics (delay, noise, etc.).

In the experiments, the PLAs are of medium size (e.g. 10–15 inputs, 1–5 outputs, 15–25 cubes).

6.4.3 SPFDs and Compatible Wire Sets

The basic idea is similar to that presented in the previous section, where a wire replaces another wire if it can provide all the required information. However since it is desirable to retain the freedom of choosing an alternate wire for one wire, independent of the choices made for the other wires, the sets of alternate wires are made **compatible**. The idea is similar to the concept of logic *don't cares*, where it is desirable to be able to choose the function for a particular node independent of the values chosen for the other nodes. So compatible sets must guarantee that the union of information coming into a node through its input wires is **always** enough to supply the information required for that node's output requirements.

Let S_{kj} denote the set of alternate sources for wire $w_{\eta_k \to \eta_j}$. The sets of alternate wires are constructed using the algorithm, **compute_comp_alts**.

Algorithm **compute_comp_alts**($\mathcal{N}$):

1. *Starting from the primary outputs and proceeding in a backward topological order, for each node η_j in the network, and each of its input wires $w_{\eta_k \to \eta_j}$, assign SPFDs using the procedure described in Section 5.1. R_j and $R_{w_{\eta_k \to \eta_j}}$ denote the required information of η_j and $w_{\eta_k \to \eta_j}$, respectively. Once this is done, each SPFD represents the set of minterms which must be distinguished by that node or wire.*

2. *Initialize for each node η_j,*

$$ex(\eta_j) = \{\eta_j\} \cup TFO(\eta_j).$$

and for each input wire $w_{\eta_k \to \eta_j}$ of η_j, let $S_{kj} = \phi$. S_{kj} will eventually represent the set of alternate wires for $w_{\eta_k \to \eta_j}$.

3. *Starting from the primary inputs and proceeding in some topological order, at each node η_j, do the following:*

   (a) *Let $C = \overline{ex(\eta_j)}$*

   (b) *For each fanin wire $w_{\eta_k \to \eta_j}$ of η_j:*

      i. *Find an $\eta_s \in C$ such that $R_{w_{\eta_k \to \eta_j}} \subseteq R_s$.*

      ii. *Include η_s in S_{kj}, $S_{kj} = S_{kj} \cup \{\eta_s\}$.*

      iii. *For each $\eta_p \in \{TFI(\eta_s) \cup \eta_s\}$, update $ex(\eta_p) = ex(\eta_j) \cup ex(\eta_p)$. (This is done to avoid cycles in the resulting network.)*

      iv. *This continues until no more nodes can be added to S_{kj}.*

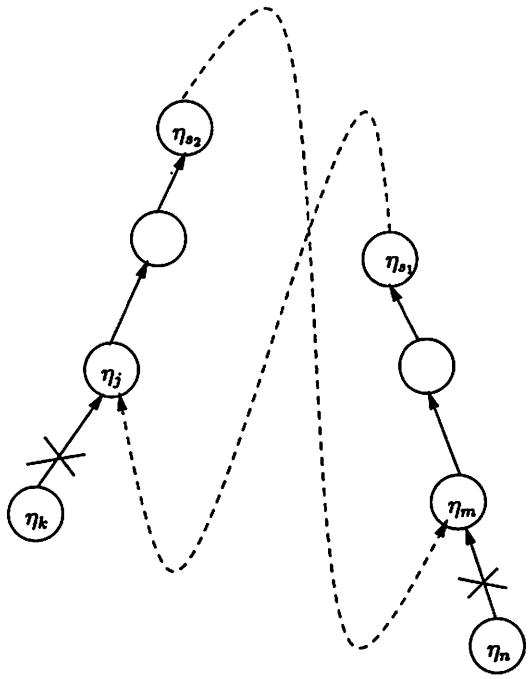


Figure 6.5: Network $\mathcal{N}'$: $w_{\eta_n \to \eta_m}$ is replaced by $w_{\eta_{s_2} \to \eta_m}$ and $w_{\eta_k \to \eta_j}$ is replaced by $w_{\eta_{s_1} \to \eta_j}$.

Given the set of alternate wires $\{S_{kj}\}$, an alternate is picked for each wire in the original circuit such that some optimization criteria such as total wirelength is minimized[1]. Let $\mathcal{N}'$ denote the new network that is obtained by replacing each original wire in $\mathcal{N}$ with its chosen alternate. Note that the functions of the nodes in $\mathcal{N}'$ cannot be the same as their corresponding functions in $\mathcal{N}$ as the fanin supports of a node could be different in $\mathcal{N}$ and $\mathcal{N}'$. The new functions of the nodes in $\mathcal{N}'$ are computed in a topological order from the primary inputs to the primary outputs. At each node $\eta_j \in \mathcal{N}'$, a new SPFD is derived by expressing its original SPFD (obtained after Step 1 of **compute_comp_alts**) in terms of the fanins of η_j in $\mathcal{N}'$. This new SPFD is then colored for obtaining a new function at η_j.

In the rest of the section, it is proved that the new network $\mathcal{N}'$ derived above always implements the same functionality as the original network $\mathcal{N}$.

Definition 6.1 *Given a set of sets of nodes $S = \{S_{kj}\}$, a* **selection** *is a ordered set of nodes $\{\eta_1, \ldots, \eta_{|S_{kj}|}\}$ such that $\eta_k \in S_{kj}$.*

Definition 6.2 *Given any selection of $\{S_{kj}\}$, the network* **derived** *from the selection is obtained*

[1]The algorithm for picking a suitable alternate for each wire in the network is described in Section 6.4.4.

from the original network $\mathcal{N}$ by replacing each wire $w_{\eta_k \to \eta_j}$ in $\mathcal{N}$ by a wire $w_{\eta_p \to \eta_j}$ such that $\eta_p \in S_{kj}$.

Definition 6.3 *A set of sets $S = \{S_{kj}\}$ is* **compatible** *if for each selection, there exist logic functions at each node such that the network derived from that selection can implement the specifications at the primary outputs.*

Definition 6.4 *The* **old global SPFD** *of a node(wire), $\eta_j(w_{\eta_k \to \eta_j})$, is the SPFD attached to it in $\mathcal{N}$ after Step 1 of* **compute_comp_alts**. *It is expressed in terms of the primary inputs of $\mathcal{N}$ and is denoted as $R_j(R_{w_{\eta_k \to \eta_j}})$.*

Definition 6.5 *The* **rewired SPFD** *of a node η_j is the SPFD obtained by expressing its old global SPFD in terms of the fanins of η_j in the new network $\mathcal{N}'$, after all the nodes in the transitive fanin of η_j have been resynthesized in $\mathcal{N}'$. It is denoted as R_j^r.*

The set $\{S_{kj}\}$ obtained by the procedure has the following property:

Lemma 6.3 *For any selection of $\{S_{kj}\}$, the network derived from the selection is acyclic.*

Proof Assume there exists a selection such that the network derived from it is cyclic. This occurs only if the situation shown in the Figure 6.5 exists in the new network. This in turn can happen only if **compute_comp_alts** puts η_{s_1} in S_{kj} and η_{s_2} in S_{nm}. It is argued below that this is impossible. In **compute_comp_alts**, the sets $\{S_{kj}\}$ are built in a particular order (Step 3). Suppose S_{kj} is constructed before S_{nm}. Since $\eta_{s_1} \in S_{kj}$, then for each η_p in the transitive fanin of η_{s_1}, $ex(\eta_p)$ includes all the nodes in $ex(\eta_j)$ (Step 3(b)(iii)) of the algorithm). Now $\eta_{s_2} \in ex(\eta_j)$ since it is in the transitive fanout of η_j in the original network. Thus $\eta_{s_2} \in ex(\eta_m)$. Hence when the set S_{nm} is being constructed in a later step in the algorithm, the node η_{s_2} will not be included in the set C (Step 3(a)) and hence can never be included in the set S_{nm}. □

Lemma 6.4 *Given a set of sets of nodes $\{S_{kj}\}$ satisfying the following two properties:*

1. *for any selection, the network derived from it is acyclic, and*
2. $\eta_s \in S_{kj} \longrightarrow R_{w_{\eta_k \to \eta_j}} \subseteq R_s$.

Let $\mathcal{N}'$ denote the new network derived from any given selection of $\{S_{kj}\}$. Any function that satisfies the rewired SPFD of a node in $\mathcal{N}'$ also satisfies its old global SPFD.

Proof By Lemma 6.3, any selection is guaranteed to produce an acyclic network. Thus $\mathcal{N}'$ can always be levelized. The above theorem is proved by induction on the levels of $\mathcal{N}'$.

Base Case Consider a node η_j of level 1. For each fanin η_k of η_j in $\mathcal{N}$, the selection picks a node, $\eta_p \in S_{kj}$. Note that η_p has to be a primary input, since η_j is a level 1 node in $\mathcal{N}'$. Condition 2 ensures that $R_{w_{\eta_k \to \eta_j}} \subseteq R_p$. Hence, $R_j^r = R_j$. Thus, any function that satisfies R_j^r also necessarily distinguishes all the edges in R_j.

Inductive Step Given that the functions of all the nodes in $\mathcal{N}'$ of level $\leq k$ satisfy their rewired SPFDs, it is necessary to prove the above theorem is true for all nodes of level $(k+1)$. Assume that's not true. Then there exists a node η_j of level $(k+1)$ such that its new function satisfies its rewired SPFD R_j^r, but an edge $e = (x, x')$ in its old global SPFD R_j is not distinguished by the new function. According to the SPFD computation algorithm in Step 1 of **compute_comp_alts**, each edge in R_j has to be assigned to a fanin of η_j in $\mathcal{N}$. Thus, $R_j \subseteq \cup_{\eta_k \in FI^{\mathcal{N}}(\eta_j)} R_{w_{\eta_k \to \eta_j}}$, where $FI^{\mathcal{N}}(\eta_j)$ denotes the fanins of η_j in $\mathcal{N}$. Condition 2 ensures that any alternate source η_p of a fanin wire $w_{\eta_k \to \eta_j}$ satisfies the following condition : $R_{w_{\eta_k \to \eta_j}} \subseteq R_p$. Hence, $R_j \subseteq \cup_{\eta_p \in FI^{\mathcal{N}'}(\eta_j)} R_p$, where $FI^{\mathcal{N}'}(\eta_j)$ denotes the fanins of η_j in $\mathcal{N}'$. Thus the edge (x, x') necessarily belongs to the old global SPFD of a fanin of η_j in $\mathcal{N}'$. Let the fanin be η_p. Since, the level of $\eta_p \leq k$ in $\mathcal{N}'$, then by the induction hypothesis, the function at η_p satisfies its old global SPFD R_p. Thus x and x' have to evaluate to different values at the output of η_p. This implies that the edge (x, x') in R_j has to produce an edge (y, y') in the rewired SPFD, since x maps to a minterm y in the new fanin space and x' maps to a different minterm y'. Hence the edge (y, y') definitely exists in the rewired SPFD. Thus any function that satisfies its rewired SPFD also satisfies its old global SPFD. □

Theorem 6.1 *Any set of sets of nodes $\{S_{kj}\}$ satisfying:*

1. *for any selection, the network derived from it is acyclic, and*
2. $\eta_s \in S_{kj} \longrightarrow R_{w_{\eta_k \to \eta_j}} \subseteq R_s$,

is compatible.

Proof Lemma 6.4 proves that if each node in $\mathcal{N}'$ is implemented using a function that satisfies its rewired SPFD, the old global SPFD of the node is also satisfied. This is also true for all the primary outputs of $\mathcal{N}'$. The old global SPFD of a primary output distinguishes its onset minterms in $\mathcal{N}$ from its offset minterms. Thus the specifications at the primary outputs (given by the original network $\mathcal{N}$) are satisfied in $\mathcal{N}'$ if each node in $\mathcal{N}'$ is implemented using a function that satisfies its rewired SPFD. Hence the set of sets of nodes $\{S_{kj}\}$ is compatible. □

Thus to form a compatible set of alternate wires it is sufficient to make sure that whatever netlist is chosen, it is acyclic and that each alternate's SPFD covers the original input's SPFD.

The rewiring done here has the property that the global SPFDs of the nodes remains unchanged. In Section 6.5, conditions where the global SPFDs of the nodes may no longer be the same after rewiring are explored. This will require more complicated algorithms for computing the functions at the nodes in the network after rewiring. In Chapter 7, an algorithm that can be used for synthesizing the nodes under these more relaxed conditions is provided.

In the next few sections, the placement algorithms that are used in the flow are described. For convenience of explanation, a slightly different terminology from the previous sections is used. Each PLA has input pins and output pins. A net consists of an output pin and all its wires. If $w_{\eta_s \to \eta_j}$ is an alternate wire for $w_{\eta_k \to \eta_j}$, then η_s is an alternate source for the input pin that supplies to η_k.

6.4.4 An Assignment Problem

In the experiments, the total wire length was used as the cost function to be minimized. Note that indirectly, this controls total area, routability, and power, but not necessarily worst case delay. For example, a significant area increase will result in an increase in total wire length. Evaluating the total wire length of a placement requires that a best selection of alternate wires be made for that placement. Thus the following problem is obtained.

Alternate Wire Choice Problem (AWC) *Given a point placement of pins, and a set* $\{R_\sigma\}$ *of candidate sources for each pin, find the selection which minimizes the sum of the half perimeters of the bounding boxes of the nets.*

Theorem 6.2 *(Chong) The Alternate Wire Choice (AWC) Problem is NP-complete.*

Branch and bound techniques can be applied to solve AWC exhaustively. However, for efficiency the following algorithm is proposed.

Procedure 6.1 *(Semi-greedy Algorithm for AWC)*
PHASE I:

1. For each pin with alternate wires, temporarily disconnect it from the current net.

2. For each net form the bounding boxes of the currently connected pins. These partial bounding boxes form a lower bound on the total wire length.

3. For each pin with alternate wires, if its pin position is inside one of the partial bounding boxes for its candidate wires (the original wire plus its alternates), assign it to that net. No increase has been caused by this assignment, and hence the partial assignment seen so far must be part of an optimum assignment.

4. For each remaining pin with alternate wires, compute the "delta" costs if it is assigned to each of the candidate nets. There is a net assignment which increases the total net length by the least amount. Choose this assignment and update the chosen net.

5. Continue step 4 until all pins have been assigned.

PHASE II:

1. For each pin which is an extreme of the bounding box of its currently assigned net, temporarily release it from its assignment, and compute the best net to put it in and its delta decrease cost in doing this. Note that the delta decrease is nonnegative.

2. Choose the pin with the maximum delta decrease and reassign the pin to the new net.

3. Repeat 1 and 2 until the best delta is 0.

Notes:

- After PHASE I, there may be pins that can be moved to different nets to improve the total cost.
- After Step 2 in PHASE II, the deltas need to be updated efficiently.
- During PHASE II, a pin may be reassigned more than once. To speed up the process, one may want to "lock" a pin once it is reassigned once.
- After PHASE II (with no locking), the solution is locally optimal, in that there is no pin which can be moved to a new net such that the total cost is decreased. However, there might be a set of pins that can be reassigned all at once which decreases the cost.

6.4.5 Two Placement Algorithms

For this work, total wirelength, measured by the half-perimeter bounding box for each net, is used as the metric for the final design. By minimizing overall wirelength, the total wiring utilization for the design and hence minimize overall congestion is reduced.

Note that the half-perimeter bounding box metric is affected by the locations of the pins on the PLAs. However, since the exact pin locations are not available, these locations are estimated by the center points of blocks.

If the alternate wire choices for each input pin are considered and the network is optimized for both area and total wire length, a two dimensional solution space-physical placement and logical wire-is obtained. Given a physical placement of the blocks as points in the physical dimension, choosing the best set of logical wires is NP-complete (Theorem 6.2). Similarly given a set of wires, choosing the best placement is also hard. Here two approaches are provided for tackling this combined problem.

6.4.5.1 Mincut Placement Approach

One of the approaches uses a mincut placement algorithm [28] to evaluate the placement of a netlist with alternate wires. This method differs from traditional mincut placement techniques by using alternate wires to change the cut costs during the recursive bipartitioning of the design. Choosing alternates for wires on a cut net may prevent that particular net from being cut at all, thus reducing the cost. Therefore the cost of a partition is evaluated by accounting for such effects. This reduction in cut cost will generally translate to a reduction in wire length for the final placement; alternate choices which prevent nets from being cut during bipartitioning will generally correspond to the selection of shorter local wires.

The FM partitioning algorithm [29] was modified to account for alternate wires. After recursive bipartitioning is applied to a design, partitions are adjoined in a quadrature fashion [28] to obtain a placement. As well, additional wire length minimization heuristics are used to guide the placement.

After mincut partitioning, a low-temperature simulated annealing, based on a sequence pair representation [30], is used to further improve the layout. In the annealing process, after each random move on the sequence pair, the layout is derived and the greedy AWC algorithm is applied to give the best wire length based on the wire choices. Once annealing is done, a greedy compaction method is applied, which evaluates the best location for each cell for minimal wire length. Finally, the AWC problem is solved for this layout using branch and bound to obtain the final wire choices.

6.4.5.2 Force-Directed Approach

Another approach uses a force-directed placement algorithm. The force-directed placer is incremental, so the AWC subroutine can be easily embedded. At each step the new position of

Design	PLAs/ IPins	Regular APins #(%)	Regular Alts #	Maximum APins #(%)	Maximum Alts #
alu2-5	18/233	32(13.7)	28.44	37(15.9)	37.43
apex6-5	37/553	21(3.8)	16.10	27(4.9)	81.56
apex7-4	12/157	9(5.7)	22.22	12(7.6)	38.75
apex7-5	11/146	5(3.4)	14.40	6(4.1)	55.83
count-4	6/67	4(6.0)	12.75	4(6.0)	30.25
count-5	6/68	3(4.4)	21.67	3(4.4)	35.00
term1-4	15/186	23(12.4)	19.61	29(15.6)	37.03
term1-5	12/170	11(6.5)	32.55	15(8.8)	44.00
ttt2-4	7/73	7(9.6)	14.00	7(9.6)	15.29
ttt2-5	8/85	9(10.6)	15.22	10(11.8)	18.30
x4-5	24/269	19(7.1)	34.05	28(10.4)	32.64

Table 6.2: Characterization of Examples.

the cells is computed in terms of the forces acting on the cells, where the forces are generated from the existing wires attached to each PLA. Then AWC is invoked to determine if better wire choices exist. All input and output ports are fixed on the chip boundary so that no trivial solution (all cells collapse into one single point) will be derived. To overcome cell overlaps, the algorithm introduced in [31] is used, while some modifications are made to improve speed. The basic idea is to form a density field in the chip area. Cells in this field tend to move towards those areas with lower density and away from areas with higher density.

6.4.6 Experimental Results

Two experiments were performed for investigating the contribution of alternate wires.

Experiment I: The first experiment was to decompose each example into a set of PLAs as described in Section 6.4.2. Table 6.2 shows the results of this decomposition. The number following the design name is related to the maximum physical width allowed for each PLA in the decomposition [27]. The resulting number of PLAs for each design is shown in the *PLAs* column, and the total number of input pins on these PLAs is shown in the *IPins* part.

Don't care wires were generated for each of these examples. The number of pins with alternate wires for each example is shown in the *APins* column under the *Regular* heading (the *Maximum* columns are described in Experiment II below). The percentage (in parentheses) of input pins which have alternate choices is also shown. The average number of alternate choices for each

Design	Init	Reg	Resyn	Max
alu2-5	6143.5	19.8	16.8	20.9
apex6-5	18053.5	0.0	3.3	0.0
apex7-4	2843.5	2.2	13.9	13.3
apex7-5	2512.0	6.2	15.5	7.0
count-4	758.0	4.2	8.0	0.0
count-5	849.0	0.0	0.0	0.0
term1-4	4748.0	8.9	34.3	14.2
term1-5	4057.0	4.2	16.4	16.0
ttt2-4	1251.0	22.4	23.1	22.4
ttt2-5	1116.0	14.2	0.0	0.0
x4-5	4590.5	0.0	0.0	0.0
average	4265.6	7.5	12.0	8.7

Table 6.3: Wirelength Improvement, Mincut.

of these pins is shown in the *Alts* column.

The following comparisons were performed:

1. The PLAs were placed without using alternate wires. The total wire lengths for these initial placements (using the two placement methods) are shown in the *Init* column of Tables 6.3 and 6.4.

2. The same placement algorithms were applied on the network of PLAs using alternate wires. The percentage improvement in wire length over the initial placement is shown in the *Reg* column in the two tables.

3. The chosen best wires were returned to logic synthesis and the functionalities of the PLAs were determined according to the wire choices. Another placement was performed using the new PLA areas, and the resulting wire lengths were compared to the initial results. The improvement in wire lengths over the initial placement is shown in the *Resyn* column in the tables.

Experiment II: In the results for Experiment I, a fairly high correlation is observed between the improvement in wire length and the percentage of wires that have alternates. Note that the percentage of wires with alternates for the examples is small (on average about 7.5%). As an additional experiment, the effect of having more wires with alternates was determined. For this, the acyclic constraint was ignored when generating alternates. In addition, for each wire $w_{\eta_k \rightarrow \eta_j}$,

Design	Init	Reg	Resyn	Max
alu2-5	6492.0	6.4	7.6	7.4
apex6-5	22253.0	7.7	1.1	9.9
apex7-4	3097.0	1.8	0.7	3.3
apex7-5	2688.0	9.5	5.7	10.4
count-4	788.0	5.1	4.7	3.9
count-5	823.0	0.8	1.5	1.1
term1-4	5374.0	11.9	2.8	4.2
term1-5	6112.0	0.8	1.5	1.8
ttt2-4	1111.0	3.5	3.1	11.4
ttt2-5	1649.0	5.3	12.4	3.8
x4-5	6148.0	3.4	1.5	1.9
average	5139.5	5.1	3.9	5.4

Table 6.4: Wirelength Improvement, Force Directed.

the minimum set of edges distinguished by it in the SPFD of η_j was computed and another wire was designated as an alternate if its SPFD covered this smaller SPFD. The resulting number of pins with alternate wires and the average number of choices for each of these is shown in the *Maximum* columns of Table 6.2. This generated only a few more wires with alternates (their average increased to 9%), although the average number of alternates on wires with at least one alternate increased substantially.

The wire length improvement over the initial placement using these extended sets of alternate wires is shown in the *Max* columns of Tables 6.3 and 6.4. This figure loosely indicates an upper bound on the possible improvement due to alternate wires alone, and should be compared to the *Reg* column since resynthesis was not done. As expected, the results obtained correlate with the increased number of wires with alternates.

6.4.6.1 Some Observations

Although not presented in Tables 6.3 and 6.4, there was a change in total areas of the placed designs when alternate wires were used. For all experiments, the worst-case final placed area increase was 8%. This small increase in area is partly due to the choice of total wirelength as a metric; since minimizing area was not the main intention, the final design area can be expected to increase after replacing a wire with its alternate. Also, after selection of alternate wires the network has to be resynthesized, and so a change in the PLA areas at that stage is also possible.

As noted, the gains in wirelength achieved is very much correlated with the percentage of pins which have alternates. When these were increased from 7.5% (*Regular*) to 9% (*Maximum*) in Experiment III, the gain in wire length improvement went from 7.5% to 8.7% for the mincut placement, and 5.1% to 5.4% for the force-directed technique.

In some cases, there was an increase in wire length when alternate wires were introduced. There are two explanations for this. First, the placement algorithms do not guarantee a global minimum, so different local minima can be obtained. Second, using the mincut placement technique, there is no direct relation between the cut sizes in the recursive bipartitioning and the final placement wirelengths. Thus a selection of alternate wires which reduces the cost of a cut may in fact increase the total wirelength. For cases where using alternate wires increases the total wirelength, the results were ignored and instead the initial placement generated without alternate wires was used.

6.5 Partial Don't Care Wires

This idea is an extension of the *don't care* wires described in the previous section. In the previous section, it was mentioned that a positive correlation was observed between the improvement in wirelength and the percentage of wires that have alternate wires. This motivated the search for more alternate wires.

The idea here is to relax the conditions an alternate wire must satisfy, thereby possibly increasing the number of wires with alternate wires. Thus, a wire $w_{\eta_m \rightarrow \eta_j}$ can be an alternate wire for another wire $w_{\eta_k \rightarrow \eta_j}$ if it can provide only a part of the information that the original wire provides. The actual amount of information that the alternate wire has to provide can be a parameter of the algorithm. The wire $w_{\eta_m \rightarrow \eta_j}$ is then called a *partial don't care* wire of $w_{\eta_k \rightarrow \eta_j}$.

The resynthesis problem can become more involved with *partial don't care* wires. This is because some of the original information of the network might be missing when the original wires are replaced by their *partial don't care* wires. It is the task of the resynthesis algorithm to fill in the missing information. An algorithm presented in the next chapter can be used for the resynthesis problem.

6.6 Summary

Rewiring using SPFDs was described in this chapter. Two different rewiring scenarios were presented. Rewiring in Boolean networks produced some significant reduction in wire count

and literal count. Analogous to a logic *don't care*, the concept of *don't care* wires was presented. Rewiring in an integrated synthesis-placement environment using *don't care* wires also provided some encouraging results. The interesting conclusion is that there was a positive correlation between an increase in the number of alternate wire and an improvement in wirelength. A generalization of *don't care* wires called *partial don't care* wires was also proposed. Experiments need to be done for evaluating the benefit of this generalization.

Some initial arguments were provided for proving that SPFD-based rewiring can be more powerful than some other previous approaches, like RAR. It will be interesting to conduct a more thorough theoretical and practical investigation for exactly determining the added advantages of SPFD-based rewiring.

Chapter 7

SPFDs and Decomposition

In this chapter, the idea of using SPFDs for a particular kind of functional decomposition called topologically constrained logic decomposition is explored. First a very brief overview of the previous work on functional decomposition is provided, and then some motivation is given about why SPFDs can be used for functional decomposition. A generalization of the traditional functional decomposition problem is proposed and it is shown how SPFDs can be used for solving the problem.

7.1 Previous Work

Decomposition is a fundamental problem in logic synthesis. Its goal is to break a function into smaller functions. The problem can be stated as:

$$\begin{aligned} F(X) &= G(H(X_1), X_2), \\ X_1 \cup X_2 &= X. \end{aligned}$$

Generally, G and H are less complex than F. It is known, in the worst case that the circuit size realizing an n-input logic function is $O(2^n/n)$. If $F(X)$ has a decomposition $G(H(X_1), X_2)$, the worst case for the decomposed circuit is $O(2^{n_1}/n_1 + 2^{n_2+1}/(n_2+1))$, where $n_1 = |X_1|$ and $n_2 = |X_2|$. Thus functional decomposition can reduce the circuit size exponentially.

The first systematic study of decomposition [32] characterized the existence of a simple disjoint decomposition of a function. This is a special case of the above equations, where $X_1 \cap X_2 = \phi$ and G is a single output function. The problem is shown in Figure 7.1.

The set X_1 is called the bound set and X_2 the free set. This procedure is described in some detail for explaining the basic idea behind functional decomposition.

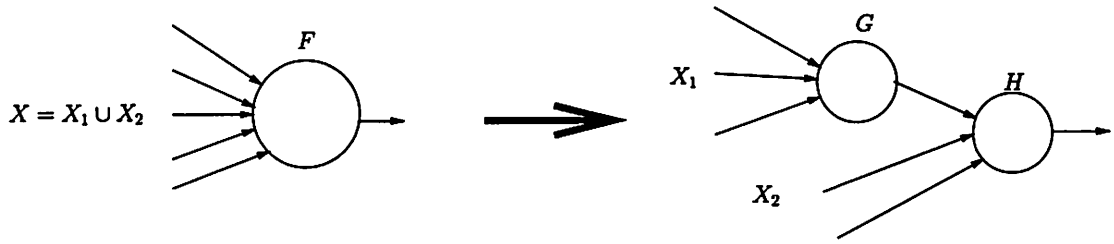


Figure 7.1: Ashenhurst decomposition.

	00	01	11	10
0	0	0	1	0
1	1	1	0	1

Figure 7.2: Decomposition chart.

The necessary and sufficient condition for the existence of a decomposition was given in terms of the decomposition chart $D(X_1|X_2)$ for F for the partition $X_1|X_2$. A decomposition chart is a truth-table of F where the vertices of $B^n = \{0,1\}^n$ are arranged in a matrix. The columns of the matrix correspond to the vertices of $B^{X_1} = B^s$, and its rows correspond to the vertices of $B^{X_2} = B^{n-s} = B^t$. The entries in $D(X_1|X_2)$ are the values that F takes for all possible input combinations. For example, if $F(a,b,c) = ab\bar{c} + \bar{a}c + \bar{b}c$, the decomposition chart for F for the partition $ab|c$ is shown in Figure 7.2.

Ashenhurst proved the following result, which relates the existence of a decomposition to the number of distinct columns in the decomposition chart $D(X_1|X_2)$.

Theorem 7.1 *(Ashenhurst) A simple disjoint decomposition exists if and only if the corresponding decomposition chart has at most two distinct columns.*

Two vertices x_1 and x_2 in B^s are compatible if they have the same column patterns. For an incompletely specified function, a *don't care* entry - cannot cause two columns to be incompatible. Thus, two columns c_i and c_j are compatible if for each row k, either $c_i(k) = -$, or $c_j(k) = -$, or $c_i(k) = c_j(k)$. For a completely specified function, compatibility is an equivalence relation and the set of vertices that are mutually compatible form an equivalence class. Hence the column multiplicity of the decomposition chart is the number of equivalence classes. For incompletely specified

functions, the compatibility relation is not an equivalence relation, i.e. there may be a case when $i \sim j \wedge j \sim k$, but $i \not\sim k$. So, a column may be contained in two or more compatible sets and a nontrivial procedure, called "minimum set covering" procedure, is needed for determining column multiplicity.

Since then, many more complicated functional decomposition models have been introduced that don't require either the bound set and the free set to be disjoint or the node G to have a single output. Recent research in the field also includes work on BDD-based methods aimed at improving the efficiency of decomposition [33, 34].

7.2 SPFDs and Decomposition

Before the connection between SPFDs and decomposition is explored, the connection between SPFDs and information content is briefly reviewed. An SPFD attached to a node specifies which pairs of primary input minterms have to be distinguished by the node. This can be thought of as the information content of the node, since it tells what information the node contributes to its surrounding network.

Example 7.1 *Consider the simple node shown in Figure 7.3. Input A has the ability to distinguish* 00 *and* 01 *from* 10 *and* 11. *Similarly, input B has the ability to distinguish* 00 *and* 10 *from* 01 *and* 11. *Thus, the two inputs together can distinguish every input minterm from every other input minterm. However, the output of the node only has the ability for distinguishing* 00 *from* 10, 01 *and* 11.

Any single-output node that depends on more than one input always results in a loss of information. Only a single-input single-output node (buffer or inverter) does not lose information. Also, an n-input n-output node, whose function is reversible (i.e. for each input combination there is exactly one output combination, and vice versa) does not lose information.

Now consider the problem of disjoint decomposition. Consider the example shown in Figure 7.1 and look at it in terms of information flow. The original function F required that the onset minterms have to be distinguished from the offset minterms. Each input of F does a part of the distinguishing job. Now, if F has to be re-implemented as the decomposed circuit shown in Figure 7.1, it is necessary that the new node G should be able to do all the distinguishing that the inputs in X_1 did for the function at F. In order to achieve this, consider the following algorithm, **com_decomp_w_spfd.**

Algorithm **com_decomp_w_spfd**(F, $X_1|X_2$):

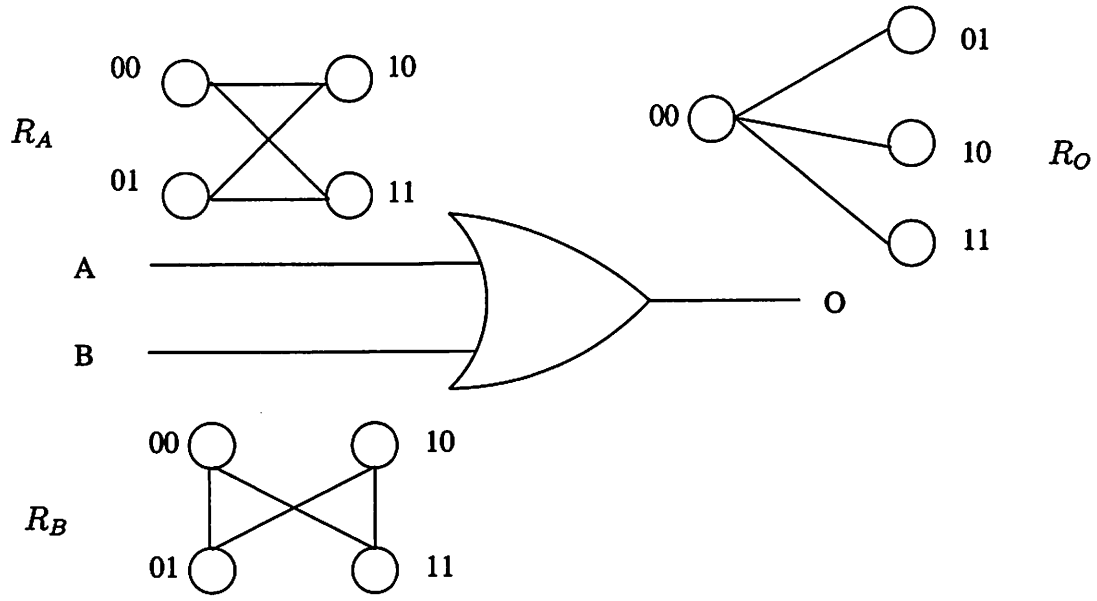


Figure 7.3: Information flow through an OR-gate: R_O is a subset of $R_{IN} = R_A \cup R_B$.

1. *Compute the SPFD of F in terms of the input space $X = X_1 \cup X_2$. Denote it as R_F.*

2. *Remove all edges of R_F that can be distinguished by the inputs in X_2. Denote this new SPFD as R'.*

3. *Existentially quantify out the variables associated with the inputs in X_2 from R' to get the SPFD of the node G. Denote this SPFD as R_G.*

The new function at G can be obtained by coloring R_G. Similarly, the new function at F can be obtained by expressing R_F in terms of $(x_g \cup X_2)$ and coloring it.

Theorem 7.2 *For a completely specified function F and a given partition $(X_1|X_2)$, any two minterms belong to the same compatible if and only if there exists a coloring of R_G such that that the two minterms can be colored with the same color.*

Proof $\rightarrow$: It is shown that all minterms in the same compatible can be colored with the same color in R_G. Assume that it is not true. Thus there exists two minterms m_1 and m_2 that belong to the same compatible but cannot be colored with the same color. This happens only if there exists an edge e between m_1 and m_2 in the SPFD R_G. This implies there exists a minterm $y \in B^t$ such that $F(m_1, y) \neq F(m_2, y)$. However, in that case m_1 and m_2 cannot belong to the same compatible.

$\leftarrow$: If two minterms m_1 and m_2 in the SPFD of R_G can be colored with the same color, then it implies that no edge exists between them. Therefore,

$$\forall_{y \in B^t}(F(m_1, y) = F(m_2, y)).$$

Thus, m_1 and m_2 belong to the same compatible. □

SPFDs can also be used for obtaining a non-disjoint decomposition. If a fanin x_i belongs only to X_2, then it has to be assigned to the SPFD of partition X_2. On the other hand, if a fanin x_i belongs to both X_1 and X_2, then it is possible to assign an edge distinguished by x_i to either the SPFD of partition X_1 or the SPFD of partition X_2. Assigning it to X_1 could increase the complexity of G whereas assigning it to X_2 could increase the complexity of H.

Given that SPFDs can be used for obtaining simple functional decompositions, an interesting decomposition scheme can be developed.

7.3 Topologically Constrained Decomposition Problem

A generalization of the decomposition idea to an arbitrary network of nodes is shown in Figure 7.4. Here, instead of specifying the free set and the bound set, the topology of the network is given i.e. the fanin and fanout connections of all the nodes in the network are provided. The problem is to determine the functionalities of the nodes so that the network implements the required output functions. The nodes in the network can have multiple outputs (or equivalently can be multi-valued). The configuration could be generated by a wireplanning algorithm, where the communication between the boxes is specified but the actual contents of each box is not specified.

In the rest of this chapter, the condition that the network topology has to satisfy in order to ensure that the network can be synthesized is provided, and a particular approach based on SPFDs is presented for synthesizing the nodes.

7.4 Problem Solution

7.4.1 Preliminaries

Definition 7.1 *A* **cut** *is a set of nodes in the network that when removed completely isolates the primary inputs from the primary outputs.*

Obviously, a network can have many cuts.

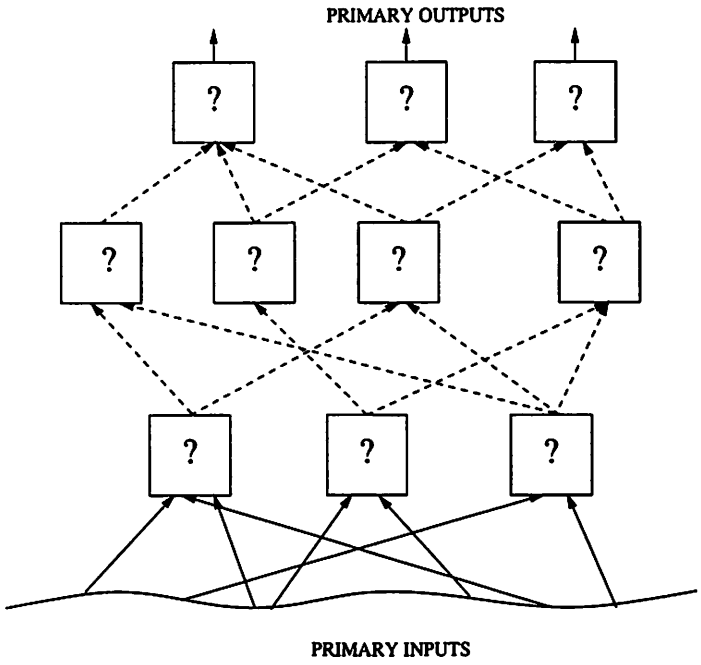


Figure 7.4: Problem definition.

Consider a node η_j in network $\mathcal{N}$. Let L_j denote its level in the network. R_j^{max} denotes the maximum SPFD of η_j. The SPFD of η_j that is used for deriving its function is denoted as R_j. The synthesized function at η_j is denoted as f_j. R_j is expressed either in terms of the $(Y_j \cup Y_j')$ space or the $(X \cup X')$ space.

7.4.2 Algorithm

In this algorithm, the analogy of information flow through the network is used. The network specification specifies what information needs to be passed on from the primary inputs to the primary outputs. As mentioned earlier, SPFDs can be used for denoting the information content of a node. So the network function specification can be thought of as the information content of the primary outputs and can be re-expressed as SPFDs associated with the primary outputs. Similarly, the information content of the primary inputs can be expressed as SPFDs associated with the primary inputs. It is instructive to think of SPFDs of the primary outputs as the **required information** and the SPFDs of the primary inputs as the **available information** (as shown in Figure 7.5). The task of the synthesis process is to determine the information flow through the nodes in the network so that the required information is present at the primary outputs. A network can be thought of as

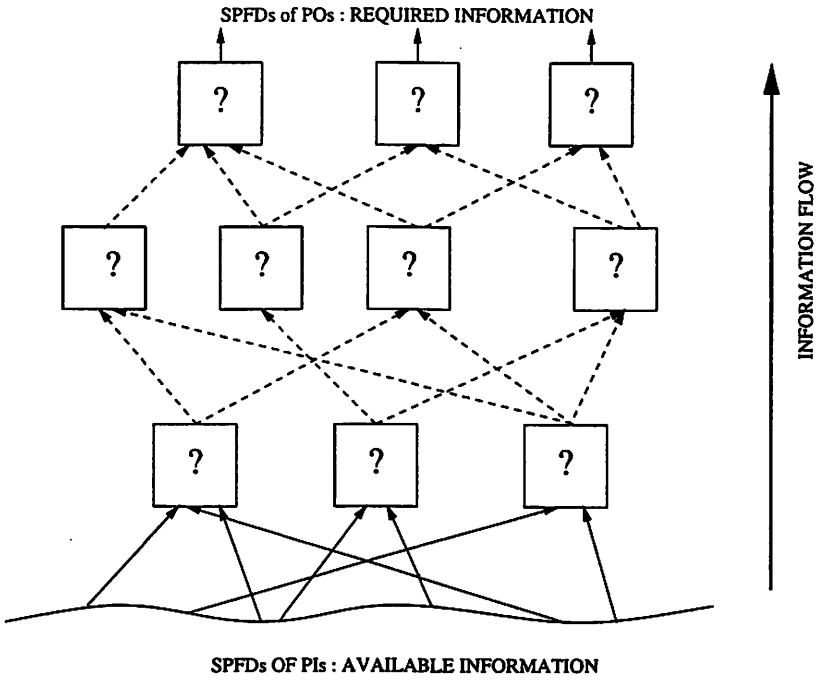


Figure 7.5: Information flows through the network.

a lossy information channel. For this method to work, it is necessary to ensure that the available information is not less than the required information. This translates into a topology constraint given in the following lemma.

Lemma 7.1 *The network $\mathcal{N}$ of empty nodes has to satisfy the following requirement : each primary output should have at least one path to each primary input in its true support.*

Proof Let x_i be a primary input in the true support of a primary output z_k. It is proved by contradiction that at least one path has to exist between x_i and z_k for ensuring correct functionality.

Assume no path exists between x_i and z_k in $\mathcal{N}$. Since x_i is in the true support of z_k, $\partial z_k/\partial x_i$ is not empty. Hence there exists at least one pair of minterms $m_1 = x_i m$ and $m_2 = \overline{x_i} m$ such that m_1 belongs to the onset of z_k and m_2 belongs to the offset of z_k. Note that m_1 and m_2 differ only in the value of x_i. According to the above assumption, the primary input x_i does not lie in the transitive fanin of z_k (since no path exists between x_i and z_k). Thus, the values of the primary inputs in the transitive fanin of z_k are identical for both m_1 and m_2 and is equal to m. Since m_1 and m_2 belong to the onset and offset of z_k, respectively, this implies m has to produce different values at the output of z_k for ensuring correct functionality. This is impossible since z_k implements a deterministic function. Thus the assumption that no path exists between x_i and z_k is incorrect. □

Note that there are many topologies that can satisfy Lemma 7.1. For example, a two level network that expresses a primary output solely in terms of the primary inputs is also a network topology that satisfies Lemma 7.1. This lemma is not useful for generating any multi-level network topologies. Interesting topologies can be generated using the concept of *partial don't care* wires, briefly described in Section 6.5. However, this is beyond the scope of this dissertation and will not be discussed further here. The condition in Lemma 7.1, however, ensures that each edge in the SPFD of a primary output is contained in the SPFDs of one or more primary inputs in its transitive fanin. This way of framing the problem in terms of SPFDs enables us to utilize some of the familiar techniques of SPFD manipulation for determining the flow of information through the network from the primary inputs to the primary outputs.

Several other papers [35, 36] exploit the connection between information flow and synthesis. In the first paper, an evolutionary approach towards network synthesis is used, where a function is corrected by adding either a few constants or variables until it becomes the specified function. The algorithm presented in the second paper uses a function expressed in terms of its primary inputs as its starting point and progressively decomposes the function at each step until some user-defined limit like the number of fanins of each node is reached. This method looks at the information content of each node for determining a function of the fanins. At each node, either a serial or parallel decomposition is allowed. However, this method does not use a fixed network topology.

The basic idea of the algorithm is to ensure that the information necessary to meet the network specification is not lost on the way from the primary inputs to the primary outputs. It accomplishes this by defining a set of cuts in the network starting from the primary inputs and moving towards the primary outputs, and ensuring that each cut has the necessary information. The general flow of the algorithm is shown in Figure 7.6.

There are two basic steps in the algorithm:

1. Defining the cuts in the network.

2. Computing the SPFDs of the nodes in the cut and synthesizing these nodes using their respective SPFDs.

Many schemes could be used for either of two steps; one scheme for each step is proposed.

7.4.3 Defining the Cuts in the Network

The procedure goes as follows:

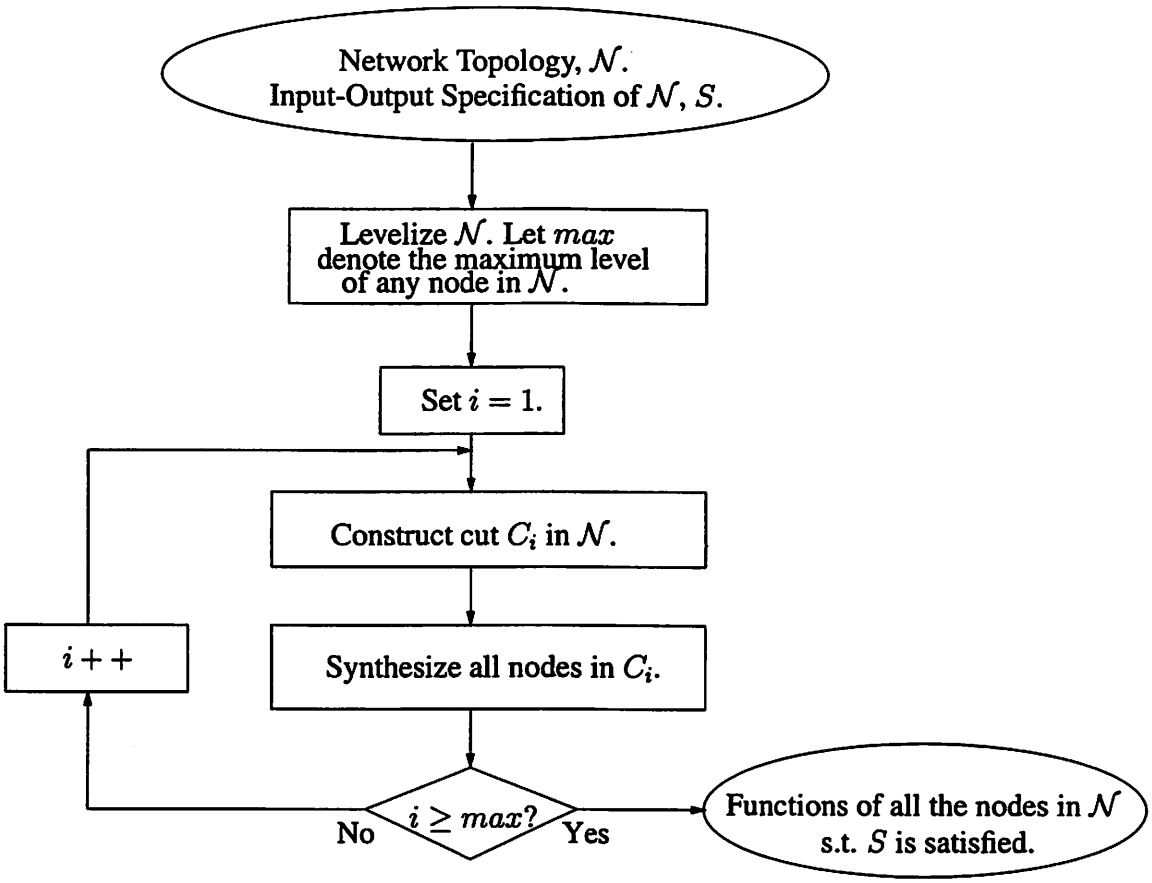


Figure 7.6: Algorithm for topologically constrained decomposition problem.

1. *Levelize all the nodes in the network starting from the primary inputs. For each primary input* η_i, $L(\eta_i) = 0$. *For any other node* $\eta_i \in \mathcal{N}$,

$$L(\eta_i) = max\{L(\eta_j) : (\eta_j \in FI(\eta_i))\} + 1.$$

2. *Let* **max** *denote the maximum level of any node in the network. Define the cuts in the network starting from the primary inputs to the primary outputs. So, for* $i = 0, \cdots,$ **max**,

   (a) *For each primary output* z_k, *compute* C_{ik} *to include*

      i. *All nodes in the transitive fanin of* z_k *with level* $= i$

      ii. *All nodes with level* $< i$ *that directly fanout to a node with level* $> i$ *in the transitive fanin of* z_k.

Thus,

$$C_{ik} = \{\eta_j | (\eta_j \in TFI(z_k)) \wedge (L(\eta_j) = i)\} \cup \{\eta_j | (L(\eta_j) < i) \wedge [\exists_{\eta_p}(L(\eta_p) > i) \wedge (\eta_p \in FO(\eta_j)) \wedge (\eta_p \in TFI(z_k))]\}.$$

(b) Construct $C_i = \cup_k C_{ik}$. Thus C_i includes all nodes in $\mathcal{N}$ with level $= i$ and all nodes with level $< i$ that directly fanout to a node with level $> i$. Note that C_i is a cut in $\mathcal{N}$ as removing these nodes will completely disconnect the primary inputs from the primary outputs. C_{ik} denotes the subset of nodes of C_i which provides all the information to a primary output z_k.

C_0 consists of the primary inputs of $\mathcal{N}$. A node η_j definitely appears in cut $C_{L(\eta_j)}$. Furthermore, let $lmax = max\{L(\eta_k) | \eta_k \in FO(\eta_j)\}$. Then η_j also appears in a cut C_i, where $L(\eta_j) < i < lmax$. Thus two cuts in $\mathcal{N}$ can share some nodes.

7.4.4 Synthesizing the nodes in the cut

Here, the algorithm for synthesizing the nodes in a particular cut C_i is described. The main requirement that has to be satisfied after the synthesis step is that for each primary output z_k, C_{ik} must be able to provide all the information that z_k requires. In the rest of the section, the algorithm that ensures that this condition is satisfied is presented.

The cuts are synthesized from the primary inputs to the primary outputs. Hence when the nodes in C_i are being synthesized, all the nodes in cuts $C_1, \cdots, C_{i-1}$ have already been synthesized. The nodes in C_i that have already been synthesized are denoted as C_i^r. These are the nodes of level$< i$. The nodes in C_i with level $= i$ have to be synthesized and are denoted as C_i^u. Note that $C_i = C_i^u \cup C_i^r$.

The algorithm **syn_cuts** first orders the nodes in C_i according to some heuristic such that that all the nodes in C_i^r are earlier in the ordering than all the nodes in C_i^u. It then computes the maximum SPFD of each node in C_i^u. This maximum SPFD denotes the total set of edges that a node can distinguish, derived solely from the distinguishing ability of its fanins. The SPFD computation then proceeds from the nodes earlier in the ordering to the ones later in the ordering. At each node η_j, its SPFD R_j is derived from its maximum SPFD as follows: For each primary output $z_k \in PO(\eta_j)$, the algorithm determines the edges in $R_k(X, X')$ that cannot be distinguished by the remaining nodes in C_{ik} (C_{ik} is the subset of nodes of C_i that provide all the information to z_k). The

node's SPFD R_j is simply the union of all these edges. The new function at η_j is derived from R_j. Then the algorithm moves to the next node in the cut.

Algorithm **syn_cuts**(C_i):

1. *Assume that each primary output z_k has an SPFD $R_k(X, X')$ associated with it.*

2. *Order the nodes in C_i. All the nodes in C_i^r should be earlier in the ordering than all the nodes in C_i^u.*

3. *For each node $\eta_j \in C_i^u$, compute the maximum SPFD of the node and denote it as R_j^{max}.*

$$R_j^{max}(X, X') = (\cup_{\eta_p \in FI(\eta_j)} R_p(X, X')),$$

where $R_p(X, X')$ is the SPFD of η_p expressed in terms of the primary input space[1]*. Thus $R_j^{max}(X, X')$ denotes the maximum set of edges that η_j can distinguish. However, if all the edges in R_j^{max} are assigned to R_j, a lot of information will be duplicated in the network. Hence the amount of redundant information in R_j is minimized in the next step.*

4. *Process the nodes in C_i^u in order, starting from the one earliest in the ordering. For each node $\eta_j \in C_i^u$, do the following:*

   (a) *For each $z_k \in PO(\eta_j)$,*

      i. *Determine the edges in the SPFD of z_k that can only be distinguished by η_j according to the ordering computed in Step 2. Hence, from*

$$R_{jk}(X, X') = R_j^{max}(X, X') \wedge R_k(X, X'),$$

      A. *Remove the edges that* **are** *distinguished by the SPFDs of the nodes in C_{ik} that are earlier in the ordering. Thus, for each $\eta_n < \eta_j$,*

$$R_{jk}(X, X') \leftarrow R_{jk}(X, X') \wedge \overline{R_n(X, X')}.$$

      B. *Remove the edges that* **can be** *distinguished by the nodes in C_{ik} that are later in the ordering. Thus, for each $\eta_m > \eta_j$,*

$$R_{jk}(X, X') \leftarrow R_{jk}(X, X') \wedge \overline{R_m^{max}(X, X')}.$$

   (b) $R_j(X, X') = \cup_{i=1}^{n} R_{ji}(X, X')$, *where* $n = |PO(\eta_j)|$.

[1] Since η_p is a fanin of η_j, hence it has already been synthesized and has an SPFD associated with it.

(c) *Compute*

$$R_j(Y_j, Y_j') = \exists_{X,X'} \mathcal{G}(X, Y_j)\mathcal{G}(X', Y_j')R_j(X, X').$$

This is the image of $R_j(X, X')$ to the local input space of η_j.

(d) *Determine the new function at η_j by coloring $R_j(Y_j, Y_j')$ and minimizing the resulting ISF using ESPRESSO-MV. Let this new function be f_j. Note that f_j can be multi-valued, in general.*

5. *Stop.*

7.4.4.1 Global SPFDs vs Local SPFDs

In all the above computations, the SPFDs were expressed in terms of the primary inputs (global SPFDs) instead of the local inputs (local SPFDs). While computations of global SPFDs can be fairly memory intensive, the disadvantages of expressing the SPFDs of the nodes in terms of the local fanin space are two-fold:

1. Expressing the SPFD in terms of the local space can add some extra useless edges. For instance, suppose the primary input edge (x, x') produces the edge (y, y') in the local fanin space of η_j. Now, if the inverse image of (y, y') is computed back to the primary input space, then in addition to (x, x'), a few more edges may be obtained. Hence, expressing R_j^{max} in terms of the local inputs could add some useless edges. This, in turn, may result in some useless edges in the SPFD R_j that is used for deriving the new function at η_j.

2. Translating the SPFD from one local space to another also results in some loss of precision due to early existential quantification. Thus, suppose it is necessary to remove the edges in the SPFD R_p of η_p from the SPFD R_j^{max}. The current algorithm would do the following (as shown in Step 4(a)(i)(A)):

$$R_j(Y_j, Y_j') = \exists_{X,X'}(R_j^{max}(X, X')\overline{R_p(X, X')})\mathcal{G}(X, Y_j)\mathcal{G}(X, Y_j').$$

On the other hand, if all the SPFDs were expressed in terms of the local fanin spaces, the computation would be the following:

$$R_j(Y_j, Y_j') = R_j^{max}(Y_j, Y_j') \wedge (\exists_{Y_p,Y_p'}\overline{R_p(Y_p, Y_p')}En(Y_j, Y_p)En(Y_j', Y_p')),$$

where $En(Y_j, Y_p) = \exists_X \mathcal{G}(X, Y_j)\mathcal{G}(X, Y_p)$. So in the second equation, first the quantification is done and then the conjunction. This could result in some additional edges. This is particularly the case if the nodes do not share any primary inputs as then $En(Y_j, Y_p) = 1$.

In practice, these disadvantages were indeed operative. Hence all the computations are performed on global SPFDs.

7.4.5 Correctness

Lemma 7.2 *Given a primary output z_k, let C^r_{ik} and C^u_{ik} denote the synthesized and unsynthesized nodes of C_{ik}, respectively. Then,*

$$C_{(i-1)k} = C^r_{ik} \cup C^{add},$$

where $C^{add} = \{\eta_p | (\eta_p \in FI(\eta_j)) \wedge (\eta_j \in C^u_{ik})\}$.

Proof $\rightarrow : C_{(i-1)k} \subseteq C^r_{ik} \cup C^{add}$:
Consider a node, $\eta_p \in C_{(i-1)k}$. Either η_p fans out to at least one node in the transitive fanin of z_k of level $> i$ or else the maximum level of its fanouts in the transitive fanin of z_k is $= i$. In the first case, it belongs to C^r_{ik}. In the second case, it belongs to the C^{add}.

$\leftarrow : C^r_{ik} \cup C^{add} \subseteq C_{(i-1)k}$:
Any node $\eta_p \in C^r_{ik}$ has level $< i$ and fans out to at least one node of level $> i$ in the transitive fanin of z_k. Thus $\eta_p \in C_{(i-1)k}$. Consider a node $\eta_p \in C^{add}$. Let it be the fanin of a node η_j in C^u_{ik}. Note that $\eta_j \in TFI(z_k)$. This is because $\eta_j \in C^u_{ik}$ and $L(\eta_j) = i$. Moreover, since $L(\eta_j) = i$, hence $L(\eta_p) \leq (i-1)$. Two cases must be distinguished:

1. $L(\eta_p) = (i-1)$: Since $\eta_j \in TFI(z_k)$ and $\eta_p \in FI(\eta_j)$, hence η_p is a transitive fanin of primary output z_k. Hence $\eta_p \in C_{(i-1)k}$.

2. $L(\eta_p) < (i-1)$: Since η_p fans out to η_j(whose level $= i$), hence it fans out to η_j with level $> (i-1)$. Also, $\eta_j \in TFI(z_k)$. Thus $\eta_p \in C_{(i-1)k}$.

□

Theorem 7.3 *If the topology constraint given by Lemma 7.1 is satisfied, each primary output z_k can always be synthesized to satisfy its network specification. The internal nodes in the network can be multi-valued after synthesis.*

Proof The above is proved for an arbitrary primary output z_k.
Base case: The topology constraint ensures that C_{0k} has all the information that z_k requires.
Inductive step: Suppose C_{ik} has all the information required by z_k. It is proved that the algorithm **syn_cuts** ensures that $C_{(i+1)k}$ will have all the information that z_k requires.

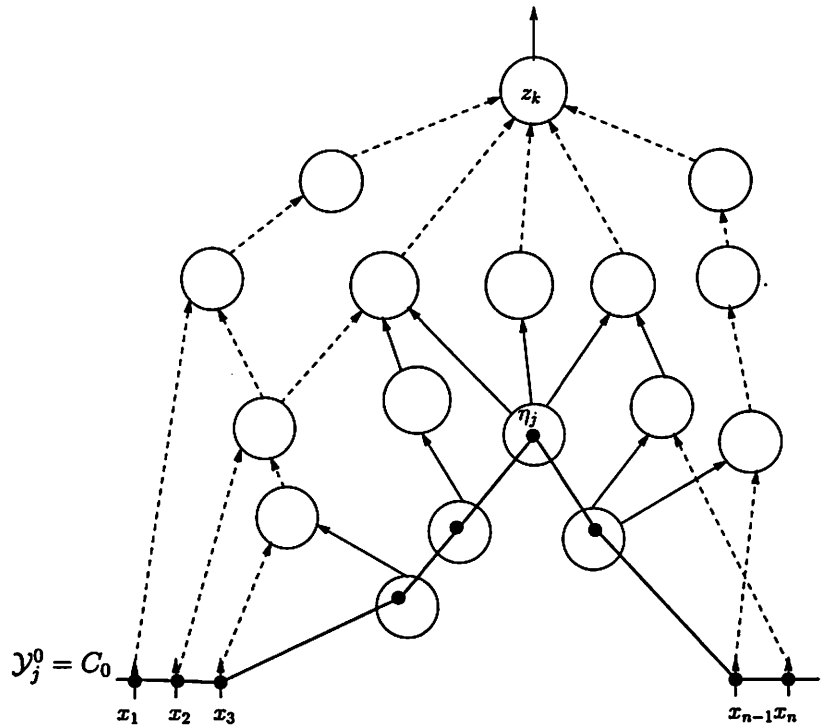


Figure 7.7: After η_j is simplified using its minimum SPFD, the nodes of the above modified network $\mathcal{N}'$ can be synthesized using **syn_spfd**.

Assume that's not true. Then there exists an edge $e = (x, x') \in R_k(X, X')$ that cannot be distinguished after synthesizing the nodes in $C_{(i+1)k}$. This happens only if e is not in the SPFDs of the nodes in $C^r_{(i+1)k}$ or in the maximum SPFD of the nodes in $C^u_{(i+1)k}$. Since the maximum SPFD of a node is simply the union of the SPFDs of its fanin nodes, e does not belong to the SPFDs of any of the fanin nodes of $C^u_{(i+1)k}$. But the fanins of the nodes in $C^u_{(i+1)k}$ together with the nodes in $C^r_{(i+1)k}$ form the nodes in C_{ik} (Lemma 7.2). Thus e does not belong to the SPFDs of the nodes in C_{ik}. But this contradicts the assumption that e can be distinguished by the nodes in C_{ik}. □

The entire algorithm (comprised of defining the cuts in a network and synthesizing the nodes in each cut using **syn_cuts**) is referred to as **syn_spfd**.

7.5 Connections with minimum SPFD

The ideas presented in the previous section also support the claim that R_j^0 computed using $\mathcal{Y}_j^0$ (shown in Figure 4.4) in **com_minspfd_for_sep** is indeed the minimum SPFD of η_j. In Theorem 4.2, it was proved that all the minterms that have an edge between them in R_j^0 have to

circuits	original	syn_spfd	% MV-nodes
apex7	292	278	9.09
cht	236	199	0
cmb	62	76	7.14
cc	99	102	0
cu	90	88	0
f51m	195	194	0
lal	224	223	8.92
ttt2	339	292	8.51
term1	625	341	24.07
x2	71	53	11.11
Average	0	-8.27	6.88

Table 7.1: Results of using **syn_spfd** on ISCAS benchmark circuits.

be distinguished at the output of η_j, after η_j has been simplified. Here, the reverse claim is made. Thus if all the minterms that don't appear in R_j^0 are assigned the same value at the output of the η_j after simplification, correct functionality at the primary outputs can still be guaranteed by simply modifying the nodes between $\mathcal{Y}_j^0$ and the primary outputs. This is because $\mathcal{Y}_j^0$ is a cut in the network and it has all the information required by the primary outputs even after simplifying η_j using R_j^0. Thus, R_j^0 is indeed the minimum SPFD of η_j.

The algorithm presented in the previous section can be applied for resynthesizing the nodes with some minor modifications. The modifications arise in the definition of the cuts. Here, the separator $\mathcal{Y}_j^0$ should be treated as the cut C_0. Applying the algorithm presented in the previous section on the modified network[2] $\mathcal{N}'$ (shown in Figure 7.7) determines the functionalities of all the nodes in between $\mathcal{Y}_j^0$ and the primary outputs of the original network. For instance, for the circuit in Figure 4.5, after setting y to zero, the new function at z is given as $z = x_1x_2 + \overline{x_2x_3}$.

This scheme can also be used when the minimum SPFD of η_j is computed using any of the other separators, say $\mathcal{Y}_j^1$, shown in Figure 4.4.

7.6 Experiments

In this section, some experiments that were performed for determining the practical feasibility of the above scheme are described. As mentioned before, the algorithm **syn_spfd** needs a

[2] All the nodes in the transitive fanin of η_j in the original network except the ones in $\mathcal{Y}_j^0$ are removed.

circuits	script.rugged	**syn_spfd**	% MV-nodes	simplify
apex7	246	260	3.03	254
cht	165	162	9.75	162
cmb	51	58	16.67	53
cc	63	64	0	64
cu	60	62	0	60
f51m	119	115	8.33	115
lal	106	107	0	107
ttt2	219	252	23.4	236
term1	176	163	6.67	157
x2	48	49	0	49
Average	0	2.99	6.78	0.36

Table 7.2: Results of using **syn_spfd** on optimized ISCAS benchmark circuits.

network topology and an input-output specifications as its starting point. For this work, circuits from the ISCAS benchmark suite (or their derivatives) were used and their topology information and input-output specification served as the starting point. The experiments were set up to test if the procedures are valid and if they can closely reproduce the original circuit.

In the first set of experiments, the initial topology of a given ISCAS benchmark circuit and its input-output specification was used as the starting point. Thus, given the topology of the original circuit, **syn_spfd** was used for synthesizing the nodes in these networks. The initial results are shown in Table 7.1. Columns 2 and 3 show the literal counts of the original circuit and the circuit after using **syn_spfd**. An average improvement of 8.27% in literal count was obtained after using **syn_spfd**[3]. One negative artifact of the greedy edge distribution scheme used in **syn_cuts** is that some of the nodes may be multi-valued [4]. This is because a node that appears later in the ordering in a cut may have to distinguish many edges and its SPFD graph may no longer be bipartite. Practical results, however, indicate that on average only 6.88% of the nodes were multi-valued (Column 4).

Table 7.2 provides results of using **syn_spfd** on optimized ISCAS benchmark circuits. In this experiment, a circuit was optimized using *script.rugged* and the topology of the optimized circuit was used as the starting point of **syn_spfd**. The input-output specification was the functionality of the original circuit. This experiment was set up to test if the algorithm works under tighter

[3]This improvement is because many of the original circuits are not optimized and the synthesis procedure has the opportunity of removing some redundancies.

[4]Note that a solution exists for the given starting topology and the input-output specification in which all the nodes are binary. This solution is the original circuit.

topology constraints. The results indicate the optimized circuit can be reproduced quite closely. Even though, the results were worse than the original optimized circuit in some cases, the average increase in literal count was only about 3%. The average percentage of nodes that were multi-valued is 6.78%. The results of Table 7.2 should be viewed from the point of view of a real application. In a real application, a solution is not available. Only the topology and the input-output specifications of the network will be provided. There would be no way for judging the quality of the solution returned by **syn_spfd**. Table 7.2 supports the claim that the solution is pretty good. In addition, network optimization methods that do not alter the topology like **full_simplify** can be applied for improving the solution. The results of running **full_simplify** [37] is shown in Column 5 in Table 7.2. Thus essentially the heavily starting optimized circuit can be recovered.

The preliminary results indicate that it is possible to use the algorithm described in this chapter for synthesizing the nodes in a network from its topology and its input-output specification. In the experiments with the topologies of the unoptimized circuits, there was also a reduction in the literal count with respect to the original circuit. Also, in practice, it was found that on average less than 7% of the nodes were multi-valued in both the optimized and unoptimized topologies.

One problem with this approach is non-robustness. This is mainly due to the large memory requirements of the global SPFDs.

7.7 Summary

In this chapter, the synthesis process for "topologically constrained decomposition" was presented. The initial results were quite encouraging. One problem was the memory usage of the global SPFDs. There are a few approaches for dealing with this problem. It may be beneficial from the memory perspective to represent the global SPFDs as asymmetric relations. Another possible help in this direction is the use of SAT as described in Chapter 5. Also, instead of working on an entire network, this algorithm could be applied to portions of a partitioned network.

The *partial don't care* wires introduced in the previous chapter could be used for obtaining the initial network topology. This algorithm could also be useful in wireplanning scenarios where the interconnect structure is planned out before the node functionalities are decided.

Chapter 8

Sequential SPFDs

In this chapter, the concept of sequential SPFDs is introduced. First, an example is provided for illustrating how sequential SPFDs can be used to reduce the number of state bits. Then a general procedure is provided for state reencoding using sequential SPFDs and the correctness of the algorithm is also established. A procedure for resynthesizing the circuit using the newly derived state encoding is also described.

8.1 Previous Work

The classical computation of equivalence classes of states for FSMs was introduced in [38]. There has been a whole body of work on the minimization of FSMs (c.f. [39]). Most of these approaches suffer state space explosion. In addition, the benefits of state minimization do not necessarily translate to the final implementation of the sequential circuit. Approaches based on structural techniques try to solve this problem by working directly on the sequential circuit. The circuit structure is used to extract the set of unreachable states which are later used as *don't cares* for circuit optimization ([40], [41]). However, these approaches still have to represent the entire state space and can potentially run into the state space explosion problem. To cope with this problem, local transformation techniques such as ATPG-based methods [42] and retiming and resynthesis [43] have been used. These are currently the most widely used techniques, but were designed with efficiency as the main consideration. As a result, sequential freedom is not completely explored, due to limited time-frame expansion. The approach presented here avoids the state space explosion problem by using a partition of the state space while exploring more sequential freedom.

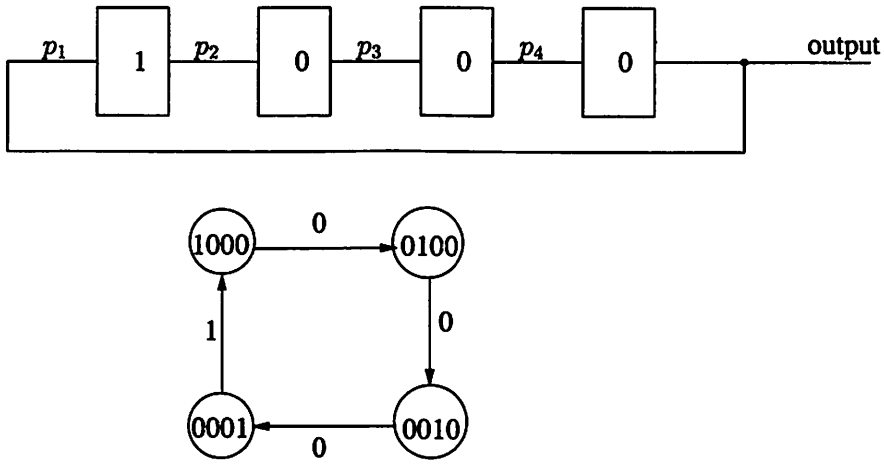


Figure 8.1: Example sequential circuit.

8.2 Motivating Example

Example 8.1 *Figure 8.1 gives a simple example of a sequential circuit and its corresponding State Transition Graph (STG). It consists of four latches connected in series to form a shift register. The output of the fourth register is the only primary output of the circuit. The initial value of the first register is 1, the others 0. This circuit is sequentially redundant and could be implemented with two registers only.*

Combinational optimization treats the register inputs as primary outputs and the register outputs as primary inputs, and optimizes the combinational network between these boundaries. For the example in Figure 8.1, the resulting combinational network is shown in Figure 8.2. Clearly, combinational optimization techniques based on CODCs or SPFDs will not produce any circuit reduction.

Another way to apply combinational optimization techniques to a sequential circuit is to ignore the register inputs of the circuit. Thus, the circuit used during combinational optimization will have the primary inputs plus register outputs of the sequential circuit as its primary inputs, and the primary outputs of the sequential circuit as its primary outputs. However, the register outputs are constrained to combinations that correspond to states that are reachable. Using this approach, the example of Figure 8.1 yields the combinational optimization problem shown in Figure 8.3.

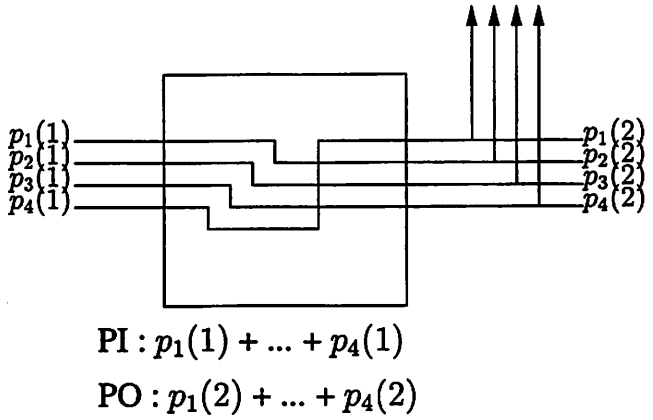


Figure 8.2: A combinational circuit derived from the sequential circuit in Example 8.1.

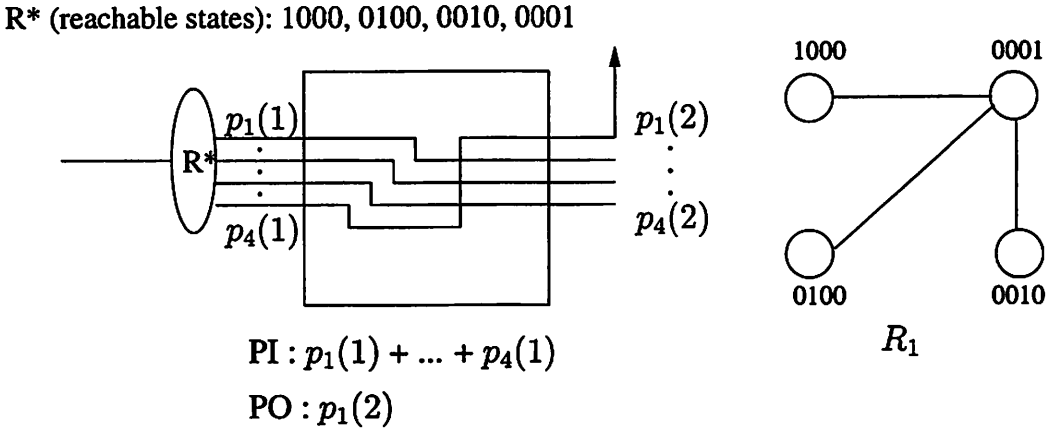


Figure 8.3: Another combinational circuit derived from the sequential circuit in Example 8.1.

Applying SPFDs on this combinational circuit, the SPFD of the primary output, which is the same as for $p_1(2)$, requires that all minterms that produce a 1 have to be distinguished from all the minterms that produce a 0. Its SPFD in terms of the present state bits $(p_1(1), p_2(1), p_3(1), p_4(1))$, denoted R_1, is shown in Figure 8.3: the minterm (0001) must be distinguished from the minterms (1000), (0100) and (0010). The SPFDs of the remaining state bits are empty. Thus, the union of the SPFDs of all state bits yields R_1. These are exactly the state pairs that produce different outputs in one transition. Thus SPFDs can provide information about the transitions of a sequential circuit, but it is not sufficient to just capture the information about one time frame. Informally speaking, it is necessary to unroll the circuit multiple times and determine the SPFDs at each node in a sequential circuit by computing the union of the SPFDs of the node in all time frames. These are called

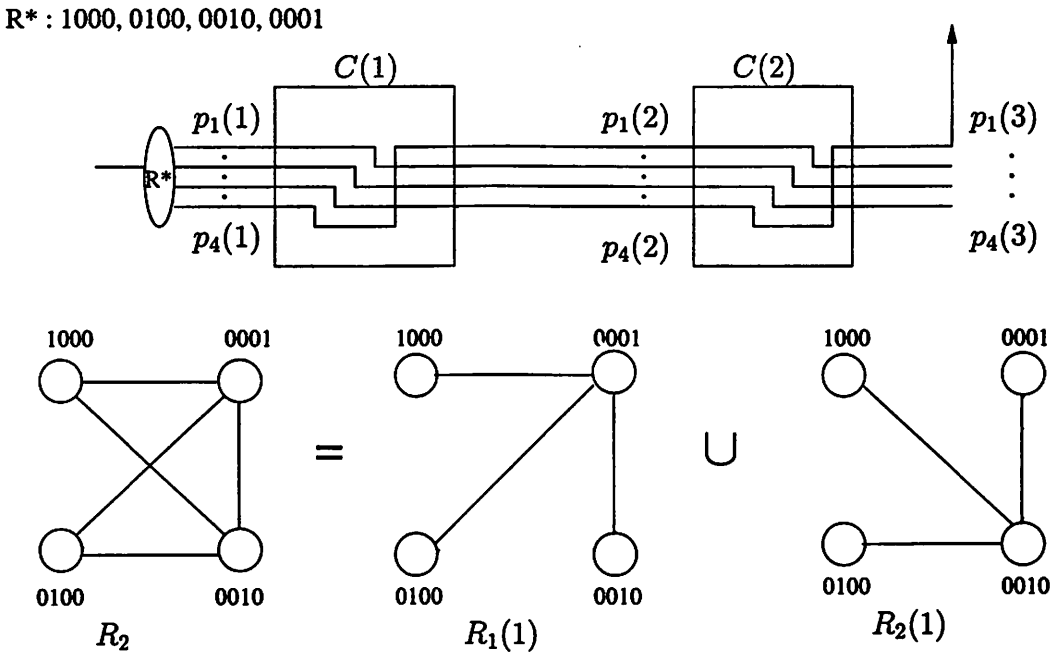


Figure 8.4: SPFDs obtained after unrolling once.

sequential SPFDs.

8.2.1 Sequential SPFDs

Consider a single unrolling of the circuit in Figure 8.1 which yields the combinational optimization problem shown in Figure 8.4. Denote the first and second copies by $C(1)$ and $C(2)$, respectively. The resulting combinational circuit has one primary output $p_1(3)$ and four primary inputs, $p_1(1)$, $p_2(1)$, $p_3(1)$ and $p_4(1)$.

Computing the SPFDs of all the nodes in the circuit and expressing the union of the SPFDs of the present state bits of $C(2)$ and $C(1)$ in terms of the present state bits of $C(2)$ and $C(1)$, respectively yields $R_1(1)$ and $R_2(1)$ as shown in Figure 8.4. $R_1(1)$ is exactly the same as R_1 in Figure 8.3. $R_2(1)$ denotes the state pairs that produce different outputs after exactly two transitions. Hence the union of the two SPFDs, $R_2 = R_2(1) + R_1(1)$, gives all those state pairs that produce different outputs in one or two transitions. Unrolling the circuit once more and computing the SPFDs of all three copies gives R_3 shown in Figure 8.5(a). It has an edge between any two states that can produce different outputs in one, two or three transitions.

Unrolling the circuit any further produces no additional edges. Thus R_3 includes all pairs of states that must be distinguished. If a pair of states s and s' has no edge between them, then the

R*:1000, 0100, 0010, 0001

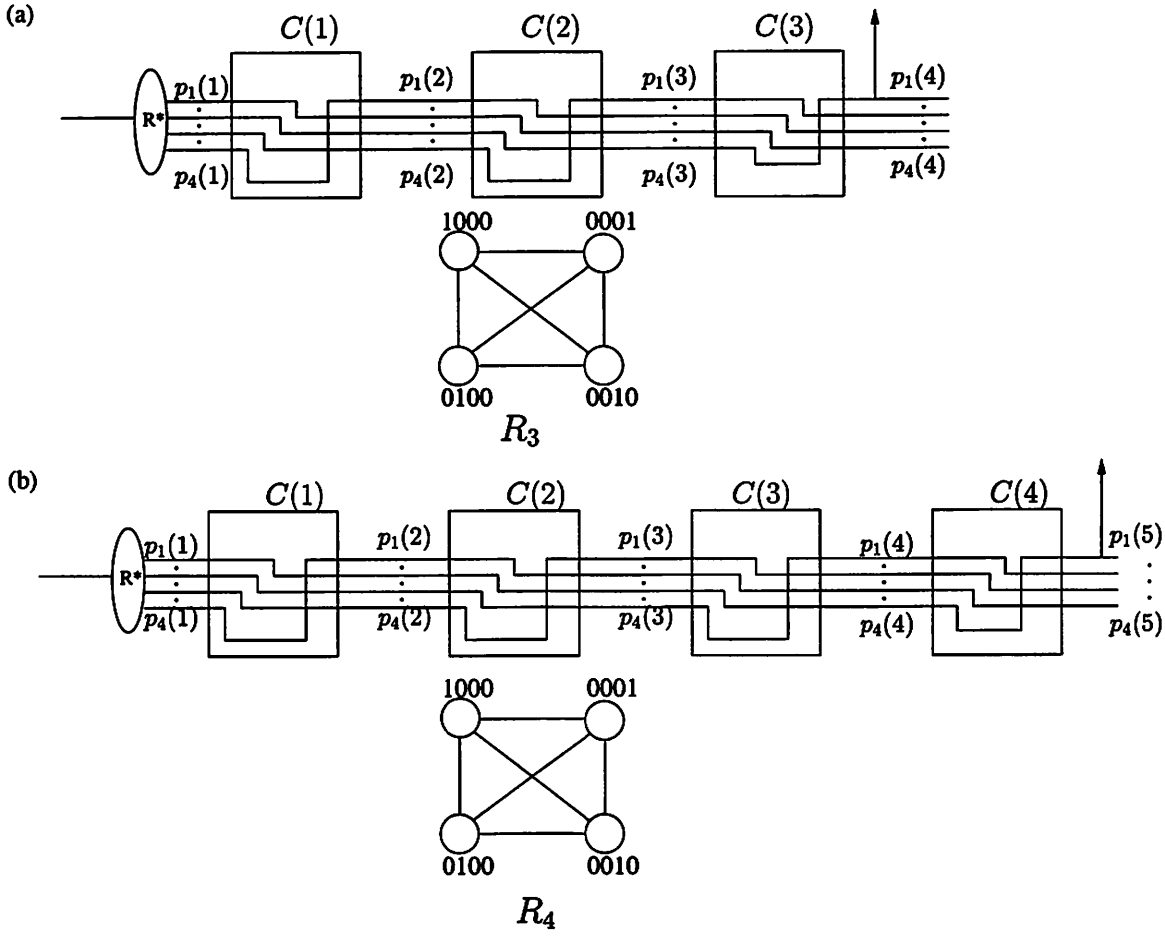


Figure 8.5: Various levels of unrolling and the corresponding SPFDs.

sequential circuit behaves identically, irrespective of whether it starts from s or s'. Hence these two states could be merged. The graph R_3 can be colored to obtain equivalence classes for the states. Four colors are needed and hence two state bits are required to implement the circuit.

This example illustrates how progressive unrolling adds edges between state pairs (s, s') that behave differently in the future. In this particular example, since all states behave differently, no additional information over the fact that the set of reachable states has four states and can be colored with four colors is gained.

The next example illustrates how sequential SPFDs can provide useful information that cannot be gained just by examining the set of reachable states.

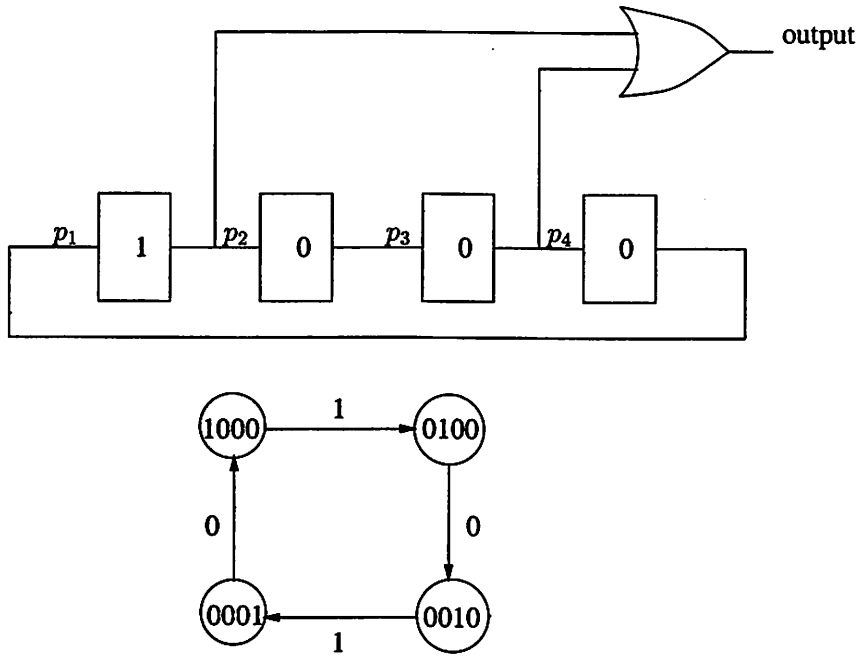


Figure 8.6: Another example sequential circuit.

Example 8.2 *Consider the circuit in Figure 8.6. It is similar to that in Figure 8.1 except that the primary output of the circuit is now the OR of the first and third register outputs. The results for no unrolling and one unrolling are shown in Figure 8.7 and are denoted* R_1 *and* R_2 *respectively.* R_1 *and* R_2 *denote the state pairs that produce different outputs in one and two transitions, respectively. Here* $R_1 = R_2$*, so the unrolling process is stopped. Unrolling the circuit any further does not produce any more state pairs that behave differently in the future. Since* R_1 *is bipartite, only one state bit is required to implement the circuit.*

Thus, SPFDs can give useful relations between states which can be exploited for deriving a new state encoding. In the following section, a general procedure which uses SPFDs for re-encoding the state space is provided. The correctness of the procedure is also established.

8.3 Sequential SPFD Computation

8.3.1 Additional Notation

For a sequential circuit M, denote the set of states by S, the set of transitions by T, the present state bits by P, and next state bits by P'. Let $p_i \in P$ denote a present state bit of M and

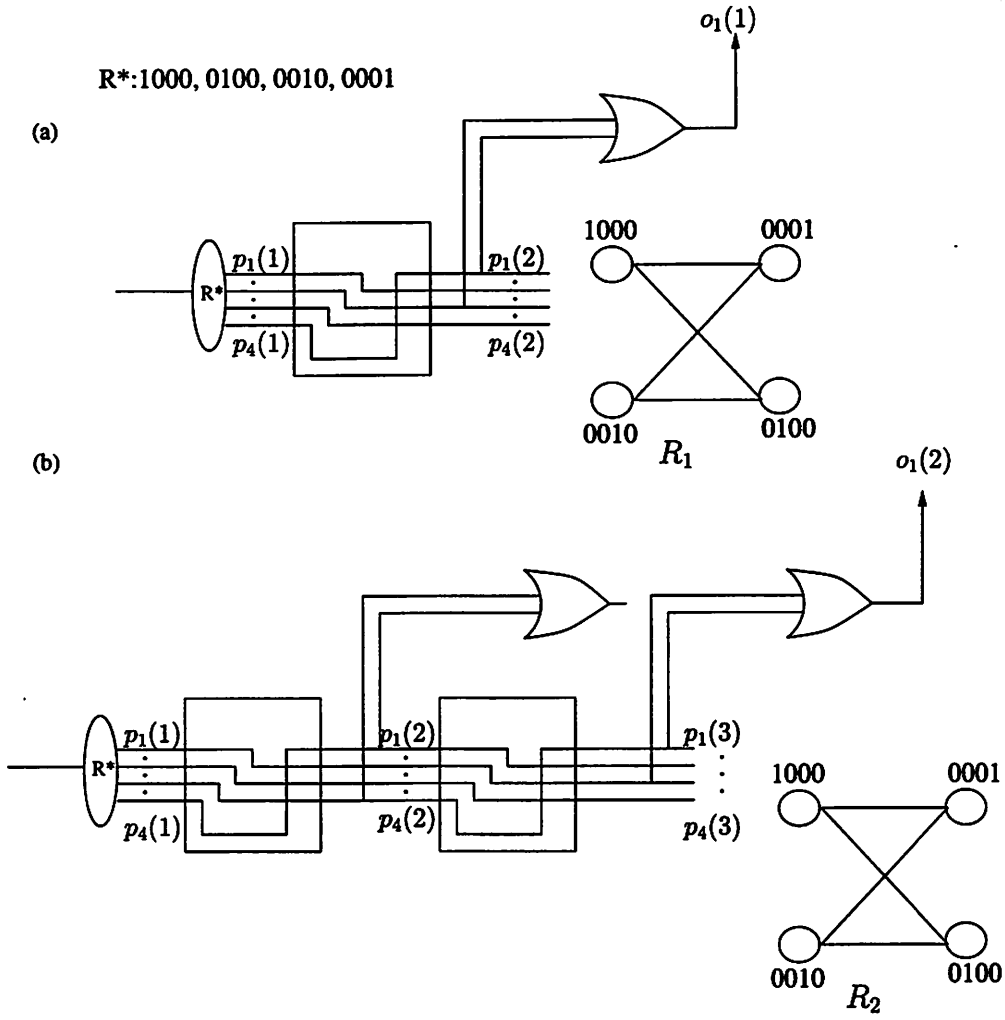


Figure 8.7: R_1 and R_2.

$p_i' \in P'$ denote the next state bit corresponding to p_i. Let $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2, \cdots, \mathcal{P}_k\}$, denote a partition of P, where each $\mathcal{P}_i$ is an individual part of $\mathcal{P}$. Each node $\eta_j \in M$ has a sequential SPFD R_j^s associated with it.

Let C be the combinational circuit obtained from M where its primary inputs are the primary inputs plus the present state inputs of M, and primary outputs are the primary outputs and the next state outputs of M.

8.3.2 Algorithm

The algorithm **com_seq_spfds** starts with the combinational circuit C obtained from M without unrolling. It computes the SPFDs of all nodes in C and uses them to update the sequential SPFDs of the nodes in M, which are initially empty. The SPFDs associated with the present state bits denote the information that they have to provide for ensuring correct functionality after one time frame. Next, the SPFD of each present state bit p_i is attached to the primary output of C that corresponds to p_i'. Then the SPFDs of C are re-computed. In general, the i^{th} step computes the sequential SPFDs of the nodes required for correctness in $\leq i$ time frames. The process stops when no more edges are added to any node in the network.

Algorithm **com_seq_spfds**($M, \mathcal{P}$):

1. $R^* = Reach\_state(M)$.
2. *For each node* $\eta_j \in M$, $R_j^s \leftarrow \phi$.
3. *Obtain the combinational circuit C from M.*
4. *Restrict the present state inputs of C so that it allows only R^*; these are used to restrict the number of input combinations that can be used during the image computation steps. The initial SPFDs on the POs of C that are also POs in M are given by the functions of the gates driving these outputs. The SPFDs of the remaining POs of C (i.e., the next state bits of M) are empty.*
5. *Compute_spfds(C).*
6. *Update_spfds(M).*
7. *repeat* {
   (a) *Modify_state_spfds($M, \mathcal{P}$).*
   (b) *Attach empty SPFDs to the POs of C that are also POs of M and the SPFDs of the present state bits of M to the POs of C that correspond to the next state bits of M.*
   (c) *Compute_spfds(C).*
   (d) *Update_spfds(M).*

   }*until (no change in SPFDs of nodes).*

8. Stop.

Reach_states computes the set of reachable states of M starting from the initial states. In general, any over-approximation of the reachable states can be used. However, the SPFDs of the nodes are the smallest if the set of reachable states is used. *Compute_spfds* computes the SPFDs of all the nodes in C as in the combinational case, described in Chapter 5. The subroutine *Update_spfds* uses the SPFDs of the nodes in C for updating the sequential SPFDs of the corresponding nodes of M. For each node η_j in M, it computes the union of the sequential SPFD of η_j stored in M with the new SPFD attached to the copy of η_j in C. During each SPFD computation phase, the present state bits are treated as primary inputs; hence the SPFD of each present state bit p_i is expressed in terms of the fanins of the fanouts of p_i. The subroutine *Modify_state_spfds* transforms this SPFD so that it is expressed in terms of the variables in $\mathcal{P}_k$, where $p_i \in \mathcal{P}_k$. The set of reachable states R^* are used to restrict the minterm combinations in the SPFD of p_i. It only contains edges between a and b such that a and b are cubes of $\mathcal{P}_k$ variables and are contained in S_k, where S_k is obtained from R^* by existentially quantifying the variables not in $\mathcal{P}_k$.

For now, the algorithm assumes that $\mathcal{P}$ has been chosen. It is important to observe that the algorithm only uses $\mathcal{P}$ in the subroutine *Modify_state_spfds*. $\mathcal{P}$ is useful if it is desirable to perform partial re-encoding of the state space since each partition can be re-encoded independently. This avoids building an incompatibility graph over the entire state space. One simple heuristic to choose $\mathcal{P}$ could group present state bits that have paths to the same set of primary outputs. In general, the structure of the circuit's topology can be used to find a good partition.

8.3.3 Theory

The ideas presented above are formalized in this section. In general, M is a Mealy machine; its primary output logic is a function of the present state and the primary inputs.

Definition 8.1 *A pair of states (s, s') in S is* **distinguishable** *if there exists an input sequence such that M produces different outputs for s and s'.*

Definition 8.2 *Given a state s, the* **projection** *of s onto the set of variables Z, denoted as s^Z, is obtained by existential quantification of all variables not in Z from s.*

Definition 8.3 *The* **sequential SPFD** *at a node η_j is the SPFD R_j^s associated with it when* **com_seq_spfds** *terminates.*

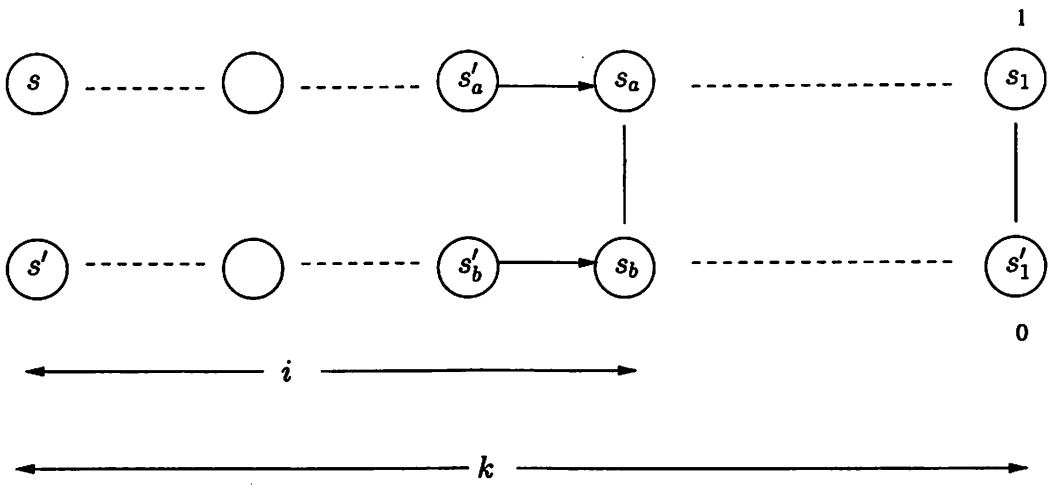


Figure 8.8: Illustration for the proof of Lemma 8.3.

In the sequel, R_j^m denotes the SPFD of η_j after m steps of **com_seq_spfds**.

Definition 8.4 *The* **SPFD of a part** $\mathcal{P}_k$ *of* $\mathcal{P}$ *is the union of the SPFDs of the present state bits in* $\mathcal{P}_k$. *It is denoted as* $R_{\mathcal{P}_k}$.

Definition 8.5 *The* **state SPFD** R *is a graph* $G = (S, E)$, *where an edge exists between two states* s *and* s' *if there exists a part,* $\mathcal{P}_i \in \mathcal{P}$, *such that* $(s^{\mathcal{P}_i}, s'^{\mathcal{P}_i}) \in R_{\mathcal{P}_i}$. *Here,* $s^{\mathcal{P}_i}$ *and* $s'^{\mathcal{P}_i}$ *are projections of* s *and* s' *respectively onto* $\mathcal{P}_i$.

First it is shown that the algorithm **com_seq_spfds** terminates and then that the state SPFD R has an edge between a pair of states if they are distinguishable.

Lemma 8.1 *The computation of* R_j^k *of a node* η_j *by* **com_seq_spfd** *is monotonic in* k.

Proof Let the SPFD of η_j after k and $(k+1)$ iterations be denoted as R_j^k and R_j^{k+1} respectively. Since R_j^{k+1} is obtained from R_j^k by adding SPFDs edges, hence $R_j^k \subseteq R_j^{k+1}$. □

Lemma 8.2 R_j^k *is finite for* $k > 0$.

Proof The input space of η_j is denoted as Y_j. R_j^k denotes input combinations that have to be distinguished after k iterations. Since η_j has a finite number of inputs, $R_j^k \subseteq Y_j \times Y_j$ is finite. □

Lemma 8.3 *If two reachable states* s *and* s' *are distinguishable, then the state SPFD* R *contains an edge between them.*

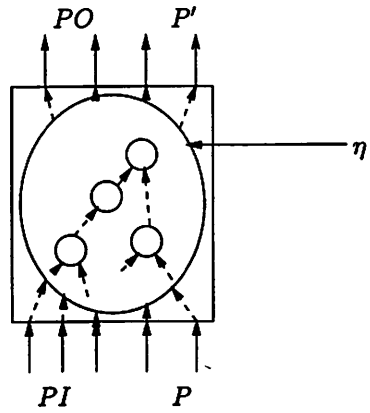


Figure 8.9: M': implementing the transition relation of M.

Proof By contradiction. Suppose s and s' are distinguishable in k steps but $(s, s') \notin R$. Then, there must be a set of states $\{s_a, s_b, s'_a, s'_b\} \in S$ such that $(s'_a, s_a) \in T$, $(s'_b, s_b) \in T$, $(s_a, s_b) \in R$ and $(s'_a, s'_b) \notin R$. This is illustrated in Figure 8.8.

Suppose the algorithm stops after m steps. The stopping criterion requires that no more additional SPFD edges are added. Since, $(s_a, s_b) \in R$, then $e = (s_a^{\mathcal{P}_k}, s_b^{\mathcal{P}_k})$ must exist in the SPFD of at least one partition $\mathcal{P}_k$. This implies that there exists a present state bit $p_j \in \mathcal{P}_k$, such that its SPFD R_j contains e. Since $e \in R_j^m$, the algorithm would have added $e' = (s_a'^{\mathcal{P}_l}, s_b'^{\mathcal{P}_l})$ to the SPFD of a present state bit p_i in the next iteration, where $p_i \in \mathcal{P}_l$. Hence, $e' \in R_{\mathcal{P}_l}$. This contradicts the assumption that $(s'_a, s'_b) \notin R$. □

Theorem 8.1 *The sequential SPFDs computed by* **com_seq_spfds** *contains the information for correct re-encoding of a sequential machine.*

Proof R_j^k is monotonic (Lemma 8.1) and finite (Lemma 8.2) for all $k > 0$. Thus R_j^k has a fixed point and hence **com_seq_spfds** terminates. By Lemma 8.3, an edge exists in the state SPFD between any two reachable distinguishable states. □

8.3.4 Previous Work

The work presented in this chapter is similar to classical state minimization of completely specified machines (c.f. [39]), which progressively partitions the state space into equivalence classes until no additional refinements can be made. At this point, the states in an equivalence class can be merged. Thus each equivalence class contains states which are not distinguishable. By Lemma 8.3,

the state SPFD contains an edge between any two states that are distinguishable. Hence the states that can be colored with the same color are a subset of an equivalence class obtained by the classical state minimization. However, two states in the same equivalence class can have an edge between them in the state SPFD, since in general, only containment is guaranteed.

Consider the circuit M', shown in Figure 8.9. M' has a single multi-valued node η which implements the transition and output relations of M. The inputs of η are the primary inputs and the present state variables of M. The outputs of η are the primary outputs and next state variables of M. The state SPFD obtained by executing **com_seq_spfds** on M' with $\mathcal{P} = \{P\}$ has an edge between two states iff they are distinguishable. In this case, the equivalence classes obtained from the state SPFD coincide with the ones obtained by the classical state minimization algorithm. Thus the additional edges are due to the particular decomposition of M and the partitioning of the state bits.

8.3.5 State Encoding Using Sequential SPFDs

The SPFD of each part in the partition can be used to perform a re-encoding of the state space. For each part $\mathcal{P}_i$, its SPFD $R_{\mathcal{P}_i}$ is solely expressed in terms of the variables of $\mathcal{P}_i$. $R_{\mathcal{P}_i}$ can thus be colored to get a new encoding of the bits of $\mathcal{P}_i$. This procedure can be repeated for each $\mathcal{P}_i$.

This method can accomplish a wide range of state encodings depending on the partition used while computing the SPFDs. On one extreme, if $\mathcal{P} = \{P\}$, then the SPFD of that part is equal to the state SPFD. Coloring the state SPFD yields a complete re-encoding of the state space. On the other extreme, re-encoding using $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2, \cdots, \mathcal{P}_m\}$, where $\mathcal{P}_i = p_i$, yields the original state encoding. A good partition that uses the initial decomposition of the circuit can be used to do partial re-encoding of the state space. This approach is computationally feasible for very large machines since it only encodes a subset of the state variables in each step. Traditional state minimization algorithms must build an incompatibility graph over the entire state space.

The following example illustrates the effect of the different partitions of P on the quality of state re-encoding.

Example 8.3 *Consider the circuit in Figure 8.6 and perform re-encoding of the state space for different partitions of P:*

1. $\mathcal{P} = \{\mathcal{P}_1\}$ *where* $\mathcal{P}_1 = (p_1, p_2, p_3, p_4)$. *After the first step, the SPFDs of* p_2 *and* p_4 *are obtained. The SPFDs of* p_2 *and* p_4 *are* $\{(0100, 1000), (0100, 0010)\}$ *and* $\{(0001, 1000), (0001, 0010)\}$, *respectively. Similarly, after the second step, the SPFDs of* p_1 *and* p_3 *are*

$\{(1000, 0100), (1000, 0001)\}$ *and* $\{(0010, 0100), (0010, 0001)\}$. *Another step of the algorithm adds no more edges and thus the algorithm stops. The SPFD of* $\mathcal{P}_1$ *is bipartite. Hence, this SPFD can be colored using two colors*[1]. *As a result, the reached states of the original state space can be encoded as:*

$$1000 \rightarrow 0; 0010 \rightarrow 0; 0100 \rightarrow 1; 0001 \rightarrow 1;$$

2. $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2\}$, *where* $\mathcal{P}_1 = (p_1, p_3)$ *and* $\mathcal{P}_2 = (p_2, p_4)$. *After the first step, the SPFDs of* p_2 *and* p_4 *are obtained. The SPFDs of* p_2 *and* p_4 *in terms of the variables in their respective parts are* $\{(10, 00)\}$ *and* $\{(01, 00)\}$, *respectively. Similarly, after the second step, the SPFDs of* p_1 *and* p_3 *in terms of variables in their respective parts are* $\{(10, 00)\}$ *and* $\{(01, 00)\}$. *The algorithm terminates in the next step. Consider the effect of re-encoding each partition separately. The SPFD of* $\mathcal{P}_1$ *is* $\{(10, 00), (01, 00)\}$. *Since it is bipartite, it can be colored using two colors. Let minterms* 00, 01 *and* 10 *in* $\mathcal{P}_1$ *map to* 1, 0 *and* 0 *respectively. Similarly,* $\mathcal{P}_2$ *can be re-encoded by coloring* $R_{\mathcal{P}_2}$. *Since* $R_{\mathcal{P}_2}$ *is also bipartite, it can also be colored with two colors. Let minterms* 00, 01 *and* 10 *in* $\mathcal{P}_2$ *map to* 0, 1 *and* 1 *respectively. Hence a circuit with two state bits can be obtained. So, the new encoding of the reached states is:*

$$1000 \rightarrow 00; 0010 \rightarrow 00; 0100 \rightarrow 11; 0001 \rightarrow 11;$$

3. $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_4\}$, *where* $\mathcal{P}_1 = p_1$, $\mathcal{P}_2 = p_2$, $\mathcal{P}_3 = p_3$ *and* $\mathcal{P}_4 = p_4$. *The algorithm terminates in three steps and computes the SPFDs of all the nodes. The SPFDs of each* p_i *in terms of* $\mathcal{P}^{p_i}$ *is* $\{(1, 0)\}$. *Re-encoding each partition separately produces no reduction in the state bits.*

8.3.6 Sequential SPFDs Using Classical Incompatibility Graph

It is interesting to note that the incompatibility graph of M derived using the classical state minimization algorithms ([39]) can be directly used to derive the sequential SPFDs of all the nodes of M in one step. The procedure is outlined below:

1. *Treat* M *as a specialized combinational circuit* C *where the POs are the POs of* M *and the PIs are the PIs of* M *plus the state bits of* M. *The next state bits of* M *are the inputs of a dummy node* D *in* C. *The SPFD of* D *is the supplied incompatibility graph.*

[1] This is exactly what was obtained at the end of Section 8.2.1.

2. *Compute the SPFDs of all nodes in C (including D) in reverse topological order from primary outputs to primary inputs using Compute_spfds.*

8.4 Resynthesis Procedure

Given the encoding relation Enc between the old states and the new states and the sequential SPFDs at all the nodes in M, the original circuit can be resynthesized using the following algorithm.

Algorithm **seq_resyn**(M, $\{R_j^s\}$):

1. *Proceed in topological fashion from primary inputs and present state bits to primary outputs and present state outputs.*

2. *For each node η_j, perform the following two steps:*

   (a) *Compute the mapping between the original and the new fanin spaces of node η_j:*

   $$En(Y_j, \hat{Y}_j) = \exists_{X,P,P^e} R^*(P)(P^e = Enc(P))\mathcal{G}(X, P, Y_j)\hat{\mathcal{G}}(X, P^e, \hat{Y}_j),$$

   where X is the set of primary inputs, P is the set of old state variables, P^e is the set of new state variables, $R^*(P)$ is the set of reachable states, Enc gives the new encoding of the states, $\mathcal{G}(X, P, Y_j)$ gives the transition relation of the original fanins and $\hat{\mathcal{G}}(X, P^e, \hat{Y}_j)$ the transition relation of the new fanins. The process is illustrated in Figure 8.10.

   (b) *Obtain the modified SPFD as:*

   $$R_j^s(\hat{Y}_j, \hat{Y}_j') = \exists_{Y_j, Y_j'} En(Y_j, \hat{Y}_j) En(Y_j', \hat{Y}_j') R_j^s(Y_j, Y_j').$$

   Color it to get an ISF for the node and minimize it.

3. *Attach a new multi-output node F at the output of the next state bits. F has n inputs and m outputs, where n is the number of original state bits and m is the number of new state bits. It can be implemented as a PLA. This node maps the re-implemented next state bits $\hat{P}'$ onto their new state encoding $Enc(P')$, as shown in Figure 8.11.*

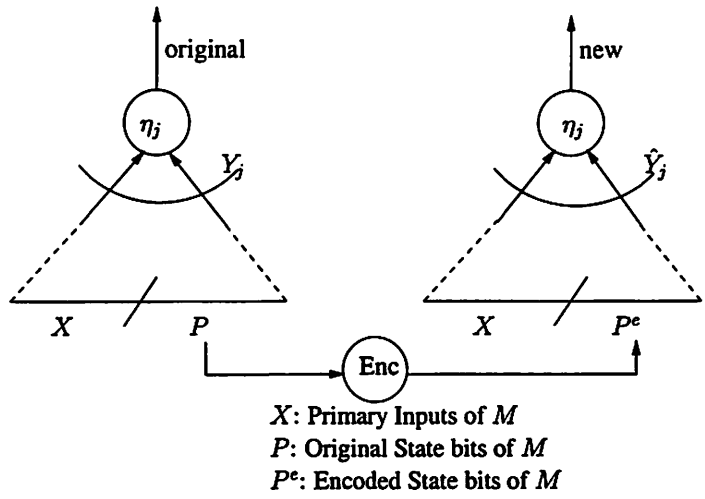


Figure 8.10: Encoding relation between the original and new fanin variables, $En(Y_j, \hat{Y}_j)$.

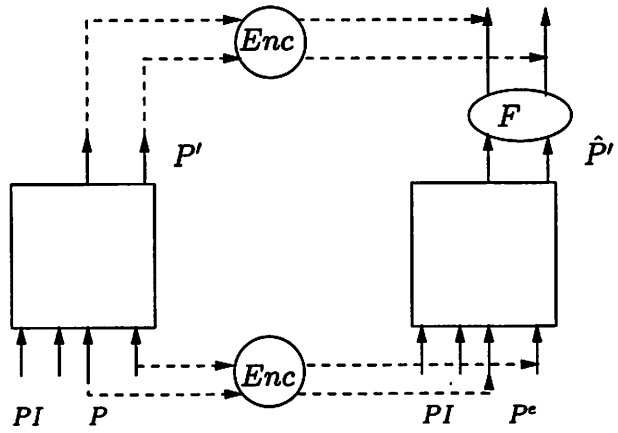


Figure 8.11: Computing the function of the multivalued node.

Example 8.4 *Figure 8.1 can be redrawn as shown in Figure 8.12. Assume that the sequential SPFDs of all the nodes are given. Further, let the encoding between the old and the new state spaces be* $\{(1000, 00), (0100, 01), (0010, 10), (0001, 11)\}$.

The sequential SPFD of f_1 *is* $\{(1000, 0100), (1000, 0010), (1000, 0001)\}$. *In terms of its inputs, the SPFD can be re-written as* $\{(0001, 1000), (0001, 0100), (0001, 0010)\}$. *The SPFD of* f_1 *in terms of its new fanins* p_1^e *and* p_2^e *is* $\{(11, 00), (11, 10), (11, 01)\}$. *Thus* f_1 *can be re-implemented as* $\hat{f}_1 = (\overline{p_1^e} + \overline{p_2^e})$. *Similarly, the new functions of* f_2, f_3 *and* f_4 *are* $\hat{f}_2 = \overline{p_1^e}\,\overline{p_2^e}$, $\hat{f}_3 = (p_1^e + \overline{p_2^e})$ *and*

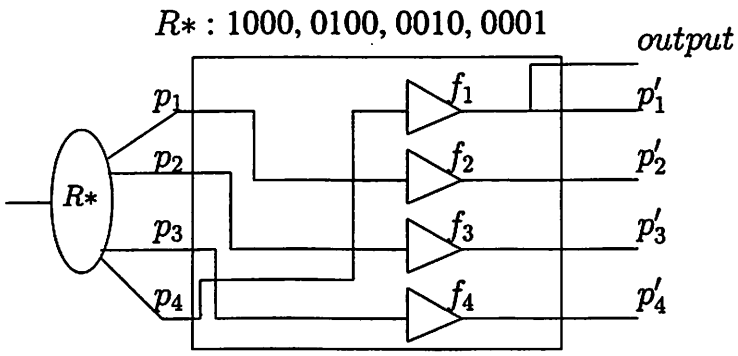


Figure 8.12: Revisiting Example 8.1.

$\hat{f}_4 = p_1^e\overline{p_2^e}$ *respectively. The encoding between* P' *and* $\hat{P}'$ *is*

$$\begin{aligned} 1000 &\rightarrow 0010 \\ 0100 &\rightarrow 1110 \\ 0010 &\rightarrow 1000 \\ 0001 &\rightarrow 1011. \end{aligned}$$

The new function of the output node F *is given below.*

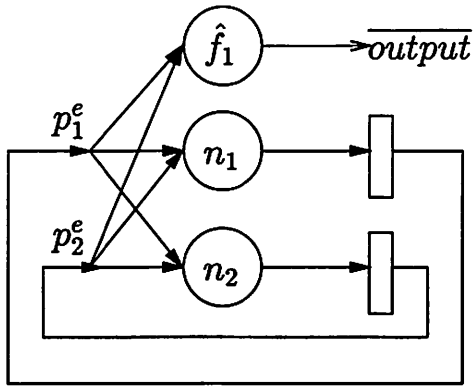
$\hat{f}_1$	$\hat{f}_2$	$\hat{f}_3$	$\hat{f}_4$	$\hat{p}'_1$	$\hat{p}'_2$
0	0	1	0	0	0
1	1	1	0	0	1
1	0	0	0	1	0
1	0	1	1	1	1

It can be implemented as two binary nodes, n_1 *and* n_2.

$$\begin{aligned} n_1 &= \hat{f}_1\overline{\hat{f}_2}(\hat{f}_3\overline{\oplus}\hat{f}_4) \\ n_2 &= \hat{f}_1\hat{f}_3(\hat{f}_2 \oplus \hat{f}_4) \end{aligned}$$

The SPFD of the output in terms of its inputs is $\{(0001, 1000), (0001, 0100), (0001, 0010)\}$. *Given the new state encoding, the modified SPFD is* $\{(11, 00), (11, 01), (11, 10)\}$. *Thus the new function of the output is* $p_1^e p_2^e$.

The above circuit can be further simplified by collapsing $\hat{f}_1$, $\hat{f}_2$, $\hat{f}_3$ *and* $\hat{f}_4$ *into* n_1 *and* n_2 *to yield the circuit shown in Figure 8.13.*



$$\hat{f}_1 = \overline{p_1^e} + \overline{p_2^e}$$
$$n_1 = p_1^e \oplus p_2^e$$
$$n_2 = \overline{p_2^e}$$
Intial state : 00

Figure 8.13: Re-implementation of Example 8.1.

Similarly, resynthesizing the circuit in Figure 8.5 using the state encoding

$$1000 \rightarrow 1; 0100 \rightarrow 0; 0010 \rightarrow 1; 0001 \rightarrow 0;$$

and simplifying it yields the circuit in Figure 8.14.

Note that in general **com_seq_spfds** followed by **seq_resyn** may be iterated to yield further reductions.

8.5 Summary

The concept of sequential SPFDs was introduced in this chapter. Given a partition of the state bits, an algorithm was presented which computes sequential SPFDs for the nodes in a sequential circuit. Each part in the partition was also associated with an SPFD. The SPFDs of these parts could be used for re-encoding the state space. This approach can be particularly useful for larger machines as it avoids building the incompatibility graph for the entire state space. The effect of different partitions on the quality of results was illustrated.

Another algorithm used the sequential SPFDs and a new state encoding for resynthesizing the sequential circuit. The resynthesis procedure could also be used in conjunction with other state minimization algorithms for obtaining a new circuit. The two algorithms could be iterated to yield

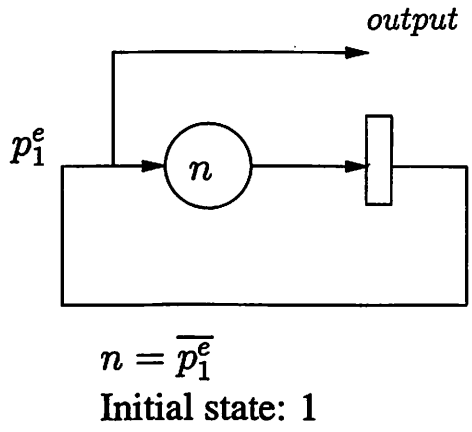


Figure 8.14: Re-implementation of Example 8.2.

new partitions and new encodings. The algorithms worked directly on the current implementation of the machine and thus only dealt with completely specified machines. A natural extension is to investigate the application of these ideas to incompletely specified machines.

Chapter 9

Conclusions

A new formalism for expressing flexibility during logic synthesis was studied in this work. The contributions of this dissertation are summarized below. Some directions for future work are also outlined.

In Chapter 4, the concept of Sets of Pairs of Functions to be Distinguished or SPFDs was introduced. The notion of representing the information content of a node/wire using SPFDs was presented. The concept of the minimum SPFD for a node was proposed. An algorithm was presented for computing the minimum SPFD of a node in a network. It was argued that node simplification using the minimum SPFD could be very computationally expensive. Hence the concept of the compatible SPFD of a node was introduced. The flexibility expressed using SPFDs was compared to several previous formalisms used for expressing flexibility. It was shown that SPFDs are a special type of Multiple Boolean Relation (MBR). They completely contain the flexibility expressed by Multi-output multifunctions and they extend the flexibility expressed by Boolean Relations but do not completely contain it.

Algorithms for generating compatible SPFDs and for resynthesizing the nodes in a network using these SPFDs were described in Chapter 5. This SPFD computation is similar to the CODC computation algorithm [20] but the resynthesis process is more involved than the resynthesis phase in CODCs. This is mainly because the SPFDs allow changes to node functionality that are not allowed by CODCs. The increased flexibility of SPFDs comes at an increased cost, both in terms of robustness and predictability. While robustness issues have arisen in other logic synthesis operations using BDDs, it can become particularly acute for SPFDs. This is mainly because more information needs to be stored in the BDDs during SPFD manipulations. The robustness problem was partially solved by using a SAT solver and a BDD engine together. SAT solvers are known to be

more robust than BDD engines. But they also suffer from the problem of reduced efficiency for set manipulations. A hybrid scheme combining the robustness of SAT with the efficiency of BDD was presented for tackling some of the more memory-intensive computations of the algorithms. The increased flexibility represented by SPFDs can often cause uncontrolled changes in the network. This could manifest itself as an unpredictability in the optimization results. The notion of a "region of change" was introduced for limiting the changes allowed by SPFDs, while using some of the additional flexibility provided by SPFDs.

With the decrease in feature sizes, interconnect effects are becoming more dominant. SPFDs provide a powerful tool for manipulating the interconnections of a network. At the heart of this ability is the notion that SPFDs represent the information content of a wire in a network, in the form of primary input minterm pairs that the wire has to distinguish. This concept was exploited for using SPFDs for changing the wiring between the nodes in a network in Chapter 6. Some preliminary intuition was provided in order to explain why SPFD-based rewiring can be much more powerful than traditional ATPG-based methods. Several different rewiring scenarios were presented. SPFD-based rewiring was used for reducing the number of interconnections(wires) in Boolean networks. The experiments showed that this method produced a 19% reduction in wire count and a 12% reduction in literal count. The concept of *don't care* wires was also proposed. These refer to alternate wire sets where the choice for one wire is completely independent of the choice for other wires. The concept is similar to that of compatible logic *don't cares*. An algorithm was presented for generating these *don't care* wires, which were subsequently used for minimizing the total wirelength of networks of PLAs. Initial experiments were done in an integrated synthesis and placement environment with favorable results. A 12% reduction in wirelength was obtained using the *don't care* wires. Moreover, a positive correlation was observed between the number of wires with *don't care* sets of wire and improvements in wirelength.

An interesting application of SPFDs to functional decomposition was presented in Chapter 7. It was proved that SPFDs can be used for solving the Ashenhurst-Curtis decomposition problem. A new type of decomposition problem called the topologically constrained decomposition problem was introduced. The problem requires synthesizing the nodes in a network so that a particular functionality is implemented, given *a priori* the final topology of the network. This type of problem may arise in interconnect-centric algorithms like wireplanning, where the interconnection between the nodes is decided before the logic in the nodes is fixed. The notion that the network acts as a lossy channel through which information flows from the primary inputs to the primary outputs was developed. The constrained decomposition problem was formulated in terms of information

flow and an algorithm for synthesizing the nodes in the network using SPFDs (which can represent information content of a node or wire) was presented.

The concept of sequential SPFDs was proposed in Chapter 8. An algorithm for state re-encoding of a general, sequential circuit was presented and the correctness of the method was established. A procedure for resynthesizing the sequential circuit using the new state encoding was also presented. The idea of partitioning the state bits of a sequential machine for dealing with large sequential machines was also proposed.

9.1 Future Work

SPFDs are interesting and seem fundamental to the synthesis process since they represent how information is passed along and processed by a network. This dissertation attempts to highlight some of the interesting applications of SPFDs to logic synthesis algorithms. Much work remains in order that they can be computed and used efficiently. In this section, we provide some directions for future work.

The basic algorithms for SPFD computation and resynthesis are still quite expensive which might affect their acceptance by the rest of the community. In this dissertation, some techniques for improving the efficiency and robustness of SPFDs were provided. However, more work is required in this area. Some other ideas that haven't been tried include:

1. Using specialized BDD operators for speeding up some repeated operations.

2. Approximating the SPFD computations thereby losing some flexibility but greatly increasing efficiency.

3. Using an incremental SAT solver (since the SAT problems generated during SPFD computations are very similar).

The other problem of SPFDs, namely uncontrolled change, was solved using the concept of a "region of change". Other techniques for controlling the changes induced in the network after optimizing a few nodes using SPFDs need to be investigated, such as creating independent partitions in the network and using SPFD optimization on each of the independent partitions.

SPFDs can be a very powerful tool for rewiring. There are many interesting rewiring applications of SPFDs. The concept of *don't care* wires was presented and some initial experiments

were presented in Chapter 6. A lot more work can be done in that area. For example, the computation of the *don't care* wires assumes a random ordering of the wires during the SPFD computation. The ordering scheme can be made more intelligent depending on the metric being optimized such that "expensive" wires have a greater chance of finding alternates. Different metrics like delay, congestion, crosstalk, etc can also be optimized using *don't care* wires. The concept of *partial don't care* wires was also introduced in the same chapter. These are a generalization of *don't care* wires, in that an alternate wire contains only a part of the information provided by the original wire. The resynthesis process can become more complicated when *partial don't care* wires are used. Experiments need to be performed for examining the benefits of using this generalization.

In Chapter 7, an algorithm was presented for using SPFDs for solving the topologically constrained decomposition problem. Interesting applications of this technique are being currently investigated. Any application that can generate an initial topology can be used. In particular, topologies that are generated using *partial don't care* wires should be examined.

Sequential SPFDs provide a whole new area for interesting research problems. The work described in Chapter 8 needs an efficient implementation of the ideas presented there. The benefits of different partitioning strategies need to be experimentally investigated. It should also be interesting to explore rewiring possibilities in a network of finite state machines using sequential SPFDs.

Bibliography

[1] W. Gosti, A. Narayan, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Wireplanning in logic synthesis. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 26–33, Nov 1998.

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

[3] R. Brayton and F. Somenzi. Boolean Relations and the incomplete specification of logic networks. In *Proceedings of the International Conference on VLSI*, pages. 231–240, Aug 1989.

[4] K-T. Cheng and L. Entrena. Multi-level logic optimization by redundancy addition and removal. In *Proceedings of European Conference Design Automation*, pages 373–377, Feb 1993.

[5] S.C. Chang, M. Marek-Sadowska and K.T. Cheng. Perturb and Simplify: Multi-level Boolean Network Optimizer. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15 (no. 12), pages 1494–1504, Nov 1996.

[6] S. Chang and K-T. Cheng. Postlayout Logic Restructuring Using Alternative Wires. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16 (no. 6), pages 587–596, Jun 1997.

[7] C-W. Chang, C-k Cheng, P. Suaris and M. Marek-Sadowska. Fast Post-placement Rewiring Using Easily Detectable Functional Symmetries. In *Proceedings of IEEE/ACM Design Automation Conference*, pages 286–289, Jun 2000.

[8] C-W. Chang and M. Marek-Sadowska. XWire : Efficient Single Wire Addition and Removal Beyond Redundancy. In *International Workshop on Logic Synthesis*, Jun 2001.

[9] C. Berman and L. Trevillyan. Global Flow Optimization in Automatic Logic Design. In *IEEE Transactions on Computers*, vol. 10 (no. 5), pages 557–564, May 1991.

[10] S. Chang, Z. Wu and H. Yu. Wire Reconnections Based on Implication Flow Graph. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 533–536, Nov 2000.

[11] S. Yamashita, H. Sawada, and A. Nagoya. A new method to express functional permissibilities for LUT based FPGAs and its applications. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 254–261, Nov 1996.

[12] R. Bryant. Graph based algorithms for Boolean function manipulation. In *IEEE Transactions on Computers*, vol. C-35 (no. 8), pages 677–691, Aug 1986.

[13] J. P. Marques-Silva and K. A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. In *IEEE Transactions on Computers*, vol. 48 (no. 5), pages 506–521, May 1999.

[14] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang and S. Malik. Engineering an efficient SAT solver. In *Proceedings of IEEE/ACM Design Automation Conference*, pages 530–535, Jun 2001.

[15] S.A. Cook. The complexity of theorem-proving procedures. In *Proceedings of 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158, May 1971.

[16] J. P. Marques-Silva and K. A. Sakallah. Boolean Satisfiability in Electronic Design Automation. In *Proceedings of the IEEE/ACM Design Automation Conference*, pages 675–680, Jun 2000.

[17] S. Malik. Analysis of Cyclic Combinational Circuits. In *Proceedings of IEEE International Conference on Computer-Aided Design*, pages 618–625, Nov 1993.

[18] R. Rudell. Logic Synthesis for VLSI Design. *Ph.D. thesis*, UC Berkeley, 1989.

[19] K. A. Bartlett, R.K. Brayton, G.D. Hatchel, R.M. Jacoby, C.R. Morrison, R. L. Rudell, A.L. Sangiovanni-Vincentelli, and A.R. Wang. Multi-level Logic Minimization Using Implicit Don't Cares. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 7 (no. 6), pages 723–740, Jun 1988.

[20] H. Savoj. Don't Cares in Multi-Level Network Optimization. *Ph.D. thesis*, UC Berkeley, 1992.

[21] S. Muroga, Y. Kambayashi, H. Lai and J. Culliney. The transduction method - design of logic networks based on permissible functions. In *IEEE Transactions on Computers*, vol. C-38 (no. 10), pages 1404–1424, Oct 1989.

[22] Y. Watanabe and R.K. Brayton. Heuristic Minimization of Multiple-Valued Relations. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 126–129, Nov 1991.

[23] E. M. Sentovich and R. K. Brayton. Multiple Boolean Relations. In *International Workshop on Logic Synthesis*, May 1993.

[24] Y. Kukimoto and M. Fujita. Rectification Method for Lookup-Table Type FPGA's. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 54–61, Nov 1992.

[25] R. Brayton. Compatibility observability don't cares revisited. In *International Workshop on Logic Synthesis*, Jun 2001.

[26] A. Veneris, M.S. Abadir and I. Ting. Design Rewiring Based on Diagnosis Techniques. In *Proceedings of ASP-DAC*, pages 474–484, 2001.

[27] S. Khatri, R. Brayton, and A. Sangiovanni-Vincentelli. A VLSI design methodology using a network of PLAs embedded in a regular layout fabric. *Technical Report UCB/ERL M99/50, Electronics Research Laboratory*, University of California, Berkeley, May 1999.

[28] Melvin A. Breuer. Min-cut placement. In *Journal of Design Automation and Fault-Tolerant Computing*, vol. 1 (no. 4), pages 343–362, Oct 1977.

[29] C.M. Fiduccia and R.M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the IEEE/ACM Design Automation Conference*, pages 174–181, Jun 1982.

[30] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani. VLSI module placement based on rectangle-packing by the sequence-pair. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15 (no. 12), pages 1518–1524, Dec 1996.

[31] H. Eisenmann and F.M. Johannes. Generic Global Placement and Floorplanning. In *Proceedings of the IEEE/ACM Design Automation Conference*, pages 269–274, Jun 1998.

[32] R.L. Ashenhurst. The Decomposition of Switching Functions. In *Proceedings of International Symposium on the Theory of Switching*, pages 74–116, Apr 1959.

[33] S. Hassoun, T. Sasao and R. Brayton Logic Synthesis and Verification. *Kluwer Academic Publishers*, 2002.

[34] M. Perkowski and S. Grygiel. A survey of literature on Functional Decomposition. *Technical Report, Department of Electrical Engineering, Portland State University*, 1995.

[35] V. Cheushev, S. Yanushkevich, V. Shmerko, C. Muraga and J. Kolodziejcyk. Information Theory Method for Flexible Network Synthesis. In *IEEE International Symposium on Multiple-Valued Logic*, pages 201–206, May 2001.

[36] L. Jozwiak. Information Relationships and Measures in Application to Logic Design. In *IEEE International Symposium on Multiple-Valued Logic*, pages 228–235, May 1999.

[37] A. Mishchenko and R.K. Brayton. Simplification of Non-Deterministic Multi-Valued Networks. To appear in *International Workshop on Logic Synthesis*, Jun 2002.

[38] Z. Kohavi. Switching and Finite Automata Theory. *McGraw Hill Publishing Company*, 1978.

[39] T. Kam. T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. Synthesis of finite state machines : logical optimization. *Boston : Kluwer Academic Publishers*, 1997.

[40] B. Lin, H. Touati and R. Newton. Don't Care Minimization of Multi-level Sequential Logic Networks. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 414–417, Nov 1990.

[41] A. Lin, K. Chen, M. Marek-Sadowska and M. Lee. Sequential Permissible Functions and their Application to Circuit Optimization. In *European Design and Test Conference*, pages 334–339, Mar 1996.

[42] L.A. Entera and K.T. Cheng. Sequential logic optimization by redundancy addition and removal. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 310–315, Nov 1993.

[43] S. Malik, E.M. Sentovich and R.K. Brayton. Retiming and Resynthesis : Optimizing sequential networks with combinational networks. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10 (no. 1), pages 74–84, Jan 1991.