BOOLEAN REASONING AND INFORMED SEARCH IN THE MINIMIZATION OF LOGIC CIRCUITS

DISSERTATION

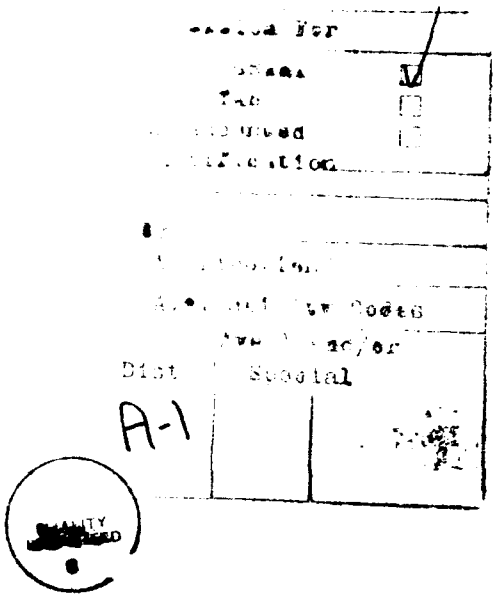
James John Kainec
Captain, US Army

AFIT/DS/ENG/92-02

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/DS/ENG/92-02



BOOLEAN REASONING AND INFORMED SEARCH IN THE MINIMIZATION OF LOGIC CIRCUITS

DISSERTATION

James John Kainec
Captain, US Army

AFIT/DS/ENG/92-02

AFIT/DS/ENG/92-02

BOOLEAN REASONING AND INFORMED SEARCH IN THE MINIMIZATION OF LOGIC CIRCUITS

DISSERTATION

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

James John Kainec, B.S., M.S.E.E

Captain, US Army

March, 1992

AFIT/DS/ENG/92-02

BOOLEAN REASONING AND INFORMED SEARCH IN THE MINIMIZATION OF LOGIC CIRCUITS

James John Kainec, B.S., M.S.E.E.

Captain, US Army

Approved:

Frank M. Brown, Chairman	5 MAR 92
Matthew Kabrisky	5 MAR 92
Mark A. Mehalic	5 MAR 92
Henry B. Potoczny	March 5 1992

Accepted:

J. S. Przemieniecki
Senior Dean

Acknowledgments

I wish to thank my committee, Dr. Frank Brown, Dr. Matthew Kabrisky, Dr. Henry Potoczny, and Captain Mark Mehalic, for reviewing this thesis and adding their constructive comments. The length of this thesis has required an inordinate effort on their part.

I would like to thank the AFIT VLSI group, in particular Captain Mehalic and Captain Keith Jones for the loan of a SUN SPARCstation for the past year. The use of the SUN and the VLSI Laser Printer has made my task much easier. I also would like to extend a particular thanks to Russ Milliron for the help he gave me in the past three years in using the VLSI computer systems (GO Buckeyes!).

I owe a special gratitude to Dr. Frank Brown for his valuable guidance while allowing me the greatest freedom possible through this endeavor. I called upon his knowledge in all phases of this effort. He spent many hours pointing out my miscues, small and large. His influence on my academic ability will be an asset for years to come.

Finally, I would like to express my appreciation to my wife, Kathy, and my daughters, Sarah and Stephanie, for their love, patience, and support which allowed me to devote the effort required to conduct this research.

James John Kolinec

Table of Contents

List of Figures

List of Tables

Abstract

The minimisation of logic circuits has been an important area of research for more than a half century. The approaches taken in this field, however, have for the most part been ad hoc. Boolean techniques have been employed to manipulate and transform formulas, but not for the task Boole intended, i.e., to perform symbolic reasoning. Boolean equations, for example, are employed in the literature on minimisation principally as icons; they are never solved. The first objective of this dissertation, accordingly, is to apply Boolean reasoning systematically and uniformly to the minimisation problem. Boolean reasoning entails the reduction of systems of Boolean equations to a single Boolean equation; the single equation is an abstraction, independent of the form of the original equations, upon which a variety of reasoning operations may be performed. The second objective of this dissertation is to apply informed search, which has arisen from research in Artificial Intelligence, to the minimization problem. In algorithms developed in this work, a circuit specification is reduced to a single equivalent Boolean equation $\phi(X, Z) = 1$ called a 1-normal form. There are a number of significant advantages to using the 1-normal form rather than traditional specification formats. It is shown that developing a particular solution $Z = F(X)$ for $\phi(X, Z) = 1$ corresponds to constructing a two-level design which meets the specification. This approach—which departs from conventional minimization techniques—has several advantages: a number of design problems which cannot be handled using conventional methods are easily treated, atypical design specifications unusable by conventional methods are dealt with in a uniform manner, and in some cases a single algorithm, rather than a set of algorithms, suffices to solve a problem.

BOOLEAN REASONING AND INFORMED SEARCH IN THE MINIMIZATION OF LOGIC CIRCUITS

I. Introduction

The minimization of logic circuits has been an important area of research for more than a half century. The importance of this field is discussed in the first section of this chapter. The significance of minimizing a special class of digital circuits called *two-level* circuits is then discussed. A survey of previous work in the area of two-level minimization is given; the shortcomings of existing techniques are highlighted. A problem statement which defines the goal of this work is given. The significant aspects of the approach developed in this work are then introduced. This chapter concludes with an overview of the remaining chapters.

Background

A goal of electrical engineers since the invention of digital computers has been the design of minimal digital logic circuits. Consequently, researchers have spent years in search of techniques which produce minimal designs. The goals of these researchers have been twofold. First and foremost is to develop methods which yield designs which are minimal, or at least reasonably close, with respect to some specified measure of cost. Second, but gaining in importance in recent years, is to develop techniques which themselves are efficient in their use of computing resources. Research continues to discover techniques which better meet these two goals.

An area of research called *logic minimization theory* includes all methods which have been discovered to minimize designs. Through the 1960s the purpose of logic minimization was to decrease the number of discrete components required to construct logic circuitry. Basic components such as gates and the basic constituents of gates—diodes and resistors—were expensive (Brayt 84:8).

Logic minimization was of great importance to find an implementation which required the fewest gates and, hence, minimized the cost of a circuit. Designers would describe a circuit at a low level, perhaps at the gate level; the description would be used as the input to a minimization technique that would produce a minimal gate-level design.

The advent of integrated circuits, specifically Very Large Scale Integration (VLSI), and hardware description languages (HDLs) changed the entire design process in the late 1980s. The introduction of VLSI and HDLs has led to a need for automated tools for handling the design process. Such tools fall under the purview of a field called *logic synthesis*. Logic synthesis consists of two components: *HDL synthesis* and *logic optimization*. HDL synthesis is the process of converting a design described in a hardware description language to a gate-level netlist. Logic optimization consists of minimizing gate-level designs which result from HDL synthesis as well as mapping the design to the intended implementation technology. The portion of logic optimization concerned with developing minimal gate-level designs is *logic minimization*. The automation of logic synthesis tools enables a designer to describe an initial design in an algorithmic or behavioral fashion and to automatically produce an optimized design which has been mapped to the planned implementation technology (deGeu 89, Hardi 89). Research continues in an effort to improve both HDL synthesis and logic optimization. This dissertation addresses the specific area of logic minimization.

The growing use of VLSI has had a great impact on digital design. The ability to place an ever-increasing number of complex functions onto a chip has created a need for techniques which handle the resulting complexity. VLSI circuit minimization is also important from a systems architecture point of view. Difficulties with off-chip communication negate performance that would otherwise be possible with VLSI technology; designers have therefore attempted to place as many functions as feasible onto a chip. Increasing the functionality of a chip reduces the number of chips required to build a system, which reduces the cost of a system as well as increasing its reliability.

This increased functionality requires each function placed on a chip to be minimized to the fullest extent possible.

While the importance of circuit minimization has grown, the requirement that minimization tools be efficient has likewise increased. Minimization algorithms which may have sufficed twenty years ago may now be impractical due to increases in the size of implemented circuits. This is not surprising considering that nearly all minimization problems in VLSI are NP-complete (Ullma 84:311). Hence, researchers must devise minimization techniques which can handle increased circuit size. Typically, assumptions are made which reduce the number of criteria which must be handled by an algorithm; no efficient algorithm has been developed which is a general-purpose circuit minimizer.

Designers need efficient minimization techniques so that they may examine the various design trade-offs by determining the resulting circuit speed and size for the different ways of constructing a circuit. Engineers often use the results to decide how a circuit will be constructed. Companies which do not perform such experimentation will produce inferior designs (Hardi 89:57). To enable such experimentation in the design process, minimization algorithms must take only minutes or hours to run; a longer period of time would make an algorithm infeasible for practical use.

Two-Level Minimization

All techniques for producing minimal digital circuits start with a *specification* which portrays in some manner how the circuit is supposed to operate. The specification represents the circuit's response to a class of stimuli. The stimuli are applied to a portion of the circuit called the *input nodes* or *inputs*. The circuit's response appears on the *output nodes* or *outputs*. A digital circuit is abstractly depicted in Figure 1.1. The inputs are represented by the n-variable input vector X; the outputs are denoted by the m-variable output vector Z. The goal of a minimization technique is to produce economical circuit implementations or *designs* which meet the specification.

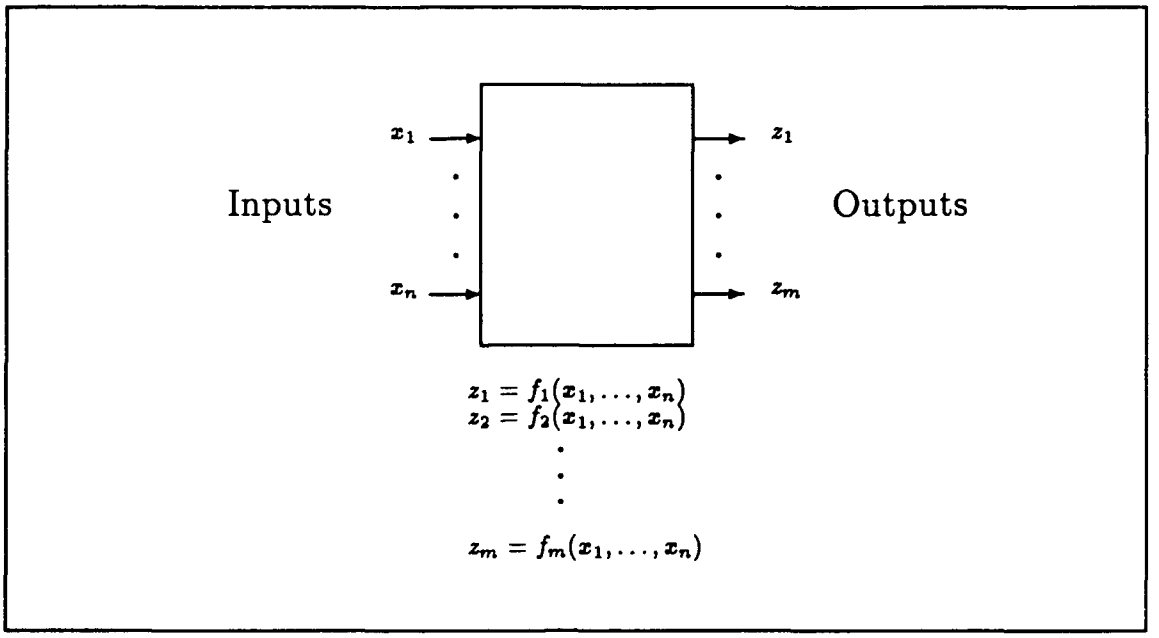


Figure 1.1. Representation of a Digital Circuit

An important type of circuit in digital design is *two-level* circuits. Two-level circuits are designs in which only two levels of gates must be traversed between the circuit inputs and the circuit outputs. Figure 1.2 depicts a two-level circuit; specifically, it is an AND-OR circuit. The AND gates form the first level; the OR gate forms the second level. The inverters are not said to form a level, because often the input signals and their complements are both available, eliminating the need for inverters. The number of levels of a circuit is defined as the maximum number of gates that must be traversed between the circuit inputs and the circuit outputs, less inverters required to complement the input signals. Any circuit which has more than two levels is called a *multi-level* circuit.

A *literal* is a letter or complemented letter such as x or y'. A *term* is 1, a literal, or a disjunction of two or more literals in which no two literals involve the same letter. A *sum-of-products* formula is 0, a single term, or a disjunction of terms. A convenient aspect of two-level circuits is

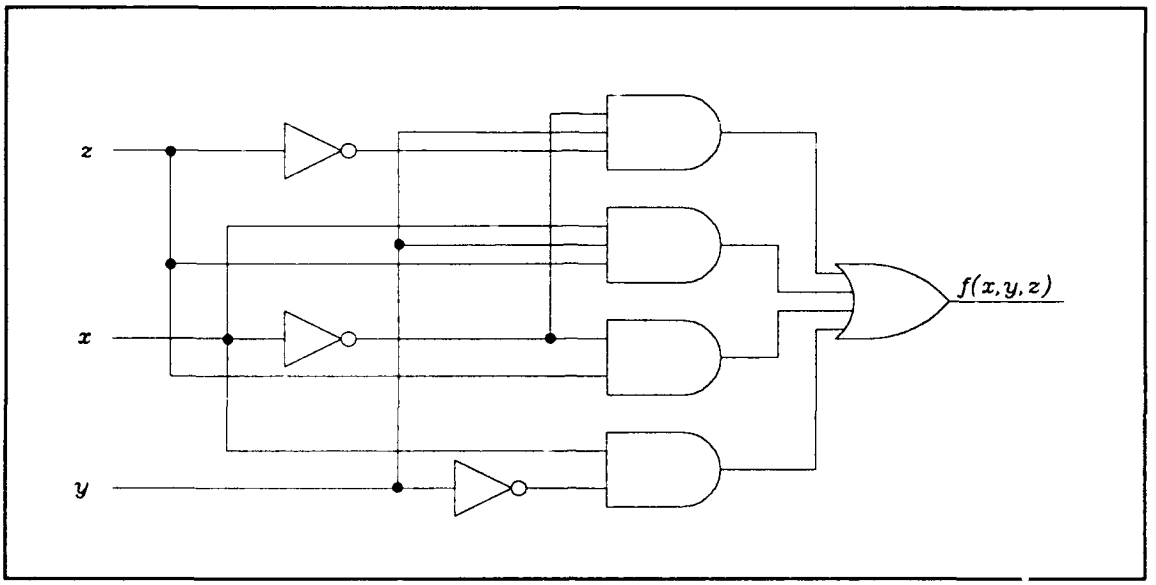


Figure 1.2. A Two-Level Circuit

that they correspond directly with *sum-of-products* (SOP) formulas representing Boolean functions. The SOP formula $x'z + x'yz' + xy' + xyz$ corresponds to the circuit depicted in Figure 1.2. An approach to developing economical two-level circuits is to specify a circuit with a Boolean function f, and then to develop a minimal SOP formula F to represent the function.[1] The resulting formula corresponds to a two-level design which meets the specification.

Measures used to determine the cost of a two-level circuit have a direct correlation with the attributes of an SOP formula. For example, the number of gates in a two-level circuit is determined by counting the number of terms in the corresponding SOP formula. Similarly, the number of gate inputs for a circuit is measured by counting the number of literals in the corresponding formula. Thus, measures used to determine the goodness of an SOP formula correlate to ways of developing the cost of a circuit. Criteria used to determine the goodness of an SOP formula include the fewest terms, fewest literals, or combinations thereof. Hence, an SOP formula with the fewest terms

[1] A function is denoted by a small letter, e.g., f, and a formula which represents the function is indicated by a capital letter, e.g., F.

which represents a function corresponds with the two-level design with the fewest AND gates. Quine (Quine 52) showed that a minimal formula with respect any criteria involving the number of literals of a formula consists only of special terms called *prime implicants* of a function. Hence, the following process is usually followed for developing a minimal SOP formula: develop the set P of all prime implicants for a function, and determine a subset $\widehat{P}$ of P which meets the given design criterion. The disjunction of the terms in $\widehat{P}$ constitutes a minimal SOP formula.

The process used to develop a single minimal formula is extended for multiple-output circuits, i.e., circuits for which the number of output variables is greater than one. In this case, for the vector $Z = (z_1, z_2, \ldots, z_m)$ of output variables, a vector $\underline{F} = (F_1, F_2, \ldots, F_m)$ of SOP formulas is developed which corresponds to a two-level design such that the combination of the distinct terms in formulas in $\underline{F}$ is minimal with respect to a given design criterion. The resulting vector $\underline{F}$ is then said to be minimal.

Two-level minimization is one of the most important problems in logic synthesis. Two-level circuits are very important in VLSI design due to the fact that they correspond directly to a Programmable Logic Array (PLA) implementation. There are several advantages with regard to using PLAs in VLSI design:

- PLAs are easy to implement,
- computer-aided design (CAD) tools exist to perform PLA layout automatically, and
- once implemented, PLAs are easy to modify.

In addition to being used to develop two-level circuits, two-level minimization techniques are employed during multi-level logic minimization. The development of a minimal two-level AND-OR circuit representation is typically one of the first steps in the process of developing a good multi-level circuit; for example, two-level minimization is one of the first steps taken in the SOCRATES system (deGeu 85).

A substantial amount of work has been devoted to developing methods for producing minimal or near-minimal two-level designs. A number of heuristic minimization techniques have been developed over the past twenty years for producing near-minimal designs. The two most notable examples are MINI (Hong 74) and ESPRESSO (Brayt 84). However, it is always desirable to develop a minimal design rather than a near-minimal one, if it is practical to do so. Unfortunately, the heuristic methods do not guarantee minimality of the resulting design,

In addition to work devoted to developing heuristic methods, there is a long line of research focused on techniques for producing minimal two-level designs. Quine (Quine 52, Quine 55, Quine 59) performed some of the earliest work on the minimization of logical formulas; his techniques were adapted by others for use in producing minimal digital designs. The methods fashioned from Quine's work use rudimentary Boolean algebraic concepts to develop minimal SOP formulas. Other methods developed over the years based on simple algebraic ideas include those developed by Ghazala (Ghaza 57), Chang and Mott (Mott 60, Chang 65), Tison (Tison 67), Reusch (Reusc 75), Cutler (Cutle 80, Cutle 87), and Hong (Hong 83, Hong 91). Most of these techniques are impractical for complex problems, although the techniques developed individually by Cutler and Hong are a significant improvement over the foregoing methods. Their methods incorporate the use of a branch-and-bound search process in the development of minimal formulas.

The most recent efforts devoted to the development of minimal two-level designs are the ESPRESSO-EXACT algorithm (Rudel 89) and McBOOLE (Dagen 86). The ESPRESSO-EXACT algorithm is theoretically similar to the ESPRESSO heuristic minimization technique. Several of the operations used in ESPRESSO, a tautology-based algorithm in particular, are extended for use in ESPRESSO-EXACT. A branch-and-bound search process is used in the final step of constructing a minimal formula. A directed graph called a *covering graph* is used in the McBOOLE algorithm for determining the relationships among prime implicants of a function. Techniques are provided for determining prime implicants to retain or discard based on the covering graph of a function.

Cycles in the graph preclude the selection of prime implicants to retain and discard; a form of search is used to select prime implicants for instances in which cycles appear in the graph. A graph-partitioning technique is used to decompose the problem. Both the ESPRESSO-EXACT and McBOOLE methods have been demonstrated to be useful for many examples which are intractable using earlier methods.

A common attribute of the aforementioned techniques is that all are constrained in the approach taken in developing minimal formulas. In each method, a minimal vector $\underline{F}$ of formulas is developed in which each formula F_j in $\underline{F}$ is comprised only of input variables (X-variables). Each formula F_j corresponds to a circuit developed for a respective output z_j. Hence, at the conclusion of the design process, a system $Z = \underline{f}(X)$ is developed in which the vector $\underline{F}$ represents the functions in $\underline{f}$. We call this the *conventional* approach to the design process.

There exist design problems which cannot be handled using a conventional approach. One example is the use of circuit outputs, i.e., Z-variables, to produce other circuit outputs. A second example is the development of designs which take advantage of previously-constructed circuits. Suppose a design exists which is represented by the equation $Y = \underline{g}(X)$ and the goal of the current design process is to develop a circuit denoted by $Z = \underline{f}(X)$. Conventional techniques are not able to use the Y-variables to develop a circuit denoted by $Z = \underline{h}(X, Y)$ such that the vector $\underline{H}$ correspond to a lower-cost design than $\underline{F}$. The limitation of the conventional approach is rooted in the absence of a sound theoretical foundation.

A second limitation of conventional methods is that there exist atypical design specifications which are unusable by traditional techniques. For example, suppose it is desired to convert between JK and RST flip-flop types. The information required to make the conversion—in either direction—is expressed by the system

$$\begin{aligned} Q'J + QK' &= S + Q'T + QR'T' \\ 0 &= RS + RT + ST \end{aligned} \tag{1.1}$$

of Boolean equations. Traditional design techniques cannot deal with a specification such as (1.1); it would be useful if a design approach were flexible enough to handle such a specification.

An additional observation based on an examination of existing methods is that *informed search* techniques have not been widely employed in methods developed for producing digital designs. Informed search techniques are often useful for efficiently dealing with complex problems such as minimization. In view of the limitations of conventional approaches to the design problem, the goal of this work is stated in the next section.

Problem Statement

Based on an examination of existing methods, we conclude that there continues to be a need for improved algorithms for logic-circuit minimization, provided that the following considerations are satisfied:

- the new methods must have a firm theoretical basis,
- they must be practical for complex problems, and
- they should be applicable to a greater variety of design problems than are existing techniques.

The primary goal of this research is to develop new theoretically-sound algorithms for producing minimal or near-minimal digital designs for a variety of design problems. The secondary goal is to devise algorithms that are as efficient as possible, and thus practical for complex problems.

Scope

This research considers the minimization of digital circuits at the gate or logic level, focusing on the minimization of two-level combinational circuits. Algorithms are presented to derive minimal combinational circuits; the circuits which are considered may have single or multiple outputs and may be completely or incompletely specified.

The main purpose of this work is to develop algorithms which yield minimal-cost circuits. The secondary goal is to produce efficient methods. Trade-offs are acceptable between the measures of circuit minimality and algorithmic efficiency. For example, a method which produces a minimal-cost design can be expected to be computationally intensive. On the other hand, a method which produces a design quickly will generally not produce a minimal design.

Assumptions

The following assumptions are made in this research:

- Any changes in circuit structure are acceptable as long as the functional relationships between inputs and outputs are maintained and design criteria are met. It is inherent in the minimization process that the circuit structure is modified.
- Cost criteria based on the number of components in a circuit are suitable for evaluating digital designs. This assumption facilitates the ability to extrapolate the cost of an SOP formula to the corresponding two-level design.
- All circuit components are considered to be "ideal"; hence, there is assumed to be no variation among like components. All components have a unit delay. Actual components, even if they perform the same function, have variances in parameter-values. These variations are ignored to allow the concentration of effort on the logical aspects of the design.

Approach

In accomplishing the goals of this work, we advance the state of minimization theory by applying the concepts of Boolean reasoning and informed search to develop new algorithms for producing minimal SOP formulas. Boolean reasoning is a methodology by which systems of Boolean equations are reduced to a single Boolean equation; the single equation is an abstraction which enables a reasoning process "independent of the form of the original equations (Brown 90:x)". This

work systematically applies the concepts of Boolean reasoning in a coherent, uniform approach to the minimization problem. Informed search techniques are generally employed in the field of Artificial Intelligence (AI); such methods are characterized by the use of *heuristic functions* which estimate how close we are to a minimal solution at a given point in the search process. Informed search techniques are incorporated as a mechanism for quickly deriving minimal or near-minimal formulas. Minimal SOP formulas correspond to minimal two-level designs. In this section, the significant aspects which are unique to the approach developed in the work are highlighted.

Every technique for producing an economical digital design begins with a specification which represents how a circuit is supposed to operate. A useful way of specifying a circuit is with a single Boolean equation

$$\phi(X, Z) = 1 \tag{1.2}$$

called the 1-*normal form*. There are a number of significant advantages to using the 1-normal form rather than traditional ways for specifying a circuit, e.g., a truth table. First, using a 1-normal form specification, the minimization of completely and incompletely-specified functions is handled in a uniform manner.[2] Thus, separate algorithms for developing formulas for completely and incompletely-specified functions are not required as is generally the case in algebraic minimization methodologies, e.g., the methods developed by Cutler (Cutle 80) and Hong (Hong 83). Secondly, the use of the 1-normal form facilitates the handling of design problems which cannot be solved using conventional minimization approaches. Moreover, it is easy to convert from traditional ways for specifying a circuit, e.g., a truth table, to a specification in 1-normal form. The development of a 1-normal form specification $\phi(X, Z) = 1$ is viewed in this work as the beginning step in the development of a digital design.

[2] Completely- and incompletely-specified functions are defined in Chapter 2.

Once a 1-normal form specification $\phi(X, Z) = 1$ is formed for a circuit, the development of a solution of the equation $\phi(X, Z) = 1$ for the vector Z corresponds to developing a design which meets the specification. A particular solution of $\phi(X, Z) = 1$ for the vector Z is a system of the form

$$\begin{aligned} z_1 &= f_1(X) \\ z_2 &= f_2(X) \\ &\vdots \\ z_m &= f_m(X), \end{aligned} \tag{1.3}$$

such that $\phi(X, \underline{f}(X)) = 1$ is an identity. Such a solution is represented compactly by $Z = \underline{f}(X)$. A design corresponds to a vector $\underline{F}$ of formulas representing the functions $\underline{f}(X)$ in a particular solution $Z = \underline{f}(X)$. A minimal design corresponds to a minimal vector $\underline{F}$ of formulas that represent the functions $\underline{f}(X)$. The use of the 1-normal form $\phi(X, Z) = 1$ as a specification and the correspondence of particular solutions $Z = \underline{f}(X)$ of $\phi(X, Z) = 1$ to designs is discussed in Chapter 4.

A particular solution $Z = \underline{f}(X)$ of $\phi(X, Z) = 1$ is developed from a system called a ***general solution*** of $\phi(X, Z) = 1$ for Z, which is a representation of the set of all particular solutions of $\phi(X, Z) = 1$. If $\phi(X, Z) = 1$ is a *tabular* specification (defined in Chapter 4), then a general solution of $\phi(X, Z) = 1$ for Z may be represented by a system of the form

$$\begin{aligned} \alpha_1(X) &\leq z_1 \leq \beta_1(X) \\ \alpha_2(X) &\leq z_2 \leq \beta_2(X) \\ \alpha_3(X) &\leq z_3 \leq \beta_3(X) \\ &\vdots \\ \alpha_m(X) &\leq z_m \leq \beta_m(X). \end{aligned} \tag{1.4}$$

Many conventional methods for producing digital designs begin with a system equivalent to (1.4), although such a system is not formed by solving an equation or *even known to be* a solution of

an equation. The formation of a 1-normal form specification $\phi(X, Z) = 1$ being the first step, the construction of a general solution is considered to be the second step in the formation of a design. The third step, the focus of most minimization techniques, is the development of a particular solution $Z = \underline{f}(X)$ from the general solution; a design corresponds to the vector $\underline{F}$ of formulas representing the functions $\underline{f}(X)$.

An alternative approach to the design problem is to develop a *recurrent general solution* of (1.2), i.e., one having the form

$$\begin{array}{rcccl} \alpha_1(X) & \leq & z_1 & \leq & \beta_1(X) \\ \alpha_2(X, z_1) & \leq & z_2 & \leq & \beta_2(X, z_1) \\ \alpha_3(X, z_1, z_2) & \leq & z_3 & \leq & \beta_3(X, z_1, z_2) \\ & & \vdots & & \\ \alpha_m(X, z_1, \ldots, z_{m-1}) & \leq & z_m & \leq & \beta_m(X, z_1, \ldots, z_{m-1}). \end{array} \tag{1.5}$$

A *recurrent solution*

$$\begin{array}{rcl} z_1 & = & f_1(X) \\ z_2 & = & f_2(X, z_1) \\ z_3 & = & f_3(X, z_1, z_2) \\ & \vdots & \\ z_m & = & f_m(X, z_1, z_2, \ldots, z_{m-1}) \end{array} \tag{1.6}$$

is then developed from the recurrent system (1.5). System (1.6) is denoted by $Z = \underline{f}(X, Z)$ with the understanding that each z_j is dependent only on $z_1, \ldots, z_{j-1}$. A design represented by $Z = \underline{f}(X, Z)$ is called a *recurrent design*. Figure 1.3 depicts a recurrent design. The advantage of a recurrent system such as (1.6) is that a design may be developed in which output signals are used as well as input signals to generate a given output signals. This allows the development in many instances of more economical designs than can be produced using a conventional approach to minimization.

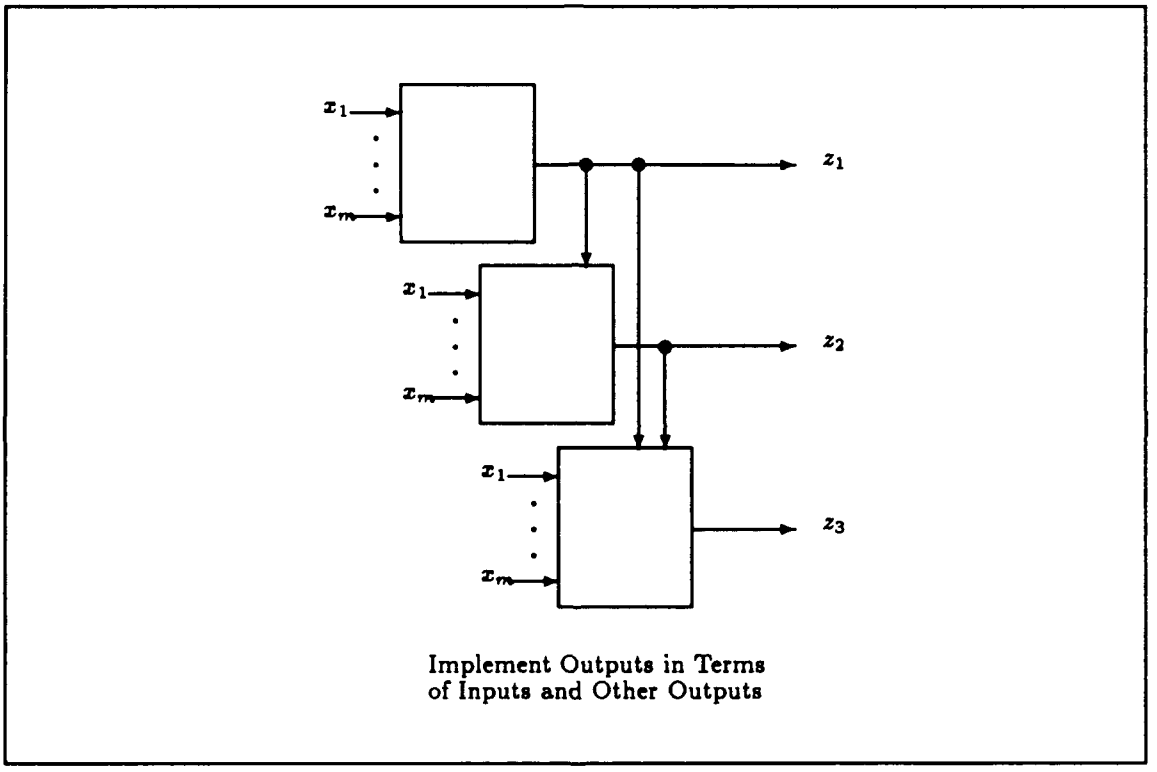


Figure 1.3. A Recurrent Design

The third step in the process of developing a design—the formation of particular solutions from a general solution—is a problem to which much effort is devoted in this work. To develop a minimal two-level design, a minimal vector $\underline{F}$ of SOP formulas is constructed to represent the functions $\underline{f}$ in systems (1.3) and (1.6). The cost criterion used in developing $\underline{F}$ is dependent on the design objective. For example, if the objective is to minimize the total number of gates in a circuit, then a vector $\underline{F}$ of formulas is developed which consists of the fewest terms.

It is well-known that a vector $\underline{F}$ of SOP formulas corresponding to a minimal two-level design consists of *prime implicants* of the functions $\underline{\beta}$ in (1.4) which cover the functions $\underline{\alpha}$. The process used in this work for determining such prime implicants consists of the following steps:

- The set of all prime implicants of $\underline{\beta}$ is formed.
- A subset of the prime implicants of $\underline{\beta}$ called a *base* is developed.
- A set IF of formulas, called *inclusion formulas,* is derived for terms t of the base which denote coverage of each term by subsets of the prime implicants of $\underline{\beta}$. An inclusion formula is a formula such as $P_1' + P_2'P_3'$ in which each literal P_i corresponds to a prime implicant of $\underline{\beta}$; the formula $P_1' + P_2'P_3'$ denotes that a term t may be covered either by the prime implicant corresponding to P_1 or by the combination of the prime implicants denoted by P_2 and P_3.
- The set IF of inclusion formulas is reduced using a set of rules which we call *reduction rules.* The reduction rules identify prime implicants of $\underline{\beta}$ to place in formulas in $\underline{F}$ as well as prime implicants to discard from consideration.
- If the reduction rules do not identify all of the prime implicants of $\underline{\beta}$ that must be contained in the formulas in $\underline{F}$, then a search process is used to identify the remaining prime implicants to place in $\underline{F}$.

These steps are combined with the first two steps—the formation of a 1-normal form specification $\phi(X, Z) = 1$ and the derivation of a general solution of $\phi(X, Z) = 1$ for Z—to form a seven-step

methodology for the development of a minimal vector $\underline{F}$ of formulas corresponding to a minimal design.

For most simple and some moderately complex specifications, all of the prime implicants which are to be contained in formulas in $\underline{F}$ may be identified during the application of reduction rules. In these cases, the last step of the process—search—is not required. Whether this occurs is dependent on the specification itself as well as the base used in the process. For highly complex specifications, search is generally required to form a vector $\underline{F}$ of minimal formulas.

If a set IF of inclusion formulas remains after rule reduction, *informed search* is applied to identify the remaining prime implicants to include in formulas in $\underline{F}$. One type of informed search used in this work is called A*-search. Under certain conditions, A* guarantees a least-cost solution. Additionally, a problem-decomposition strategy is applied which uses a *graph partitioning* technique for breaking up the search problem into smaller pieces, each of which is much easier to handle than the global search problem. When a solution is desired quickly, search techniques are used which quickly yield a near-minimal set of prime implicants to include in formulas of $\underline{F}$.

The theoretical foundation for techniques used to develop inclusion formulas is presented in the Implication Relations section of Chapter 3. These techniques are an example of the utility of the concepts of Boolean reasoning. The application of the seven-step methodology, with the exception of the search process, for developing a minimal formula F corresponding to single-output design is presented in Chapter 6. The formation of a minimal vector $\underline{F}$ for multiple-output circuits with tabular specifications is introduced in Chapter 7. The search process, which is required for many problems, is discussed in Chapter 9. Finally, the approach used to develop recurrent circuits is outlined in Chapter 10.

Overview

This work is organized in three parts, together with introductory and concluding chapters. Chapters 2 through 7 form Part I. Contained in these chapters are the Boolean reasoning concepts used in this work. Part II, consisting of Chapters 8 and 9, discusses the use of search in the minimisation process. Chapter 10, Alternative Minimization Techniques, forms the third segment of the work.

This chapter has provided the background and motivation of this project as well as a definition of the problem. The scope of the effort was presented, as well as the assumptions found to be necessary. A general approach to the solution of the problem was outlined with specific emphasis on the aspects of the methodology used in this work which distinguish it from conventional approaches.

The concepts of Boolean reasoning are presented in Chapter 2, which provides the mathematical foundation on which many of the ideas developed throughout this work are based. The purpose of Chapter 2 is *fourfold:*

1. to present terminology used throughout this work;
2. to familiarize the reader with concepts and operations that are generally not well-known;
3. to describe the underlying principles of techniques that provide the building blocks for algorithms presented in later chapters; and
4. to present a number of new procedures developed in the course of this work.

Chapter 2 may be skipped by a reader who is primarily interested in the minimization techniques presented in later chapters, and then referred to only as required.

In Chapter 3, a unified set of techniques based on the Boolean-reasoning concepts of reduction and elimination is introduced for deducing specific relationships among subsets of a set of Boolean functions. Procedures are presented for the determination of normal and evanescent subsets of a set of functions. Additionally, techniques are presented for the deduction of implication relations, i.e., the coverage of a Boolean function by subsets of a set of functions. Chapter 3 is primarily of

theoretical interest with the exception of a method which is introduced for deducing the coverage of a term by subsets of a set of terms—a specialization of the method for determining implication relations. Variations of the technique for determining the coverage of a term are used in subsequent chapters. An understanding of Chapter 3 is not required for a reader whose only interest is in the minimization methods found in later chapters.

The solutions of Boolean equations, the modeling of digital circuits with Boolean algebra, and the relationship between solving Boolean equations and the minimization problem are discussed in Chapter 4. The correspondence between developing a good solution for a Boolean equation and the process of developing an economical digital design which meets a specification is highlighted. Concepts developed in this chapter are employed in later chapters. A comprehension of the second and third sections of Chapter 4 is particularly important to understanding the approach taken to the minimization problem in this work.

In Chapters 5 and 6, the development of SOP formulas F to represent functions f belonging to the interval $[g, h]$ is presented. We initially focus on this case in order to clearly develop the basic strategy used throughout this work. In Chapter 5, methods are introduced for generating all irredundant SOP formulas which represent functions belonging to the interval $[g, h]$. However, since a function may be represented by thousands or even millions of irredundant formulas, it is often not feasible to form all irredundant formulas. In these cases, we endeavor to find a single minimal formula F which represents a function f in the interval $[g, h]$. Techniques for finding a single minimal formula are presented in Chapter 6. In both chapters we concentrate on providing a firm theoretical foundation for procedures while striving to incorporate concepts which reduce computational effort. Chapter 5 may be skipped by the reader who is only interested in methods for developing a single minimal SOP formula.

The techniques presented in Chapter 6 for single-output circuits are extended to multiple-output circuits in Chapter 7. Algorithms described in Chapter 7 are restricted to development of circuits with tabular specifications.

Chapters 8 and 9 form the second part of this work. Chapter 8 is primarily an introductory chapter which gives an overview of the artificial intelligence concepts used in this research. It is provided for those who may be unfamiliar with the concepts of informed search. Background material is presented on search strategies. The use of state spaces and heuristic functions during search is discussed. Chapter 8 may be skipped by a reader who has a basic understanding of search techniques. In Chapter 9 the application of search in this work is presented. Heuristics developed for use in the search process are discussed. Additionally, a decomposition strategy for breaking up the search problem is introduced.

Chapter 10 is the third major section of this work. In this chapter a number of approaches are introduced for forming minimal or near-minimal circuit designs which cannot be arrived at using conventional minimization techniques. A procedure is presented for the generation of cascade circuits, circuits in which a number of circuit outputs are used in the same manner as circuit inputs (loops are prohibited). The resulting circuits are typically smaller than circuits formed using conventional techniques. A second technique presented in this chapter is the formation of minimal designs which take advantage of previously-constructed subcircuits.

Chapter 11 presents a summary of this effort. An assessment is made of the algorithms presented in this work. The conclusions of this effort are stated. Recommendations for future work are elaborated.

Previous research efforts in two-level logic circuit minimization are described in Appendix A. Terms and concepts are presented as well as a discussion of the efficacy of previous methods.

Appendix B, Example Functions, contains descriptions of several sets of functions to which we have applied procedures introduced in this work. The computational results of applying algorithms to the functions listed in Appendix B are presented in Appendix C, Computational Results.

The format for procedures and algorithms found in the text is given in the first section of Appendix D. We make a distinction between "procedures" and "algorithms" in this work. ***Procedures*** are simple techniques that are used as the "building blocks" for larger methods. We designate as ***algorithms*** the methods used to produce minimal digital designs. In this distinction, procedures compose algorithms. Procedures and algorithms are written in a manner that should facilitate easy computer implementation. Additionally, because Chapters 2 and 8 are primarily background chapters, the procedures for which the theoretical basis is described in Chapters 2 and 8 are found only in Appendix D.

II. Fundamentals of Boolean Reasoning

The algorithms presented in this dissertation are based on the concepts of Boolean reasoning. Because these concepts are not well-known, the fundamentals of Boolean reasoning are presented in this chapter to facilitate comprehension of the methods developed in this work. The theory and terminology presented in this chapter provide the foundation for later chapters. However, one whose only interest is in the minimization techniques presented in later chapters may skip this chapter and refer to it only as required for understanding subsequent chapters. Sources for much of the terminology and notation found in this chapter include (Johns 87), (Murog 79), (Lipsc 76), and (Nagle 75). For a more in-depth coverage of Boolean reasoning, see (Brown 90).

This chapter provides the theoretical background for a number of Boolean operations. Algorithms and heuristics used to perform such operations are described; for the sake of brevity, however, procedures which implement specific operations in a step-by-step fashion are located in Appendix D.

Many researchers have worked to devise techniques which are better in some way than previously-existing methods. Developments include algorithms which produce better results, the introduction of heuristics, and more efficient implementations of the algorithms. For example, Brayton et al. report algorithms for a number of Boolean operations; among their significant developments is a heuristic which greatly improves the efficiency of several operations and a method for obtaining improved results (Brayt 82). Brown presents algorithms for many techniques in Boolean reasoning in (Brown 90). An outcome of this dissertation is the development of several new algorithms or heuristics for Boolean operations such that the resulting implementations yield better results faster than do existing methods. A number of new developments are presented in this chapter; these are highlighted in the summary of the chapter.

Basic Concepts of Boolean Algebra

Definitions. An algebra is characterized by three components:

1. A set, called a *carrier*,
2. *Operations* defined on the carrier, and
3. Distinct members of the carrier which are called *constants* of the algebra.

In addition to these components, an algebra has associated *axioms*. A *closed algebraic system* is governed by the Law of Substitution which states that two expressions are said to be equal if one can be replaced by the other.

A *Boolean algebra* is a closed algebraic system denoted by the quintuple

$$< \mathbf{B}, +, \cdot, 0, 1 > \tag{2.1}$$

where

- **B** is the carrier of the algebra,
- $+$ and $\cdot$ are binary operations defined on **B**, and
- 0 and 1 are the constants of **B**.

The operator $\cdot$ is called AND. An expression of the form $a \cdot b$ is called a *conjunction.*

The operator $+$ is called OR. An expression of the form $a + b$ is called a *disjunction.*

The $*$ symbol is often used in lieu of the $\cdot$ symbol. Additionally, $a \cdot b$ may be replaced by the juxtaposition ab for simplicity.

Axioms. A Boolean algebra is based on a set of axioms known as Huntington's Postulates (Hunti 04). These axioms are:

1. **Commutative Laws.** *For all* $a, b \in \mathbf{B}$,

$$a + b = b + a \tag{2.2}$$
$$a \cdot b = b \cdot a. \tag{2.3}$$

2. **Distributive Laws.** *For all* $a, b, c \in \mathbf{B}$,

$$a + (b \cdot c) = (a + b) \cdot (a + c) \tag{2.4}$$
$$a \cdot (b + c) = (a \cdot b) + (a \cdot c). \tag{2.5}$$

3. **Identities.** *For all* $a \in \mathbf{B}$,

$$0 + a = a \tag{2.6}$$
$$1 \cdot a = a. \tag{2.7}$$

0 is the identity for the + operator. 1 is the identity for the · operator.

4. **Complements.** *For every* $a \in \mathbf{B}$, *there exists an* $a' \in \mathbf{B}$ *such that*

$$a + a' = 1 \tag{2.8}$$
$$a \cdot a' = 0. \tag{2.9}$$

The " ′ " symbol denotes *complementation.* It can be shown that a' is unique.

Boolean algebras are governed by the *principle of duality* by which a given valid expression has an associated valid dual expression. The dual of an expression is found by interchanging all + and · operators and exchanging identity elements 0 and 1; additionally, the left and right sides of

inclusions (defined in the next section) are interchanged. Note that each of the preceding postulates has two expressions; these expressions are duals of each other.

The Inclusion Relation. The *inclusion relation*, $\leq$, is defined as follows. For all $a, b \in \mathbf{B}$

$$a \leq b \quad \Leftrightarrow \quad a \cdot b' = 0. \tag{2.10}$$

(Rudea 74:8)

Statement (2.10) is isomorphic to the following property in the algebra of subsets of a set:

$$A \subseteq B \quad \Leftrightarrow \quad A \cap B' = \emptyset, \tag{2.11}$$

where A and B are arbitrary classes, i.e., subsets of a universal set S. The relation $A \cap B' = \emptyset$ is best visualized by use of the Euler diagram given in Figure 2.1.

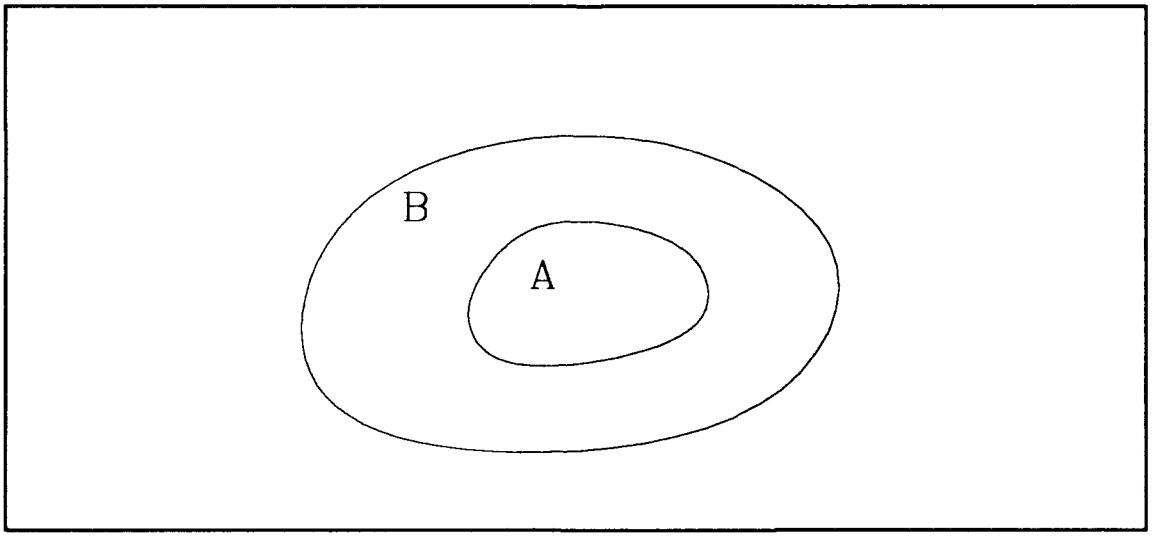


Figure 2.1. Euler Diagram of $A \cap B' = \emptyset$

The following statements are equivalent:

$$a \leq b \tag{2.12}$$
$$ab' = 0 \tag{2.13}$$
$$a' + b = 1 \tag{2.14}$$
$$b' \leq a' \tag{2.15}$$
$$a + b = b \tag{2.16}$$
$$ab = a. \tag{2.17}$$

Intervals. Let a and b be members of a Boolean algebra $\mathbf{B}$, and assume that $a \leq b$. The *interval* (also called a *segment*) $[a, b]$ is the set of elements of $\mathbf{B}$ lying between a and b, i.e.,

$$[a, b] = \{x | x \in \mathbf{B} \quad \text{and} \quad a \leq x \leq b\} \tag{2.18}$$

The element a is called the lower bound of x; b is called the upper bound of x.

Theorems. Theorems which can be proven from the axioms and the definition of the inclusion relation are:

1. **Associativity.** *For all* $a, b, c \in \mathbf{B}$,

$$a + (b + c) = (a + b) + c \tag{2.19}$$
$$a \cdot (b \cdot c) = (a \cdot b) \cdot c. \tag{2.20}$$

2. **Idempotence.** *For all* $a \in \mathbf{B}$,

$$a + a = a \tag{2.21}$$
$$a \cdot a = a. \tag{2.22}$$

3. **Boundedness.** *For all* $a \in \mathbf{B}$,

$$a + 1 = 1 \tag{2.23}$$
$$a \cdot 0 = 0. \tag{2.24}$$

4. **Absorption.** *For all* $a, b \in \mathbf{B}$,

$$a + (a \cdot b) = a \tag{2.25}$$
$$a \cdot (a + b) = a. \tag{2.26}$$

5. **Involution.** *For all* $a \in \mathbf{B}$,

$$(a')' = a. \tag{2.27}$$

6. **DeMorgan's Laws.** *For all* $a, b \in \mathbf{B}$,

$$(a + b)' = a' \cdot b' \tag{2.28}$$
$$(a \cdot b)' = a' + b'. \tag{2.29}$$

7. *For all* $a, b \in \mathbf{B}$,

$$a + (a' \cdot b) = a + b \tag{2.30}$$
$$a \cdot (a' + b) = a \cdot b. \tag{2.31}$$

8. **Consensus.** *For all* $a, b, c \in \mathbf{B}$,

$$(a \cdot b) + (a' \cdot c) + (b \cdot c) = (a \cdot b) + (a' \cdot c) \tag{2.32}$$
$$(a + b) \cdot (a' + c) \cdot (b + c) = (a + b) \cdot (a' + c). \tag{2.33}$$

9. **Interchange.** *For all* $a, b, c \in \mathbf{B}$,

$$(a \cdot b) + (a' \cdot c) = (a + c) \cdot (a' + b) \tag{2.34}$$
$$(a + b) \cdot (a' + c) = (a \cdot c) + (a' \cdot b). \tag{2.35}$$

10. *For all* $a, b \in \mathbf{B}$,

$$a \leq a + b \tag{2.36}$$
$$a \cdot b \leq a. \tag{2.37}$$

Equivalent Boolean Equations. It is often useful to express a Boolean equation in an equivalent expression of the form $f = 0$ or $g = 1$. The following properties can be proven from the Boolean axioms and theorems.

1. An arbitrary Boolean equation is equivalently expressed by the form $f = 0$ using the following equivalence:

$$a = b \iff a' \cdot b + a \cdot b' = 0 \tag{2.38}$$

   $(a' \cdot b + a \cdot b')$ is the Exclusive-OR of a and b and is denoted by either $(a \oplus b)$ or a XOR b.

2. An arbitrary Boolean equation is equivalently expressed by the form $g = 1$ using the following equivalence:

$$a = b \iff a' \cdot b' + a \cdot b = 1. \tag{2.39}$$

   $(a' \cdot b' + a \cdot b)$ is the Exclusive-NOR of a and b and is denoted be either $(a \odot b)$ or a XNOR b.

3. Systems of Boolean equations are equivalently expressed by a single equation using the following equivalences:

$$a = 0 \;\&\; b = 0 \iff a + b = 0 \tag{2.40}$$
$$a = 1 \;\&\; b = 1 \iff a \cdot b = 1. \tag{2.41}$$

Theorem Involving the Inclusion Relation. The theorem in this section, which involves the inclusion relation $\leq$, is useful in various applications of Boolean reasoning.

Theorem 2.1: *For all elements a, b, c, and d of a Boolean algebra* **B**, *if*

$$\begin{aligned} a &\leq b \\ c &\leq d \end{aligned} \tag{2.42}$$

then

$$a + c \leq b + d. \tag{2.43}$$

Proof. Suppose (2.42) to hold. Then we conclude by the definition of the inclusion relation that

$$\begin{aligned} ab' &= 0, \\ cd' &= 0. \end{aligned} \tag{2.44}$$

In view of (2.24), the validity of the following statements may be asserted:

$$\begin{aligned} ab'd' &= 0, \\ b'cd' &= 0. \end{aligned} \tag{2.45}$$

Whence, the equation

$$ab'd' + cb'd' = 0 \tag{2.46}$$

is valid. The following statements are equivalent:

$$\begin{aligned} ab'd' + cb'd' &= 0 && (2.47)\\ (a+c)b'd' &= 0 && (2.48)\\ a+c &\leq (b'd')' && (2.49)\\ a+c &\leq b+d. && (2.50) \end{aligned}$$

Statements (2.47) and (2.48) are equivalent by distributivity. The definition of the inclusion relation is used to form (2.49) from (2.48). Finally, DeMorgan's law and involution are used to form (2.50). This completes the proof. □

When there are an arbitrary number of statements of the form (2.42), Theorem 2.1 is applied repeatedly to develop an inclusion such as (2.43).

Definition of Subtraction. The operation $a - b$ is defined as the portion of a that is not covered by b, i.e.,

$$a - b = a \cdot b'. \tag{2.51}$$

The operation $a - b$ is sometimes called the *relative complement of b with respect to a*. In set theory, the relative complement of a set B with respect to a set A, denoted $A - B$, is also called the *difference* of A and B. However, the term "difference" has been expropriated in engineering literature for another Boolean operation. Hence, we follow Brayton (Brayt 82) in calling $a - b$ the *subtraction* of b from a.

Literals, Terms, and Formulas

Literals and Terms. A *literal* is a letter or complemented letter such as a, b, a', b', x, and x', where a letter denotes a variable or a member of a Boolean algebra. (The convention used in this work is that letters such as $a, b, c, \ldots$ are members of a Boolean algebra and variables are represented by letters such as $\ldots, x, y, z$.) A *term* is 1, a literal, or a conjunction of two or more literals in which no two literals involve the same letter. Examples of terms include $ab'x, ac$, and $x'yz'$. An *alterm* is 0, a literal, or a disjunction of literals in which no two literals involve the same letter. Examples include $(a + y), (a + c')$, and $(x + y + z')$.

Boolean Formulas. The set of n-variable *Boolean formulas* on n symbols $x_1, \ldots, x_n$ is defined by the following rules:

1. The elements of **B** are Boolean formulas, and
2. The symbols $x_1, \ldots, x_n$ are Boolean formulas, and
3. If f and g are Boolean formulas, then so are
   (a) $f + g$,
   (b) $f \cdot g$,
   (c) f', and
4. A string is a Boolean formula if and only if it is formed by a finite number of applications of the first three rules.

Examples of formulas include $a, x', a + y$, and $(x \cdot (b' + z))' + a$.

A *sum-of-products* formula is 0, a single term, or a disjunction of terms. A *product-of-sums* formula is 1, a single alterm, or a conjunction of alterms. A sum-of-products formula is often called an SOP formula; a product-of-sums formula is called a POS formula. Unless noted otherwise, the type of formula referred to throughout this work is a sum-of-products formula rather than a formula of arbitrary form.

Unate and Binate Variables. If a variable appears in both complemented and uncomplemented form in the same formula, the variable is said to be *opposed* within the formula. If the variable appears in only uncomplemented or only complemented form, then the variable is said to be *unopposed* in the formula.

If a variable is unopposed in a formula, the variable is called a *unate* or *monoform* variable. If a variable is opposed in a formula, then the variable is called a *binate* or *biform* variable. If a variable is unate and appears in only uncomplemented form in a formula, then it is called a *positive* variable. If a variable is unate and appears in only complemented form in a formula, then it is called a *negative* variable.

Boolean Functions

General Case. An n-variable *Boolean function*, $f : \mathbf{B}^n \rightarrow \mathbf{B}$, is the mapping associated with an n-variable Boolean formula. Rudeanu, in his work on Boolean functions and equations, gives an informal definition of a Boolean function:

> Roughly speaking, a *Boolean function* (also called *Boolean polynomial* by certain authors) is a function with arguments and values in a Boolean algebra $\mathbf{B}$, such that f can be obtained from variables and constants of $\mathbf{B}$ by superpositions of the basic operations $+, \cdot$, and $'$ of $\mathbf{B}$. (Rudea 74:16)

A more formal definition of the set of n-variable Boolean functions parallels the definition of a Boolean formula:

1. For any element $a \in \mathbf{B}$, a *constant function* $f : \mathbf{B}^n \rightarrow \mathbf{B}$ defined by
$$f(x_1, \ldots, x_n) = a \qquad \forall (x_1, \ldots, x_n) \in \mathbf{B}^n \tag{2.52}$$
is an n-variable Boolean function.
2. For n variables, $x_1, \ldots, x_n$, a *projection function* $f : \mathbf{B}^n \rightarrow \mathbf{B}$ defined by
$$f(x_1, \ldots, x_n) = x_i \qquad \forall (x_1, \ldots, x_n) \in \mathbf{B}^n, \qquad i \in \{1 \ldots n\}, \tag{2.53}$$

is an n-variable Boolean function.

3. If $g, h : \mathbf{B}^n \rightarrow \mathbf{B}$ are n-variable Boolean functions, then the functions $g+h$, gh, and g' defined by

   (a)
   $$(g+h)(x_1, \ldots, x_n) = g(x_1, \ldots, x_n) + h(x_1, \ldots, x_n) \tag{2.54}$$

   (b)
   $$gh(x_1, \ldots, x_n) = g(x_1, \ldots, x_n)\, h(x_1, \ldots, x_n) \tag{2.55}$$

   (c)
   $$g'(x_1, \ldots, x_n) = (g(x_1, \ldots, x_n))' \tag{2.56}$$

   $\forall (x_1, \ldots, x_n) \in \mathbf{B}^n$, are also n-variable Boolean functions.

4. A function is a Boolean function if and only if it is formed by a finite number of applications of the first three rules. (Rudea 74:17)

Switching Functions. Rudeanu makes a clear distinction between Boolean functions in the general case, and the special case of Boolean functions involving no constants except 0 and 1 which he calls *simple Boolean functions* (Rudea 74:xvi). One form of simple Boolean functions is that involving the two-element Boolean algebra $\mathbf{B}_2 = \{0, 1\}$. Rudeanu states:

> In the particular case of the two-element Boolean algebra $\mathbf{B}_2 = \{0, 1\}$, every function $f : \mathbf{B}_2^n \rightarrow \mathbf{B}_2$ is a simple Boolean function and will be termed a *truth function* (also called a "switching function" or "Boolean function" by switching theorists ...) (Rudea 74:xvi)

Thus, a switching function is a Boolean function for which $\mathbf{B}_2 = \{0, 1\}$.

Representations of Boolean Functions. Every n-variable Boolean formula maps into a corresponding n-variable Boolean function; the function is said to be *represented* by the formula. A function, $f : \mathbf{B}^n \rightarrow \mathbf{B}$, is a Boolean function if and only if it can be represented by a Boolean formula. A Boolean formula which represents the function f is referred to by the capitalised form of the symbol, i.e., F.

Equivalent Formulas. The number of n-variable Boolean formulas which represent an n-variable Boolean function is infinite. Formulas that represent the same function are called

equivalent formulas. An important task—the central theme of this work—is to determine a "good" formula to represent a function. Typical metrics for determining the "goodness" of a formula include the number of terms in the formula and the number of literals in the formula.

Congruent Formulas. A special case of equivalent formulas is that of congruent formulas. Two SOP formulas are called *congruent* "in case one can be transformed into the other using only the commutative law (Brown 90:239)". The formulas

$$a'b'c + cde + efg'$$
$$edc + g'ef + ca'b'$$

are congruent.

Absorptive Formulas. A term t is said to *absorb* a second term u if the first term consists of a subset of the literals of the second term. In such a case, the inclusion

$$u \leq t. \tag{2.57}$$

is valid. To prove (2.57), it is observed that u can be reformulated as $u_1 \cdot u_2$, in which u_1 consists of all of the literals in t, and where

- u_2 consists of all of the literals not in t, or
- u_2 is defined to be 1 in case $u = t$.

Hence, $u = t \cdot u_2$. By (2.37), $t \cdot u_2 \leq t$. Replacing $t \cdot u_2$ by u proves (2.57).

An SOP formula F is said to be *absorptive* in case no term in F is absorbed by any other term in F. If one term of a formula absorbs a second term, (2.25) shows that an equivalent formula is derived by deleting the absorbed term. An absorptive formula that is equivalent to F is obtained

by successive deletion of all terms absorbed by other terms in F; the resulting formula is denoted by $ABS(F)$. The formula $ABS(F)$ is unique to within congruence (see (Brown 90:240) for proof).

Unate Functions. A function $f(x_1, \ldots, x_n)$ is said to be *positive in* a variable x_i if the function can be represented by a formula in which x_i is positive. Similarly, a function $f(x_1, \ldots, x_n)$ is said to be *negative in* a variable x_i if the function can be represented by a formula in which x_i is negative. If f is either positive or negative in x_i, then the function is said to be *unate in* x_i. If a function $f(x_1, \ldots, x_n)$ is unate in all of its variables $x_1, \ldots, x_n$, then it is called a *unate* function.

Unate functions are an important class of functions because many operations that may be performed on the general case of Boolean functions are performed more efficiently on unate functions. For example, given a formula F which is unate in all of its variables, it can be shown that the minimum-term equivalent formula for F is $ABS(F)$.

Two unate Boolean functions f and g are said to be *collectively unate* if each is unate and $f + g$ is unate.

Orthogonal Functions. A set $\{\phi_1, \phi_2, \ldots, \phi_k\}$ of n-variable Boolean functions is called *orthogonal* if the condition

$$\phi_i \cdot \phi_j = 0 \qquad i \neq j \tag{2.58}$$

is satisfied. If the condition

$$\sum_{i=1}^{k} \phi_i = 1 \tag{2.59}$$

is satisfied, then the set is called *normal*. If both (2.58) and (2.59) are satisfied, then the set of functions is called *orthonormal*.

Evanescent Functions. A set $\{\phi_1, \phi_2, \ldots, \phi_k\}$ of n-variable Boolean functions is called *evanescent* if the following condition is valid:

$$\phi_i \cdot \phi_2 \cdots \phi_k = 0. \tag{2.60}$$

Boole's Expansion Theorem

The most important functional theorem in Boolean algebra is *Boole's Expansion Theorem.* This theorem is commonly—but not properly—attributed to Shannon (Shann 49) in most current-day texts on switching theory. It is stated as follows:

If f is an n-variable Boolean function, then f has the expansion

$$f(x_1, x_2, \ldots, x_n) = x_1' f(0, x_2, \ldots, x_n) + x_1 f(1, x_2, \ldots, x_n), \tag{2.61}$$

$\forall (x_1, \ldots, x_n) \in \mathbf{B}^n$, (Boole 54).

The dual of (2.61) is given by the expansion

$$f(x_1, x_2, \ldots, x_n) = [x_1' + f(1, x_2, \ldots, x_n)][x_1 + f(0, x_2, \ldots, x_n)], \quad \forall (x_1, \ldots, x_n) \in \mathbf{B}^n. \tag{2.62}$$

This theorem is used as the basis for efficient operations on Boolean functions (see (Brown 90) or (Brayt 82)). The variable x_i about which a function f is expanded is called the *splitting variable* (Brayt 82:59). In most cases Boolean expansion is recursively applied to derive a result. When f is expanded about a variable, two functions are derived which do not involve the variable; the new functions derived are called *leaf* functions. For example, if a 3-variable Boolean function $f(x, y, z)$ is expanded about x, the statement

$$f(x, y, z) = x' f(0, y, z) + x f(1, y, z) \tag{2.63}$$

is derived in which $f(0,y,z)$ and $f(1,y,z)$ are functions which do not involve x. In this case, x is the splitting variable, and $f(0,y,z)$ and $f(1,y,z)$ are the leaf functions. The functions $f(0,y,z)$ and $f(1,y,z)$ can then be expanded further about either y or z.

Expansion is often performed about each variable in turn until one or both of the resulting leaf functions possess a specific property, e.g., are unate or are elements of the underlying Boolean algebra **B**. Of particular concern is the efficiency by which the expansion process occurs. In most cases a judicious choice of splitting variable at each stage of the expansion process will greatly affect the amount of recursion which takes place as well as the simplicity of the formulas which represent the leaf functions. For many applications of the expansion theorem, heuristics have been devised to wisely choose the splitting variable at each stage of expansion; these heuristics—several of which are reported for the first time in this work—will be discussed in later sections.

Minterm and Maxterm Canonical Forms

It is often desirable to use a restricted class of formula in which any Boolean function has only one corresponding formula. Formulas in such classes are called *canonical forms*. Two canonical forms generated using Boole's Expansion Theorem include the *minterm canonical form* and the *maxterm canonical form*.

Minterm Canonical Form. If Boole's expansion theorem is recursively applied to a 3-variable Boolean function $f(x,y,z)$ about variables x, y, and z, one derives

$$\begin{aligned} f(x,y,z) &= x'f(0,y,z) + xf(1,y,z) \\ &= x'[y'f(0,0,z) + yf(0,1,z)] + x[y'f(1,0,z) + yf(1,1,z)] \\ &= x'y'z'f(0,0,0) + x'y'zf(0,0,1) \\ &\quad + x'yz'f(0,1,0) + x'yzf(0,1,1) \\ &\quad + xy'z'f(1,0,0) + xy'zf(1,0,1) \\ &\quad + xyz'f(1,1,0) + xyzf(1,1,1). \end{aligned} \tag{2.64}$$

The values

$$f(0,0,0), f(0,0,1), f(0,1,0), \ldots, f(1,1,1) \tag{2.65}$$

are elements of B called the *discriminants* of the function f. The products

$$x'y'z', x'y'z, x'yz', \ldots, xyz \tag{2.66}$$

consisting of all of the variables of $f(x,y,z)$ in each term of the right side of (2.64) are called the *minterms* of $X = (x,y,z)$.

This concept may be extended for an arbitrary n-variable Boolean function. The function $f(x_1, \ldots, x_n)$ is expanded as follows:

$$\begin{aligned} f(x_1, \ldots, x_n) \;=\; & f(0,\ldots,0,0)x_1' \cdots x_{n-1}'x_n' \\ & + f(0,\ldots,0,1)x_1' \cdots x_{n-1}'x_n \\ & \vdots \\ & + f(1,\ldots,1,1)x_1 \cdots x_{n-1}x_n. \end{aligned} \tag{2.67}$$

The values

$$f(0,\ldots,0,0), f(0,\ldots,0,1), \ldots, f(1,\ldots,1,1) \tag{2.68}$$

are the discriminants of f, and the terms

$$x_1' \cdots x_{n-1}'x_n', x_1' \cdots x_{n-1}'x_n, \ldots, x_1 \cdots x_{n-1}x_n \tag{2.69}$$

are the minterms of $X = (x_1, \ldots, x_n)$. The expansion (2.67) is called the *minterm canonical form* of f and is designated by $MCF(f)$. The minterm canonical form also is called the *canonical sum-of-products form* or *full disjunctive normal form.*

An important result is that a function f is a Boolean function if and only if it can be expressed in minterm canonical form (see (Brown 90:40) for proof).

Minterm Canonical Form for Switching Functions. In the special case of an n-variable switching function f, its minterm canonical form is an SOP formula in which all of the terms are minterms. In this instance, a minterm appears as a term in the formula if and only if the corresponding discriminant has the value of 1. There exist 2^n possible minterms in a formula which represents an n-variable switching function; thus, there are 2^{2^n} possible switching functions of n variables.

Example 2.1: Given the three-variable Boolean function $f : \mathbf{B}_2^3 \rightarrow \mathbf{B}_2$ represented by the formula $xyz + x'z'$, the following minterm canonical form represents f:

$$x'y'z' + x'yz' + xyz. \tag{2.70}$$

Often, a shorthand notation is used to represent a minterm. One such form is m_i, where i is the decimal integer of the binary code for the minterm. The shorthand notation for three-variable minterms is given in Table 2.1.

Using this notation, the formula in Example 2.1 can be written as $f(x, y, z) = m_0 + m_2 + m_7$. This notation can be shortened further to *minterm list form,* i.e., $f(x, y, z) = \sum m(0, 2, 7)$.

Truth Tables. A *function table* or *truth table* is often used to specify a function. A function table is generated for an n-variable Boolean function by substituting all possible combinations in $\mathbf{B}^n$ for the variables in a formula which represents the function. For every element

Term	Binary Code	Shorthand Notation
$x'y'z'$	0 0 0	m_0
$x'y'z$	0 0 1	m_1
$x'yz'$	0 1 0	m_2
$x'yz$	0 1 1	m_3
$xy'z'$	1 0 0	m_4
$xy'z$	1 0 1	m_5
xyz'	1 1 0	m_6
xyz	1 1 1	m_7

Table 2.1. Shorthand Notation for Minterms

$(a_1, \ldots, a_n) \in \mathbf{B}^n$, the function table displays $f(a_1, \ldots, a_n)$. A truth table is generated for a Boolean function by substituting all $0, 1$ combinations for the variables in a formula which represents the function. A truth table is a proper subset of a function table, except in the case of the two-element Boolean algebra, $\mathbf{B} = \{0, 1\}$. In this case a truth table is identical to a function table.

Example 2.2: Given the two-element Boolean algebra, $\mathbf{B} = \{0, 1\}$, Table 2.2 is the truth table for the three-variable Boolean function $f : \mathbf{B}_2^3 \rightarrow \mathbf{B}_2$ represented by the Boolean formula $xyz + x'z'$.

$x\ y\ z$	$f(x, y, z)$
0 0 0	1
0 0 1	0
0 1 0	1
0 1 1	0
1 0 0	0
1 0 1	0
1 1 0	0
1 1 1	1

Table 2.2. Truth Table for Example 2.2

Just as a Boolean function's truth table may be generated from a Boolean formula, a formula to represent the function can be generated using a truth table. In this case, for each minterm we obtain the corresponding discriminant directly from the truth table. This yields the function's minterm canonical form. Because a truth table may be used to generate the minterm canonical form of a Boolean function, a truth table completely defines a Boolean function.

Example 2.3: Given the truth table in Example 2.2, we construct the minterm canonical form of the function specified by the table:

$$MCF(f) = x'y'z'(1) + x'y'z(0) + x'yz'(1) + x'yz(0) + xy'z'(0) + xy'z(0) + xyz'(0) + xyz(1). \quad (2.71)$$

After simplification by (2.6),(2.7), and (2.24) we form the equation

$$MCF(f) = x'y'z' + x'yz' + xyz. \quad (2.72)$$

Maxterm Canonical Form. The maxterm canonical form is defined analogously to the minterm canonical form. It is generated using the dual form (2.62) of Boole's Expansion Theorem. In some texts the maxterm canonical form is called a *canonical product-of-sums form* or *full conjunctive normal form.*

A *maxterm* is an alterm which contains all variables of the formula either in complemented or uncomplemented form. The *maxterm canonical form* for a switching function f is a product-of-sums formula for f in which all of the alterms are maxterms. In this case, a maxterm appears as an alterm in the formula if and only if the corresponding discriminant has the value of 0. Hence, the maxterm canonical form is analogous to the minterm canonical form.

Example 2.4: The three-variable Boolean function $f : \mathbf{B}_2^3 \rightarrow \mathbf{B}_2$ from Example 2.1 is represented by the following formula in POS form:

$$(x + z')(x' + y)(x' + z). \quad (2.73)$$

This formula has the following formula maxterm canonical form:

$$(x + y + z')(x + y' + z')(x' + y + z)(x' + y + z')(x' + y' + z). \quad (2.74)$$

As with minterms, a shorthand notation is often used to represent maxterms. One such form is M_i, where i is the decimal integer of one's complement of the binary code for the maxterm. The shorthand notation for three-variable maxterms is given in Table 2.3. Using this notation,

Alterm	Binary Code	Shorthand Notation
$x+y+z$	0 0 0	M_0
$x+y+z'$	0 0 1	M_1
$x+y'+z$	0 1 0	M_2
$x+y'+z'$	0 1 1	M_3
$x'+y+z$	1 0 0	M_4
$x'+y+z'$	1 0 1	M_5
$x'+y'+z$	1 1 0	M_6
$x'+y'+z'$	1 1 1	M_7

Table 2.3. Shorthand Notation for Maxterms

the formula in Example 2.4 can be written as $f(x,y,z) = M_1M_3M_4M_5M_6$. This notation can be shortened further to *maxterm list form*, i.e., $f(x,y,z) = \prod M(1,3,4,5,6)$.

Incompletely-Specified Functions

It is often the case that instead of working with a single Boolean function, f, an interval of Boolean functions is specified. This interval of functions is actually a set $\mathcal{F}$ of functions defined by

$$\mathcal{F} = \{\phi \mid g(X) \leq \phi(X) \leq h(X), \quad \forall X \in \mathbf{B}^n\}, \tag{2.75}$$

for which $g : \mathbf{B}^n \rightarrow \mathbf{B}$ and $h : \mathbf{B}^n \rightarrow \mathbf{B}$ are Boolean functions such that $g(X) \leq h(X), \ \forall X \in \mathbf{B}^n$. Since g and h are completely defined by their truth tables, (2.75) may be restated as:

$$\mathcal{F} = \{\phi \mid g(X) \leq \phi(X) \leq h(X), \quad \forall X \in \{0,1\}^n\}. \tag{2.76}$$

Let $\mathcal{S}$ be the set of intervals on $\mathbf{B}$, i.e.,

$$\mathcal{S} = \{[a, b] \mid a \in \mathbf{B}, b \in \mathbf{B}, a \leq b\}. \tag{2.77}$$

We may represent the set $\mathcal{F}$ by the single mapping $f : \mathbf{B}^n \rightarrow \mathcal{S}$, defined as follows:

$$f(X) = [g(X), h(X)], \quad \forall X \in \{0,1\}^n. \tag{2.78}$$

Thus, for each $X \in \{0,1\}^n$, the value of $f(X)$ is $[g(X), h(X)]$ rather than an element of the Boolean algebra. When a mapping is specified in this fashion, f is referred to as an *incompletely-specified* (ICS) Boolean function. A mapping which complies with the earlier stated definition of a Boolean function is called a *completely-specified* (CS) Boolean function. The function g is called the *lower bound* of function f; h is called the *upper bound* of f.

Typically, incompletely-specified Boolean functions are used in switching theory, where the two-element Boolean algebra $\mathbf{B} = \{0,1\}$ is employed. g and h are switching functions for which $g(X) \leq h(X)$, $\forall X \in \{0,1\}^n$. Therefore, for each $X \in \{0,1\}^n$, $f(X)$ is one of the intervals $[0,0]$, $[0,1]$, or $[1,1]$. The normal convention is to rename the intervals $[0,0]$, $[0,1]$, and $[1,1]$ as 0, X, and 1, respectively. The symbol X signifies that either 0 or 1 may be chosen as the value of f; this is called a *don't-care* condition. X is said to be a "don't care" value.

Example 2.5: Given the equations

$$g(x,y,z) = x'y' + xyz \tag{2.79}$$
$$h(x,y,z) = x'y' + xy + yz + x'z, \tag{2.80}$$

the incompletely-specified function $f(x,y,z)$ defined by $g(x,y,z) \leq f(x,y,z) \leq h(x,y,z)$ is given by the truth table in Table 2.4.

$x\ y\ z$	$g(x,y,z)$	$h(x,y,z)$	$f(x,y,z)$	$f(x,y,z)$
0 0 0	1	1	[1, 1]	1
0 0 1	1	1	[1, 1]	1
0 1 0	0	0	[0, 0]	0
0 1 1	0	1	[0, 1]	X
1 0 0	0	0	[0, 0]	0
1 0 1	0	0	[0, 0]	0
1 1 0	0	1	[0, 1]	X
1 1 1	1	1	[1, 1]	1

Table 2.4. Incompletely-Specified Function $f(x, y, z)$

Terminology has been developed by switching theorists to discuss incompletely-specified switching functions. One way the minimization problem has been approached is to find a completely-specified function $\hat{f}$ in the interval $[g, h]$ for which a formula which represents $\hat{f}$ has the fewest possible terms. The function g forms the lower bound of the possible functions for $\hat{f}$; therefore, for every $X \in \{0,1\}^n$ for which $g(X) = 1$, $\hat{f} = 1$ must be valid. In other words, every term in $MCF(g)$ also must appear in $MCF(\hat{f})$. The set of minterms which appear in $MCF(g)$ is referred to as the *on-set* of the incompletely-specified function f. Formally, the on-set of f is defined as

$$\text{ON-SET} = \{m_i \mid m_i \in MCF(g)\}. \tag{2.81}$$

The function h forms the upper bound of the possible functions for $\hat{f}$; therefore, for every $X \in \{0,1\}^n$ for which $h(X) = 0$, $\hat{f} = 0$ must be an identity. Where the function h is 0, the function h' is equal to 1. Thus, no term of $MCF(h')$ can appear in $MCF(\hat{f})$. The set of minterms which appear in $MCF(h')$ is referred to as the *off-set* of f. Formally, the off-set of f is defined as

$$\text{OFF-SET} = \{m_i \mid m_i \in MCF(h')\}. \tag{2.82}$$

A third set of minterms, which is empty in the case of completely-specified Boolean functions, is called the *don't care-set*. Given the interval (2.76) a completely-specified function may be formed

which specifies the cases in which $\hat{f}$ may be chosen to be either 0 or 1, i.e., the cases where f has a don't care value. This function is defined as $h - g$. A term in $MCF(h-g)$ may or may not appear in $MCF(\hat{f})$. The set of minterms which appear in $MCF(h-g)$ is referred to as the *don't-care set* of f. Formally, the don't-care set of f is defined as

$$\text{DON'T-CARE SET} = \{m_i \mid m_i \in MCF(h-g)\} \tag{2.83}$$

The don't-care set is sometimes referred to as the *dc-set*. In some cases, the minimization problem is solved by determining a good choice of minterms from the don't care-set to add to the on-set to produce a function $\hat{f}$ such that the number of terms in a formula representing $\hat{f}$ is minimal.

Example 2.6: For the incompletely-specified function $f(x, y, z)$ defined in Example 2.5:

- the on-set of f is $\{m_0, m_1, m_7\}$,
- the off-set of f is $\{m_2, m_4, m_5\}$, and
- the dc-set of f is $\{m_3, m_6\}$.

Free Boolean Algebras

The variables $x_1, x_2, \ldots, x_n$ may be used to construct a 2^{2^n}-element Boolean algebra in the following manner:

- each element of the algebra is the disjunction of a subset of the 2^n minterms built from the n variables,
- the null disjunction is the 0-element, and
- the disjunction of all minterms is the 1-element.

The resulting structure is called the *free Boolean algebra on the n generators* $x_1, x_2, \ldots, x_n$. The free Boolean algebra is denoted $FB(x_1, x_2, \ldots, x_n)$.

Operations on Boolean Functions

In this section, operations are discussed which are classified as *basic* operations and *expansion-based* operations. Operations on Boolean functions are not applied to the functions, but rather to the formulas which represent the functions. Typically, the result of an operation is a formula which represents a new function. All of the operations discussed in this section apply to functions which are represented by sum-of-products formulas; moreover, the result of each operation is an SOP formula.

Basic Operations. Basic operations on Boolean formulas are those which operate on a term-by-term basis. Typically, the resulting formulas are simplified by forming equivalent absorptive formulas.

Boolean Addition. To add two Boolean functions f and g, their corresponding SOP formulas F and G are simply appended. Absorbed terms are deleted to derive an equivalent absorptive result. The addition of two functions is implemented by Procedure 2.1 (Addition), listed in Appendix D. Assuming F and G are absorptive at the outset, Procedure 2.1 returns $ABS(F+G)$.

Cross-Product. The cross-product of two Boolean functions f and g is the term-by-term product of their corresponding formulas F and G. If $F = \sum_i s_i$ and $G = \sum_j t_j$, then the cross-product of f and g is given by

$$F \times G = \sum_i \sum_j s_i \cdot t_j. \tag{2.84}$$

This operation is used in lieu of the expansion-based product (described later) in some circumstances. The cross-product of two functions is implemented by Procedure 2.2 (Cross-Product),

listed in Appendix D. A quick way to simplify the result of Procedure 2.2 is to make absorptive the resulting formula.

In circumstances involving collectively unate functions represented by formulas which are unate in all variables, a more efficient version of Procedure 2.2 may be used to perform the cross-product operation. Such a technique is given by Procedure 2.3 (Unate Cross-Product) in Appendix D. The result of the procedure is the minimum-term formula representing the product, $f \times g$, of collectively unate functions.

Boolean Division. Given a function f and a term t, the *division* of f with respect to t is the function formed from f by enforcing the constraint $t = 1$ on the function. The division of f by t is indicated by f/t. Boolean division is also called the Boolean *quotient* in (Brown 90:53) and a *ratio* by Ghazala (Ghaza 57). The operation f/t is implemented by Procedure 2.4 (Boolean Division), listed in Appendix D.

A number of useful theorems involving Boolean division are found in (Brown 90:54-56). They are restated here without proof.

1. *Let f and g be n-variable Boolean functions and let t be an m-variable term ($m \leq n$). Then*

$$f \leq g \Rightarrow f/t \leq g/t. \tag{2.85}$$

2. *Let f be a Boolean function and let t be a term. Then the statements*

$$f'/t = (f/t)' \tag{2.86}$$
$$t \cdot f = t \cdot (f/t) \tag{2.87}$$
$$t' + f = t' + (f/t) \tag{2.88}$$

are identities.

3. *Let f be a Boolean function and let t be a term. Then*

$$t \cdot f \leq f/t \leq t' + f. \tag{2.89}$$

4. *Let p, q, and r be terms such that $pq \neq 0$. Then*

$$pq \leq r \Rightarrow q \leq r/p. \tag{2.90}$$

5. *Let f and g be Boolean functions and let t be a term. Then*

$$g \leq f/t \Rightarrow t \cdot g \leq f. \tag{2.91}$$

Boolean division is convenient in applications of Boole's Expansion Theorem. For example, the expansion of a 3-variable Boolean function $f(x, y, z)$ about x is

$$f(x, y, z) = x' f(0, y, z) + x f(1, y, z), \tag{2.92}$$

which can be rewritten as

$$f(x, y, z) = x'(f/x') + x(f/x). \tag{2.93}$$

Brayton et al. call f/x the *cofactor* of a Boolean function f with respect to x and denote it f_x (Brayt 82).

Expansion-Based Operations. Many operations which are performed on Boolean functions are executed most efficiently using algorithms based on Boole's Expansion Theorem. In these algorithms, expansion is applied recursively to reduce the functions on which an operation is ap-

plied. Expansion is performed until the leaf functions reach a base-case to terminate the recursion. Heuristics are used to determine the splitting variable at each stage of the recursion.

Brayton's Results. The algorithms for complementation, the product of functions, and subtraction stated in this section are taken from (Brayt 82). The key idea underlying these algorithms is the goal of quickly deriving leaf functions which are unate; special algorithms are then applied to the unate leaf functions. A heuristic is used to choose the "most binate" splitting variable x (Brayt 82:59); this choice tends to keep the number of terms in the formulas representing f/x and f/x' small as well as balanced. The most-binate heuristic results in a net reduction of the number of recursive expansion operations which are performed with respect to the number of expansions which generally would occur if the splitting variable x were chosen arbitrarily. Additionally, a *merge* operation is applied to the functions returned at each stage of the recursion to ensure that a simplified formula is returned. Brayton et al. call this process "a recursive paradigm based on cofactor and merging operations." For a unary functional operation, the general approach is

$$\text{operate}(f) = \text{merge}(\text{operate}(f/x), \text{operate}(f/x')). \tag{2.94}$$

Similarly, for a binary functional operation

$$\text{operate}(f, g) = \text{merge}(\text{operate}(f/x, g/x), \text{operate}(f/x', g/x')). \tag{2.95}$$

(Brayt 82:59)

Splitting-Variable Heuristic. Brayton et al. developed a heuristic which works well in determining a good splitting variable when using Boole's Expansion Theorem in the complementation, product, subtraction, simplification, and tautology algorithms. If the heuristic is used for a unary operation on a function f, it is applied to the formula F representing f. If used

for a binary operation on functions f and g, the heuristic is applied to the formula constructed by appending the formulas F and G which represent the functions. A method which produces a splitting variable based on Brayton's heuristic is implemented by Procedure 2.5 (Splitting-Variable Heuristic), listed in Appendix D.

Merge Operation. The merge operation is used to produce a simplified formula when using Boole's Expansion Theorem to perform a given primary operation. When using Boolean expansion, the primary operation is applied to the leaf functions recursively. After returning from the recursion, leaf functions are derived which are the result of applying the primary operation. For example, if the function f were to be complemented, Boole's Expansion Theorem would be applied in the following manner:

$$f' = x'(f/x')' + x(f/x)'. \tag{2.96}$$

In this case, the primary operation of complementation is applied to the leaf functions f/x' and f/x recursively. Let us call the functions which result from applying complementation to f/x' and f/x, g_0 and g_1, respectively. Equation (2.96) then becomes

$$f' = x'g_0 + xg_1. \tag{2.97}$$

One way of forming a formula to represent f' in (2.97) is to append x' to each term of a formula representing g_0 and x to each term of a formula representing g_1. Suppose s and t are arbitrary terms of formulas representing g_0 and g_1, respectively. These terms are used to form the terms $x's$ and xt in the formula representing f'. In many instances, consensus (2.32) may be used to form a new term st; in certain cases the term st absorbs either or both terms $x's$ and xt. In essence, what the merge operation does is to create the consensus term st only in those cases where at least one

of the parent terms, $x's$ and xt, is absorbed. The equivalent absorptive formula is returned by the merge operation.

Let h_0 be the result returned by applying an operation to the leaf function associated with x' and let h_1 be the result returned by applying an operation to the leaf function associated with x. Hence, for a unary functional operation, h_0 is the result of applying the operation to f/x'; h_1 is the result of applying the operation to f/x. Similarly, for a binary operation h_0 is the result of applying the operation to f/x' and g/x'; h_1 is the result of applying the operation to f/x and g/x. The function

$$x'h_0 + xh_1 \tag{2.98}$$

may then be formed. The merge operation is used to reformulate (2.98) by rewriting it as

$$x'\widehat{h_0} + x\widehat{h_1} + h_2. \tag{2.99}$$

This action is performed by removing terms of the formulas representing h_0 and h_1 and placing the appropriate consensus terms in the formula which represents h_2. Because terms are removed from the formulas representing h_0 and h_1, new functions $\widehat{h_0}$ and $\widehat{h_1}$ are formed. Likewise, the function h_2 is created.

Suppose s and t are terms in formulas representing h_0 and h_1, respectively. Then $x's$ and xt are terms in the formulas representing $x'h_0$ and xh_1, respectively, in (2.98). For each term s of h_0, it is determined whether s is either a subset or superset with respect to contained literals of at least one term t of h_1. If so, then the formula

$$x's + xt \tag{2.100}$$

is replaced by the equivalent formula

$$x's + xt + st. \tag{2.101}$$

The term st is the consensus term of $x's$ and xt. When this replacement is made, the term st is placed in the formula representing h_2. Additionally, one of the following actions is also possible:

- s and t may be removed from the formulas representing h_0 and h_1, respectively;
- s may be removed from the formula representing h_0; or
- t may be removed from the formula representing h_1.

Each of these cases is now examined.

Case 1: $s = t$

In this instance the term st is equivalent to s (or t) by idempotence. Thus, (2.101) can be rewritten as

$$x's + xs + s. \tag{2.102}$$

This is simplified by absorption to s. Thus, s and t are removed from the formulas representing h_0 and h_1 and the term s (or t) is added to formula representing h_2.

Case 2: $s \leq t, s \neq t$

In this instance the term st is equivalent to s since it contains a superset of the literals of t. Thus, (2.101) can be rewritten as

$$x's + xt + s. \tag{2.103}$$

This is simplified by absorption to $xt + s$. Thus, s is removed from the formula representing h_0 and the term s is added to formula representing h_2.

Case 3: $t \leq s, s \neq t$

In this instance the term st is equivalent to t since it contains a superset of the literals of s. Thus, (2.101) can be rewritten as

$$x's + xt + t. \tag{2.104}$$

This is simplified by absorption to $x's + t$. Thus, t is removed from the formula representing h_1 and the term t is added to formula representing h_2.

For each term s, only one term t must exist for which it is a superset to place the term into the formula representing h_2. However, if a term s is a subset of a term t, then it must be tested against all terms t_j to determine which terms of the formula representing h_1 should be moved to the formula representing h_2. If both subset and superset conditions exist between a term s and a term t, then both are removed from the formulas representing h_0 and h_1, respectively, and s (or t) is placed into the formula representing h_2. An algorithm for performing the merge operation is given by Procedure 2.6 (Merge Operation), listed in Appendix D.

Complement. The complementation algorithm presented in this section uses both the splitting-variable heuristic and the merge operation described previously. Boole's Expansion Theorem is used to complement a function f in the following manner:

$$f' = x'(f/x')' + x(f/x)'. \tag{2.105}$$

The complementation operation then is applied recursively to the leaf functions f/x' and f/x. Expansion is used to complement a function unless a function meets one of the following conditions:

- it is identically equal to 0;
- it is identically equal to 1; or
- the function is unate.

In the first case, 1 is returned as the complement of 0. Similarly, 0 is returned as the complement of 1. If the function is unate, a special unate complementation procedure—one more efficient than basic complementation—is applied to the function. An implementation of the general complementation algorithm is given by Procedure 2.7 (Complementation), listed in Appendix D.

Complementing a Unate Function. Brayton et al. developed a special algorithm to complement a unate function (Brayt 82:61-62). This algorithm uses the relationship

$$f' = x'(f/x')' + (f/x)' \tag{2.106}$$

as the basis for computation in which

- the formula representing f/x consists of all terms in F which do not include the literal x and the terms in F which include the literal x with the literal removed, and
- the formula representing f/x' consists only of the terms in F which do not include the literal x.

The validity of (2.106) is now demonstrated. Suppose a formula F which represents f is positive in x. Then there are formulas G and H, not involving x, such that

$$f = xG + H. \tag{2.107}$$

Complementing both sides of (2.107), we form

$$f' = x'H' + G'H'. \tag{2.108}$$

But

$$\begin{aligned} f/x' &= H \\ f/x &= G + H; \end{aligned} \tag{2.109}$$

hence,

$$\begin{aligned} (f/x')' &= H' \\ (f/x)' &= G'H'. \end{aligned} \tag{2.110}$$

Then, by substitution in (2.108):

$$f' = x'(f/x')' + (f/x)'. \tag{2.111}$$

Similarly, if F is negative in x, then

$$f' = x(f/x)' + (f/x')'. \tag{2.112}$$

To minimize computation, Brayton et al. use two heuristics to choose the literal x in (2.106). First, in the formula F the term with the fewest literals is found. Second, of the literals in the term selected by the first heuristic, the literal which occurs most frequently in F is selected. The first heuristic keeps the number of recursions as small as possible since for every recursive call the number of literals of the term with the fewest literals is reduced by one. Selecting the literal which occurs most frequently in F makes the number of terms in the formula representing f/x' as small as possible. If many terms include x, then the number which do not—all of the terms of the formula representing f/x'—are few. The unate complementation process is easily performed when the number of terms in a formula is small. (Brayt 82:61)

An implementation of the unate complementation algorithm is given by Procedure 2.8 (Unate Complementation), listed in Appendix D.

Product. The product algorithm described in this section is taken from (Brayt 82). It uses both the splitting variable heuristic and the merge operation previously described. Boole's Expansion Theorem is used to multiply two Boolean functions f and g as follows:

$$f \cdot g = x'(f/x' \cdot g/x') + x(f/x \cdot g/x). \tag{2.113}$$

The product operation is applied recursively to the leaf functions $(f/x' \cdot g/x')$ and $(f/x \cdot g/x)$. Expansion is used to multiply the functions unless the functions meet one of the following conditions:

1. either function is identically equal to 0;
2. either function is identically equal to 1; or
3. the functions are collectively unate.

If either function is identically equal to 0, then 0 is the result of the product of the functions. In the event that either function is identically equal to 1, then the result of the product is the value of the other function. In this case, the equivalent absorptive for nula is returned for the formula which

represents the second function. If the functions are collectively unate, then the most efficient way to multiply the two functions is to use Procedure 2.3 to perform a unate cross-product operation. The product algorithm is implemented by Procedure 2.9 (Product) in Appendix D.

Subtraction. The subtraction of a Boolean function g from a function f, $f - g$, is defined as the portion of f not covered by g, i.e., $f \cdot g'$. The subtraction algorithm presented in this section is taken from (Brayt 82). As in the product operation, it uses both the splitting-variable heuristic and the merge operation previously described. Boole's Expansion Theorem is used to subtract function g from function f as follows:

$$f - g = x'(f/x' - g/x') + x(f/x - g/x). \tag{2.114}$$

The subtraction operation is applied recursively to the leaf functions $(f/x' - g/x')$ and $(f/x - g/x)$. Expansion is used to perform the subtraction until the functions meet one of the following conditions:

1. either function is identically equal to 0;
2. either function is identically equal to 1; or
3. the functions are collectively unate.

If either function f is identically equal to 0 or function g is identically equal to 1, then 0 is the result of the subtraction. If function g is identically equal to 0, then the result of the subtraction is the function f. If function f is identically equal to 1, then the result is the complement of function g. If the functions are collectively unate, then $f - g$ is formed by multiplying the function f by the complement of g. Since g is unate, the complement of g is formed using the unate complementation algorithm implemented by Procedure 2.8. The product is performed using the cross-product operation implemented by Procedure 2.2. The result is simplified by forming $ABS(F - G)$. The subtraction algorithm is given by Procedure 2.10 (Subtraction) in Appendix D.

Exclusive-OR. In this section, two algorithms are presented based on Boole's Expansion Theorem to perform the Exclusive-OR of two Boolean functions f and g. In the first method, a splitting variable is arbitrarily chosen for expansion. The merge operation is not used in this procedure after expansion and recursive calculations. In the second technique, the methods developed by Brayton are extended for use in new procedure for performing the XOR operation. The splitting variable is chosen using Brayton's heuristic. After expansion and recursive operations, the merge operation is used to form the resulting formula.

The difference between the two procedures is the amount of time required to form a result and the simplicity of the result. The first algorithm is significantly faster than the second—roughly one to four times faster. However, the second technique produces a simpler formula; typically, the number of terms in this formula is fewer than two-thirds the number of terms in a formula produced by the first technique. The choice of one procedure over the other is dependent on considerations such as the need for efficiency and the nature of the application.

In both XOR procedures, Boole's Expansion Theorem is used to form the Exclusive-OR of functions f and g as follows:

$$f \oplus g = x'(f/x' \oplus g/x') + x(f/x \oplus g/x). \tag{2.115}$$

The XOR operation is applied recursively to the leaf functions $(f/x' \oplus g/x')$ and $(f/x \oplus g/x)$. Expansion is used until the functions meet one of the following conditions:

1. either function is identically equal to 0; or
2. either function is identically equal to 1.

If either function is identically equal to 0, then the result of the XOR operation is the other function. If either is identically equal to 1, then the result is the complement of the other function.

In the first XOR algorithm, of the formulas which represent f and g, the splitting variable is found by arbitrarily choosing a variable which appears in the smaller of the formulas F and G. A variable is chosen in the smaller formula to force the decomposition of the function which it represents; hence, one of the termination conditions stated above is reached more quickly. The first XOR algorithm is implemented by Procedure 2.11 (Exclusive-OR), listed in Appendix D.

The second XOR procedure is a new algorithm which adapts the methods developed by Brayton to the XOR operation. The splitting variable is chosen using Brayton's heuristic. If the functions f and g are found to be collectively unate, then the recursion is terminated and the Exclusive-OR is calculated directly via the form $f' \cdot g + f \cdot g'$. Since it is known that the functions are unate, the unate complementation algorithm is used to complement the functions. Additionally, to perform the product operations, the cross-product operation of Procedure 2.2 is used. After expansion and recursive operations, the merge operation is used to form the resulting formula. Procedure 2.12 (Exclusive-OR) in Appendix D states the second XOR operation.

Comparison of Functions. In view of (2.38), i.e.,

$$f = g \iff f \oplus g = 0, \tag{2.116}$$

the Exclusive-OR operation may be used to compare two Boolean functions. If the right-hand side of (2.116) is true, then $f = g$. On the other hand, if the right-hand is false, then $f \neq g$. Hence, to determine whether Boolean functions f and g are equal, the following actions are taken:

1. $f \oplus g$ is formed; and
2. $f \oplus g$ is tested to determine if it is identically equal to zero.

If $f \oplus g$ is identically equal to zero, then the functions are equal. Otherwise, they are not equal.

An interesting by-product of the XOR test for equality is produced in the case where functions are not equal. Given two n-variable Boolean functions f and g, an n-variable function h can be constructed which shows all circumstances in which functions f and g are different. h is defined in the following way:

$$f \oplus g = h \tag{2.117}$$

Forming the minterm canonical form for h, the minterms of h for which the associated discriminants are not zero correspond to the minterms of f and g in which their associated discriminants differ. Conversely, for minterms of f and g in which the respective discriminants are equal, the discriminants for the corresponding minterms of h are equal to zero. In the case of switching functions, for minterms of f and g where the associated discriminants differ, the respective discriminant in h is equal to one.

Example 2.4: Given the equations $f(x, y) = x$ and $g(x, y) = y$, $h(x, y)$ is found as follows:

$$\begin{aligned} h(x,y) &= f(x,y) \oplus g(x,y) \\ &= x \oplus y \\ &= x'y + xy'. \end{aligned} \tag{2.118}$$

Minterms and discriminants are summarized in Table 2.5. For the minterm $x'y$ the associated discriminant is 1 in $g(x, y)$ and 0 in $f(x, y)$, i.e., they differ. Hence, the discriminant of the minterm $x'y$ in $h(x, y)$ is equal to 1. For the minterm $x'y'$ the associated discriminant is 0 in $g(x, y)$ and $f(x, y)$, i.e., they are the same. Thus, the discriminant of the corresponding minterm in $h(x, y)$ is equal to 0

Exclusive-NOR. Two algorithms based on Boole's Expansion Theorem to compute the Exclusive-NOR of two Boolean functions f and g are listed in Appendix D. These procedures

$x\ y$	$f(x,y)$	$g(x,y)$	$h(x,y)$
0 0	0	0	0
0 1	0	1	1
1 0	1	0	1
1 1	1	1	0

Table 2.5. Results of Example 2.4

are analogous to the two described for the Exclusive-OR operation. The first $XNOR$ algorithm is implemented by Procedure 2.13 (Exclusive-NOR).

As in the second XOR method, the second $XNOR$ procedure is a new technique which adapts the methods developed by Brayton. The splitting variable is chosen using Brayton's heuristic. If the functions f and g are found to be collectively unate, then the recursion is terminated and the Exclusive-NOR is calculated directly via the form $f' \cdot g' + f \cdot g$. Since it is known that the functions are unate, unate complementation is used to complement the functions. Additionally, to perform the product operations, the unate cross-product operation is used. After expansion and recursive operations, the merge operation is used to form the resulting formula. Procedure 2.14 (Exclusive-NOR) states the second $XNOR$ method.

Simplification

A formula F representing a Boolean function f may not be economical. It would be better to generate an equivalent formula which is simpler is some sense—typically with respect to the number of terms. Less memory is required to store a formula with fewer terms. Additionally, when performing operations such a complementation or product, fewer computations are required if the formulas on which the operations are applied are small. However, although it is advantageous to have the best formula possible, expending much effort producing such a formula is not desirable. In such circumstances, a simplification routine is used which produces a good formula (but not necessarily the best) in an efficient manner.

In this section a number of simplification routines are described which produce good formulas efficiently. Included among these procedures is one which is based on Boole's Expansion Theorem as well as the merge operation described in previous sections and an absorption-based simplification routine.

Expansion-Based Simplification. The simplification algorithm discussed in this section is taken from (Brayt 82). Both the splitting variable heuristic and the merge operation described in preceding sections are used. Boole's Expansion Theorem is used to simplify a function f in the following manner:

$$f = x'(f/x') + x(f/x). \tag{2.119}$$

The simplification operation is applied recursively to the leaf functions f/x' and f/x until a point is reached at which the functions are unate; the functions are known to be unate when the formulas which represent the functions are unate in all of their variables. At this point the equivalent absorptive formula is returned as the result. The merge operation is applied to develop a formula to represent f by combining the simplified formulas representing f/x' and f/x. The resulting formula which represents f generally is much simpler with respect to the number of terms and literals than the original formula; however, this outcome is not guaranteed. The simplification algorithm is implemented by Procedure 2.15 (Simplification) in Appendix D.

Absorption-Based Simplification Techniques.

Quick Simplification. In cases where speed of simplification is important—at the expense of not generating as good a formula as in expansion-based simplification—an absorption-based simplification technique may be used to generate a simplified formula. Such a procedure is described in this section.

One of the underlying concepts used by the procedure in this section is that of the consensus of two terms. Two terms are said to be *opposed* in the event that a variable is opposed, i.e., exists complemented in one term and uncomplemented in the second, between the two terms. When exactly one variable is opposed between the two terms, then by (2.32) a new term may be formed which is called the *consensus* of the terms. The consensus of two terms s and t, denoted $c(s,t)$, is created by deleting the literals opposed in the terms and forming the conjunction of the remaining literals of each term. If any literals appear in both s and t, the duplicate literals are deleted. Term $c(s,t)$ is referred to as a *consensus term;* terms s and t are called its *parent terms.*

Given two terms s and t which have one opposed variable, in some cases it is advantageous to form $c(s,t)$ and in other situations it is not. In general, it is desirable to form the term $c(s,t)$ if it absorbs at least one of the terms s or t. If $c(s,t)$ absorbs at least one of the terms, then by (2.25) the absorbed term may be deleted. By following this methodology, a formula is developed which has fewer terms—with many of the remaining terms having fewer literals—than the original formula. Procedure 2.16 (Simplification) in Appendix D is a simplification procedure in which terms are compared on a term-by-term basis. When it is possible to form a consensus term which absorbs at least one of its parent terms, the consensus term is formed and the absorbed parent(s) are deleted.

Relative Simplification. In this section, a new concept called *relative simplification* is presented which has proven to be useful in procedures discussed in subsequent chapters. Of particular importance is the *Relative Simplification Theorem* which facilitates the replacement of the sum of two formulas F and G, i.e., $F + G$, with a simpler equivalent formula.

Given two Boolean functions f and g and SOP formulas, F and G, which represent them, the formulas may be examined on a term-by-term basis in which every term in F is compared to every term in G. This may be done in order to generate all of the consensus terms that can be formed between the terms in F and the terms in G. In a special case, consensus terms are generated

only when they absorb the parent in one of the formulas. If consensus terms are formed only when they absorb their respective parent terms in F, and these terms replace their parent terms to form a new formula $\tilde{F}$, this is called simplifying F *relative to* G. We call this process ***relative simplification.*** In forming the new formula $\tilde{F}$ a new function $\tilde{f}$ is created. Procedure 2.17 (Relative Simplification) in Appendix D implements the process of relative simplification. An application of relative simplification is now introduced.

Let $ABSREL(F,G)$ be an operator which returns a formula $\dot{F}$ constructed from F by removing all terms absorbed by G. Let $SIMPREL(F,G)$ be an operator which returns a formula $\tilde{F}$ formed by simplifying F relative to G. Furthermore, let $\tilde{F}$ denote the formula developed by simplifying $\dot{F}$ relative to G, i.e.,

$$\tilde{F} = SIMPREL(\dot{F}, G). \tag{2.120}$$

Given the functions f, g, and $\tilde{f}$ represented by F, G, and $\tilde{F}$, respectively, it can be demonstrated that $f+g = \tilde{f}+g$. We show that $f+g$ and $\tilde{f}+g$ are equal by establishing that they are the same function; this is demonstrated by showing that the formulas $F+G$ and $\tilde{F}+G$ are equivalent.

Relative Simplification Theorem: *Given functions f and g, represented by formulas F and G, respectively, and a function $\tilde{f}$ represented by the formula $\tilde{F}$ which is defined as follows:*

$$\dot{F} = ABSREL(F,G) \tag{2.121}$$
$$\tilde{F} = SIMPREL(\dot{F},G), \tag{2.122}$$

the following statement is true

$$f+g = \tilde{f}+g. \tag{2.123}$$

Proof. Using the property of absorption (2.25), the formula $F + G$ is equivalently expressed by the formula $\dot{F} + G$.

By simplifying $\dot{F}$ relative to G, consensus terms are created between terms in $\dot{F}$ and G only when the resulting terms absorb their parent terms in $\dot{F}$; the resulting terms replace their parent terms in $\dot{F}$ to form $\tilde{F}$.

The property of consensus (2.32) allows the replacement of a formula by an equivalent formula which contains consensus terms formed from parent terms in the original formula. Let H be the set of consensus terms resulting from the consensus of terms in $\dot{F}$ and G which are formed only if the consensus terms absorb their parent terms in $\dot{F}$. Then $\dot{F} + G$ may be expressed equivalently by $\dot{F} + G + H$. By definition, all terms in H absorb terms in $\dot{F}$. Again applying the property of absorption, $\dot{F} + H$ may be equivalently expressed by a formula comprised of the terms in H plus the terms in $\dot{F}$ not absorbed by the terms in H. The resulting formula is defined identically as the formula $\tilde{F}$, i.e., it is the formula $\tilde{F}$. It follows that $\dot{F} + G + H$ is equivalent to $\tilde{F} + G$.

Thus, $F + G \equiv \tilde{F} + G$; whence, $f + g = \tilde{f} + g$. This completes the proof. □

An application of the Relative Simplification Theorem is to replace the formula $F + G$ representing the function $f + g$ by the equivalent formula $\tilde{F} + G$ representing $\tilde{f} + g$. Since $\tilde{F}$ contains fewer terms and literals than does F, it is beneficial that an operation that is to be applied to the function $f + g$ be instead applied to the function $\tilde{f} + g$. In general, fewer operations must be applied when using a smaller formula than when using a larger one.

The Blake Canonical Form

A Boolean function f is said to be *included in* a Boolean function g, denoted $f \leq g$, if

$$f \cdot g' = 0. \tag{2.124}$$

A term p is called an *implicant* of a Boolean function f if $p \leq f$. If a function f is expressed in sum-of-products form, all terms in the formula are implicants of f. A *prime implicant* (PI) of a Boolean function f is an implicant of f such that it is no longer an implicant if any of its literals is removed (Quine 52). The following theorem formalizes the relationship between implicants and prime implicants:

> *If r is an implicant of f, then there is a prime implicant p of f such that $r \leq p$.* (Brown 90:245)

Thus, the existence of an implicant implies the existence of a prime implicant which includes it.

Boolean axioms, and theorems such as consensus and absorption, are used in the literature to reduce a Boolean formula which represents a function to a form which consists of the prime implicants of the function. An application is minimization, one approach to which is to reduce an SOP formula to an equivalent formula which includes the smallest number of prime implicants that still represent the same function. The impetus for minimization is to represent a Boolean function by a formula that can be implemented in hardware with the smallest number of components.

Example 2.6: The only term in the n-variable Boolean formula F given by

$$xyz + x'yz' + x'y'z' + xy'z' \tag{2.125}$$

that is a prime implicant of the function f is xyz. The formula may be transformed to an equivalent formula consisting only of prime implicants by application of Boolean axioms and theorems. An equivalent formula which consists only of prime implicants is:

$$xyz + y'z' + x'z'. \tag{2.126}$$

Another application for the prime implicants of a formula is for *Boolean inference*. Boolean inference is "the extraction of conclusions from a collection of Boolean data" (Brown 90:72). The basis for Boolean inference is the *Blake canonical form*. The Blake canonical form, denoted BCF(f), of a function f is the disjunction of all of the prime implicants of f. The Blake canonical form is a complete and simplified representation of all possible conclusions that can be inferred from a Boolean equation. Methods for generating BCF(f) are the *exhaustion of implicants*, *iterated consensus*, and *multiplication*. Blake invented the methods of iterated consensus and multiplication (Blake 37). Iterated consensus also is discussed in (Quine 52); additionally, the multiplication method is found in (Samso 54). Many references on switching theory refer to the sum of all of the prime implicants of a Boolean function as the *complete sum* of the function; however, it is more suitably termed a canonical form. Brown has designated the complete sum the "Blake" canonical form in deference to Blake whose seminal work on the generation of prime implicants of a Boolean function preceded and encompasses most work recognized today as the "original" work on the generation of prime implicants of a function (Blake 37)(Brown 90).

Example 2.7: The formula

$$xyz + y'z' + x'z' \tag{2.127}$$

which represents the n-variable Boolean function defined in Table 2.2 is its Blake canonical form because the formula consists of all of the prime implicants of the function.

In the process of reducing a given formula to prime implicants, *superfluous* terms are often generated. A term p is superfluous in a sum-of-products formula, $p + r$, if $p + r$ is equivalent to the formula r (Quine 52:522). A superfluous term is also called a *redundant* term. A literal of a term in a sum-of-products formula is *superfluous* if it can be removed without changing the formula to a non-equivalent formula. Quine called a "formula *irredundant* if it has no superfluous clauses and none of its clauses has superfluous literals" (Quine 52:523).

Once the Blake canonical form $BCF(f)$ is generated for a function f, many prime implicants may be deleted as redundant. A prime implicant p which cannot be deleted from the Blake canonical form without changing the formula to a non-equivalent formula is called an *essential prime implicant*. Therefore, an essential prime implicant appears in every irredundant formula which can represent a Boolean function. Let f_{ess} be the sum of the essential prime implicants of f. If a prime implicant p of f which is not an essential prime implicant meets the condition

$$p \leq f_{ess}, \tag{2.128}$$

then p is called an *absolutely inessential prime implicant* or an *inessential* prime implicant. An irredundant formula cannot contain an inessential prime implicant. All prime implicants which are neither essential nor inessential are called *conditionally-eliminable* prime implicants.[1] The fact that a prime implicant is conditionally eliminable rather than inessential does not guarantee that it will appear in an irredundant formula which represents a function f; there may be conditionally-eliminable prime implicants which do not appear in any irredundant formula (Gaine 64:179).

Example 2.8: Suppose a function f is given for which $BCF(f)$ is the formula

$$ac'e' + a'c'd + cd'e + c'de' + bce + bcd + a'b'cd' + a'b'ce' + a'de' + bde' + a'bd. \tag{2.129}$$

The terms $ac'e', a'c'd$, and $cd'e$ are essential prime implicants. The term $c'de'$ is an inessential prime implicant because

$$c'de' \leq ac'e' + a'c'd + cd'e. \tag{2.130}$$

[1] The term *conditionally eliminable* is borrowed from Chang and Mott (Chang 65) who used this term to denote prime implicants that appear in at least one—but not all—irredundant SOP formulas that may represent a function. The term is used in a somewhat broader sense in this work.

The remaining prime implicants—$bce, bcd, a'b'cd', a'b'ce', a'de', bde'$, and $a'bd$—are conditionally eliminable. The irredundant formulas that represent f are

$$\begin{aligned} &ac'e' + a'c'd + cd'e + a'de' + a'b'cd' + bcd \\ &ac'e' + a'c'd + cd'e + a'b'ce' + bde' + bce \\ &ac'e' + a'c'd + cd'e + a'de' + a'b'cd' + bde' + bce \\ &ac'e' + a'c'd + cd'e + a'b'ce' + bcd. \end{aligned} \tag{2.131}$$

The essential prime implicants appear in every irredundant formula. Of the conditionally-eliminable prime implicants, all except the term $a'bd$ appear in at least one irredundant formula.

Two techniques will be described to generate the Blake canonical form of a Boolean function. The first method is based on the specialized form of iterated consensus called *successive extraction*. The second procedure uses recursive multiplication to form $BCF(f)$. In this method, a new heuristic is introduced which significantly improves the efficiency and practicality of the recursive multiplication method.

Successive Extraction. Iterated consensus generates the Blake canonical form of an SOP formula by repeated application of the following rule:

> **If the formula contains a pair r, s of terms whose consensus $c(r, s)$ exists and is not included in any term of the formula, then adjoin $c(r, s)$ to the formula.** (Brown 90:77)

Blake invented the method of iterated consensus (Blake 37); the technique is also discussed in (Quine 52), (Quine 55), and (Samso 54).

A refinement of iterated consensus is a process called *successive extraction*. In this form of iterated consensus, all of the variables in the given formula are found which have opposed literals. Using this set of variables, consensus is performed within the formula on a letter-by-letter basis. For example, suppose the variables a and b are opposed within a formula. First, all possible consensus

terms would be formed between terms where the variable a is opposed. Then, all possible consensus terms would be formed with respect to sets of terms where b is opposed. If there were more opposed variables, the process would continue until the set of opposed variables was exhausted. Blake mentioned this approach to iterated consensus in his work (Blake 37), although it is commonly attributed to Tison (Tison 67). Procedure 2.18 in Appendix D generates the Blake canonical form of a function using successive extraction.

Recursive Multiplication. As is the case of iterated consensus and successive extraction, Blake also invented techniques based on multiplication to generate the sum of all of the prime implicants of a function (Blake 37). A theorem in Brown (Brown 90:81) states that if we are given a function f and let x be one of its variables, "then the Blake canonical form of f is given by

$$BCF(f) = ABS((x' + BCF(f/x)) \times (x + BCF(f/x'))).\text{"} \tag{2.132}$$

The $\times$ operation is the cross-product operation implemented by Procedure 2.2. Note that $BCF(f)$ is formed in (2.132) by recursively forming $BCF(f/x)$ and $BCF(f/x')$. This recursion is performed repeatedly until f meets one of the following conditions:

- f is identically equal to 1;
- f is identically equal to 0; or
- f is recognized to be a unate function, i.e., f is represented by a formula F in which F consists only of unate variables.

When f is identically equal to 0 or 1, the formula which represents 0 or 1 is returned as the result of $BCF(f)$. In the event that f is represented by a formula F consisting only of unate variables, then $ABS(F)$ is returned as the result of $BCF(f)$.

Brown (Brown 90:82-83) has shown that the efficiency of forming $BCF(f)$ using recursive multiplication may be improved by changing the form of (2.132) and restricting the scope of the ABS operation. This is done in the following manner:

$$BCF(f) = ABS(G + H) \tag{2.133}$$

in which

$$G = x' \times BCF(f/x') + x \times BCF(f/x) \tag{2.134}$$
$$H = BCF(f/x) \times BCF(f/x'). \tag{2.135}$$

However, G is an absorptive formula because $BCF(f/x')$ and $BCF(f/x)$ are in Blake canonical form. Additionally, no term in G will absorb a term in H because every term in G includes the literal x or x' whereas every term in H will include neither x nor x'. Therefore, the only absorptions which must be perfo:ıed are:

- absorptions within H; and
- absorptions of terms in G by terms in H.

Hence, equation (2.132) may be expressed as

$$BCF(f) = ABS(H) + ABSREL(G, ABS(H)) \tag{2.136}$$

where

- G and H are given by (2.134) and (2.135), respectively; and
- $ABSREL(P, Q)$ is an operator which returns the formula constructed from P by removing all terms absorbed by Q.

Previous implementations of this recursive multiplication algorithm arbitrarily chose one of the opposed variables in the formula F representing f to be x in (2.132). Upon closer examination of (2.134), it may be noted that G is formed in a manner analogous to the expansion-based operations described in earlier sections. Based on a rationale similar to the splitting-variable heuristic implemented by Procedure 2.5, we present a new heuristic for choosing the variable x in (2.132); this heuristic significantly improves the efficiency and practicality of the recursive multiplication algorithm. The choice of x keeps the number of terms in the formulas representing f/x and f/x' small as well as balanced. Additionally, the choice of x causes f/x and f/x' to become unate quickly; this property is useful due to the fact that having a unate function is one of the termination conditions for the recursion.

To apply this heuristic to an SOP formula F, the following characteristic numbers are determined for each variable x_i that appears in F:

- n_0^i: the number of terms in F in which the literal x_i' appears; and
- n_1^i: the number of terms in F in which the literal x_i appears.

Each variable is scored using the metric

$$n_0^i * n_1^i. \tag{2.137}$$

The variable x_i associated with the i with the highest value for (2.137) is returned as the variable x used in (2.136). A variable so chosen is called a ***modified splitting-variable.*** The heuristic for determining the modified splitting-variable is implemented by Procedure 2.19 in Appendix D. An algorithm which uses Procedure 2.19 and (2.136) to form $BCF(f)$ is implemented by Procedure 2.20, listed in Appendix D.

Procedure 2.20 has been applied by the author to generate the Blake canonical form for a variety of Boolean functions. For certain classes of functions, it is significantly faster than

Procedure 2.18. This was not the case, however, prior to the development of the heuristic for choice of variable x at each stage of recursion. The modified splitting-variable heuristic decreases both the time and memory required to form $BCF(f)$ via recursive multiplication.

Boolean Analysis

Boolean Systems. An *n-variable Boolean system* $S(X)$ on a Boolean algebra $\mathbf{B}$ is a collection

$$\begin{aligned} g_1(X) &= h_1(X) \\ g_2(X) &= h_2(X) \\ &\vdots \\ g_k(X) &= h_k(X) \end{aligned} \tag{2.138}$$

of simultaneously-asserted equations, for which $g_i(X)$ and $h_i(X)$ are n-variable Boolean functions on the Boolean algebra $\mathbf{B}$. The notation X denotes the vector $(x_1, x_2, \ldots, x_n)$. $S(X)$ is a predicate defined by system (2.138). For a substitution $A \in \mathbf{B}^n$ for X in system $S(X)$, $S(A)$ is said to be *true* if each equation in (2.138) is an identity; otherwise, $S(A)$ is said to be *false*. A substitution $A \in \mathbf{B}^n$ for X in system $S(X)$, which causes $S(X)$ to be true is called a *solution* of system $S(X)$. A Boolean system is called *consistent* if it has at least one solution. If it does not have any solutions, then it is said to be *inconsistent*.

Let $S_1(X)$ and $S_2(X)$ be two n-variable Boolean systems. $S_1(X)$ is called an *antecedent* of $S_2(X)$ if every substitution $A \in \mathbf{B}^n$ for X that causes $S_1(X)$ to be true also causes $S_2(X)$ to be true. That $S_1(X)$ is an antecedent of $S_2(X)$ is denoted by $S_1(X) \Rightarrow S_2(X)$. $S_2(X)$ is called a *consequent* of $S_1(X)$. If $S_1(X)$ and $S_2(X)$ are both a antecedent and consequent of each other, i.e., if each has exactly the same set of solutions, then they are called *equivalent* systems. This is denoted $S_1(X) \Longleftrightarrow S_2(X)$.

Reduction. Any system of Boolean equations can be reduced to a single Boolean equation of the form $f(X) = g(X)$, where $g(X)$ is any preassigned Boolean function (Rudea 74:116-117). In particular, we may choose $g(X)$ to be 0 or 1.

0-Normal Form. The reduced form $f(X) = 0$ is derived in the following manner. A system

$$\begin{aligned} g_1(X) &= h_1(X) \\ g_2(X) &= h_2(X) \\ &\vdots \\ g_k(X) &= h_k(X) \end{aligned} \tag{2.139}$$

of Boolean equations can be transformed, using property (2.38), into the equivalent system

$$\begin{aligned} g_1(X) \oplus h_1(X) &= 0 \\ g_2(X) \oplus h_2(X) &= 0 \\ &\vdots \\ g_k(X) \oplus h_k(X) &= 0. \end{aligned} \tag{2.140}$$

This system of equations can then be transformed into a single Boolean equation by Property (2.40). Since all of the equations must be simultaneously true, they are "&'ed" together as in Equation (2.40). However, the "&" symbol is dropped for notational simplicity. The resulting single Boolean equation is

$$f(X) = 0 \tag{2.141}$$

for which f is defined by

$$f = \sum_{i=1}^{k} (g_i \oplus h_i). \tag{2.142}$$

When a system (2.139) of equations is reduced to an equivalent form (2.141), (2.141) is called a *0-normal form* of (2.139).

1-Normal Form. The reduced form $p(X) = 1$ for a system of equations is similarly derived. The system of equations (2.139) can be transformed into an equivalent system using property (2.39):

$$\begin{aligned} g_1(X) \odot h_1(X) &= 1 \\ g_2(X) \odot h_2(X) &= 1 \\ &\vdots \\ g_k(X) \odot h_k(X) &= 1. \end{aligned} \tag{2.143}$$

This system of equations is transformed into a single Boolean equation by property (2.41). Again, the "&" symbol is dropped for notational simplicity. The resulting single Boolean equation is

$$p(X) = 1 \tag{2.144}$$

for which p is defined by

$$p = \prod_{i=1}^{k} (g_i \odot h_i). \tag{2.145}$$

When a system (2.139) of equations is reduced to an equivalent form (2.144), (2.144) is called a *1-normal form* of system (2.139).

0-Normal Form Versus 1-Normal Form. The utility of the $f(X) = 0$ form versus the $p(X) = 1$ form is dependent on the application (Rudea 74:52). Conversion between the two forms is done by complementation of both sides of the equality, i.e.,

$$f'(X) = 0 \Leftrightarrow f(X) = 1 \tag{2.146}$$

and

$$p'(X) = 1 \Leftrightarrow p(X) = 0. \tag{2.147}$$

Extended Verification Theorem. An important theorem in Boolean algebra is the *Extended Verification Theorem*, stated as follows:

Let $f, g : \mathbf{B}^n \rightarrow \mathbf{B}$ be Boolean functions, and assume that the equation $f(X) = 0$ is consistent. Then the following statements are equivalent:

$$f(X) = 0 \implies g(X) = 0 \quad \forall X \in \mathbf{B}^n, \tag{2.148}$$

$$g(X) \leq f(X) \quad \forall X \in \mathbf{B}^n, \tag{2.149}$$

$$g(X) \leq f(X) \quad \forall X \in \{0,1\}^n. \tag{2.150}$$

(Rudea 74:100)

Eliminants and Elimination.

The Conjunctive Eliminant. Let $f : \mathbf{B}^n \rightarrow \mathbf{B}$ be an n-variable Boolean function expressed in terms of variables $x_1, \ldots, x_n$ and let $T = \{x_1, \ldots x_m\}$ consist of the first m elements of $\{x_1, \ldots, x_n\}$. The *conjunctive eliminant* of f with respect to T, denoted by $ECON(f, T)$, is defined by

$$ECON(f, \{x_1, \ldots, x_m\}) = \prod_{(a_1, \ldots, a_m) \in \{0,1\}^m} f(a_1, \ldots, a_m, x_{m+1}, \ldots, x_n). \tag{2.151}$$

Although the first m variables are used in the above definition, the conjunctive eliminant of a function may be found with respect to an arbitrary subset of the variables.

The following theorem follows from the definition of the conjunctive eliminant:

Let $f : \mathbf{B}^n \rightarrow \mathbf{B}$ be an n-variable Boolean function expressed in terms of variables $x_1, \ldots, x_n$, and let R, S, and T be subsets of $\{x_1, \ldots x_n\}$. Then the following statements are true:

1. $ECON(f, \emptyset) = f$;
2. $ECON(f, \{x_1\}) = f(0, x_2, \ldots, x_n) \cdot f(1, x_2, \ldots, x_n)$; *and*
3. $ECON(f, R \cup S) = ECON(ECON(f, R), S)$.

(See (Brown 90:100).)

The Disjunctive Eliminant. Let $f : \mathbf{B}^n \rightarrow \mathbf{B}$ be an n-variable Boolean function expressed in terms of variables $x_1, \ldots, x_n$ and let $T = \{x_1, \ldots x_m\}$ consist of the first m elements of $\{x_1, \ldots, x_n\}$. The *disjunctive eliminant* of f with respect to T, denoted by $EDIS(f, T)$, is defined by

$$EDIS(f, \{x_1, \ldots, x_m\}) = \sum_{(a_1, \ldots, a_m) \in \{0,1\}^m} f(a_1, \ldots, a_m, x_{m+1}, \ldots, x_n). \tag{2.152}$$

As in the conjunctive eliminant, the disjunctive eliminant of a function may be found with respect to an arbitrary subset of the variables by which the function is expressed.

The following theorem follows from the definition of the disjunctive eliminant:

Let $f : \mathbf{B}^n \rightarrow \mathbf{B}$ be an n-variable Boolean function expressed in terms of variables $x_1, \ldots, x_n$, and let R, S, and T be subsets of $\{x_1, \ldots x_n\}$. Then the following statements are true:

1. $EDIS(f, \emptyset) = f$;
2. $EDIS(f, \{x_1\}) = f(0, x_2, \ldots, x_n) + f(1, x_2, \ldots, x_n)$; *and*
3. $EDIS(f, R \cup S) = EDIS(EDIS(f, R), S)$.

(See (Brown 90:100).)

A simple method for deriving the disjunctive eliminant of a Boolean function f is by transforming the formula that represents the function to any equivalent sum-of-products form and then replacing the literals of the variables to be eliminated, whether in complemented or uncomplemented form, by 1 (Mitch 83).

Elimination. Given a Boolean equation, it is possible to determine constraints on certain variables in the absence of information with respect to the other variables using a process called *elimination.* Equations deduced as the result of elimination are called *resultants of elimination.*

Using the definition of the conjunctive eliminant, a variable may be eliminated from an equation to form an implied equation:

$$f(X) = 0 \Rightarrow ECON(f, \{x_i\}) = 0. \tag{2.153}$$

The equation $ECON(f, \{x_i\}) = 0$ is called the resultant of elimination of x_i from equation $f(X) = 0$.

Using the definition of the disjunctive eliminant, a variable may be similarly eliminated from an equation to deduce an implied equation:

$$p(X) = 1 \Rightarrow EDIS(p, \{x_i\}) = 1. \tag{2.154}$$

The equation $EDIS(p, \{x_i\}) = 1$, like the equation $ECON(f, \{x_i\}) = 0$, is the resultant of elimination of x_i from equation $p(X) = 1$.

Theorems Involving Eliminants. A number of theorems pertaining to eliminants are useful in Boolean reasoning. These theorems are taken from (Brown 90:103-107).

Let $f : \mathbf{B}^n \rightarrow \mathbf{B}$ be an n-variable Boolean function expressed in terms of variables $x_1, \ldots, x_n$ and let T be a subset of $\{x_1, \ldots x_n\}$.

1.

$$BCF(ECON(f,T)) = \sum(\text{terms of } BCF(f) \text{ not involving arguments in } T). \quad (2.155)$$

The resulting formula is in Blake canonical form.

2.

$$ECON(f,T) \leq f \leq EDIS(f,T). \quad (2.156)$$

3. *Let U be a p-element subset of $\{x_1, \ldots x_n\}$, and let t be a q-argument term whose arguments are disjoint from those in U. Then*

$$ECON(f/t,U) = (ECON(f,U))/t, \text{ and} \quad (2.157)$$
$$EDIS(f/t,U) = (EDIS(f,U))/t. \quad (2.158)$$

4.

$$(ECON(f,T))' = EDIS(f',T), \text{ and} \quad (2.159)$$
$$(EDIS(f,T))' = ECON(f',T). \quad (2.160)$$

Formation of the Conjunctive Eliminant. By its definition, one way to form the conjunctive eliminant, $ECON(f, \{x\})$, is

$$ECON(f, \{x\}) = f/x' \cdot f/x. \tag{2.161}$$

However, it is more efficient to form the conjunctive eliminant of a function f with respect to a single variable x in the following manner:

1. Partition the SOP formula F which represents f into terms which include the literal x', terms which include the literal x, and terms which include neither x nor x'.
2. Given the partitioning, express F in the following manner:

$$f(x, y, \ldots) = x'p(y, \ldots) + xq(y, \ldots) + r(y, \ldots) \tag{2.162}$$

   where

   - p comprises the terms in F which include the literal x', with the literal divided out,
   - q comprises the terms in F which include the literal x, with the literal divided out, and
   - r comprises the terms in F which include neither x nor x'.

3. Form $ECON(f, \{x\})$:

$$ECON(f, \{x\}) = p \cdot q + r. \tag{2.163}$$

This basic process for forming a conjunctive eliminant with respect to a variable was originally discussed by (Schrö 90). Constructing the eliminant in this fashion reduces the number of terms involved in the multiplication operation; only terms of the original formula which include the variable x are involved in the multiplication $p \cdot q$. When the conjunctive eliminant is formed with respect to a set of variables, an operation of the form (2.163) is executed in turn for each variable. After each iteration, simplification of the resulting formula reduces the number of computations that are performed in later stages.

Let T be the set of variables with respect to which we are forming the conjunctive eliminant. The order in which variables are selected from T to form the conjunctive eliminant can be computationally significant. A heuristic which works well is to select the argument at each iteration

for which the multiplication operation $p \cdot q$ in (2.163) will be the least computationally intensive. One way to predict the complexity of the potential multiplication operation for each variable is to determine the number of term-by-term multiplications that would have to performed between P and Q using the cross-product operation. Hence, the following metrics are determined for each variable x in T:

- n_0: the number of terms in F in which the literal x' appears, and
- n_1: the number of terms in F in which the literal x appears.

The product

$$n_0 * n_1 \tag{2.164}$$

for each variable x is the number of multiplications that would be performed using the cross-product operation. The variable x is selected for which $n_0 * n_1$ is the lowest.

The best variable x to select from T to form the right side of (2.163) is one for which $n_0 * n_1 = 0$; such a variable, if it exists, is unate in the formula F which represents f. In this case, one of either P or Q would consist of no terms. Equation (2.163) would become

$$ECON(f, \{x\}) = r \tag{2.165}$$

where r comprises the terms in F which do not include the variable x. Hence, the formation of an eliminant with respect to a unate variable x consists of simple term deletions.

Suppose the set T of variables contains a subset $U \subseteq T$ consisting of two or more unate variables. In this case, the variable $x \in U$ which appears in the greatest number of terms in F with respect to the set of variables in U would be selected to form (2.165). This would cause the greatest number of terms to be deleted from F at each iteration; the resulting formula which represents r

would consist of the fewest possible terms. The subsequent operations for the remaining variables in T would tend to require fewer computations than those necessary if a different ordering of variables in U were used.

In some cases variables $x, y \in T$ may exist for which the corresponding products $n_0 * n_1$ are nonzero and equal. In such an instance the variable would be selected for which the numbers of terms in formulas representing p and q differ more. Suppose that for variable x the formula P consists of two terms and Q consists of two terms. Assume that for y the formula P includes four terms while Q has only one term. For both x and y, we would generate $n_0 * n_1 = 4$. However, we select y rather than x because the numbers of terms in the formulas P and Q differ more for y. The rationale for this choice is that when the product $p \cdot q$ in (2.163) is formed using Procedure 2.9, there is a tendency to reach a termination condition for the recursive operations faster when one of the formulas, P or Q, contains a small number of terms. The formula containing the least number of terms is the smaller of P and Q when the numbers of terms in P and Q differ the most, e.g., for y the formula Q contains only one term.

After reviewing all of the conditions for which one variable is chosen over another in T to compute (2.163), it is observed that the variable that is selected in all cases is the one which appears least opposed in the formula F. This is opposite of the criterion used to select the splitting variable in Procedure 2.5. Hence, the variable that is selected from T at each iteration is called the "least binate" variable relative to the set of variables in T.

To calculate the least binate variable, the following calculation is performed for each $x \in T$:

$$\gamma * (n_0 * n_1) - \max(n_0, n_1) \tag{2.166}$$

where γ is a large constant. A value of $\gamma \geq 10$ is sufficient. The variable x is selected for which (2.166) has the smallest value. If $U \subseteq T$ is unate in F, then $\gamma * (n_0 * n_1) = 0$ for variable in U. Subtracting $\max(n_0, n_1)$ forces the selection of the variable which appears in the most terms.

Similarly, for variables where $\gamma * (n_0 * n_1)$ is equal, the variable will be selected for which $\max(n_0, n_1)$ is the greatest—the variable which appears least opposed in F.

Some examples of using (2.166) to select a variable x from T are given in Table 2.6. Note that for $\gamma = 1$, a variable for which $n_0 = 1$ and $n_1 = 10$ would be selected rather than one for which $n_0 = 2$ and $n_1 = 2$. A value of $\gamma = 10$ precludes such an anomaly. Procedure 2.21 (Least-Binate Argument) in Appendix D is an original technique based on the foregoing discussion which selects the least binate variable from a set T of variables.

n_0	n_1	$n_0 * n_1$	$\max(n_0, n_1)$	Value of (2.166) ($\gamma = 1$)	Value of (2.166) ($\gamma = 10$)
0	5	0	5	$(1 \cdot 0) - 5 = -5$	$(10 \cdot 0) - 5 = -5$
0	6	0	6	$(1 \cdot 0) - 6 = -6$	$(10 \cdot 0) - 5 = -6$
1	4	4	4	$(1 \cdot 4) - 4 = 0$	$(10 \cdot 4) - 4 = 36$
2	2	4	4	$(1 \cdot 4) - 2 = 2$	$(10 \cdot 4) - 2 = 38$
1	10	10	10	$(1 \cdot 10) - 10 = 0$	$(10 \cdot 10) - 10 = 90$
2	5	10	5	$(1 \cdot 10) - 5 = 5$	$(10 \cdot 10) - 5 = 95$
2	10	20	20	$(1 \cdot 20) - 20 = 10$	$(10 \cdot 20) - 10 = 190$
4	5	20	5	$(1 \cdot 20) - 5 = 15$	$(10 \cdot 20) - 5 = 195$

Table 2.6. Calculation of Least-Binate Variable

Procedure 2.21 is applied in Procedure 2.22 (Conjunctive Eliminant - ECON), listed in Appendix D, which forms the conjunctive eliminant of a function f with respect to a set T of variables.

The Tautology Problem. An important problem in Boolean reasoning is to determine whether an equation of the form

$$t_1 + t_2 + \cdots + t_k = 1, \tag{2.167}$$

where each t_i is a term, is an identity. If (2.167) is an identity, then the function $t_1 + t_2 + \cdots + t_k$ is called a *tautology*. Two algorithms are presented in this section for determining whether a function is a tautology. Additionally, a problem which applies these algorithms is introduced.

Algorithms for Determining Tautology. Brayton et al. have developed a tautology algorithm which uses both Boole's Expansion Theorem and the splitting-variable heuristic presented earlier (Brayt 82). A key idea incorporated in the algorithm is that if any unate variables are identified in a formula F representing a function being tested for tautology, all terms which include the unate variables may be deleted. It can be shown that if a term of F includes a unate variable, then F is a tautology if and only if the remaining terms make F a tautology (Brayt 82:60-61). Boole's Expansion Theorem is applied recursively until a leaf function meets one of the following conditions:

- f is identically equal to 0, or
- f is identically equal to 1.

If all resulting leaf functions meet the $f = 1$ criterion, then the function is a tautology; otherwise, the function is not a tautology. Procedure 2.23 (Test for Tautology) in Appendix D implements the tautology algorithm developed by Brayton.

The second algorithm for determining whether a function f is a tautology was developed by Zakrevskii (Zakre 69:207-213). Zakrevskii's algorithm is similar to the one developed by Brayton in that Boole's Expansion Theorem is applied in a like fashion. Zakrevskii tests for the special circumstance that a term exists in the formula F which consists of a single literal x; Zakrevskii's test is actually a special case of Brayton's test for unate variables inasmuch as the function f is unate in the literal x. Boole's Expansion Theorem is applied to derive

$$f = x' \cdot f/x' + x \cdot f/x \tag{2.168}$$
$$= x' \cdot f/x' + x \cdot 1. \tag{2.169}$$

Since F includes the term x, when f/x is formed a function is derived which is identically equal to 1. In this situation, f is a tautology if and only if f/x' is a tautology. Zakrevskii's test for tautology is implemented by Procedure 2.24 (Test for Tautology) in Appendix D.

Of the two algorithms presented, Zakrevskii's algorithm is typically more efficient when the number of terms in a formula is less than 50. Brayton's algorithm is best applied when the number of terms is larger.

Sum-to-One Theorem. An application of the tautology test used in this work is to determine whether a given term t is included in a function f, i.e.,

$$t \leq f. \tag{2.170}$$

The relation of (2.170) to the tautology problem is given by the following theorem:

Let f be a Boolean function and let t be a term. Then

$$t \leq f \iff f/t = 1. \tag{2.171}$$

Thus, to determine if a term t is included in a function f, the function f is divided by t and f/t is tested to determine if it is a tautology.

Tests for Inclusion and Equivalence

Tests for Inclusion. One way to determine if a Boolean function g is included in a Boolean function h, i.e.,

$$g \leq h, \tag{2.172}$$

is to determine if each term t in the formula G which represents g is included in h. This determination may be made by applying (2.171). Hence, if $h/t = 1$ for every term t in G, then the function g is included in the function h. If a function g in included in a function h, h is said to *cover* g. The foregoing test for inclusion is implemented by Procedure 2.25 (Test for Inclusion), listed in Appendix D.

A second test for inclusion of one function in another is to use the definition of the inclusion relation and the subtraction operation. By the definition of the inclusion relation,

$$g \leq h \iff gh' = 0. \tag{2.173}$$

Hence,

$$g \leq h \Longleftrightarrow g - h = 0. \tag{2.174}$$

Thus, the inclusion of g in h is tested by using the subtraction operation to form $g - h$ and examining the result for equivalence to zero. A procedure which implements this test is given by Procedure 2.26 (Test for Inclusion) in Appendix D.

Tests for Equivalence. To test two Boolean functions g and h for equivalence, the functions may be tested for reciprocal inclusion, i.e.,

$$g = h \Longleftrightarrow g \leq h \text{ and } h \leq g. \tag{2.175}$$

Procedure 2.27 (Test for Equivalence), listed in Appendix D, applies (2.175) to test two functions for equivalence.

The test (2.175) for equivalence may be expressed equivalently as follows:

$$g = h \iff gh' + g'h = 0. \tag{2.176}$$

Since $gh' + g'h$ defines the Exclusive-OR operation, XOR may be used to test two functions for equivalence. Procedure 2.28 (Test for Equivalence) in Appendix D uses the XOR operation to test two functions for equivalence.

Tests for Membership in an Interval. When working with intervals, it often must be determined if a function f exists within a specified interval $[g, h]$, i.e.,

$$g \leq f \leq h. \tag{2.177}$$

Procedure 2.29 (Test for Membership in Interval) in Appendix D tests if a function f exists within an interval $[g, h]$ using Procedure 2.25.

An alternative method of testing for membership in an interval would be to use Procedure 2.26 to test the validity of $g - f = 0$ and $f - h = 0$. If $g - f = 0$ and $f - h = 0$, then $g \leq f \leq h$. Such a technique is given by Procedure 2.30 in Appendix D.

Computational Results. A typical application of the test for equivalence is to verify that a new formula $\widehat{F}$ developed as the result of minimization represents the same function f as an initial formula F. Procedure 2.27 is a general-purpose procedure used to test for equivalence. However, the difference in performance between Procedure 2.27 and that of Procedure 2.28 is considerable in certain cases. When a positive result is expected, i.e., $g = h$, Procedure 2.28 calculates a result much faster than does Procedure 2.27. The reason for this outcome is that when Procedure 2.27 calculates a positive result, every term in G is tested for inclusion in H and every term in H is tested for inclusion in G. On the other hand, when a negative result is expected, Procedure 2.27

executes considerably faster than Procedure 2.28. The failure of the inclusion test for one term is all that is required to determine that formulas are not equivalent. A heuristic which works well in forcing a negative result more quickly is to test terms for inclusion which have a large number of literals prior to testing terms which have a small number of literals.

In the general case, Procedure 2.29 is used to test if $g \leq f \leq h$. Using Procedure 2.29, every term of G is tested for inclusion in f and every term of F is tested for inclusion in h. Similar to the test for equivalence, when a positive result is expected the method based on Procedure 2.30 produces a result considerably faster than when using Procedure 2.29. Using Procedure 2.29, every term must be tested for inclusion to positively determine inclusion of functions. However, when a negative result is expected, Procedure 2.29 executes faster than Procedure 2.30.

Irredundant Formulas

In this section, techniques are described for generating irredundant sums of products formulas to represent both functions and intervals.

Sub-Minimal Formulas for Functions. Quine called a formula *irredundant* if it has no superfluous terms and none of its terms has superfluous literals (Quine 52). Thus, all irredundant formulas consist of prime implicants. A way to generate an irredundant formula to represent a function f is to form $BCF(f)$ and then to test each term of $BCF(f)$ for redundancy. If a term is redundant, it is deleted; otherwise, it is kept. However, the deletion of a term may cause another term, previously redundant, no longer to be redundant. Because of this characteristic, it follows that the ordering of terms in the process of testing and deleting terms is important.

Consider any subformula $\widetilde{F}$ of $BCF(f)$ that represents f. A term t that appears in $\widetilde{F}$ whose removal from $\widetilde{F}$ causes the formula to no longer represent f is called a *conditionally-essential* prime implicant provided that t is not an essential prime implicant of f. In developing an irredundant

formula, the objective is to identify a near-minimal set of conditionally-essential prime implicants to be contained in the irredundant formula.

A systematic way to form such a set is to sort the terms of $BCF(f)$ a priori such that the terms with the greatest number of literals are tested for redundancy prior to testing the terms with the fewest number of literals. The reason for this methodology is that if the cost of a formula is based on the number of literals of its constituent terms, it is advantageous to test and delete the terms with the greatest number of literals prior to testing the terms with fewer literals. It is not desirable for many terms with large numbers of literals to become conditionally essential due to the deletion of a few terms with small numbers of literals. This process does not guarantee that a least-cost formula is developed, although the results are generally good if not minimal. Hence, a formula developed in this fashion is called a *near-minimal* or *sub-minimal* formula. An irredundant formula also is called an *irredundant disjunctive form* (IDF) for a function. There typically are many irredundant disjunctive forms which may represent a function. Procedure 2.31 in Appendix D implements a process for generating a sub-minimal formula for a function; it is assumed that the Blake canonical form for the function is generated prior to using the procedure.

In some circumstances, the prime implicants of a function f which are essential may be known in advance of forming an irredundant formula. For such occurrences, Procedure 2.31 may be modified such that only the prime implicants which are not essential have to be tested for redundancy in the process of forming an IDF. Procedure 2.32 in Appendix D includes this revision.

Sub-Minimal Formulas for Intervals. Just as for functions, irredundant formulas are generated to represent an interval $[g, h]$. An irredundant formula for an interval $[g, h]$ is a formula which represents a function f belonging to the interval $[g, h]$.

It is well-known that irredundant formulas for intervals consist of sums of prime implicants of h which cover g. Therefore, a way to generate an irredundant formula to represent an interval is to form $BCF(h)$ and then test each term successively to determine if g is included in the resulting

sum of prime implicants. A prime implicant of h is *redundant* if it can be deleted and g is included in the sum of the remaining prime implicants of h. Otherwise, the prime implicant is irredundant and the next term is tested for redundancy. Just as in the case of a function, the ordering of the prime implicants of h when testing for redundancy is significant. Also similar to the case of a function, an irredundant formula which represents an interval is called an *irredundant disjunctive form* (IDF) for the interval. There typically are many irredundant disjunctive forms which may represent functions in the interval.

The terminology used to classify prime implicants of functions is adapted to describe prime implicants of h in intervals. A prime implicant of h is called an *essential* prime implicant if and only if it is necessary to cover some term of g. Hence, an essential prime implicant of h will appear in every irredundant formula which represents an f in $[g, h]$. Similarly, a function h_{ess} is formed which consists of the sum of the essential prime implicants of h. A prime implicant p of h which is not an essential prime implicant and which meets the condition

$$p \leq h_{ess} \tag{2.178}$$

is called an *inessential* prime implicant. An irredundant formula which represents f in $[g, h]$ will never contain an inessential prime implicant. Prime implicants of h which are neither essential nor inessential are called *conditionally-eliminable* prime implicants. A term t which appears in a subformula $\tilde{H}$ of $BCF(h)$ is called a *conditionally-essential* prime implicant if its removal from $\tilde{H}$ causes the condition $g \not\leq \tilde{h}$ and t is not an essential prime implicant of h. Procedure 2.33 in Appendix D implements a process for generating irredundant formulas for intervals given the Blake canonical form for h.

In some situations, we may know which prime implicants of h are essential in advance of forming an irredundant formula. In this case Procedure 2.33 is modified such that only the prime implicants which are not essential have to be tested for redundancy in the process of forming an

IDF. Procedure 2.34 in Appendix D is a modified version of Procedure 2.33 which includes this revision.

Summary

In this chapter the theoretical background and terminology which provide a foundation for subsequent chapters was presented. A great portion of this chapter was devoted to summarizing existing ideas which are used in algorithms proposed in later sections. However, a number of ideas discussed in this chapter were new results:

- A heuristic (Procedure 2.19) was described which greatly improves the efficiency of the generation via recursive multiplication of all of the prime implicants of a Boolean function (Procedure 2.20). This heuristic improves the speed with which the Blake canonical form is developed and decreases the memory usage of the recursive multiplication technique.
- It was discovered that when forming the conjunctive eliminant of a function with respect to a set of arguments, the ordering of the arguments is significant. A heuristic (Procedure 2.21) has been developed which orders the arguments in a manner that generally reduces computations and memory usage.
- A set of procedures (Procedures 2.25–2.28) was presented for evaluating inclusion and equivalence of functions. Of significance is the recognition of the efficiency of one method over another when there is a reasonable expectation of the result, i.e., when the outcome of the test is expected to be either true or false.
- The Relative Simplification Theorem was presented which allows the replacement of the function $f + g$ by $\bar{f} + g$. In general, fewer computations are necessary when applying an operation to $\bar{f} + g$ than to $f + g$.

- New Exclusive-OR (Procedure 2.12) and Exclusive-NOR (Procedure 2.14) procedures were described which are based on Boole's Expansion Theorem and which incorporate Brayton's splitting-variable heuristic and merge operation.

III. Functional Relations

A central problem in Boolean reasoning is to deduce relationships among a collection of Boolean functions. Brown presented in his book a method to deduce all relations among a collection of functions (Brown 90:138-140). This method is based on the concepts of reduction, elimination, and use of the Blake canonical form. However, often there is only a need to deduce specific relationships among a set of functions. For these circumstances, the techniques presented by Brown can be adapted to form efficient techniques for the deduction of particular relationships among a set of functions. A unified set of such techniques is presented in this chapter. Procedures are provided for:

- the deduction of normal subsets of a set of functions;
- the deduction of evanescent subsets of a set of functions; and
- the generation of implication relations.

These procedures are a prime example of the use of Boolean inference—the extraction of conclusions from a collection of Boolean data.

This chapter is primarily of theoretical interest with the exception of a method introduced for deducing the coverage of a term by subsets of a set of terms—a specialization of the method for determining implication relations. Variations of the technique for determining the coverage of a term are used in subsequent chapters. However, an understanding of Chapter 3 is not required for a reader whose main interest is in the minimization methods found in ensuing chapters.

The Label-and-Eliminate Procedure

If an equation of the form $f = 0$ is consistent, and there exists an equation $g = 0$ that

$$f = 0 \implies g = 0, \tag{3.1}$$

then the equation $g = 0$ is called a *consequent* or *conclusion* of $f = 0$. Given (3.1), we use the Extended Verification Theorem to form the equivalent statement

$$g \leq f. \tag{3.2}$$

Hence, if $f = 0$ is consistent, consequents of $f = 0$ may be formed using any function g which is included in f, i.e.,

$$g \leq f \iff (f = 0 \Rightarrow g = 0). \tag{3.3}$$

If p is a prime implicant of f, then $p \leq f$. Consequently, the statement

$$f = 0 \implies p = 0 \tag{3.4}$$

is valid. A consequent of the form $p = 0$, where p is a prime implicant of f, is called a *prime consequent* of $f = 0$.

Our particular concern is to determine relationships among a collection $f_1, f_2, \ldots, f_k : \mathbf{B}^n \rightarrow \mathbf{B}$ of n-variable Boolean functions. A method for determining such relationships will be discussed in the remainder of this section. The basis of this method is the construction of systems reducible to the form $f = 0$ such that certain prime consequents of $f = 0$ represent the relationships among the functions.

Suppose we would like to determine the relationships among the functions in the set $F = \{f_1, f_2, \ldots, f_k\}$. Let the functions be expressed in terms of the argument-vector $X = (x_1, x_2, \ldots, x_n)$. Each function is equated to an associated *label* to form a system of equations:

$$\begin{aligned} A_1 &= f_1(X) \\ A_2 &= f_2(X) \\ &\vdots \\ A_k &= f_k(X). \end{aligned} \tag{3.5}$$

The labels in the vector $A = (A_1, A_2, \ldots, A_k)$ are called *A-arguments*. The variables in the vector X are called *X-arguments*. Using the process of reduction presented in Chapter 2, system (3.5) is reduced to a single equivalent equation of the form $f(A, X) = 0$, for which $f(A, X)$ is given by

$$f(A, X) = \sum_{i=1}^{k} (A_i \oplus f_i(X)). \tag{3.6}$$

As presented in Chapter 2, a variable x_i may be eliminated from an equation $f = 0$ to form a new equation

$$ECON(f, \{x_i\}) = 0. \tag{3.7}$$

The equation $ECON(f, \{x_i\}) = 0$ is the resultant of elimination of x_i from equation $f = 0$. Clearly, the equation $ECON(f, \{x_i\}) = 0$ is a consequent of $f = 0$. We may eliminate all of the X-arguments from $f(A, X) = 0$ to form a consequent $g(A) = 0$, i.e.,

$$f(A, X) = 0 \implies g(A) = 0, \tag{3.8}$$

for which

$$g(A) = ECON(f(A, X), X). \tag{3.9}$$

In view of property (2.40), it follows that prime implicants $p(A)$ of $g(A)$ may be used to develop equations of the form

$$p(A) = 0. \tag{3.10}$$

We conclude that each equation $p(A) = 0$ is a prime consequent of $f(A, X) = 0$ due to the theorem

$$BCF(ECON(f, T)) = \sum(\text{terms of } BCF(f) \text{ not involving arguments in } T). \tag{3.11}$$

Because of this theorem, we deduce that the prime implicants $p(A)$ of the function $g(A)$ in the resultant of elimination $g(A) = 0$ of the X-arguments from the equation $f(A, X) = 0$ are also prime implicants of the function $f(A, X)$.

Each prime implicant $p(A)$ of $g(A)$ consists entirely of variables $A_1, A_2, \ldots, A_k$. Additionally, the $p(A)$ prime implicants of $f(A, X)$ are the only prime implicants of f which consist entirely of A-arguments. Any implicant r of the function $f(A, X)$ which consists entirely of A-arguments forms a consequent $r = 0$ of $f = 0$ called an *A-consequent* of $f = 0$. The term r is called an *A-consequent term*. A prime consequent which is also an A-consequent is called a *prime A-consequent*. Each prime consequent $p(A) = 0$ of $f(A, X) = 0$ is a prime A-consequent.

The prime consequents $p(A) = 0$ represent the relationships which exist among the functions in F. The labels $A_1, A_2, \ldots, A_k$ which are equated to the functions are a form of encoding through which we abstractly determine the relationships among the set of functions. The general process outlined above forms the basis for the "label-and-eliminate procedure" presented by Brown (Brown 90:139) for generating all of the prime A-consequent terms of the system (3.5). The steps of the procedure are given as follows:

Step 1. Reduce system (3.5) to an equation of the form $f(A, X) = 0$.
Step 2. Eliminate X from $f(A, X) = 0$ to form the consequent $g(A) = 0$, for which

$$g(A) = ECON(f(A, X), X). \tag{3.12}$$

Step 3. Form $BCF(g(A))$ to generate all of the prime A-consequents of (3.5).

In the next section, we explore the specific relationships among subsets of a set of functions which are represented by the A-consequents of $f(A, X) = 0$. However, we first present an example which demonstrates the application of the label-and-eliminate procedure for deducing A-consequents of $f(A, X) = 0$. In a follow-up example at the conclusion of the next section, we will discuss the specific relationships among subsets of a set of functions represented by the A-consequents of $f(A, X) = 0$ formed in Example 3.1.

Example 3.1: Suppose we would like to determine the relationships among the functions of the set $\{f_1, f_2, f_3, f_4\}$, defined by

$$\begin{aligned} f_1 &= x'yz + xyz' \\ f_2 &= xyz \\ f_3 &= yz \\ f_4 &= y' + x'z'. \end{aligned} \tag{3.13}$$

We equate the labels A_1, A_2, A_3, and A_4 to the functions, i.e.,

$$\begin{aligned} A_1 &= x'yz + xyz' \\ A_2 &= xyz \\ A_3 &= yz \\ A_4 &= y' + x'z' \end{aligned} \tag{3.14}$$

and reduce the system (3.14) to the form $f(A_1, A_2, A_3, A_4, x, y, z) = 0$. The function f is defined by

$$\begin{aligned} f(A_1, A_2, A_3, A_4, x, y, z) = {} & A_1'x'yz + A_1'xyz' + A_1y' + A_1x'z' + A_1xz + \\ & A_2'xyz + A_2x' + A_2y' + A_2z' + \\ & A_3'yz + A_3y' + A_3z' + \\ & A_4'y' + A_4'x'z' + A_4xy + A_4yz. \end{aligned} \tag{3.15}$$

Eliminating the variables x, y, z from $f = 0$ produces the consequent $g(A_1, A_2, A_3, A_4) = 0$, for which

$$g(A_1, A_2, A_3, A_4) = A_1'A_2'A_4' + A_1A_2 + A_2A_4 + A_1A_4 + A_1'A_2'A_3 + A_2A_3'. \tag{3.16}$$

Prime implicants of $g(A)$ are used to generate the prime A-consequents of $f(A, X) = 0$. The Blake canonical form of $g(A)$ is expressed as follows:

$$BCF(g(A)) = A_1'A_2'A_4' + A_1'A_3'A_4' + A_1A_2 + A_2A_4 + A_3A_4 + A_1A_4 + A_1'A_2'A_3 + A_2A_3'. \tag{3.17}$$

Using (3.17), the prime A-consequents of $f = 0$ are derived:

$$\begin{aligned} A_1'A_2'A_4' &= 0 \\ A_1'A_3'A_4' &= 0 \\ A_1A_2 &= 0 \\ A_2A_4 &= 0 \\ A_3A_4 &= 0 \\ A_1A_4 &= 0 \\ A_1'A_2'A_3 &= 0 \\ A_2A_3' &= 0. \end{aligned} \tag{3.18}$$

Each equation in (3.18) represents a specific relationship among a subset of the functions in $\{f_1, f_2, f_3, f_4\}$. Example 3.2 at the end of the next section explains the relationships represented by the equations (3.18).

Relationships Among Boolean Functions

Each A-consequent term $p(A)$ has the general form

$$a_1 \cdots a_m \cdot b_1' \cdots b_n' \tag{3.19}$$

where $a_1, \ldots, a_m$ are the uncomplemented literals of p and $b_1' \cdots b_n'$ are the complemented literals of p. The set $\{a_1, \ldots, a_m, b_1, \ldots, b_n\}$ of variables in each term p is a subset of the set $A = \{A_1, A_2, \ldots, A_k\}$ of labels. Specific forms of the term (3.19) denote specific relationships among a set $F = \{f_1, f_2, \ldots, f_k\}$ of functions. Several such relationships are normal subsets, evanescent subsets, and implications relations.

If the sum of functions in F is identically equal to one, we say that F is *normal*. A subset of F which is normal is called a *normal subset*. When an A-consequent term $p(A)$ consists only of complemented literals $b_1', b_2', \ldots, b_n'$, we deduce an A-consequent of the form

$$b_1' b_2' \cdots b_n' = 0, \tag{3.20}$$

which may be expressed equivalently as

$$b_1 + b_2 + \cdots + b_n = 1. \tag{3.21}$$

Each of the labels $b_1, b_2, \ldots, b_n$ may be replaced with its associated function from (3.5). Hence, we deduce sets of functions which are normal. An A-consequent term which appears in Example 3.1 is $A_1' A_2' A_4'$. Thus, the functions f_1, f_2, and f_4 form a normal set of functions, i.e.,

$$(x'yz + xyz') + (xyz) + (y' + x'z') = 1. \tag{3.22}$$

Each A-consequent term in $BCF(g(A))$ is a prime implicant. Since a literal cannot be removed from a prime implicant of a function and still be an implicant of the function, no literal can be deleted from the resulting A-consequent terms. Consequently, each prime A-consequent term consisting only of complemented literals represents a minimal subset of F which is normal. The collection of all A-consequent terms of the form $b_1' b_2' \cdots b_n'$ denotes all possible ways in which a subset of the

set F may be put together to form a normal subset. As a special case of normal subsets, normal subsets of a set F of functions are called *sum-to-one* subsets if each function in F is a single term.

If the product of functions in F is identically the zero-function, F is said to be *evanescent*. An *evanescent subset* is a subset of F which is evanescent. When an A-consequent term $p(A)$ consists only of uncomplemented literals $a_1, a_2, \ldots, a_m$, the corresponding A-consequent has the form

$$a_1 a_2 \cdots a_m = 0. \tag{3.23}$$

By replacing each of the labels $a_1, a_2, \ldots, a_m$ with its associated function from (3.5), we deduce sets of functions which are evanescent. In Example 3.1, one A-consequent term is $A_1 A_2$. Thus the functions f_1 and f_2 form an evanescent subset, i.e.,

$$(x'yz + xyz') \cdot (xyz) = 0. \tag{3.24}$$

Forming A-consequent terms from terms in $BCF(g(A))$, each A-consequent term is a prime implicant. Since a literal cannot be removed from a prime implicant of a function and still be an implicant of the function, no literal can be deleted from the resulting A-consequent terms. Thus, each A-consequent term in $BCF(g(A))$ consisting only of uncomplemented literals represents a minimal evanescent subset of F. The collection of all A-consequent terms of the form $a_1 a_2 \cdots a_m$ in $BCF(g(A))$ denotes all possible ways in which subsets of F form minimal evanescent subsets. If each function in a set F of functions is a single term, evanescent subsets of F are denoted by the term *product-of-zero subsets*.

We consider a third relationship among functions: that of implication. Given the A-consequent

$$a_1 \cdots a_m \cdot b_1' \cdots b_n' = 0, \tag{3.25}$$

DeMorgan's Law and the definition of the inclusion relation may be applied to derive the equivalent statement:

$$a_1 \cdots a_m \leq b_1 + \cdots + b_n. \tag{3.26}$$

When an A-consequent term consists of a single uncomplemented literal a_j and complemented literals $b'_1, b'_2, \ldots, b'_n$, statement (3.26) becomes

$$a_j \leq b_1 + \cdots + b_n. \tag{3.27}$$

Relation (3.27) is sometimes written as

$$a_j \implies b_1 + \cdots + b_n. \tag{3.28}$$

Hence, we say that an A-consequent term of the form $a_j b'_1 \cdots b'_n$ denotes an *implication relation.* Suppose the label a_j is associated with the function f_j in (3.5); moreover, let the labels $b_1, \ldots, b_n$ be associated with the functions $f_1, \ldots, f_n$. An A-consequent term such as (3.27) then represents the coverage of f_j by the sum of the functions $f_1, \ldots, f_n$, i.e,

$$f_j \leq f_1 + \cdots + f_n. \tag{3.29}$$

If the A-consequent terms are formed from prime implicants of $g(A)$, then no literal can be deleted from the resulting A-consequent terms and still have the term be an implicant of $g(A)$. Consequently, each A-consequent term of the form $a_j b'_1 \cdots b'_n$ denotes a minimal subset of functions necessary to cover the function associated with a_j. Implication relations in which a function is included in a minimal subset of functions are called *irredundant implication relations* (IIRs). All

A-consequent terms of the form $a_j b_1' \cdots b_n'$ generated using the label-and-eliminate procedure represent IIRs.

Example 3.2 discusses the relationships among a set of functions represented by the prime A-consequents developed in Example 3.1.

Example 3.2: In Example 3.1, the following prime A-consequents of $f = 0$ were developed:

$$\begin{aligned} A_1'A_2'A_4' &= 0 \\ A_1'A_3'A_4' &= 0 \\ A_1A_2 &= 0 \\ A_2A_4 &= 0 \\ A_3A_4 &= 0 \\ A_1A_4 &= 0 \\ A_1'A_2'A_3 &= 0 \\ A_2A_3' &= 0. \end{aligned} \tag{3.30}$$

The A-consequents $A_1'A_2'A_4' = 0$ and $A_1'A_3'A_4' = 0$ denote the existence of normal subsets $\{f_1, f_2, f_4\}$ and $\{f_1, f_3, f_4\}$. The A-consequents $A_1A_2 = 0$, $A_2A_4 = 0$, $A_3A_4 = 0$, and $A_1A_4 = 0$ signify the existence of the evanescent subsets $\{f_1, f_2\}$, $\{f_2, f_4\}$, $\{f_3, f_4\}$, and $\{f_1, f_4\}$, respectively. Finally, the A-consequents $A_1'A_2'A_3 = 0$ and $A_2A_3' = 0$ represent the implication relations $f_3 \leq f_1 + f_2$ and $f_2 \leq f_3$, respectively.

Normal Subsets

In many circumstances, we desire to know the normal subsets of a set $F = \{f_1, f_2 \ldots, f_k\}$ of functions. For example, a common problem is to determine the sum-to-one subsets among a set of terms. The label-and-eliminate procedure may be used to determine normal subsets by forming all prime A-consequent terms and selecting only those A-consequent terms in which all of the literals are complemented. Unfortunately, the label-and-eliminate procedure is inefficient when we only require particular A-consequent terms. A better approach to determining normal subsets is to

formulate a procedure which produces only prime A-consequent terms of the form which denotes normal subsets of F. Such a procedure is presented in this section.

The Partial Labeling-and-Reduction Process. In the label-and-eliminate procedure, a system (3.5) is developed in which a label is associated with each function in F by forming an equality between a label and its associated function. The system is then reduced to an equation of the form $f(A, X) = 0$. We denote this process the *full labeling-and-reduction* of a set of functions. In the formula $F(A, X)$ representing $f(A, X)$ terms exist which include both complemented and uncomplemented literals of variables in the set $A = \{A_1, A_2, \ldots, A_k\}$ of labels. $BCF(f)$ may include prime implicants consisting entirely of variables in A in which literals are both complemented and uncomplemented. However, it would be advantageous if we could develop a function $f(A, X)$ such that we could ensure that prime implicants of f consisting entirely of variables in A contain only complemented literals—our intent is to produce only A-consequents of the form (3.20).

To derive such a function, we make the supposition that the initial formula $F(A, X)$ which represents $f(A, X)$ should not contain terms which include uncomplemented literals of variables in the set A of labels. Hence, we formulate a revised system consisting of a set of inequalities which is reduced to an equation of the form $f(A, X) = 0$. For a set F of functions, we form the system

$$\begin{aligned} f_1(X) &\leq A_1 \\ f_2(X) &\leq A_2 \\ &\vdots \\ f_k(X) &\leq A_k. \end{aligned} \tag{3.31}$$

The definition of inclusion is used to produce the equivalent system

$$\begin{aligned} f_1(X) \cdot A_1' &= 0 \\ f_2(X) \cdot A_2' &= 0 \\ &\vdots \\ f_k(X) \cdot A_k' &= 0. \end{aligned} \tag{3.32}$$

Thus, (3.31) is equivalent to $f(A, X) = 0$, for which f is defined by

$$f(A, X) = \sum_{i=1}^{k} (f_i(X) \cdot A_i'). \tag{3.33}$$

Given the function $f(A, X)$ defined by (3.33), we now must show that $f(A, X)$ produces A-consequent terms which represent normal subsets. Additionally, we have to demonstrate that the A-consequent terms realized from $f(A, X) = 0$ contain only complemented literals from A.

In Theorem 3.1 we prove that $f(A, X)$ as defined by (3.33) may be used to produce A-consequent terms which represent normal subsets. The idea behind this theorem is that a subset of the set F of functions forms a normal subset if and only if the A-consequent term which denotes the normal subset is an implicant of $f(A, X)$. If an implicant exists which consists entirely of complemented literals from the set A of variables, then a prime implicant consisting of a subset of the literals must necessarily exist. For the purposes of this theorem, whether an implicant is or is not prime is not as critical as showing that an implicant exists if and only if a normal subset exists. Terms in $BCF(f(A, X))$ which consist only of A-arguments represent minimal normal subsets.

Theorem 3.1: *Given a set $F = \{f_1, f_2, \ldots, f_k\}$ of functions and a system*

$$\begin{aligned} f_1(X) &\leq A_1 \\ f_2(X) &\leq A_2 \\ &\vdots \\ f_k(X) &\leq A_k \end{aligned} \tag{3.34}$$

reduced to the form $f(A, X) = 0$, *a subset* $\{f_1, f_2, \ldots, f_s\}$ *of the set* F *is a normal subset if and only if* $A'_1 A'_2 \cdots A'_s$ *is an implicant of* $f(A, X)$.

Proof. We first show that if a subset $\{f_1, f_2, \ldots, f_s\}$ of F is normal, then $A'_1 A'_2 \cdots A'_s \leq f(A, X)$. If $\{f_1, f_2, \ldots, f_s\}$ is normal, then

$$f'_1(X) f'_2(X) \cdots f'_s(X) = 0. \tag{3.35}$$

Hence, $f'_1(X) f'_2(X) \cdots f'_s(X)$ is equal to the zero-function. Expressing $f(A, X)$ in the form

$$f_1(X)A'_1 + f_2(X)A'_2 + \cdots + f_s(X)A'_s + f_{s+1}(X)A'_{s+1} + \cdots + f_k(X)A'_k \qquad (s \leq k), \tag{3.36}$$

the zero-function may be added to (3.36) and the result will still represent $f(A, X)$. Hence, $f(A, X)$ may be represented by the form

$$\begin{aligned} &f'_1(X) f'_2(X) \cdots f'_s(X) + f_1(X)A'_1 + f_2(X)A'_2 \\ &+ \cdots + f_s(X)A'_s + f_{s+1}(X)A'_{s+1} + \cdots + f_k(X)A'_k \qquad (s \leq k). \end{aligned} \tag{3.37}$$

Since $f'_1(X) f'_2(X) \cdots f'_s(X)$ and $f_1(X)A'_1$ form a consensus term $f'_2(X) \cdots f'_s(X)A'_1$, by the property of consensus (2.32) we may represent (3.37) equivalently by

$$\begin{aligned} &f'_2(X) \cdots f'_s(X)A'_1 + f'_1(X) f'_2(X) \cdots f'_s(X) + f_1(X)A'_1 \\ &+ f_2(X)A'_2 + \cdots + f_s(X)A'_s + f_{s+1}(X)A'_{s+1} + \cdots + f_k(X)A'_k \qquad (s \leq k). \end{aligned} \tag{3.38}$$

Similarly, $f_2'(X)\cdots f_s'(X)A_1'$ and $f_2(X)A_2'$ form a consensus term $f_3'(X)\cdots f_s'(X)A_1'A_2'$. It follows that (3.38) is equivalent to the form

$$\begin{aligned} &f_3'(X)\cdots f_s'(X)A_1'A_2' + f_2'(X)\cdots f_s'(X)A_1' + f_1'(X)f_2'(X)\cdots f_s'(X) \\ &+ f_1(X)A_1' + f_2(X)A_2' + \cdots + f_s(X)A_s' + f_{s+1}(X)A_{s+1}' + \cdots + f_k(X)A_k' \qquad (s \le k). \end{aligned} \tag{3.39}$$

We continue forming consensus terms in the same fashion until $f_s'(X)A_1'A_2'\cdots A_{s-1}'$ and $f_s(X)A_s'$ form the consensus term $A_1'A_2'\cdots A_s'$. It follows that (3.39) is expressed equivalently by

$$\begin{aligned} &A_1'A_2'\cdots A_s' + f_s'(X)A_1'A_2'\cdots A_{s-1}' + \cdots + f_2'(X)\cdots f_s'(X)A_1' \\ &+ f_1'(X)f_2'(X)\cdots f_s'(X) + f_1(X)A_1' + f_2(X)A_2' \\ &+ \cdots + f_s(X)A_s' + f_{s+1}(X)A_{s+1}' + \cdots + f_k(X)A_k' \qquad (s \le k). \end{aligned} \tag{3.40}$$

We thus conclude that the term $A_1'A_2'\cdots A_s'$ appears in a sum-of-products formula which represents $f(A,X)$. Since any term which appears in an SOP formula which represents a function f is an implicant of f, it follows that $A_1'A_2'\cdots A_s'$ is an implicant of $f(A,X)$.

Assume on the other hand that $A_1'A_2'\cdots A_s'$ is an implicant of $f(A,X)$. Then the statement

$$A_1'A_2'\cdots A_s' \le \sum_{i=1}^{k}(f_i(X)\cdot A_i') \qquad (s \le k). \tag{3.41}$$

is valid for all substitutions for X and for $(A_1, A_2, \ldots, A_k)$. Equivalently,

$$\sum_{i=1}^{s}(A_i) + \sum_{i=1}^{k}(f_i(X)\cdot A_i') = 1 \qquad (s \le k). \tag{3.42}$$

The equation

$$\sum_{i=1}^{s}(A_i + f_i(X)) + \sum_{i=s+1}^{k}(f_i(X)\cdot A_i') = 1 \qquad (s \le k) \tag{3.43}$$

is equivalent to (3.42) by Property (2.30). Since (3.43) is an identity, i.e., it holds for all substitutions for X and for $(A_1, A_2, \ldots, A_k)$, it holds if

$$A_1 = A_2 = \cdots = A_s = 0 \tag{3.44}$$

and

$$A_{s+1} = A_{s+2} = \cdots = A_k = 1. \tag{3.45}$$

Thus, the equation

$$\sum_{i=1}^{s}(f_i(X)) = 1 \tag{3.46}$$

is an identity; whence, $\{f_1, f_2, \ldots, f_s\}$ is a normal subset of F. This completes the proof. □

Lemma 3.1 shows that if a variable in a formula is unate, then the variable is unate in the Blake canonical form of the corresponding function. This lemma is required to show that only A-consequent terms of the desired form are deduced from system (3.31).

Lemma 3.1: *Given a formula F which represents a Boolean function f, any variable which is unate in F is also unate in $BCF(f)$.*

Proof. $BCF(f)$ may be formed from an SOP representation F for f as follows:

Step 1. Generate consensus terms, beginning with F, until no new terms can be formed.
Step 2. Delete absorbed terms.

Any formula may be expressed equivalently by a formula which contains consensus terms formed from terms within it. Similarly, an equivalent formula may be formed by deleting absorbed terms. It follows that a formula developed by applying Steps 1 and 2 is equivalent to F; the resulting

formula is $BCF(f)$. (This method is similar to the successive extraction technique for forming a Blake canonical form described in Chapter 2.)

From terms $x's$ and xt, Step 1 produces a consensus term st. Thus, Step 1 cannot introduce a literal not already present. It is clear that Step 2 as well cannot introduce a new literal. Thus, the Blake canonical form developed from an SOP formula F cannot contain a literal that is not present in F. This completes the proof. □

Theorem 3.2 shows that the A-consequent terms deducible from system (3.31) exist only in the form which represents normal subsets of a set F of functions.

Theorem 3.2: *Given a system*

$$\begin{array}{rcl} f_1(X) & \leq & A_1 \\ f_2(X) & \leq & A_2 \\ & \vdots & \\ f_k(X) & \leq & A_k. \end{array} \tag{3.47}$$

reducible to the form $f(A, X) = 0$, *the A-consequent terms of the prime A-consequents* $p(A) = 0$ *of* $f(A, X) = 0$ *consist only of complemented literals of the set* $A = \{A_1, A_2, \ldots, A_k\}$ *of variables.*

Proof. System (3.47) may be expressed equivalently as

$$\begin{array}{rcl} f_1(X) \cdot A_1' & = & 0 \\ f_2(X) \cdot A_2' & = & 0 \\ & \vdots & \\ f_k(X) \cdot A_k' & = & 0. \end{array} \tag{3.48}$$

By (2.40), system (3.48) is reduced to an equation of the form $f(A, X) = 0$, for which

$$f(A, X) = f_1(X)A_1' + f_2(X)A_2' + \cdots + f_k(X)A_k'. \tag{3.49}$$

A formula to represent $f_1(X)A_1' + f_2(X)A_2' + \cdots + f_k(X)A_k'$ is constructed in the following manner:

1. For each function f_i, prefix the complement of its associated label A_i' to each term of the formula which represents f_i. The resulting formula represents the function $f_i \cdot A_i'$.
2. Append together all formulas formed in Step 1. The resulting formula represents $f_1(X)A_1' + f_2(X)A_2' + \cdots + f_k(X)A_k'$. Let us call this formula $F(A, X)$.

Because the formulas which represent each function $f_i(X)$ do not involve any variable of the set A, and the formula $F(A, X)$ is formed so that only the complemented forms of the variables $A_1, A_2, \ldots, A_k$ appear in it, the set A of variables are unate in $F(A, X)$. By Lemma 3.1, the variables in A are unate in $BCF(f(A, X))$. It follows that if any prime A-consequents $p(A) = 0$ of $f(A, X) = 0$ exist, then the corresponding A-consequent terms $p(A)$ consist only of complemented literals of the set A of variables. This completes the proof. □

We have proven the two properties of the equation $f(A, X) = 0$ reduced from system (3.31) that demonstrate that system (3.31) may be *used* as the basis for economically determining only those subsets of a given set of functions which are normal. We summarize these properties:

1. If a subset of a set F of functions is normal, then we can deduce A-consequents terms which represent normal subsets from the equation $f(A, X) = 0$ reduced from system (3.31). Such A-consequent terms are implicants of $f(A, X)$. This property was demonstrated by Theorem 3.1.
2. All A-consequent terms deduced from the equation $f(A, X) = 0$ reduced from system (3.31) represent normal subsets. This property was demonstrated by Theorem 3.2 in which it was shown that the A-consequent terms of the prime A-consequents $p(A) = 0$ of $f(A, X) = 0$ consist only of complemented literals of the set A of variables, i.e., the A-consequent terms deducible from system (3.31) exist only in the form which represents normal subsets of a set F of functions.

We denote the process by which we use the inclusion relation to associate labels with functions in (3.31) and the entailing reduction to the form $f(A, X) = 0$ the *partial labeling-and-reduction* of a set F of functions. There are several significant advantages of the partial labeling-and-reduction process versus full labeling-and-reduction. Foremost is the fact that less time and space is needed

to generate A-consequent terms using the partial label-and-reduce process. When a label A_i is equated to a function f_i, we reduce to an equation of the form $f(A, X) = 0$ via (2.38), i.e.,

$$f_i = A_i \iff f_i A_i' + f_i' A_i = 0. \tag{3.50}$$

When an inclusion is used to associate a label with a function, we apply the relationship

$$f_i \leq A_i \iff f_i A_i' = 0. \tag{3.51}$$

Hence, we do not have to perform the computations to form each $f_i' A_i$, and the terms which represent each $f_i' A_i$ do not have to be stored. The function $f(A, X)$ produced using the partial label-and-reduce process also is preferable when forming the consequent $g(A) = 0$, for which

$$g(A) = ECON(f(A, X), X), \tag{3.52}$$

and for forming $BCF(g(A))$. The reason for this is discussed in the next section.

Goal-Directed Elimination. After the system

$$\begin{aligned} f_1(X) &\leq A_1 \\ f_2(X) &\leq A_2 \\ &\vdots \\ f_k(X) &\leq A_k \end{aligned} \tag{3.53}$$

is reduced to the form $f(A, X) = 0$, the next step for deducing normal subsets is to form the consequent $g(A) = 0$, for which

$$g(A) = ECON(f(A, X), X). \tag{3.54}$$

The formula which represents function $f(A, X)$ contains terms comprising A-arguments as well as X-arguments. When $f(A, X)$ is formed using the partial label-and-reduce process, the A-arguments are unate in $F(A, X)$. The X-arguments generally are binate in $F(A, X)$. When the conjunctive eliminant of f with respect to X is formed, a multiplication process is performed iteratively. We review the process of forming a conjunctive eliminant for a single variable x:

1. Partition the formula F which represents f into terms which include the literal x', terms which include the literal x, and terms which include neither x nor x'.
2. Given the partitioning, express F in the following manner:

$$f(x, y, \ldots) = x'p(y, \ldots) + xq(y, \ldots) + r(y, \ldots) \tag{3.55}$$

   where
   - p comprises the terms of F which include the literal x' with the literal divided out,
   - q comprises the terms of F which include the literal x with the literal divided out, and
   - r comprises the terms of F which include neither x nor x'.
3. Form $ECON(f, \{x\})$:

$$ECON(f, \{x\}) = p \cdot q + r \tag{3.56}$$

An operation of the form (3.56) with a product $p \cdot q$ is performed for each variable $x \in \{x_1, \ldots, x_n\}$ when forming $ECON(f(A, X), X)$. We assume that the product operation used when forming a conjunctive eliminant maintains the "unateness" of a collectively unate variable.[1] The resulting eliminant $ECON(f(A, X), X)$ consists only of A-arguments. Since the A-arguments are unate in $F(A, X)$, they are unate in the formula representing $ECON(f(A, X), X)$.

During each iteration of forming $ECON(f(A, X), X)$ from the function $f(A, X)$, one variable $x \in \{x_1, \ldots, x_n\}$ is eliminated. Since the X-arguments typically are binate, each iteration typically produces a function which has fewer binate variables. Hence, the resulting function at each iteration is typically "less" binate and "more" unate—until the final iteration, when the function is unate in the A-variables. The fact that the function resulting from each iteration of forming the eliminant

[1] In the product operations presented earlier, a collectively unate variable in the multiplied formulas is unate in the resulting formula. See Procedures 2.2, 2.3, and 2.9.

becomes more unate is useful in formulating a special process for constructing a conjunctive eliminant which is more efficient than the procedure described in Chapter 2. We call this special process the "quick" method for forming the conjunctive eliminant; for brevity, we will refer to the quick method for forming the conjunctive eliminant as "quick econ" (quick *E*liminant-*CON*junctive).

Several heuristics are employed in quick econ which differentiate it from other methods for forming a conjunctive eliminant. There are two differences between quick econ and the technique for forming the conjunctive eliminant described by Procedure 2.22. These are:

- the use of the cross-product (Procedure 2.2) to perform multiplication as opposed to expansion-based multiplication (Procedure 2.9); and
- the use of absorption to simplify the resulting formula after each iteration of the process rather than the simplification technique implemented by Procedure 2.15.

A similarity between quick econ and the technique given by Procedure 2.22 is that the least-binate-argument heuristic is used to order the arguments with respect to which we are forming the conjunctive eliminant. Example 3.3 demonstrates the application of quick econ to a function $f(A, X)$.

Example 3.3: Suppose we are given a system

$$\begin{aligned} xy + x'y' + z &\leq A_1 \\ x'y + xy' &\leq A_2 \\ x + z &\leq A_3 \\ x' + y' &\leq A_4 \end{aligned} \tag{3.57}$$

created by associating labels A_1, A_2, A_3, and A_4 with a set of functions as in (3.31) which is reducible to the form $f(A, X) = 0$. The A-variables are defined by the vector $A = (A_1, A_2, A_3, A_4)$ and the X-variables are defined by $X = (x, y, z)$. The function $f(A, X)$ is thus defined by the equation

$$
\begin{aligned}
f(A,X) \;=\;& A'_1xy + A'_1x'y' + A'_1z + \\
& A'_2x'y + A'_2xy' + \\
& A'_3x + A'_3z + \\
& A'_4x' + A'_4y'.
\end{aligned}
\tag{3.58}
$$

We eliminate the X-variables from $f(A,X) = 0$ to form the consequent $g(A) = 0$. We first eliminate the variable z, i.e., form the function $ECON(f(A,X),\{z\})$. Using the form (3.56) to generate $ECON(f(A,X),\{z\})$, we develop

$$
\begin{aligned}
p_z &= 0 \\
q_z &= A'_1 + A'_3 \\
r_z &= A'_1xy + A'_1x'y' + A'_2x'y + A'_2xy' + A'_3x + A'_4x' + A'_4y'.
\end{aligned}
\tag{3.59}
$$

Since $p_z \cdot q_z = 0$, it follows that $ECON(f(A,X),\{z\}) = r_z$. Hence, we deduce the equation

$$
ECON(f(A,X),\{z\}) = A'_1xy + A'_1x'y' + A'_2x'y + A'_2xy' + A'_3x + A'_4x' + A'_4y'. \tag{3.60}
$$

We eliminated z rather than x or y because z is unate in the formula on the right-hand side of (3.58); hence $ECON(f(A,X),\{z\})$ is formed by deleting the terms in $F(A,X)$ which contain the literal z.

The variable y is eliminated to form the function $ECON(ECON(f(A,X),\{z\}),\{y\})$. We thus derive

$$
\begin{aligned}
p_y &= A'_1x' + A'_2x + A'_4 \\
q_y &= A'_1x + A'_2x' \\
r_y &= A'_3x + A'_4x'.
\end{aligned}
\tag{3.61}
$$

We use the term-by-term product of P_y and Q_y to form $p_y \cdot q_y$. We thus develop the function $ECON(ECON(f(A,X),\{z\}),\{y\})$ defined by the equation

$$ECON(ECON(f(A,X),\{z\}),\{y\}) = A'_1A'_2x' + A'_1A'_2x + A'_1A'_4x + A'_2A'_4x' + A'_3x + A'_4x'. \quad (3.62)$$

The variable y is eliminated rather than x because the formula on the right-hand side of (3.60) includes three terms containing the literal x and three terms containing the literal x'; on the other hand, the formula includes two terms containing the literal y and three terms containing the literal y'. Thus, the number of terms resulting from the $P \times Q$ operation is nine if the variable x were to be eliminated, but only six if the variable y is eliminated. We eliminate y since the number of term-by-term products is less than if x were eliminated.

Finally, the function $ECON(ECON(ECON(f(A,X),\{z\}),\{y\}),\{x\})$ is formed by eliminating the variable x. Prior to eliminating x, however, we delete the absorbed term $A'_2A'_4x'$ in (3.62). We then develop

$$\begin{aligned} p_x &= A'_1A'_2 + A'_4 \\ q_x &= A'_1A'_2 + A'_1A'_4 + A'_3 \\ r_x &= 0. \end{aligned} \quad (3.63)$$

The function $ECON(ECON(ECON(f(A,X),\{z\}),\{y\}),\{x\})$ is equal to $p_x \cdot q_x + r_x$. Using the term-by-term product of P_x and Q_x, we derive the formula

$$A'_1A'_2 + A'_1A'_2A'_4 + A'_1A'_2A'_3 + A'_1A'_2A'_4 + A'_1A'_4 + A'_3A'_4. \quad (3.64)$$

Formula (3.64) is not absorptive. The equivalent absorptive formula,

$$A_1'A_2' + A_1'A_4' + A_3'A_4', \tag{3.65}$$

represents $ECON(ECON(ECON(f(A,X),\{z\}),\{y\}),\{x\}) = ECON(f(A,X),X)$. Since $g(A) = ECON(f(A,X),X)$, it follows that

$$g(A) = A_1'A_2' + A_1'A_4' + A_3'A_4'. \tag{3.66}$$

The formula which represents $ECON(ECON(f(A,X),\{z\}),\{y\})$ is more unate than the formula which represents $ECON(f(A,X),\{z\})$ since the right-hand side of (3.62) contains only one binate variable, x, while the right-hand side of (3.60) is binate in x and y. Furthermore, the formula that results after eliminating all of the X-variables, (3.64), is unate in all of its component A-variables. Hence, the formula which represents the function resulting after each iteration of elimination is "more" unate than the formula resulting from the preceding iteration. Moreover, less work is performed in developing the formula which represents $ECON(f(A,X),X)$ if the X-variables are eliminated in the order of "binateness", i.e., eliminate unate variables first and the most-binate variables last.

Quick econ is particularly useful when the function resulting after each iteration is more unate than the previous function. Because the formula which represents the function contains fewer opposed literals, the chances that a simplification procedure can reduce the number of terms decrease after each iteration. Making the formula absorptive after each iteration works about as well as simplification to reduce the number of terms to a manageable level. After all iterations have been carried out, the unate formula which results is reduced to an minimal number of terms by developing the equivalent absorptive formula. Additionally, because the function is more unate after each iteration, as we proceed the cross-product technique for performing multiplication does

not produce significantly more terms in the resulting formula than the expansion-based product operation. The primary motive for using the expansion-based product versus a cross-product operation is that the number of terms in the formula produced by the expansion-based product operation is significantly fewer than when using the cross-product operation; the cross-product always returns a result quicker than does the expansion-based product. Extraneous terms which are created using the cross-product operation generally are absorbed by other terms.

The heuristic for ordering variables with respect to which the conjunctive eliminant is formed is the least-binate-variable heuristic described in Chapter 2. The motivation for this heuristic was illustrated in Example 3.3. The least-binate-variable heuristic is a measure of the number of term-by-term products required to perform the $p \cdot q$ product in a given iteration of forming the conjunctive eliminant. Hence, the least-binate-variable heuristic is a direct measure of the complexity of the product operation when using the cross-product operation. The use of this heuristic is based on the supposition that the least amount of work will be done in the process of forming the conjunctive eliminant with respect to a set of variables if we always choose the variable at the outset of an iteration which will cause us to do the least work in that iteration. In practice, this seems to work well, especially when absorbed terms are removed from the formula after each iteration.

The quick method for forming the conjunctive eliminant of a function with respect to a set of variables is given by Procedure 3.1.

Procedure 3.1 (Quick Conjunctive Eliminant - QUICK-ECON): Given a Boolean function f and a set T of literals, the conjunctive eliminant of f with respect to T, $ECON(f,T)$, is found as follows:

Step 1.

- If T is empty, then return F. It is $ECON(f,T)$.
- Of the variables in set T, determine the least-binate variable using Procedure 2.21. Call this variable x.

Step 2.

- If $F \equiv 0$, then return a formula F which represents 0. It is $ECON(f, T)$.
- If $F \equiv 1$, then return a formula F which represents 1. It is $ECON(f, T)$.
- Otherwise, continue to Step 3.

Step 3. Partition the terms of F into the following sets:

- P, the terms of F which include the literal x' with the literal divided out;
- Q, the terms of F which include the literal x with the literal divided out; and
- R, the terms of F which include neither x nor x'.

Using Procedure 2.2 (Cross-Product), form $P \times Q$. Append the result to R. The resulting formula represents $ECON(f, \{x\})$.

Step 4. Form the equivalent absorptive formula of the formula generated in Step 3. Replace the contents of F with the absorptive formula and return to Step 1.

Our intent in using the quick method for forming the conjunctive eliminant is to efficiently generate the consequent $g(A) = 0$, the resultant of elimination of X from $f(A, X) = 0$, for which

$$g(A) = ECON(f(A, X), X). \tag{3.67}$$

Formation of the conjunctive eliminant of f with respect to X is a vehicle for eliminating the X-arguments from $f(A, X) = 0$. A unique aspect of the elimination process in this instance is that the forms of the functions $f(A, X)$ and $g(A)$ are known a priori. This allows us to guide the elimination process so that we efficiently proceed from $f(A, X) = 0$ to the equation $g(A) = 0$ with fewer computations than in the general case of forming an eliminant; thus, we achieve our goal of obtaining $g(A)$ as efficiently as possible. Therefore, we call this methodology of eliminating the variables X from $f(A, X) = 0$ to derive the equation $g(A) = 0$—in particular to derive the function $g(A)$—*goal-directed elimination*. In the next section, we discuss how $g(A) = 0$ is used to produce the A-consequent terms with which we identify normal subsets.

Formation of A-Consequent Terms. To determine all normal subsets of a set F of functions, we must form all of the prime A-consequent terms of $f(A, X) = 0$, where $f(A, X) = 0$ is equivalent to the system

$$\begin{aligned} f_1(X) &\leq A_1 \\ f_2(X) &\leq A_2 \\ &\vdots \\ f_k(X) &\leq A_k. \end{aligned} \tag{3.68}$$

One way to obtain the terms $p(A)$ of each of the prime A-consequents $p(A) = 0$ of $f(A, X) = 0$ is by forming $BCF(f(A, X))$ and selecting those terms which consist only of A-arguments. From an earlier stated theorem (2.155), we know that these terms also may be obtained by forming the conjunctive eliminant of $f(A, X)$ with respect to X and placing the result in Blake canonical form, i.e., $BCF(ECON(f(A, X), X))$. As the outcome of goal-directed elimination we develop the equation $g(A) = 0$, for which

$$g(A) = ECON(f(A, X), X). \tag{3.69}$$

Hence, to obtain all of the prime A-consequent terms of $f(A, X)$ we form $BCF(g(A))$.

Terms of $BCF(g(A))$ consist only of complemented literals of the set A. The formula G which represents $g(A)$ produced in the process of goal-directed elimination consists only of unate A-arguments. Since there are no opposed literals in G we cannot form any consensus terms; therefore all prime implicants of $g(A)$ appear in formula G. Any term of G which is not a prime implicant of $g(A)$ is necessarily absorbed by another term in G. Thus, to form $BCF(g(A))$ we form $ABS(G)$, which produces an absorptive formula equivalent to G to represent $g(A)$. By formulating a system in the form (3.68), reducing the system to the form $f(A, X) = 0$, and eliminating the X-arguments using goal-directed elimination, we realize the added benefit of producing a unate function $g(A)$ for which we generate the Blake canonical form via simple absorption. Furthermore, since the quick method for forming the conjunctive eliminant makes the resulting formula absorptive after each iteration, the formula which represents the function $ECON(f(A, X), X)$ is absorptive.

Because $g(A) = ECON(f(A,X),X)$, the function $g(A)$ is represented by its in Blake canonical form after the goal-directed elimination process. Hence, the elimination of X produces the consequent $BCF(g(A)) = 0$. Each term,

$$A_1' A_2' \cdots A_s', \tag{3.70}$$

of $BCF(g(A))$ denotes the existence of a normal subset

$$\{f_1, f_2, \ldots, f_s\}. \tag{3.71}$$

Determination of Minimal Normal Subsets. Given the partial label-and-reduce and goal-directed elimination processes, we have the mechanisms to form a procedure for determining all minimal normal subsets among a set F of functions. This method is stated by Procedure 3.2. *Example 3.4 demonstrates the use* of Procedure 3.2 to determine the normal subsets among a set of functions.

Procedure 3.2 (Minimal Normal Subsets): Given a set $F = \{f_1, f_2, \ldots, f_k\}$ of Boolean functions, we determine all of the normal subsets among the set F of functions in the following manner:

Step 1. For each function $f_i \in F$:

1. Generate an associated label A_i.
2. Prefix each term of the formula which represents f_i with the complemented literal A_i'.

The resulting formula represents $A_i' \cdot f_i$.

Step 2. Append together each of the formulas developed in Step 1. The resulting formula represents $f(A,X)$.

Step 3. Using the quick method for forming the conjunctive eliminant (Procedure 3.1), form $ECON(f(A,X),X)$. The resulting function is equal to $g(A)$; the formula which represents $g(A)$ is $BCF(g(A))$.

Step 4. Each term of $BCF(g(A))$ represents a normal subset. For each term of $BCF(g(A))$, determine the labels contained in the term and replace the labels by the functions with which they are associated. Return each subset of functions.

Example 3.4: Suppose we would like to determine the normal subsets among the functions of the set $\{f_1, f_2, f_3, f_4, f_5\}$ defined by

$$\begin{aligned} f_1 &= x_1'x_2 + x_1x_2' \\ f_2 &= x_1x_2 + x_1'x_2' \\ f_3 &= x_1'x_2 \\ f_4 &= x_1x_2' \\ f_5 &= x_1. \end{aligned} \tag{3.72}$$

Step 1. A system of the form (3.31) is created by associating labels A_1, A_2, A_3, A_4, and A_5 with the respective functions, i.e.,

$$\begin{aligned} x_1'x_2 + x_1x_2' &\leq A_1 \\ x_1x_2 + x_1'x_2' &\leq A_2 \\ x_1'x_2 &\leq A_3 \\ x_1x_2' &\leq A_4 \\ x_1 &\leq A_5. \end{aligned} \tag{3.73}$$

Step 2. The system is reduced to the form $f(A_1, A_2, A_3, A_4, A_5, x_1, x_2) = 0$, where f is defined by

$$\begin{aligned} f(A_1, A_2, A_3, A_4, A_5, x_1, x_2) = {} & A_1'x_1'x_2 + A_1'x_1x_2' + \\ & A_2'x_1x_2 + A_2'x_1'x_2' + \\ & A_3'x_1'x_2 + \\ & A_4'x_1x_2' + \\ & A_5'x_1. \end{aligned} \tag{3.74}$$

Step 3. Using goal-directed elimination to eliminate the variables x_1 and x_2 produces the consequent $g(A_1, A_2, A_3, A_4, A_5) = 0$, where g is defined by

$$g(A_1, A_2, A_3, A_4, A_5) = A_1'A_2' + A_2'A_3'A_4' + A_2'A_3'A_5'. \tag{3.75}$$

The function $g(A_1, A_2, A_3, A_4, A_5)$ is in Blake canonical form.

Step 4. The terms in the right-hand side of (3.75) are A-consequent terms of $f(A, X) = 0$ which denote the existence of the minimal normal subsets $\{f_1, f_2\}$, $\{f_2, f_3, f_4\}$, and $\{f_2, f_3, f_5\}$. Thus, the relations

$$\begin{aligned} f_1 + f_2 &= 1 \\ f_2 + f_3 + f_4 &= 1 \\ f_2 + f_3 + f_5 &= 1 \end{aligned} \tag{3.76}$$

are identities.

An application of Procedure 3.2 is to determine the sum-to-one subsets among a set of functions each of which is a single term. A specialization of the sum-to-one subsets concept is to determine subsets of a set of terms which cover a given term. The sum-to-one theorem (2.171) states that the equivalence

$$t \leq f \Leftrightarrow f/t = 1 \tag{3.77}$$

is valid. Given a set $T = \{t_1, t_2, \ldots, t_k\}$ of terms and a term t, we would like to determine all minimal subsets of T which cover t. The function f in (3.77) is formed by summing the terms in T, i.e.,

$$f = t_1 + t_2 + \cdots + t_k. \tag{3.78}$$

Each term of $t_1 + t_2 + \cdots + t_k$ is divided by the term t to form the function $t_1/t + t_2/t + \cdots + t_k/t$. If t is included in $t_1 + t_2 + \cdots + t_k$, then $t_1/t + t_2/t + \cdots + t_k/t$ is a tautology. Moreover, if a subset $\{t_1/t, t_2/t, \ldots, t_s/t\}$ is normal, then the subset $\{t_1, t_2, \ldots, t_s\}$ covers t. Hence, forming the sum-to-one subsets of $\{t_1/t, t_2/t, \ldots, t_k/t\}$ yields the minimal subsets of $\{t_1, t_2, \ldots, t_k\}$ which cover t. Example 3.5 demonstrates this process.

Example 3.5: Suppose a set $T = \{t_1, t_2, t_3, t_4, t_5\}$ of terms is defined by

$$\begin{aligned} t_1 &= ac \\ t_2 &= ab' \\ t_3 &= bc \\ t_4 &= a \\ t_5 &= ac'. \end{aligned} \tag{3.79}$$

We would like to determine the minimal subsets of T which cover the term $t = ac$. To do so, we divide each member of T by t:

$$\begin{aligned} t_1/t &= 1 \\ t_2/t &= b' \\ t_3/t &= b \\ t_4/t &= 1 \\ t_5/t &= 0. \end{aligned} \tag{3.80}$$

Forming the sum-to-one subsets of $\{t_1/t, t_2/t, \ldots, t_5/t\}$ using Procedure 3.2 yields the set of subsets

$$\{\{t_1/t\}, \{t_4/t\}, \{t_2/t, t_3/t\}\}. \tag{3.81}$$

Hence, the term $t = ac$ is covered minimally by the subsets $\{t_1\}$, $\{t_4\}$, and $\{t_2, t_3\}$, i.e.,

$$\begin{aligned} ac &\leq ac \\ ac &\leq a \\ ac &\leq ab' + bc, \end{aligned} \tag{3.82}$$

respectively.

Evanescent Subsets

Similar to normal subsets, our concern may be to know the evanescent subsets, i.e., subsets for which the product is the zero-function, of a set $F = \{f_1, f_2, \ldots, f_k\}$ of functions. The label-and-eliminate procedure may be used to determine these subsets by forming all prime A-consequent terms $p(A)$ and selecting only those terms in which all of the literals are not complemented. Similar to the method presented for identifying normal subsets, a more efficient plan is to devise a procedure which produces only prime A-consequent terms which consist of uncomplemented A-arguments. A technique for recognizing evanescent subsets which parallels the methodology for identifying normal subsets is presented in this section.

The Partial Labeling-and-Reduction Process. A method for identifying evanescent subsets among a set F of functions is to develop a system reducible to the form $f(A, X) = 0$ from which A-consequents of the form

$$a_1 a_2 \cdots a_m = 0 \tag{3.83}$$

are deduced. Each A-consequent term $a_1 a_2 \cdots a_m$ represents an evanescent subset of F. To make the process of generating A-consequent terms efficient, we desire to enforce the requirement that the A-consequent terms of $f(A, X) = 0$ consist only of uncomplemented A-variables. To enforce this requirement, we make the supposition that the initial formula $F(A, X)$ which represents $f(A, X)$ should not contain terms which include complemented literals of variables in the set $A = \{A_1, A_2, \ldots, A_k\}$ of labels. We devise a system of inequalities which is then reduced to an equation $f(A, X) = 0$. For a set F of functions, we form the system

$$\begin{aligned} A_1 &\leq f_1(X) \\ A_2 &\leq f_2(X) \\ &\vdots \\ A_k &\leq f_k(X). \end{aligned} \tag{3.84}$$

The equivalent system

$$\begin{aligned} A_1 \cdot f_1'(X) &= 0 \\ A_2 \cdot f_2'(X) &= 0 \\ &\vdots \\ A_k \cdot f_k'(X) &= 0 \end{aligned} \tag{3.85}$$

is equivalent in turn to the equation $f(A, X) = 0$, for which

$$f(A, X) = \sum_{i=1}^{k} (A_i \cdot f_i'(X)). \tag{3.86}$$

Similar to the proofs given in the section on normal subsets, we present proofs which demonstrate that the terms of the A-consequents of $f(A, X) = 0$ represent evanescent subsets. Hence, the system (3.84) may be used as the basis for economically determining those subsets of a given set F of functions which are evanescent. Although applied to a system of a different form than when forming normal subsets, we again call this process the *partial labeling-and-reduction* of a set of functions.

Theorem 3.3 demonstrates that a subset of the set F of functions forms an evanescent set if and only if the A-consequent term which denotes the evanescent set is an implicant of $f(A, X)$. Our concern here is to show the existence of an implicant if and only if the corresponding subset is evanescent. If an implicant exists which consists entirely of uncomplemented literals from the set A of labels, then a prime implicant of $f(A, X)$ consisting of a subset of the literals must exist.

Terms in the Blake canonical form of $f(A, X)$ which consist only of A-arguments represent minimal evanescent subsets.

Theorem 3.3: *Given a set $F = \{f_1, f_2, \ldots, f_k\}$ of functions and a system*

$$\begin{aligned} A_1 &\leq f_1(X) \\ A_2 &\leq f_2(X) \\ &\vdots \\ A_k &\leq f_k(X). \end{aligned} \tag{3.87}$$

reduced to the 0-normal form $f(A, X) = 0$, a subset $\{f_1, f_2, \ldots, f_s\}$ of the set F is an evanescent subset if and only if $A_1 A_2 \cdots A_s$ is an implicant of $f(A, X)$.

Proof. We first show that if a subset $\{f_1, f_2, \ldots, f_s\}$ is evanescent, then $A_1 A_2 \cdots A_s \leq f(A, X)$. If $\{f_1, f_2, \ldots, f_s\}$ is evanescent, then

$$f_1(X) f_2(X) \cdots f_s(X) = 0. \tag{3.88}$$

Hence, $f_1(X) f_2(X) \cdots f_s(X)$ is equal to the sero-function. Expressing $f(A, X)$ in the form

$$A_1 f_1'(X) + A_2 f_2'(X) + \cdots + A_k f_k'(X) \qquad (s \leq k), \tag{3.89}$$

we can add the sero-function to (3.89) and the result will still represent $f(A, X)$. Hence, $f(A, X)$ may be represented by the form

$$\begin{aligned} &f_1(X) f_2(X) \cdots f_s(X) + A_1 f_1'(X) + A_2 f_2'(X) \\ &+ \cdots + A_s f_s'(X) + A_{s+1} f_{s+1}'(X) + \cdots + A_k f_k'(X) \qquad (s \leq k). \end{aligned} \tag{3.90}$$

Since $f_1(X)f_2(X)\cdots f_s(X)$ and $A_1f_1'(X)$ form the consensus term $A_1f_2(X)\cdots f_s(X)$, we may represent (3.90) equivalently by

$$\begin{aligned} &A_1f_2(X)\cdots f_s(X) + f_1(X)f_2(X)\cdots f_s(X) + A_1f_1'(X) \\ &+ A_2f_2'(X) + \cdots + A_sf_s'(X) + A_{s+1}f_{s+1}'(X) + \cdots + A_kf_k'(X) \qquad (s \le k). \end{aligned} \tag{3.91}$$

Similarly, $A_1f_2(X)\cdots f_s(X)$ and $A_2f_2'(X)$ form a consensus term $A_1A_2f_3(X)\cdots f_s(X)$. It follows that (3.91) is equivalent to

$$\begin{aligned} &A_1A_2f_3(X)\cdots f_s(X) + A_1f_2(X)\cdots f_s(X) + f_1(X)f_2(X)\cdots f_s(X) \\ &+ A_1f_1'(X) + A_2f_2'(X) + \cdots + A_sf_s'(X) + A_{s+1}f_{s+1}'(X) + \cdots + A_kf_k'(X) \qquad (s \le k). \end{aligned} \tag{3.92}$$

We continue forming consensus terms in the same fashion until $A_1A_2\cdots A_{s-1}f_s(X)$ and $A_sf_s'(X)$ form the consensus term $A_1A_2\cdots A_s$. It follows that (3.92) is expressed equivalently by

$$\begin{aligned} &A_1A_2\cdots A_s + A_1A_2\cdots A_{s-1}f_s'(X) + \cdots + A_1f_2'(X)\cdots f_s'(X) \\ &+ f_1(X)f_2(X)\cdots f_s(X) + A_1f_1'(X) + A_2f_2'(X) \\ &+ \cdots + A_sf_s'(X) + A_{s+1}f_{s+1}'(X) + \cdots + A_kf_k'(X) \qquad (s \le k). \end{aligned} \tag{3.93}$$

We thus conclude that the term $A_1A_2\cdots A_s$ appears in a sum-of-products formula which represents $f(A,X)$. Since any term which appears in an SOP formula which represents a function f is an implicant of f, $A_1A_2\cdots A_s$ is an implicant of $f(A,X)$.

Assume on the other hand that $A_1A_2\cdots A_s$ is an implicant of $f(A,X)$. The statement

$$A_1A_2\cdots A_s \le \sum_{i=1}^{k}(A_i \cdot f_i'(X)) \qquad (s \le k). \tag{3.94}$$

is then valid for all substitutions for X and for $(A_1, A_2, \ldots, A_k)$. We deduce the equivalent statement

$$\sum_{i=1}^{s}(A_i') + \sum_{i=1}^{k}(A_i \cdot f_i'(X)) = 1 \qquad (s \leq k), \tag{3.95}$$

from which follows the equation

$$\sum_{i=1}^{s}(A_i' + f_i'(X)) + \sum_{i=s+1}^{k}(A_i \cdot f_i'(X)) = 1, \qquad (s \leq k). \tag{3.96}$$

Equation (3.96) holds for all substitutions for X and for $(A_1, A_2, \ldots, A_k)$. Thus, it holds if

$$A_1 = A_2 = \cdots = A_s = 1 \tag{3.97}$$

and

$$A_{s+1} = A_{s+2} = \cdots = A_k = 0; \tag{3.98}$$

whence, the statement

$$\sum_{i=1}^{s}(f_i'(X)) = 1 \tag{3.99}$$

is an identity. It then follows that

$$\prod_{i=1}^{s}(f_i(X)) = 0 \tag{3.100}$$

is valid. Hence, $\{f_1, f_2, \ldots, f_s\}$ is an evanescent subset of F. This completes the proof. □

We now show that the prime A-consequent terms $p(A)$ deducible from $f(A, X) = 0$, where f is defined by (3.86), contain only uncomplemented literals of the A-variables. Hence, the terms $p(A)$ exist only in the form which denotes evanescent subsets of a set F of functions. This statement is established in Theorem 3.4.

Theorem 3.4: *Given a system*

$$\begin{aligned} A_1 &\leq f_1(X) \\ A_2 &\leq f_2(X) \\ &\vdots \\ A_k &\leq f_k(X). \end{aligned} \tag{3.101}$$

reducible to the form $f(A, X) = 0$, *the A-consequent terms of the prime A-consequents* $p(A) = 0$ *of* $f(A, X) = 0$ *consist only of uncomplemented literals of the set* $A = \{A_1, A_2, \ldots, A_k\}$ *of variables.*

Proof. System (3.101) may be expressed equivalently as

$$\begin{aligned} A_1 \cdot f_1'(X) &= 0 \\ A_2 \cdot f_2'(X) &= 0 \\ &\vdots \\ A_k \cdot f_k'(X) &= 0. \end{aligned} \tag{3.102}$$

System (3.102) is reduced to an equation of the form $f(A, X) = 0$, where

$$f(A, X) = A_1 f_1'(X) + A_2 f_2'(X) + \cdots + A_k f_k'(X). \tag{3.103}$$

We construct a formula to represent $f(A, X)$ as follows:

1. For each function f_i, complement the function f_i. Prefix the associated label A_i to each term of the formula which represents f_i'. The resulting formula represents the function $A_i \cdot f_i'$.
2. Append together all formulas formed in Step 1. The resulting formula represents $A_1 f_1'(X) + A_2 f_2'(X) + \cdots + A_k f_k'(X)$. Let us call this formula $F(A, X)$.

Because the formulas which represent each function f_i' do not involve any variable of the set A, and the formula $F(A, X)$ is formed so that only the uncomplemented forms of the variables $A_1, A_2, \ldots, A_k$ appear, the set A of variables are unate in $F(A, X)$. By Lemma 3.1, the variables in

A are unate in $BCF(f(A,X))$. Consequently, if any prime A-consequents $p(A)=0$ of $f(A,X)=0$ exist, then the corresponding A-consequent terms $p(A)$ only consist of uncomplemented literals of the set A of variables. This completes the proof. □

The benefit of using the partial label-and-reduce process to generate an equation of the form $f(A,X)=0$ is a reduction in the time and space complexity of generating A-consequent terms of the desired form. When reducing system (3.84), the associated function $A_i'f_i$ is not included in $f(A,X)$ for each function f_i because an inclusion is used rather than an equality to associate a label A_i with a function f_i. Therefore, the computations required to form $A_i'f_i$ are not performed; nor do we have to store the formula which represents $A_i'f_i$. In the next section the formation of the consequent $g(A)=0$ from $f(A,X)=0$ is discussed. Prime implicants of $g(A)$ are used to form A-consequent terms which represent evanescent subsets.

Elimination of X-Arguments and Formation of A-Consequents. Due to the manner in which we form a system of inequalities, i.e.,

$$\begin{aligned} A_1 &\leq f_1(X) \\ A_2 &\leq f_2(X) \\ &\vdots \\ A_k &\leq f_k(X), \end{aligned} \tag{3.104}$$

after reduction to the $f(A,X)=0$ form, the A-consequent terms of the prime A-consequents $p(A)=0$ of $f(A,X)=0$ consist only of uncomplemented literals of the set A of variables. This was proven in Theorem 3.4. We can derive the A-consequent terms of each of the prime A-consequents of $f(A,X)=0$ by forming the conjunctive eliminant of $f(A,X)$ with respect to X and placing the result in Blake canonical form, i.e., $BCF(ECON(f(A,X),X))$. Terms of $BCF(ECON(f(A,X),X))$ compose the complete set of terms $p(A)$ of the prime A-consequents $p(A)=0$ of $f(A,X)=0$.

Eliminating the X-arguments from $f(A,X)=0$ yields the consequent $g(A)=0$, where

$$g(A)=ECON(f(A,X),X). \tag{3.105}$$

The formula which represents $f(A,X)$ consists of unate A-arguments and generally binate X-arguments. The formula which represents $g(A)$, if the means for forming the conjunctive eliminant of $f(A,X)$ with respect to X preserves the unateness of the A-arguments, will be unate in the A-arguments. Hence, we know the form of the functions $f(A,X)$ and $g(A)$ prior to elimination the X-arguments from $f(A,X)=0$. Thus, we apply the goal-directed elimination technique discussed in the section on deducing normal subsets. In goal-directed elimination, we form the function $g(A)$ by applying the quick method[2] for forming the conjuctive eliminant of $f(A,X)$ with respect to X. The formula representing $g(A)$ which results from goal-directed elimination is its Blake canonical form. Each term of $BCF(g(A))$ forms a prime A-consequent term $p(A)$ of $f(A,X)=0$. Prime A-consequent terms $p(A)$ represent minimal evanescent subsets of the set F of functions. Each term,

$$A_1A_2\cdots A_s, \tag{3.106}$$

of $BCF(g(A))$ denotes the existence of the evanescent subset

$$\{f_1,f_2,\ldots,f_s\}. \tag{3.107}$$

Determination of Minimal Evanescent Subsets. Using the the partial label-and-reduce and goal-directed elimination processes, we have demonstrated a means for determining all minimal evanescent subsets among a set F of functions. Procedure 3.3 is a method for determining all

[2]Procedure 3.1

evanescent subsets. An example which demonstrates the use of Procedure 3.3 to determine the evanescent subsets among a set of functions is given by Example 3.6.

Procedure 3.3 (Minimal Evanescent Subsets): Given a set $F = \{f_1, f_2, \ldots, f_k\}$ of Boolean functions, we determine all of the minimal evanescent subsets among the set of functions in the following manner:

Step 1. For each function $f_i \in F$:

1. Generate an associated label A_i.
2. Complement each function f_i.
3. Prefix each term of the formula which represents f_i' with the literal A_i.

The resulting formula represents $A_i \cdot f_i'$.

Step 2. Append together each of the formulas developed in Step 1. The resulting formula represents $f(A, X)$.

Step 3. Using the quick method for forming the conjunctive eliminant (Procedure 3.1), form $ECON(f(A, X), X)$. The resulting function is equal to $g(A)$. The formula which represents $g(A)$ is $BCF(g(A))$.

Step 4. Each term of $BCF(g(A))$ represents a minimal evanescent subset. For each term of $BCF(g(A))$, determine the labels contained in the term and replace the labels by the functions with which they are associated. Return each subset of functions.

Example 3.6: Suppose we would like to determine the evanescent subsets among the functions of the set $\{f_1, f_2, f_3, f_4, f_5\}$ defined by

$$\begin{aligned} f_1 &= x_1'x_2 + x_1x_2' \\ f_2 &= x_1x_2 + x_1'x_2' \\ f_3 &= x_1'x_2 \\ f_4 &= x_1x_2' \\ f_5 &= x_1. \end{aligned} \tag{3.108}$$

Step 1. We create a system of the form (3.84) by associating labels A_1, A_2, A_3, A_4, and A_5 with the respective functions, i.e.,

$$\begin{aligned} A_1 &\leq x_1'x_2 + x_1x_2' \\ A_2 &\leq x_1x_2 + x_1'x_2' \\ A_3 &\leq x_1'x_2 \\ A_4 &\leq x_1x_2' \\ A_5 &\leq x_1. \end{aligned} \tag{3.109}$$

Step 2. The system is reduced to the form $f(A, X) = 0$, for which f is defined by the equation

$$\begin{aligned} f(A_1, A_2, A_3, A_4, A_5, x_1, x_2) = {} & A_1x_1'x_2' + A_1x_1x_2 + \\ & A_2x_1'x_2 + A_2x_1x_2' + \\ & A_3x_1 + A_3x_2' + \\ & A_4x_1' + A_4x_2 + \\ & A_5x_1'. \end{aligned} \tag{3.110}$$

Step 3. Eliminating the variables x_1 and x_2 from $f(A, X)$ produces the consequent $g(A) = 0$, for which

$$g(A_1, A_2, A_3, A_4, A_5) = A_1A_2 + A_2A_3 + A_3A_4 + A_2A_4 + A_3A_5. \tag{3.111}$$

The function $g(A)$ is in Blake canonical form.

Step 4. The terms in $BCF(g(A))$ are the A-consequent terms of $f(A, X) = 0$ which denote the existence of the minimal evanescent subsets $\{f_1, f_2\}$, $\{f_2, f_3\}$, $\{f_3, f_4\}$, $\{f_2, f_4\}$, and $\{f_3, f_5\}$. Thus, the relations

$$\begin{aligned} f_1f_2 &= 0 \\ f_2f_3 &= 0 \\ f_3f_4 &= 0 \\ f_2f_4 &= 0 \\ f_3f_5 &= 0 \end{aligned} \tag{3.112}$$

are identities.

Implication Relations

The third relationship among a set $F = \{f_1, f_2, \ldots, f_k\}$ of functions with which we are concerned is that of implication. An implication relation is one of the form

$$f_0 \leq f_1 + \cdots + f_s, \tag{3.113}$$

in which $\{f_1, f_2, \ldots, f_s\}$ is a subset of F. When using the label-and-eliminate procedure to determine all of the relationships among a set of functions, the A-consequents which denote implication relations take the form

$$a_j b_1' \cdots b_s' = 0. \tag{3.114}$$

As with normal and evanescent subsets, a more efficient approach is to formulate the problem so that only those A-consequents which have the desired form are developed. In this section, we introduce an efficient technique for producing only those A-consequents which have the form (3.114).

The General Method. To determine coverage of a function f_0 by a subset of F, a system reducible to the form $f(A, X) = 0$ must be devised from which A-consequents of the form $A_0 A_1' \cdots A_s' = 0$ are deduced. Such A-consequents may be expressed by the equivalent statement

$$A_0 \leq A_1 + \cdots + A_s \tag{3.115}$$

where $A_0, A_1, \ldots, A_s$ are members of the set $A = \{A_0, A_1, \ldots, A_k\}$ of labels. We replace the labels with their associated functions to deduce the coverage of f_0 by a subset of F.

We would like to enforce the condition that all A-consequent terms consist of one uncomplemented literal, i.e., the label A_0 associated with the function f_0, and one or more complemented literals which are labels representing a subset of F which covers f_0. To produce A-consequents of

this form, we make the supposition that the initial formula $F(A, X)$ which represents the function $f(A, X)$ should not contain terms which include the complemented literal A_0' or other A-literals in uncomplemented form. Therefore, a system of inequalities is formulated which is reduced to an equation $f(A, X) = 0$. For a function to be covered, f_0, and a set F of functions which may cover f_0, we form the system

$$\begin{aligned} A_0 &\leq f_0(X) \\ f_1(X) &\leq A_1 \\ f_2(X) &\leq A_2 \\ &\vdots \\ f_k(X) &\leq A_k. \end{aligned} \tag{3.116}$$

Applying the definition of the inclusion relation, (3.116) is represented by the equivalent system

$$\begin{aligned} f_0(X)' \cdot A_0 &= 0 \\ f_1(X) \cdot A_1' &= 0 \\ f_2(X) \cdot A_2' &= 0 \\ &\vdots \\ f_k(X) \cdot A_k' &= 0. \end{aligned} \tag{3.117}$$

This system is equivalent in turn to the equation $f(A, X) = 0$, for which

$$f(A, X) = A_0 \cdot f_0(X)' + \sum_{i=1}^{k} (f_i(X) \cdot A_i'). \tag{3.118}$$

We now present proofs which demonstrate that the terms of the A-consequents of $f(A, X) = 0$ represent coverage of the function f_0 by subsets of F. Hence, system (3.116) may be used to economically determine the subsets of a set of functions which cover a given function. As when determining normal and evanescent subsets, we call the process of proceeding from a system of

equations of the form (3.116) to an equivalent equation of the form $f(A, X) = 0$ the *partial labeling-and-reduction* of a set of functions.

Theorem 3.5 establishes that a function f_0 is covered by a subset of the set F of functions if and only if the A-consequent term which denotes the respective implication relation is an implicant of $f(A, X)$.

Theorem 3.5: *Given a function f_0, a set $F = \{f_1, f_2, \ldots, f_k\}$ of functions, and a system*

$$\begin{aligned} A_0 &\leq f_0(X) \\ f_1(X) &\leq A_1 \\ f_2(X) &\leq A_2 \\ &\vdots \\ f_k(X) &\leq A_k. \end{aligned} \tag{3.119}$$

reduced to the 0-normal form $f(A, X) = 0$, the function f_0 is included in a subset $\{f_1, f_2, \ldots, f_s\}$ of the set F if and only if $A_0 A_1' A_2' \cdots A_s'$ is an implicant of $f(A, X)$.

Proof. We first show that if a function f_0 is included in a subset $\{f_1, f_2, \ldots, f_s\}$, then $A_0 A_1' A_2' \cdots A_s'$ is an implicant of $f(A, X)$. Since f_0 is included in the sum of the functions in $\{f_1, f_2, \ldots, f_s\}$, it follows that

$$f_0(X) f_1'(X) f_2'(X) \cdots f_s'(X) = 0 \tag{3.120}$$

is valid. Hence, $f_0(X) f_1'(X) f_2'(X) \cdots f_s'(X)$ is equal to the zero-function. Expressing $f(A, X)$ in the form

$$A_0 f_0'(X) + f_1(X) A_1' + f_2(X) A_2' + \cdots + f_s(X) A_s' + f_{s+1}(X) A_{s+1}' + \cdots + f_k(X) A_k' \quad (s \leq k), \tag{3.121}$$

we can add $f_0(X)f_1'(X)f_2'(X)\cdots f_s'(X)$ to develop an equivalent representation for $f(A,X)$:

$$\begin{aligned} &f_0(X)f_1'(X)f_2'(X)\cdots f_s'(X) + A_0f_0'(X) + f_1(X)A_1' + f_2(X)A_2' \\ &+\cdots + f_s(X)A_s' + f_{s+1}(X)A_{s+1}' + \cdots + f_k(X)A_k' \qquad (s \le k). \end{aligned} \tag{3.122}$$

Using the same methodology as the proofs for Theorems 3.1 and 3.3, we apply consensus to form the equivalent representation for (3.122):

$$\begin{aligned} &A_0A_1'A_2'\cdots A_s' + f_s'(X)A_0A_1'A_2'\cdots A_{s-1}' + \cdots + f_1'(X)\cdots f_s'(X)A_0 \\ &+ f_0(X)f_1'(X)\cdots f_s'(X) + A_0f_0'(X) + f_1(X)A_1' \\ &+\cdots + f_s(X)A_s' + f_{s+1}(X)A_{s+1}' + \cdots + f_k(X)A_k' \qquad (s \le k). \end{aligned} \tag{3.123}$$

We thus conclude that the term $A_0A_1'\cdots A_s'$ appears in an SOP formula which represents $f(A,X)$. It thus follows that $A_0A_1'\cdots A_s'$ is an implicant of $f(A,X)$.

Assume on the other hand that $A_0A_1'A_2'\cdots A_s'$ is an implicant of $f(A,X)$. Then the statement

$$A_0A_1'A_2'\cdots A_s' \le A_0 \cdot f_0'(X) + \sum_{i=1}^{k}(f_i(X)\cdot A_i') \qquad (s \le k) \tag{3.124}$$

is valid for all substitutions for X and for $(A_1, A_2, \ldots, A_k)$. From (3.124) we deduce the equivalent statement

$$A_0' + \sum_{i=1}^{s}(A_i) + A_0 \cdot f_0'(X) + \sum_{i=1}^{k}(f_i(X)\cdot A_i') = 1 \qquad (s \le k), \tag{3.125}$$

which in turn is equivalent to

$$A_0' + \sum_{i=1}^{s}(A_i + f_i(X)) + f_0'(X) + \sum_{i=s+1}^{k}(f_i(X)\cdot A_i') = 1 \qquad (s \le k). \tag{3.126}$$

Since (3.126) is an identity, i.e., it is true for all substitutions for X and for $(A_1, A_2, \ldots, A_k)$, it holds if

$$A_1 = A_2 = \cdots = A_s = 0 \tag{3.127}$$

and

$$A_0 = A_{s+1} = A_{s+2} = \cdots = A_k = 1. \tag{3.128}$$

Thus,

$$f_0'(X) + \sum_{i=1}^{s}(f_i(X)) = 1 \tag{3.129}$$

is an identity. Thus, statement (3.124) implies equation (3.129). In other words, f_0 is included in the subset $\{f_1, f_2, \ldots, f_s\}$ of F. This completes the proof. □

Theorem 3.6 demonstrates that the prime A-consequent terms deducible from (3.116) contain only the uncomplemented literal A_0 and complemented literals from the set $\{A_1, A_2, \ldots, A_k\}$.

Theorem 3.6: *Given a system*

$$\begin{aligned} A_0 &\leq f_0(X) \\ f_1(X) &\leq A_1 \\ f_2(X) &\leq A_2 \\ &\vdots \\ f_k(X) &\leq A_k. \end{aligned} \tag{3.130}$$

reducible to the form $f(A, X) = 0$, the A-consequent terms of the prime A-consequents $p(A) = 0$ of $f(A, X) = 0$ consist only of the uncomplemented literal A_0 or complemented literals from the set $A = \{A_1, A_2, \ldots, A_k\}$ of variables.

Proof. System (3.130) may be expressed equivalently as

$$\begin{aligned} A_0 \cdot f_0'(X) &= 0 \\ f_1(X) \cdot A_1' &= 0 \\ f_2(X) \cdot A_2' &= 0 \\ &\vdots \\ f_k(X) \cdot A_k' &= 0. \end{aligned} \tag{3.131}$$

By (2.40), system (3.131) is reduced to an equation of the form $f(A, X) = 0$, where

$$f(A, X) = A_0 f_0'(X) + f_1(X)A_1' + \cdots + f_k(X)A_k'. \tag{3.132}$$

A formula to represent $A_0 f_0'(X) + f_1(X)A_1' + \cdots + f_k(X)A_k'$ is constructed in the following manner:

1. Complement function f_0. Prefix the label A_0 to each term of the formula which represents f_0'. The resulting formula represents $A_0 \cdot f_0'$.
2. For each function $f_i \in \{f_1, \ldots, f_k\}$, prefix the complement of its associated label A_i' to each term of the formula which represents f_i. The resulting formula represents the function $f_i \cdot A_i'$.
3. Append together the formulas formed in Steps 1 and 2. The resulting formula represents $A_0 f_0'(X) + f_1(X)A_1' + \cdots + f_k(X)A_k'$. Let us call this formula $F(A, X)$.

Because the formulas which represent each of the functions $f_0(X), f_1(X), \ldots, f_k(X)$ consist only of X-variabies, and the formula $F(A, X)$ is formed so that only the uncomplemented form of A_0 and the complemented forms of variables in A appear in it, A_0 and the variables in A are unate in $F(A, X)$. By Lemma 3.1, A_0 and the variables in A are unate in $BCF(f(A, X))$. It follows that if any prime A-consequents $p(A) = 0$ of $f(A, X) = 0$ exist, then the corresponding A-consequent terms $p(A)$ consist only of the uncomplemented form of A_0 and the complemented forms of variables in A. This completes the proof. □

Using the partial label-and-reduce process to form the equation $f(A, X) = 0$ rather than the full label-and-reduce process decreases the time and space complexity of generating A-consequents

of the desired form. When reducing system (3.116), the associated function $A_0 f_0'(X)$ for function $f_0(X)$ is not included in $f(A, X)$. Likewise, for each function $f_i(X) \in F$, the associated function $f_i(X)A_i'$ is not included in the function $f(A, X)$. Functions $A_0 f_0'(X)$ and $f_i(X)A_i'$ are not included in $f(A, X)$ because inclusions are used rather than equalities to associate labels with functions. The computations required to form $A_0 f_0'(X)$ or $f_i(X)A_i'$ are not performed; additionally, the respective formulas do not have to be stored.

Elimination of X-Arguments and Formation of A-Consequents. Because of the form of system

$$\begin{aligned} A_0 &\leq f_0(X) \\ f_1(X) &\leq A_1 \\ f_2(X) &\leq A_2 \\ &\vdots \\ f_k(X) &\leq A_k, \end{aligned} \tag{3.133}$$

after reduction to 0-normal form, $f(A, X) = 0$, the A-consequent terms of the prime A-consequents of $f(A, X) = 0$ may contain only the uncomplemented literal A_0 or complemented literals in the set A of labels. As in the methods for deducing normal and evanescent subsets, the prime A-consequent terms are deduced using the goal-directed elimination technique for eliminating the X-variables from $f(A, X) = 0$. As a result of elimination, we deduce the consequent $g(A) = 0$, for which

$$g(A) = ECON(f(A, X), X). \tag{3.134}$$

Because $g(A)$ is formed using the quick method[3] for forming the conjunctive eliminant of $f(A, X)$ with respect to X, the formula produced by the elimination process to represent $g(A)$ is its Blake

[3]Procedure 3.1

canonical form. Each term of $BCF(g(A))$ corresponds to a prime A-consequent $p(A) = 0$ of $f(A, X) = 0$.

The prime A-consequent terms $p(A)$ of $f(A, X)$ do not necessarily represent irredundant implication relations. In some cases—depending on the functions in F—prime A-consequent terms represent normal subsets. Prime A-consequent terms deduced from $f(A, X) = 0$ may take one of two forms:

- $A_0 A_1' \cdots A_s'$, or
- $A_1' \cdots A_s'$.

If a prime A-consequent term is of the form $A_0 A_1' \cdots A_s'$, then the term represents an irredundant implication relation. The literals of the prime implicant correspond to an irredundant subset of F which covers the function f_0. On the other hand, if a prime A-consequent term is of the form $A_1' \cdots A_s'$, then the term represents a normal subset of F. Any function is included in the sum of functions which form a normal subset. When a prime A-consequent term of the form $A_1' A_2' \cdots A_s'$ is deduced, Boole's Expansion Theorem may be applied to form

$$A_0' A_1' A_2' \cdots A_s' + A_0 A_1' A_2' \cdots A_s'. \tag{3.135}$$

It follows that the equation

$$A_0 A_1' A_2' \cdots A_s' = 0. \tag{3.136}$$

is valid and a consequent of $f(A, X) = 0$. Consequent (3.136), although not a prime A-consequent of $f(A, X) = 0$, represents an implication relation denoting the coverage of function f_0 by a subset of F. The A-consequent term $A_0 A_1' A_2' \cdots A_s'$ represents an irredundant implication relation if it is not absorbed by a prime A-consequent term which represents an irredundant implication relation.

To generate all A-consequent terms of $f(A, X) = 0$ which represent irredundant implication relations, the following steps are taken after the function $g(A)$ is formed using goal-directed elimination:

1. The terms of $BCF(g(A))$ are separated into two groups:
   (a) terms which contain the literal A_0, and
   (b) terms which do not contain the literal A_0.
2. The literal A_0 is prefixed to the terms which do not contain the literal A_0.
3. A formula, $\widehat{G}$, is formed composed of the terms resulting from Step 2 and the terms of $BCF(g(A))$ which contain the literal A_0.
4. The equivalent absorptive formula, $ABS(\widehat{G})$, is formed for $\widehat{G}$.

Terms of $\widehat{G}$ are the A-consequent terms of $f(A, X) = 0$ which represent the complete set of irredundant implication relations denoting the coverage of the function f_0 by subsets of F. We shortly will present an example which illustrates this process.

Determination of Irredundant Implication Relations. Procedure 3.4 outlines a method for deducing all irredundant implication relations in which a function f_0 is included in a subset of the set F of functions. Example 3.7 demonstrates the use of Procedure 3.4 to determine implication relations in which a function is included in a subset of a set of functions.

Procedure 3.4 (Irredundant Implication Relations): Given a Boolean function f_0 and a set $F = \{f_1, f_2, \ldots, f_k\}$ of Boolean functions, we determine all irredundant implication relations in which the function f_0 is included in a subset of the set F of functions as follows:

Step 1. For function f_0:

1. Generate an associated label A_0.
2. Complement function f_0.
3. Prefix each term of the formula which represents f_0' with the literal A_0.

Step 2. For each function $f_i \in F$:

1. Generate an associated label A_i.
2. Prefix each term of the formula which represents f_i with the complemented literal A_i'.

The resulting formula represents $A_i' \cdot f_i$.

Step 3. Append together each of the formulas developed in Steps 1 and 2. The resulting formula represents $f(A, X)$.

Step 4. Using the quick method for forming the conjunctive eliminant (Procedure 3.1), form $ECON(f(A, X), X)$. The resulting function equals $g(A)$; the formula which represents $g(A)$ is $BCF(g(A))$.

Step 5. Each term of $BCF(g(A))$ consisting only of complemented literals represents a minimal normal subset; append the literal A_0 to each of these terms. (Do nothing to the remaining terms.) Call the resulting formula $\widehat{G}$.

Step 6. Form the equivalent absorptive, $ABS(\widehat{G})$, for $\widehat{G}$. Each term of $ABS(\widehat{G})$ represents an irredundant implication relation.

Step 7. For each term of $ABS(\widehat{G})$, remove the literal A_0 and form a set which consists of the functions associated with the remaining labels. Return each subset of functions.

Example 3.7: Suppose we would like to determine the irredundant implication relations which denote coverage of the function $f_0 = xyz + x'y'z'$ by subsets of the set $F = \{f_1, f_2, \ldots, f_6\}$ of functions defined by

$$\begin{aligned} f_1 &= z \\ f_2 &= y'z' + w \\ f_3 &= x'z' \\ f_4 &= x + z' \\ f_5 &= w \\ f_6 &= w'. \end{aligned} \tag{3.137}$$

Steps 1-2. A system of the form (3.116) is created by associating the labels $A_0, A_1, \ldots, A_6$ with the respective functions

$$\begin{aligned} A_0 &\leq xyz + x'y'z' \\ z &\leq A_1 \\ y'z' + w &\leq A_2 \\ x'z' &\leq A_3 \\ x + z' &\leq A_4 \\ w &\leq A_5 \\ w' &\leq A_6. \end{aligned} \tag{3.138}$$

Step 3. Reducing to the form $f(A, X) = 0$, we form the equation

$$\begin{aligned} f(A_1, A_2, A_3, A_4, A_5, A_6, w, x, y, z) \;=\; & A_0xy' + A_0x'y + A_0x'z + A_0xz' + A_0y'z + A_0yz' + \\ & A_1'z + \\ & A_2'y'z' + A_2'w + \\ & A_3'x'z' + \\ & A_4'x + A_4'z' + \\ & A_5'w + \\ & A_6'w'. \end{aligned} \tag{3.139}$$

Step 4. Using goal-directed elimination to eliminate the variables w, x, y and z results in the consequent $g(A_0, A_1, \ldots, A_6) = 0$, where

$$g(A_0, A_1, A_2, A_3, A_4, A_5, A_6) = A_0A_1'A_3' + A_0A_1'A_2' + A_0A_4' + A_2'A_6' + A_1'A_4' + A_5'A_6'. \tag{3.140}$$

The function g is in Blake canonical form.

Steps 5-7. The following A-consequents of $f(A, X) = 0$ are developed:

$$\begin{aligned} A_0A_1'A_3' &= 0 \\ A_0A_1'A_2' &= 0 \\ A_0A_4' &= 0 \\ A_2'A_6' &= 0 \\ A_1'A_4' &= 0 \\ A_5'A_6' &= 0. \end{aligned} \tag{3.141}$$

The A-consequents in (3.141) denote the existence of the irredundant implication relations:

$$\begin{aligned} f_0 &\leq f_1 + f_3 \\ f_0 &\leq f_1 + f_2 \\ f_0 &\leq f_4 \end{aligned} \tag{3.142}$$

and the normal subsets $\{f_2, f_6\}, \{f_1, f_4\}$, and $\{f_5, f_6\}$. The normal subsets indicate the validity of the statements

$$\begin{aligned} f_2 + f_6 &= 1 \\ f_1 + f_4 &= 1 \\ f_5 + f_6 &= 1. \end{aligned} \tag{3.143}$$

When the literal A_0 is appended to the last three A-consequent terms of (3.141), we deduce the A-consequents

$$A_0 A_2' A_6' = 0 \tag{3.144}$$
$$A_0 A_1' A_4' = 0 \tag{3.145}$$
$$A_0 A_5' A_6' = 0. \tag{3.146}$$

Equations (3.144) and (3.146) represent irredundant implication relations

$$\begin{aligned} f_0 &\leq f_2 + f_6 \\ f_0 &\leq f_5 + f_6. \end{aligned} \tag{3.147}$$

However, equation (3.145) represents the implication relation

$$f_0 \leq f_1 + f_4 \tag{3.148}$$

which is not irredundant since $f_0 \leq f_4$. The A-consequent term $A_0 A_1' A_4'$ is absorbed by $A_0 A_4'$.

A Modified Approach. When determining the inclusion of the function f_0 in subsets of the set $F = \{f_1, f_2, \ldots, f_k\}$ of functions, it is advantageous to make each formula F_i which represents each function $f_i \in F$ as simple as possible prior to the label-and-reduce and elimination processes. To make a formula simple, terms of the formula are deleted and literals are removed from the remaining terms where possible. Simplifying each formula prior to the label-and-reduce and elimination processes minimizes the total amount of work that must be performed during elimination. Memory usage also will decrease because the formulas to be manipulated are smaller. In this section we develop an approach for simplifying formulas representing each function f_i.

Suppose we have two functions f and g in which $f \leq g$. A question that may be posed is whether there exists a third function h such that

$$f \leq g \iff f \leq h \tag{3.149}$$

is valid. Additionally, if such an h exists, then what functions form a suitable h. We show in Theorem 3.7 that a range of such functions exists given that $f \leq g$ is true.

Theorem 3.7: *If $f \leq g$ is valid, then there exists an h defined by the interval*

$$f \cdot g \leq h \leq f' + g \tag{3.150}$$

such that we may equivalently state that

$$f \leq g \iff f \leq h. \tag{3.151}$$

Proof. We may state (3.151) equivalently as

$$fg' = 0 \iff fh' = 0. \tag{3.152}$$

Since $f \leq g$, the Extended Verification Theorem allows us to equivalently assert that

$$fg' = fh'. \tag{3.153}$$

Reducing (3.153) to an equivalent 1-normal form, we develop the equation

$$(fg') \oplus (fh') = 0, \tag{3.154}$$

which may be restated in turn by the equation

$$(fg')h + (fg)h' = 0. \tag{3.155}$$

Equation (3.155) is equivalently stated by the interval

$$fg \leq h \leq f' + g. \tag{3.156}$$

This completes the proof. □

The existence of a range of h-functions for which (3.149) is true allows us to select a function h to use in lieu of g as we please. We now show how to apply this principle to our current problem.

Our goal is to make each formula F_i which represents each function $f_i \in F$ simpler with respect to contained terms and literals prior to applying the label-and-reduce and elimination processes for determining the inclusion of the function f_0 in subsets of F. A way to accomplish this is to find a set $\tilde{F} = \{\tilde{f}_1, \tilde{f}_2, \ldots, \tilde{f}_k\}$ of functions to use in lieu of F in the label-and-reduce and

elimination processes, such that each formula $\tilde{F}_i$ which represents $\tilde{f}_i \in \tilde{F}$ is simpler with respect to contained terms and literals than the corresponding F_i which represents $f_i \in F$. Theorem 3.7 facilitates the development of such a set.

If f_0 is included in a subset of F, then the statement

$$f_0 \leq f_1 + f_2 + \cdots + f_k \tag{3.157}$$

must be valid. We would like to find a sum $\tilde{f}_1 + \tilde{f}_2 + \cdots + \tilde{f}_k$ to use in lieu of $f_1 + f_2 + \cdots + f_k$. If $f_0 \mapsto f$ and $f_1 + f_2 + \cdots + f_k \mapsto g$ in Theorem 3.7, then a suitable sum is given by the interval

$$f_0 \cdot (f_1 + f_2 + \cdots + f_k) \leq \tilde{f}_1 + \tilde{f}_2 + \cdots + \tilde{f}_k \leq f_0' + f_1 + f_2 + \cdots + f_k. \tag{3.158}$$

We are free to choose any sum which falls between the lower and upper bounds of (3.158). It follows that we may select each function $\tilde{f}_i$ which is a member of the interval

$$[(f_0 \cdot f_i), (f_0' + f_i)]. \tag{3.159}$$

In the next two sections, we will describe a strategy for selecting a reasonable $\tilde{f}_i$ from this interval.

Relative Absorption. One way to develop a function to use in lieu of f_i is to delete terms included in the function f_0' in the formula F_i which represents it. These terms of are no use in covering the function f_0 since they are implicants of f_0'. We may easily identify such terms in each F_i, because each contains a superset of the literals contained in at least one term in $BCF(f_0')$. Any implicant of a function in necessarily included in a prime implicant of the function. Using the $ABSREL$ operator[4] introduced in Chapter 2, we may remove terms from each F_i included in the

[4] $ABSREL(P, Q)$ is an operator which returns the formula constructed from P by removing all terms absorbed by Q.

function f_0'. For each function f_i, the $ABSREL$ operator is used to form a new formula $\dot{F}_i$ which represents a function $\dot{f}_i$,

$$\dot{F}_i = ABSREL(F_i, BCF(f_0')), \tag{3.160}$$

in which $\dot{F}_i$ consists of those terms in F_i which are not included in f_0'. Each function f_i may be split into the sum of two functions, i.e.,

$$f_i = \dot{f}_i + \hat{f}_i, \tag{3.161}$$

one of which is $\dot{f}_i$ and the other, call it $\hat{f}_i$, is represented by the terms in F_i absorbed by $BCF(f_0')$.

We must demonstrate that the resulting function $\dot{f}_i$ is in interval (3.159). Clearly, $\dot{f}_i \leq f_i$ is valid. Thus, $\dot{f}_i$ is less than the upper bound of (3.159). We now have to show that $f_0 f_i \leq \dot{f}_i$. Since, $f_0 f_i \leq f_i$ is valid, it is also true that

$$f_0 f_i \leq \dot{f}_i + \hat{f}_i. \tag{3.162}$$

It follows that

$$f_0 f_i \dot{f}_i' \hat{f}_i' = 0. \tag{3.163}$$

The valid statement $\hat{f}_i \leq f_0'$ is equivalent to stating that $\hat{f}_i f_0 = 0$. We then form the equation

$$f_0 f_i \dot{f}_i' \hat{f}_i' + \hat{f}_i f_0 = 0 \tag{3.164}$$

which is in turn equivalent to

$$f_0 f_i \dot{f}_i{}' = 0; \tag{3.165}$$

whence,

$$f_0 f_i \leq \dot{f}_i. \tag{3.166}$$

Thus, $\dot{f}_i$ is in interval (3.159).

Using relative absorption, we thus form a function $\dot{f}_i$ by deleting the terms in F_i which are included in function f_0'. We now show how relative simplification is used to simplify each formula $\dot{F}_i$ relative to $BCF(f_0')$, thus deleting literals from terms in $\dot{F}_i$. The formula resulting from relative simplification is the formula $\tilde{F}_i$ which represents the function $\tilde{f}_i$.

Relative Simplification. Each formula $\dot{F}_i$ may be simplified relative to terms in the formula $BCF(f_0')$ using the relative simplification process implemented by Procedure 2.17. We develop modified formulas $\tilde{F}_i$ by forming consensus terms between terms in $\dot{F}_i$ and terms of $BCF(f_0')$ when terms in $\dot{F}_i$ are absorbed by the resulting consensus terms. The consensus terms replace their parent terms in each formula $\dot{F}_i$ to form a new formula $\tilde{F}_i$; hence, each term in $\tilde{F}_i$ has a corresponding term in $\dot{F}_i$. Some terms in $\tilde{F}_i$ are the same as those in the formula $\dot{F}_i$; other terms in $\tilde{F}_i$ include (i.e., $\leq$) a corresponding term in $\dot{F}_i$. Hence, the new formulas, $\tilde{F}_i$, represent modified functions $\tilde{f}_i$, in which

$$\dot{f}_i \leq \tilde{f}_i. \tag{3.167}$$

The advantage of using the newly created terms in each $\ddot{F}_i$ rather than the parent terms in each $\dot{F}_i$ is that the new terms consist of fewer literals, hence less work must be performed to eliminate the X-arguments once we derive the equivalent 0-normal form, $f(A, X) = 0$, of system (3.116).

In essence, what is being accomplished in relative simplification is that some portion of the function f_0' is being added to each $\dot{f}_i$ to form a new function $\ddot{f}_i$. We demonstrate this in Example 3.8.

Example 3.8: Suppose we are given a function $f_0 = xy + yz$ which we would like to cover by the set $F = \{f_1, f_2\}$ of functions, in which the functions in F are defined by

$$\begin{aligned} f_1 &= x'y \\ f_2 &= xy. \end{aligned} \tag{3.168}$$

Forming $BCF(f_0')$, we derive

$$BCF(f_0') = y' + x'z'. \tag{3.169}$$

Neither f_1 nor f_2 is absorbed by any term of $BCF(f_0')$. Thus, we form the set of functions

$$\begin{aligned} \dot{f}_1 &= x'y \\ \dot{f}_2 &= xy. \end{aligned} \tag{3.170}$$

Using relative simplification, the consensus term x' would be created from the term $x'y$ of $\dot{f}_1$ and y' of $BCF(f_0')$. The term x' absorbs the term $x'y$, hence we form the function

$$\ddot{f}_1 = x'. \tag{3.171}$$

Similarly, we form the function

$$\bar{f}_2 = x. \tag{3.172}$$

By Boole's Expansion Theorem, $x' = x'y' + x'y$; thus, the relative simplification process added the term $x'y'$ to $\dot{f}_1$ to form $\bar{f}_1$. The term xy' was added to $\dot{f}_2$ to form $\bar{f}_2$. Both $x'y'$ and xy' are included in f_0', hence a portion of f_0' was added to $\dot{f}_1$ to form $\bar{f}_1$ and to $\dot{f}_2$ to create $\bar{f}_2$.

The sum $\dot{f}_i + f_0'$ is in interval (3.159). We expand f_0' with respect to $\dot{f}_i$ to develop the equivalent statement

$$\dot{f}_i + \dot{f}_i' f_0' + \dot{f}_i f_0'. \tag{3.173}$$

The function $\dot{f}_i f_0'$ represents the portion of f_0' which is included in $\dot{f}_i$; we delete this function by absorption. The function $\dot{f}_i' f_0'$ represents the portion of f_0' that is not included in $\dot{f}_i$; it also denotes the maximum part of function f_0' that may be added to each $\dot{f}_i$ to form $\bar{f}_i$. The sum $\dot{f}_i + \dot{f}_i' f_0'$ therefore represents the upper bound for function $\bar{f}_i$. Thus, we deduce an interval which defines the limits on the function $\bar{f}_i$:

$$\dot{f}_i \leq \bar{f}_i \leq \dot{f}_i + \dot{f}_i' f_0'. \tag{3.174}$$

Selecting a function $\bar{f}_i$ in the range given by (3.174) yields a function which in interval (3.159). $\bar{f}_i$ is greater than $\dot{f}_i$ which we demonstrated was greater than the lower bound of (3.159). Moreover, we established that $\dot{f}_i \leq f_i$. Finally, since $\dot{f}_i' f_0' \leq f_0'$ is valid, it follows that $\bar{f}_i$ is less than the upper bound of (3.159).

As a final step in forming a simplified formula for each function $\bar{f}_i$, the formula $ABS(\bar{F}_i)$ is formed to remove absorbed terms in the formula $\bar{F}_i$.

Identification of Implication Relations. When determining coverage of the function f_0 by subsets of the set F of functions, we may use the revised set $\tilde{F} = \{\tilde{f}_1, \tilde{f}_2, \ldots, \tilde{f}_k\}$ of functions in Procedure 3.4. Using the set $\tilde{F}$ to determine subsets of F which cover f_0, when a subset of functions in $\tilde{F}$ covers f_0, the corresponding functions in F also form a subset which covers f_0. The only portion of each function f_i which is not in each $\tilde{f}_i$ are those terms in F_i which are completely covered by f_0'; hence, they are of no utility in covering f_0.

Typically, each formula $\tilde{F}_i$ which represents the function $\tilde{f}_i$ consists of fewer terms than the formula F_i representing the respective function f_i. These terms are deleted from F_i to form $\tilde{F}_i$ during the relative absorption process. Additionally, terms in each $\tilde{F}_i$ consist of equal or fewer literals than the corresponding terms in the respective formula F_i. The literals are removed from terms during the relative simplification process. Hence, we have attained the goal of making the formulas F_i representing each function f_i simpler with respect to both terms and literals prior to the label-and-reduce and elimination processes. We formalize the concepts developed in this section with Theorem 3.8.

Theorem 3.8: *If a function f_0 is included in a subset of the set $\tilde{F} = \{\tilde{f}_1, \tilde{f}_2, \ldots, \tilde{f}_k\}$ of functions, then the function f_0 is included in the corresponding subset of the set $F = \{f_1, f_2, \ldots, f_k\}$ of functions, where each function $\tilde{f}_i$ is represented by*

$$\tilde{F}_i = SIMPREL(ABSREL(F_i, BCF(f_0')), BCF(f_0')). \tag{3.175}$$

$SIMPREL$ represents the application of the relative simplification operation.

Proof. The proof of this statement is demonstrated in the foregoing discussion. We summarize the steps for forming each formula $\tilde{F}_i$ which represents a function $\tilde{f}_i$.

1. Form the complement of f_0.
2. Generate $BCF(f_0')$.

3. For each function f_i, take the following steps:

   (a) develop a function $\dot{f}_i$, where

$$\dot{F}_i = ABSREL(F_i, BCF(f_0')); \tag{3.176}$$

   (b) form $\tilde{f}_i$, where

$$\tilde{F}_i = SIMPREL(\dot{F}_i, BCF(f_0')); \text{ and} \tag{3.177}$$

   (c) derive an equivalent formula to represent $\tilde{f}_i$ by forming $ABS(\tilde{F}_i)$.

This completes the proof. □

When developing the A-consequent terms which represent implication relations in Procedure 3.4, many of the prime A-consequent terms will represent normal subsets when the set $\tilde{F}$ of functions is used, whereas corresponding A-consequent terms may represent IIRs when the set F is used. The reason for this occurrence is the fact that the functions $\tilde{f}_1, \tilde{f}_2, \ldots, \tilde{f}_k$ have some portion of f_0' added to them during relative simplification. By the definition of the inclusion relation, the following statement is valid:

$$f_0 \leq f_1 + f_2 + \cdots + f_k \iff f_0' + f_1 + f_2 + \cdots + f_k = 1. \tag{3.178}$$

Hence, if some portion of f_0' is added to each f_i to form each $\tilde{f}_i$, then the functions in $\tilde{F}_i$ will likely form normal subsets, since f_0 is included in $f_1 + f_2 + \cdots + f_k$.

We now present a modified version of Procedure 3.4 which uses the set $\tilde{F}$ of functions to determine the coverage of a function f_0 by subsets of F.

Procedure 3.5 (Irredundant Implication Relations): Given a Boolean function f_0 and a set $F = \{f_1, f_2, \ldots, f_k\}$ of Boolean functions, we determine all irredundant implication relations denoting the inclusion of f_0 in a subset of F in the following manner:

Step 1. For function f_0:

1. Generate an associated label A_0.
2. Complement function f_0.
3. Form $BCF(f_0')$ for use in Step 2.

4. Prefix each term of the formula which represents f_0' with the literal A_0.

Step 2. For each function $f_i \in F$:

1. Form a function $\dot{f}_i$, where $\dot{F}_i = ABSREL(F_i, BCF(f_0'))$.
2. Form a function $\bar{f}_i$, where $\bar{F}_i = SIMPREL(\dot{F}_i, BCF(f_0'))$.
3. Generate an associated label A_i.
4. Prefix each term of the formula which represents $\bar{f}_i$ with the complemented literal A_i'.

The resulting formula represents $A_i' \cdot \bar{f}_i$.

Step 3. Append together each of the formulas developed in Steps 1 and 2. The resulting formula represents $f(A, X)$.

Step 4. Using the quick method for forming the conjunctive eliminant (Procedure 3.1), form $ECON(f(A, X), X)$. The resulting function equals $g(A)$, which is represented by $BCF(g(A))$.

Step 5. Each term of $BCF(g(A))$ consisting only of complemented literals represents a minimal normal subset; append the literal A_0 to each of these terms. (Do nothing to the remaining terms.) Call the resulting formula $\widehat{G}$.

Step 6. Form the equivalent absorptive, $ABS(\widehat{G})$, for $\widehat{G}$. Each term of $ABS(\widehat{G})$ represents an irredundant implication relation.

Step 7. For each term of $ABS(\widehat{G})$, form a set which consists of the functions in F; each function of the set corresponds to a respective function in $\{\bar{f}_1, \bar{f}_2, \ldots, \bar{f}_k\}$ which were associated with the labels in the term in Step 2. Return each subset of functions.

Example 3.9: Suppose we would like to form the irredundant implication relations which denote coverage of the function $f_0 = xyz + x'y'z'$ by subsets of the set $F = \{f_1, f_2, \ldots, f_6\}$ of functions defined by

$$\begin{aligned} f_1 &= xz \\ f_2 &= y'z' + xz' \\ f_3 &= x'z' \\ f_4 &= x'yz' + xy'z \\ f_5 &= x + z' \\ f_6 &= xy + x'y'. \end{aligned} \tag{3.179}$$

Step 1. We form

$$f_0' = x'y + x'z + xy' + y'z + xz' + yz'; \tag{3.180}$$

the right side of (3.180) is the Blake canonical form for f_0'.

Step 2. Using $BCF(f_0')$, we form the set $\dot{F} = \{\dot{f}_1, \dot{f}_2, \ldots, \dot{f}_k\}$ of functions, for which

$$\dot{F}_i = ABSREL(F_i, BCF(f_0')).$$

We derive the system

$$\begin{aligned} \dot{f}_1 &= xz \\ \dot{f}_2 &= y'z' \\ \dot{f}_3 &= x'z' \\ \dot{f}_4 &= 0 \\ \dot{f}_5 &= x + z' \\ \dot{f}_6 &= xy + x'y'. \end{aligned} \tag{3.181}$$

The set $\tilde{F} = \{\tilde{f}_1, \tilde{f}_2, \ldots, \tilde{f}_k\}$ of functions is then devised, for which

$$\tilde{F}_i = SIMPREL(\dot{F}_i, BCF(f_0')).$$

The functions $\tilde{f}_i$ are defined as follows:

$$\begin{aligned} \tilde{f}_1 &= z \\ \tilde{f}_2 &= y' \\ \tilde{f}_3 &= x' \\ \tilde{f}_4 &= 0 \\ \tilde{f}_5 &= x + z' \\ \tilde{f}_6 &= x + x'. \end{aligned} \tag{3.182}$$

Step 3. A system

$$\begin{aligned} A_0 &\leq xyz + x'y'z' \\ z &\leq A_1 \\ y' &\leq A_2 \\ x' &\leq A_3 \\ 0 &\leq A_4 \\ x + z' &\leq A_5 \\ x + x' &\leq A_6. \end{aligned} \tag{3.183}$$

of the form (3.116) is devised by associating the labels $A_0, A_1, \ldots, A_6$ with the respective functions. The system is in turn reduced to the form $f(A, X) = 0$:

$$\begin{aligned} f(A_1, A_2, A_3, A_4, A_5, A_6, x, y, z) = {} & A_0xy' + A_0x'y + A_0x'z + A_0xz' + A_0y'z + A_0yz' \\ & + A_1'z \\ & + A_2'y' \\ & + A_3'x' \\ & + A_4'0 \\ & + A_5'x + A_5'z' \\ & + A_6'x + A_6'x'. \end{aligned} \tag{3.184}$$

Step 4. Using goal-directed elimination to eliminate the variables x, y, and z produces the consequent $g(A) = 0$, for which g is defined by the equation

$$g(A_0, A_1, A_2, A_3, A_4, A_5, A_6) = A_0A_1'A_3' + A_0A_1'A_2' + A_0A_5' + A_1'A_5' + A_3'A_5' + A_6'. \tag{3.185}$$

The function g is in Blake canonical form.

Steps 5-7. The following A-consequents of $f = 0$ are deduced:

$$\begin{aligned} A_0 A_1' A_3' &= 0 \\ A_0 A_1' A_2' &= 0 \\ A_0 A_5' &= 0 \\ A_1' A_5' &= 0 \\ A_3' A_5' &= 0 \\ A_6' &= 0. \end{aligned} \tag{3.186}$$

The A-consequents in (3.186) denote the existence of the irredundant implication relations

$$\begin{aligned} f_0 &\leq \bar{f}_1 + \bar{f}_3 \\ f_0 &\leq \bar{f}_1 + \bar{f}_2 \\ f_0 &\leq \bar{f}_5 \end{aligned} \tag{3.187}$$

and the minimal normal subsets $\{\bar{f}_1, \bar{f}_5\}, \{\bar{f}_3, \bar{f}_5\}$, and $\{\bar{f}_6\}$. The normal subsets indicate that the equations

$$\begin{aligned} \bar{f}_1 + \bar{f}_5 &= 1 \\ \bar{f}_3 + \bar{f}_5 &= 1 \\ \bar{f}_6 &= 1 \end{aligned} \tag{3.188}$$

are identities. When the literal A_0 is appended to the last three A-consequent terms in (3.186), we deduce the A-consequents

$$A_0 A_1' A_5' = 0 \tag{3.189}$$

$$A_0 A_3' A_5' = 0 \tag{3.190}$$

$$A_0 A_6' = 0. \tag{3.191}$$

Equation (3.191) represents the irredundant implication relation

$$f_0 \leq \bar{f}_6. \tag{3.192}$$

However, equations (3.189) and (3.190) represent the implication relations

$$\begin{aligned} f_0 &\leq \bar{f}_1 + \bar{f}_5 \\ f_0 &\leq \bar{f}_3 + \bar{f}_5, \end{aligned} \tag{3.193}$$

which are not irredundant since $f_0 \leq \bar{f}_5$. The A-consequent terms $A_0A_1'A_5'$ and $A_0A_3'A_5'$ are absorbed by A_0A_5'.

Based on the subsets of the set $\{\bar{f}_1, \bar{f}_2, \ldots, \bar{f}_k\}$ which cover f_0, we then conclude that the following subsets of F cover f_0:

$$\{\{f_1, f_2\}, \{f_1, f_3\}, \{f_5\}, \{f_6\}\}. \tag{3.194}$$

Hence, the statements

$$\begin{aligned} xyz + x'y'z' &\leq (xz) + (y'z' + xz') \\ xyz + x'y'z' &\leq (xz) + (x'z') \\ xyz + x'y'z' &\leq x + z' \\ xyz + x'y'z' &\leq xy + x'y' \end{aligned} \tag{3.195}$$

are identities.

Covering a Term. We now discuss the special case of forming irredundant implication relations to denote coverage of a function f_0 by subsets of F when f_0 and the elements of F are single terms. A number of ideas may be incorporated to make efficient the process of determining the coverage of a single term t_0 by subsets of a set $T = \{t_1, t_2, \ldots, t_k\}$ of terms. We may identify a

priori the members of T which are not members of any minimal subset of T which covers t_0; such terms in T are called *irrelevant* with respect to t_0. The remaining terms in T—terms which cover some portion of t_0—are the only terms that must be considered when determining the coverage of t_0 by minimal subsets of T. We call these terms *relevant* with respect to t_0. Additionally, we may remove certain literals from the relevant terms prior to the process for developing the minimal subsets. We present in this section a procedure for forming minimal subsets in which irrelevant terms with respect to t_0 are deleted and certain literals are removed from relevant terms prior to the subset-identification process.

There exist three types of terms in a set T which are irrelevant with respect to t_0. These categories are:

- terms in T which contain an argument which is opposed to an argument contained in t_0;
- terms in T containing an argument not contained in t_0 which is unate in the formula formed from the disjunction of terms in T; and
- terms in T which only contain variables not "related" to t_0.

We discuss in turn each of these categories. Irrelevant terms in T are removed from consideration prior to the process for determining the coverage of t_0 by minimal subsets of T.

We may apply the methodology developed in the modified approach for identifying irredundant implication relations. Thus, each $t_i \in T$ may be replaced by a term $\tilde{t}_i$ which is a member of the interval

$$[(t_0 \cdot t_i), (t_0' + t_i)]. \tag{3.196}$$

If t_0 consists of the literals $l_1, l_2, \ldots, l_l$, then t_0' is equal to the disjunction of the complement of the literals, i.e.,

$$t_0' = l_1' + l_2' + \cdots + l_l'. \tag{3.197}$$

Any term t_i which contains at least one of the literals $l_1', l_2', \ldots, l_l'$ is absorbed in the sum $t_0' + t_i$. Additionally, since t_0 consists of the literals $l_1, l_2, \ldots, l_l$, any term t_i absorbed by one of the literals $l_1', l_2', \ldots, l_l'$ contains a literal that is opposed to a literal in t_0. Thus, $t_0 \cdot t_i = 0$. It follows that the interval (3.196) formed for a t_i which contains a literal opposed to a literal contained in t_0 is $[0, t_0']$. Thus, we may select $\tilde{t}_i = 0$, which means that terms t_i in T which contain a literal opposed to a literal in t_0 are irrelevant.

In addition to absorbing terms, the literals $l_1', l_2', \ldots, l_l'$ may be used to delete certain literals from terms t_i. By property (2.30), we may remove literals from terms t_i in T which are the complement of the literals $l_1', l_2', \ldots, l_l'$. This operation is a specialization of the relative simplification applied in the modified approach for identifying irredundant implication relations. Denoting the term $\tilde{t}_i$ as the term formed by removing the literals $l_1, l_2, \ldots, l_l$ from t_i, we thus form the interval

$$[(t_0 \cdot t_i), (l_1' + l_2' + \cdots + l_l' + \tilde{t}_i)]. \tag{3.198}$$

The statement $t_i \leq \tilde{t}_i$ is valid, since t_i contains a superset of the literals in $\tilde{t}_i$. It follows that a suitable term which is a member of interval (3.198) is $\tilde{t}_i$.

Thus, for each $t_i \in T$, we form a term $\tilde{t}_i$ in the following manner:

- $\tilde{t}_i = 0$, if t_i contains a literal opposed to a literal in t_0; or
- $\tilde{t}_i$ contains the literals of t_i not contained in t_0.

If we divide each term in T by t_0, we would deduce the same result as the process described above. This what is done in the method described in the Normal Subsets section of this chapter and illustrated by Example 3.5 for determining coverage of a term t_0 by subsets of a set T of terms.

After dividing each term of T by t_0, minimal normal subsets of the set $\tilde{T} = \{\tilde{t_1}, \tilde{t_2}, \ldots, \tilde{t_k}\}$ of terms correspond to minimal subsets of T which cover t_0.

Terms $t_j \in T$ which contain a variable x which is unate in the formula formed from the disjunction of terms in T are irrelevant—provided that x is not contained in t_0. Such terms exist only in non-minimal subsets of T which cover t_0. Hence, if a term t_j is included in a set of terms which covers t_0, we may delete it from the set and the resulting subset will still cover t_0. This idea was applied by Cutler (Cutle 80:48). Moreover, he states that after all terms containing variables which are unate in the formula formed by the disjunction of terms in T are removed from T, the process may be performed again to remove terms containing variables which may become unate—due to the removal of terms—in the disjunction of the remaining terms in T. The process is executed iteratively until there are no terms t_j containing arguments not contained in t_0 which are unate in the formula formed by the disjunction of the remaining terms in T.

The third category of terms in T which are irrelevant with respect to t_0 are terms in T which contain only variables not "related" to t_0. Variables *related* to t_0 are defined by the following recursive definition:

1. Variables in t_0 are related to t_0.
2. All variables contained in terms in T which contain variables related to t_0 are related to t_0.
3. All variables contained in terms in T which do not contain a related variable are *unrelated* variables.

Furthermore, we say that terms in T which contain variables related to t_0 are *related terms*. Likewise, if a term contains only variables not related to t_0, then we say that it is an *unrelated term*. If the collection of unrelated terms does not form a normal subset, all terms in T which are unrelated to t_0 may be removed from T prior to identifying minimal subsets of T which cover t_0. This concept is formalised by Theorem 3.9.

Theorem 3.9 (Unrelated Terms): *Given a term t_0, a set $T = \{t_1, \ldots, t_k\}$ of terms, and a subset U of T which consists of all the terms in T which are unrelated to t_0, if U does not form a normal subset, then no term in U is included in a minimal subset of T which covers t_0.*

Proof. Let us denote variables related to t_0 by X_1 and variables unrelated to t_0 by X_2. Let $\{t_1, \ldots, t_s\}$ be the subset of T consisting of terms related to t_0. Furthermore, let $U = \{t_{s+1}, \ldots, t_k\}$. We form the system

$$\begin{aligned} A_0 &\leq t_0(X_1) \\ t_1(X_1) &\leq A_1 \\ &\vdots \\ t_s(X_1) &\leq A_s \\ t_{s+1}(X_2) &\leq A_{s+1} \\ &\vdots \\ t_k(X_2) &\leq A_k \end{aligned} \tag{3.199}$$

which is reducible to the form $f(A, X_1, X_2) = 0$. The function $f(A, X_1, X_2)$ is represented by the formula

$$A_0 t_0'(X_1) + \sum_{i=1}^{s} t_i(X_1) A_i' + \sum_{i=s+1}^{k} t_i(X_2) A_i'. \tag{3.200}$$

Elimination of the X_1-arguments from $f(A, X_1, X_2) = 0$ yields a resultant of the form

$$g_1(A_0, A_1, \ldots, A_s) + \sum_{i=s+1}^{k} t_i(X_2) A_i' = 0 \tag{3.201}$$

since $\sum_{i=s+1}^{k} t_i(X_2) A_i'$ is independent of the X_1-arguments. Similarly, eliminating the X_2-arguments from (3.201) yields the consequent

$$g_1(A_0, A_1, \ldots, A_s) + g_2(A_{s+1}, \ldots, A_k) = 0 \tag{3.202}$$

since the function $g_1(A_0, A_1, \ldots, A_s)$ is independent of the X_2 arguments.

Since $f(A, X_1, X_2) = 0$, it follows that the statement

$$\sum_{i=s+1}^{k} t_i(X_2) A_i' = 0 \tag{3.203}$$

is an identity. Equation (3.203) is equivalent to the system

$$\begin{aligned} t_{s+1}(X_2) &\leq A_{s+1} \\ &\vdots \\ t_k(X_2) &\leq A_k. \end{aligned} \tag{3.204}$$

System (3.204) is included in system (3.199), therefore any consequent of (3.204) is also a consequent of (3.199). By Theorem 3.2, the A-consequent terms in (3.203), i.e., the terms in $g_2(A_{s+1}, \ldots, A_k)$, consist only of complemented literals of the set $\{A_{s+1}, \ldots, A_k\}$. However, in view of Theorem 3.1, A-consequent terms in (3.203) and hence $f(A, X_1, X_2) = 0$ consisting only of complemented literals of $\{A_{s+1}, \ldots, A_k\}$ exist if and only if the associated terms $\{t_{s+1}, \ldots, t_k\}$ form a normal subset. Since it was assumed that the terms in U do not form a normal subset, it follows that no A-consequent terms of $f(A, X_1, X_2) = 0$ exist consisting only of complemented literals of $\{A_{s+1}, \ldots, A_k\}$. Hence, the function $g_2(A_{s+1}, \ldots, A_k)$ is identically equal to zero.

Since $g_2(A_{s+1}, \ldots, A_k)$ is identically equal to zero, all A-consequent terms denoting the coverage of t_0 are deduced from the equation

$$g_1(A_0, A_1, \ldots, A_s) = 0. \tag{3.205}$$

The consequent $g_1(A_0, A_1, \ldots, A_s) = 0$ is derived from system (3.199) less the inclusions in system (3.204). The labels in g_1 are associated with the term t_0 and the terms in T which are related to

t_0. Hence, only terms in T which are related to t_0 are included in minimal subsets of T which cover t_0. This completes the proof. □

We now present a procedure which produces minimal subsets of a set T of terms which cover a term t_0. Once irrelevant terms with respect to t_0 are omitted from T, and literals of t_0 are removed from terms in T (via a divide operation), we determine sum-to-one subsets of the revised set $\hat{T}$; sum-to-one subsets of $\hat{T}$ correspond to minimal subsets of the original set T which cover t_0. If it is known beforehand that all terms in T are related to the term t_0, then Step 4 may be skipped in the procedure since it will have no effect on the result. Example 3.10 illustrates the application of Procedure 3.6.

Procedure 3.6 (Coverage of a Term): Given a term t_0 and a set $T = \{t_1, t_2, \ldots, t_k\}$ of terms, we determine all minimal subsets of T which cover the term t_0 in the following manner:

Step 1.

1. Divide each term in T by the term t_0. Denote the terms t_i/t_0 by the notation $\tilde{t}_i$.
2. Remove from T each element $\tilde{t}_i$ which is equal to 0.

Step 2. Determine the set V of variables which are unate in the formula formed by the disjunction of terms in T.

Step 3. Remove from V variables contained in the term t_0.

- If $V = \emptyset$, then continue to Step 4.
- Otherwise, remove from T terms containing any variable in V and return to Step 2.

Step 4.

1. Determine the set U of terms in T which are unrelated to t_0. (We assume that terms which are unrelated to t_0 do not form a normal subset.)
2. Remove from T terms included in the set U.

Call the revised set $\hat{T}$.

Step 5. For each term $\tilde{t}_i(X)$ remaining in T:

1. Generate an associated label T_i.
2. Prefix each term $\tilde{t}_i(X)$ with the complemented literal T_i'.

The resulting term is $T_i' \cdot \tilde{t}_i(X)$.

Step 6. Append together each of the terms formed in Step 5. The resulting formula represents $f(A, X)$, where the vector A is defined as $(T_1, \ldots, T_k)$.

Step 7. Using the quick method for forming the conjunctive eliminant (Procedure 3.1), form $ECON(f(A,X),X)$. The resulting function is equal to $g(A)$; the formula which represents $g(A)$ is $BCF(g(A))$.

Step 8. Each term of $BCF(g(A))$ consists only of complemented literals; each represents a minimal normal subset denoting the coverage of t_0 by minimal subsets of T. Call the resulting formula $\widehat{G}$.

Step 9. For each term in $\widehat{G}$, form a set which consists of the terms $\{t_1, t_2, \ldots, t_k\}$; each term of the set corresponds to a respective term in $\{\vec{t_1}, \vec{t_2}, \ldots, \vec{t_k}\}$ which were associated with the labels in the term. Return each subset of terms.

Example 3.10: Given a term wz and a set $T = \{w'z', w'x, wx', wy, wz, x'z', xy, xz, yz', u'v, uv'\}$, we use Procedure 3.6 to determine minimal subsets of T which cover wz.

Step 1. In the first step, we divide each term of T by the term wz. The set

$$\{0, 0, x', y, 1, 0, xy, x, 0, u'v, uv'\} \tag{3.206}$$

results from the division process. Removing the elements of (3.206) which are equal to zero, we develop the set

$$\{x', y, 1, xy, x, u'v, uv'\}. \tag{3.207}$$

Steps 2-3. Forming the disjunction of terms in (3.207), we find that the literal y is unate in the resulting formula. Hence, we may remove terms containing y from (3.207) to form the set

$$\{x', 1, x, u'v, uv'\}. \tag{3.208}$$

Step 4. Given the term wz and the original set T of terms, the variables u and v are unrelated to the term wz. Hence, the terms $u'v$ and uv' are unrelated to wz. Since $\{u'v, uv'\}$ is not a normal set, the terms $u'v$ and uv' are removed from (3.208). We thus form the set $\widehat{T} = \{x', 1, x\}$, which is used to determine subsets of T which cover wz.

Steps 5-6. We form the system

$$\begin{array}{rcl} x' & \leq & T_3 \\ 1 & \leq & T_5 \\ x & \leq & T_8 \end{array} \tag{3.209}$$

which is reduced to the equation $f(A, X) = 0$, for which

$$f(A, X) = x'T_3' + T_5' + xT_8'. \tag{3.210}$$

The subscripts used for each label T_i corresponding to a term in $\widehat{T}$ are chosen to to denote the term in the original set T from which we derive the respective term in $\widehat{T}$. This allows us to identify the subsets of T which cover wz.

Step 7. Eliminating the X-argument from $f(A, X) = 0$, we form the resultant $g(A) = 0$, where

$$g(A) = T_3'T_8' + T_5'. \tag{3.211}$$

Steps 8-9. Thus, term wz is covered by the fifth term in T—the term wz. Additionally, wz is covered by the disjunction of terms in the subset $\{wx', xz\}$ of T, i.e.,

$$wz \leq wx' + xz. \tag{3.212}$$

Contrast of Label-and-Eliminate Procedure with Specialized Procedures

Significant differences exist between the general label-and-eliminate procedure and the specialised methods developed in this chapter. Each uses a label-and-reduce process; however, the label-and-eliminate procedure employs full labeling-and-reduction while the special-purpose methods use partial labeling-and-reduction. Using partial labeling-and-reduction rather than full labeling-and-

reduction reduces the number of computations as well as memory usage. Once we generate an equation of the form $f(A, X) = 0$, the unified set of procedures use the goal-directed elimination process to deduce consequents of a specific form. This process uses the quick method for forming the conjunctive eliminant of a function with respect to a set of variables (Procedure 3.1). On the other hand, the label-and-eliminate procedure uses a general method for forming the conjunctive eliminant (Procedure 2.22). To reduce the complexity of eliminant formation, the label-and-eliminate technique simplifies the formula resulting after each iteration of formation of the conjunctive eliminant; the specialised procedures make absorptive the resulting formula after each iteration of conjunctive eliminant formation. Finally, after forming a consequent resulting from the elimination of the X-arguments, the label-and-eliminate procedure must formulate a Blake canonical form to produce all possible A-consequent terms. The unified set of procedures produces a consequent which is represented by its Blake canonical form at the conclusion of the elimination process; the consequent includes only those A-consequent terms which denote specific relationships among the set of functions. Table 3.1 summarises the differences between the label-and-eliminate procedure and the unified set of specialised procedures presented in this chapter.

	Label-and-Eliminate Procedure	Specialised Procedures
Label-and-Reduce Process	Full Label-and-Reduce	Partial Label-and-Reduce
Conjunctive Eliminant	Procedure 2.22	Procedure 3.1 (Quick-Econ)
Simplification Method	Procedure 2.15 (Simplify)	Absorption
Blake Canonical Form	Formation Required	Implicit in Process
A-Consequent Terms	All A-Consequent Terms	Specific A-Consequent Terms

Table 3.1. Contrast of Label-and-Eliminate Method with Specialised Procedures

We now present an example which illustrates the difference between the label-and-eliminate procedure and the unified set of procedures for deducing specific relationships among functions.

Example 3.11: Suppose we would like to determine the relationships among members of the set $\{f_0, f_1, \ldots, f_6\}$ of functions defined by

$$
\begin{aligned}
f_0 &= xyz + x'y'z' \\
f_1 &= z \\
f_2 &= y'z' + d \\
f_3 &= x'z' \\
f_4 &= x + z' \\
f_5 &= w \\
f_6 &= w'.
\end{aligned}
\tag{3.213}
$$

Using the label-and-eliminate procedure, we form the following prime A-consequent terms which denote all relationships among the set of functions:

1. $A_2'A_6'$, $A_1'A_4'$, $A_5'A_6'$;
2. A_5A_6, $A_1A_2A_6$, A_1A_3;
3. $A_0A_1'A_2'$, $A_0A_1'A_3'$, A_0A_4';
4. $A_2A_4'A_5'$, A_3A_4', A_5A_7';
5. $A_0A_3A_2'$, $A_2A_6A_4'$, $A_1A_2A_5'$, $A_2A_3A_6A_0'$; and
6. $A_2A_3A_0'A_5'$, $A_0A_2A_6A_3'$, $A_0A_2A_3'A_5'$.

The terms in line 1 denote normal subsets. Terms in line 2 represent evanescent subsets. Implication relations in which f_0 is included in subsets of the other functions are represented by terms in line 3; terms in line 4 represent all other implication relations. Terms in the remaining lines portray a relationship in which products of functions are included in sums of functions.

Using Procedure 3.2 to find all normal subsets, we would derive only those A-consequent terms which appear in line 1. Procedure 3.3 would generate only those terms found in line 2 when determining evanescent subsets. If we were to use Procedure 3.4 to determine the subsets of functions in which the function f_0 is included, we would derive the terms found in lines 1 and 3. When deducing normal subsets, evanescent subsets, or irredundant implication relations for the inclusion of f_0, we avoid generating the consequents listed in lines 4, 5, and 6.

Summary

In this chapter, a new set of procedures was presented for deducing specific relationships among subsets of a set of functions. Procedures in this set provide a more efficient way to deduce specific relationships among a set of functions than does the general label-and-eliminate procedure. Having the same basic theoretical foundation, these techniques compose a "unified" set of specialised procedures to determine particular relationships among functions of a set of functions. In later chapters, techniques developed in this chapter will be applied to construct efficient methods for dealing with the minimization problem.

IV. Solutions of Boolean Equations and the Minimization Problem

In this chapter we discuss the solutions of Boolean equations, the modeling of digital circuits with Boolean algebra, and the relationship between solving Boolean equations and the minimization problem. The correspondence between developing a good solution for a Boolean equation and the process of developing an economical digital design which meets a specification is highlighted. A comprehension of the second and third sections of the chapter is particularly important to understanding the approach taken to the minimization problem in subsequent chapters.

The solutions of Boolean equations of the form $f(X) = 1$ are presented in the first section. Rudeanu (Rudea 74) and Brown (Brown 90) develop solutions for Boolean equations of the form $f(X) = 0$. We present a development of solutions for $f(X) = 1$ similar to their presentation for the $f(X) = 0$ case, since solutions of the Boolean equation $f(X) = 1$ is our specific interest.

In the second section the modeling of circuits with Boolean algebra is discussed. Circuit specifications correspond to Boolean functions. Designs, on the other hand, correspond to formulas which represent a function. The object of the design problem is to develop an economical design which implies its specification. We show in the third section that the design process is in correspondence with finding good solutions for a Boolean equation. We first solve a Boolean equation in 1-normal form to develop an interval, i.e., a range of functions, for each output of a circuit. Finding a good formula to represent a function in the range of functions, i.e., finding a "good" solution, yields an economical design. The resulting design implies its specification, just as $f(X) = 1$ is implied by any of its solutions.

Solutions of Boolean Equations

Functional Antecedents. Consider an equation

$$f(X) = 1, \tag{4.1}$$

where $X = (x_1, \ldots, x_n)$, and a system of the form

$$\begin{aligned} x_1 &= h_1 \\ x_2 &= h_2 \\ &\vdots \\ x_n &= h_n, \end{aligned} \tag{4.2}$$

where each h_i is a formula in a free Boolean algebra, $FB(i_1, i_2, \ldots, i_k)$, on k generators $i_1, i_2, \ldots, i_k$. If system (4.2) is an antecedent of system (4.1), then (4.2) is called a *functional antecedent* or *solution* of (4.1). A substitution $A \in \mathbf{B}^n$ for X which causes (4.1) to be an identity is also called a solution of (4.1); specifically, it is called a *particular solution* of (4.1). A *general solution* of a Boolean equation is a representation of the set of all particular solutions of the equation. There exist various ways to represent a general solution; in this work we are concerned with interval-based representations. For other representations, see (Brown 90) or (Rudea 74).

Consistency of a Boolean Equation. A Boolean equation is *consistent* if it has at least one solution. Otherwise, the equation is said to be *inconsistent*. A necessary and sufficient condition for the consistency of $f(X) = 1$ is given in Theorem 4.1.

Theorem 4.1 (Consistency Condition): *The Boolean equation $f(X) = 1$ is consistent if and only if the condition*

$$EDIS(f(X), X) = 1 \tag{4.3}$$

is satisfied.

Proof. We prove this statement inductively.

We first show the case of $n = 1$. Assume that $a \in \mathbf{B}$ is a solution of $f(x) = 1$; then $f(a) = 1$. Using Boole's Expansion Theorem, we form

$$a'f(0) + af(1) = 1. \tag{4.4}$$

Complementing both sides of (4.4) yields $a'f'(0) + af'(1) + f'(0)f'(1) = 0$. Thus, $f'(0)f'(1) = 0$; this may be rewritten as $f(0) + f(1) = 1$. Hence, $EDIS(f(x), \{x\}) = 1$ by the definition of the disjunctive eliminant. Suppose now that $EDIS(f(x), \{x\}) = 1$; i.e., $f(0) + f(1) = 1$. In this case the element $f(1)$ is a solution of $f(x) = 1$, because

$$f(f(1)) = (f(1))'f(0) + f(1)f(1) = f(0) + f(1) = 1 \tag{4.5}$$

by Boole's Expansion Theorem, idempotence, and Property (2.30). Thus, $f(x) = 1$ is consistent. Hence, we have proven the theorem for $n = 1$.

For $n = k$, where $k > 1$, we assume for our induction hypothesis that $g(x_1, x_2, \ldots, x_k) = 1$ is consistent if and only if $EDIS(g(x_1, \ldots, x_k), \{x_1, \ldots, x_k\}) = 1$.

We now must prove the theorem for $n = k+1$. We first assume that $f(x_1, \ldots, x_{k+1}) = 1$ is consistent, i.e., an $A \in \mathbf{B}^n$ exists for which $f(A) = 1$ is an identity. Because $f(X) \leq EDIS(f(X), T)$ for a subset $T \subseteq X$ of variables in f by (2.156), $EDIS(f(x_1, \ldots, x_{k+1}), \{x_1, \ldots, x_{k+1}\}) = 1$ is an identity.

We now demonstrate that if

$$EDIS(f(x_1, \ldots, x_{k+1}), \{x_1, \ldots, x_{k+1}\}) = 1 \tag{4.6}$$

is true, then $f(x_1,\ldots,x_{k+1}) = 1$ is consistent. $EDIS(f(x_1,\ldots,x_{k+1}),\{x_1,\ldots,x_{k+1}\})$ may be rewritten as $EDIS(f(x_1,\ldots,x_{k+1}),\{x_1,\ldots,x_k\}\cup\{x_{k+1}\})$. Then by the definition of the disjunctive eliminant, the function

$$EDIS(EDIS(f(x_1,\ldots,x_{k+1}),\{x_{k+1}\}),\{x_1,\ldots,x_k\}) \tag{4.7}$$

is equal to

$$EDIS(f(x_1,\ldots,x_{k+1}),\{x_1,\ldots,x_k\}\cup\{x_{k+1}\}). \tag{4.8}$$

It follows that the statement

$$EDIS(EDIS(f(x_1,\ldots,x_{k+1}),\{x_{k+1}\}),\{x_1,\ldots,x_k\}) = 1 \tag{4.9}$$

is also true. By our induction hypothesis, if the disjunctive eliminant of a k-variable function g with respect to its k variables is equal to 1, then the equation $g(x_1,\ldots,x_k) = 1$ is consistent. Since $EDIS(f(x_1,\ldots,x_{k+1}),\{x_{k+1}\})$ is a function of k variables and the disjunctive eliminant of $EDIS(f(x_1,\ldots,x_{k+1}),\{x_{k+1}\})$ with respect to its k variables is equal to 1 by (4.9), it follows that $EDIS(f(x_1,\ldots,x_{k+1}),\{x_{k+1}\}) = 1$ is consistent.

Let us define a k-variable function $g(x_1,\ldots,x_k)$ by

$$g(x_1,\ldots,x_k) = EDIS(f(X),\{x_{k+1}\}), \tag{4.10}$$

i.e.,

$$g(x_1,\ldots,x_k) = f(x_1,\ldots,x_k,0) + f(x_1,\ldots,x_k,1). \tag{4.11}$$

Let $(a_1, \ldots, a_k) \in \mathbf{B}^k$ be a solution of the consistent equation $g(x_1, \ldots, x_k) = 1$. Thus,

$$f(a_1, \ldots, a_k, 0) + f(a_1, \ldots, a_k, 1) = 1 \tag{4.12}$$

is an identity, and therefore $f(a_1, \ldots, a_k, x_{k+1}) = 1$ has a solution, as shown in the case of $n = 1$. Thus, the equation $f(x_1, \ldots, x_k, x_{k+1}) = 1$ is consistent, and we have proven the theorem for the case of $n = k + 1$. Hence, the theorem is true for arbitrary n. This completes the proof. □

The equation $EDIS(f(X), X) = 1$ is called the *consistency condition* for $f(X) = 1$, i.e., the condition which is necessary and sufficient for $f(X) = 1$ to be consistent.

General Solutions. A *general solution* of a Boolean equation is a representation of the set of all particular solutions of the equation. We now endeavor to develop an interval-based general solution for a Boolean equation $f(X) = 1$. Such a solution is based on a lower and upper bound, i.e., a range. Each value within the range is a particular solution, and all particular solutions are found within the range.

The Single-Variable Case. In Lemma 4.1, an interval-based general solution for the single-variable equation $f(x) = 1$ is developed.

Lemma 4.1 (Solution - Single Variable): *Let $f : \mathbf{B} \to \mathbf{B}$ be a Boolean function for which the equation $f(x) = 1$ is consistent. Then the set of solutions for $f(x) = 1$ is given by*

$$\{x \mid f'(0) \leq x \leq f(1)\}. \tag{4.13}$$

Proof. We show that $f(x) = 1 \Leftrightarrow f'(0) \leq x \leq f(1)$. By Boolean expansion (2.62), $f(x) = 1$ is expressed equivalently as

$$[x' + f(1)] \cdot [x + f(0)] = 1. \tag{4.14}$$

In view of (2.41), (4.14) is equivalent to the system

$$\begin{aligned} x' + f(1) &= 1 \\ x + f(0) &= 1, \end{aligned} \tag{4.15}$$

which, by the definition of the inclusion relation, is in turn equivalent to the system

$$\begin{aligned} x &\leq f(1) \\ f'(0) &\leq x. \end{aligned} \tag{4.16}$$

System (4.16) is equivalent to the interval

$$f'(0) \leq x \leq f(1). \tag{4.17}$$

This completes the proof. □

Given the set of solutions $\{x \mid f'(0) \leq x \leq f(1)\}$ for $f(x) = 1$ and the consistency condition

$$EDIS(f(x), \{x\}) = 1, \tag{4.18}$$

we can develop a general solution for $f(x) = 1$ which "extends" the range expressed by $f'(0) \leq x \leq f(1)$. When we say that we are "extending" the range for x, we mean that we form an upper bound which is greater than a previous upper bound and a lower bound that is less than a previous lower bound. This result is shown in Theorem 4.2.

Theorem 4.2 (Extended Range - Single Variable): *Let $f : \mathbf{B} \rightarrow \mathbf{B}$ be a Boolean function for which the equation $f(x) = 1$ is consistent. Then*

$$f(x) = 1 \iff f'(0) \cdot f(1) \leq x \leq f'(0) + f(1). \tag{4.19}$$

Proof. In Lemma 4.1 it was shown that

$$f(x) = 1 \iff f'(0) \leq x \leq f(1). \tag{4.20}$$

If $f(x) = 1$ is consistent, then $EDIS(f(x), \{x\}) = 1$. By the definition of the disjunctive eliminant

$$EDIS(f(x), \{x\}) = f(0) + f(1). \tag{4.21}$$

Hence, $f(0) + f(1) = 1$ or $f'(0)f'(1) = 0$.

Using Boole's Expansion Theorem, we expand $f'(0)$ with respect to $f(1)$ to form the identity

$$f'(0) = f'(1)f'(0) + f(1)f'(0). \tag{4.22}$$

Additionally, we can add $f'(0)f'(1)$ to $f(1)$ to form the identity

$$f(1) = f(1) + f'(0)f'(1). \tag{4.23}$$

Substituting for $f'(0)$ and $f(1)$ in $f'(0) \leq x \leq f(1)$, we conclude that

$$f'(1)f'(0) + f(1)f'(0) \leq x \leq f(1) + f'(0)f'(1) \tag{4.24}$$

is equivalent to $f(x) = 1$ provided the latter is consistent. Because $f'(0)f'(1) = 0$, and by property (2.30), we can simplify both the upper and lower bounds of (4.24) to form the equivalent interval

$$f'(0)f(1) \leq x \leq f'(0) + f(1). \tag{4.25}$$

Hence, $f(x) = 1 \iff f'(0) \cdot f(1) \leq x \leq f'(0) + f(1)$. This completes the proof. □

The n-Variable Case. We now generalise the approach used to develop an interval-based solution for the single-variable equation $f(x) = 1$ to an n-variable Boolean equation $f(X) = 1$. Our object is to develop a general solution of $f(X) = 1$ of the form

$$\begin{array}{rcccl} & & 1 & \leq & t_0 \\ s_1 & \leq & x_1 & \leq & t_1 \\ s_2(x_1) & \leq & x_2 & \leq & t_2(x_1) \\ s_3(x_1, x_2) & \leq & x_3 & \leq & t_3(x_1, x_2) \\ & & \vdots & & \\ s_n(x_1, \ldots, x_{n-1}) & \leq & x_n & \leq & t_n(x_1, \ldots, x_{n-1}) \end{array} \tag{4.26}$$

where t_0, s_1, and t_1 are constants and all other s_i and t_i are functions with variables as denoted in (4.26).

A system in the form of (4.26) is called a ***subsumptive general solution*** of the n-variable Boolean equation $f(X) = 1$ if $1 \leq t_0$[1] is the consistency condition of $f(X) = 1$, and if $f(X) = 1$ is consistent, then every particular solution $(a_1, a_2, \ldots, a_n)$ of $f(X) = 1$ is generated by the following procedure:

1. select a_1 in the range $s_1 \leq x_1 \leq t_1$;
2. select a_2 in the range $s_2(a_1) \leq x_2 \leq t_2(a_1)$;
3. and so on, until we select a_n in the range $s_n(a_1, \ldots, a_{n-1}) \leq x_n \leq t_n(a_1, \ldots, a_{n-1})$.

A method for developing a subsumptive general solution of $f(X) = 1$ is via a ***successive elimination of variables.*** Theorem 4.3 employs this method. The proof parallels a proof in Rudeanu (Rudea 74:69-71) for the generation of solutions for $f(X) = 0$.

[1] Since any function is less then the 1 element, we may simply say that $t_0 = 1$ is the consistency condition of $f(X) = 1$.

Theorem 4.3 (Successive Elimination of Variables): *Let $f : \mathbf{B}^n \to \mathbf{B}$ be a Boolean function for which the equation $f(x_1, \ldots, x_n) = 1$ is consistent, and let*

$$f_i(x_1, \ldots, x_i) = EDIS(f(x_1, \ldots, x_n), \{x_{i+1}, \ldots, x_n\}), \qquad (i = 0, \ldots, n). \tag{4.27}$$

Then, the set of solutions of

$$f(x_1, \ldots, x_n) = 1 \tag{4.28}$$

is described by the system

$$f_i'(x_1, \ldots, x_{i-1}, 0) \leq x_i \leq f_i(x_1, \ldots, x_{i-1}, 1) \qquad (i = 1, \ldots, n). \tag{4.29}$$

Proof. Given an equation $f(x_1, \ldots, x_n) = 1$, for any $i \in \{0, \ldots, n\}$ we form the resultant of elimination

$$EDIS(f(x_1, \ldots, x_n), \{x_{i+1}, \ldots, x_n\}) = 1. \tag{4.30}$$

The function $EDIS(f(x_1, \ldots, x_n), \{x_{i+1}, \ldots, x_n\})$ is a function of i variables, which by (4.27) is equal to $f_i(x_1, \ldots, x_i)$. Since (4.30) is true, it follows that

$$f_i(x_1, \ldots, x_i) = 1. \tag{4.31}$$

For every $i = n, n-1, \ldots, 1$, we apply Boole's Expansion Theorem (2.62) to equation (4.31) to form

$$[x_i' + f_i(x_1, \ldots, x_{i-1}, 1)][x_i + f_i(x_1, \ldots, x_{i-1}, 0)] = 1, \tag{4.32}$$

which is expressed equivalently by the system

$$\begin{aligned} x_i' + f_i(x_1, \ldots, x_{i-1}, 1) &= 1 \\ x_i + f_i(x_1, \ldots, x_{i-1}, 0) &= 1. \end{aligned} \tag{4.33}$$

System (4.33) is equivalent in turn to the interval

$$f_i'(x_1, \ldots, x_{i-1}, 0) \leq x_i \leq f_i(x_1, \ldots, x_{i-1}, 1) \qquad (i = 1, \ldots, n). \tag{4.34}$$

Since $a \leq b \Leftrightarrow a' + b = 1$, (4.34) is valid only if the equation $f_i(x_1, \ldots, x_{i-1}, 0) + f_i(x_1, \ldots, x_{i-1}, 1) = 1$, stated equivalently as

$$EDIS(f_i(x_1, \ldots, x_i), \{x_i\}) = 1, \tag{4.35}$$

is an identity. In view of (4.27), we substitute for f_i in (4.35) to form the equivalent statement

$$EDIS(EDIS(f(x_1, \ldots, x_n), \{x_{i+1}, \ldots, x_n\}), \{x_i\}) = 1, \tag{4.36}$$

which in turn is equivalent to

$$EDIS(f(x_1, \ldots, x_n), \{x_i, \ldots, x_n\}) = 1. \tag{4.37}$$

Equation (4.37) is an identity, because it is the resultant of elimination of $\{x_i, \ldots, x_n\}$ from $f(x_1, \ldots, x_n) = 1$. Consequently, if $f(x_1, \ldots, x_n) = 1$ is an identity, then so is (4.29).

Since we are assuming that $f(x_1, \ldots, x_n) = 1$ is consistent, then the condition

$$EDIS(f(x_1, \ldots, x_n), \{x_1, \ldots, x_n\}) = 1 \tag{4.38}$$

is satisfied. The left-hand side of (4.38) is equal to f_0, hence $f_0 = 1$. Because $f_0 = 1$ is the consistency condition for $f_1(x_1) = 1$, then $f_1'(0) \leq x_1 \leq f_1(1)$ is a solution for $f_1(x_1) = 1$.

Continuing this methodology, we find that given an x_1, we develop an interval $f_2'(x_1, 0) \leq x_2 \leq f_2(x_1, 1)$. Then, because $f_1(x_1) = 1$ is the consistency condition for $f_2(x_1, x_2) = 1$, $f_2(x_1, x_2) = 1$ is consistent. We apply this process iteratively through the case of x_{n-1}, finding that $f_2(x_1, x_2) = 1$ through $f_{n-1}(x_1, \ldots, x_{n-1}) = 1$ are all consistent. In the process, intervals are formed for $x_2, \ldots, x_{n-1}$.

Given elements $x_1, \ldots, x_{n-1}$, we derive an interval for x_n:

$$f_n'(x_1, \ldots, x_{n-1}, 0) \leq x_n \leq f_n(x_1, \ldots, x_{n-1}, 1). \tag{4.39}$$

For the interval (4.39) to be true, the equation

$$f_n(x_1, \ldots, x_{n-1}, 0) + f_n(x_1, \ldots, x_{n-1}, 1) = 1 \tag{4.40}$$

must be valid. The left-hand side of (4.40) is $EDIS(f_n, \{x_n\})$ which is equal to $f_{n-1}(x_1, \ldots, x_{n-1})$. Since $f_{n-1}(x_1, \ldots, x_{n-1}) = 1$ is consistent, it follows that the consistency condition (4.40) holds for $f_n(x_1, \ldots, x_n) = 1$. Thus, any substitution $(a_1, \ldots, a_n)$ for $(x_1, \ldots, x_n)$ which makes the statements

$$
\begin{array}{rcccl}
 & & 1 & \leq & f_0 \\
f_1'(0) & \leq & x_1 & \leq & f_1(1) \\
f_2'(x_1,0) & \leq & x_2 & \leq & f_2(x_1,1) \\
f_3'(x_1,x_2,0) & \leq & x_3 & \leq & f_3(x_1,x_2,1) \\
 & & \vdots & & \\
f_n'(x_1,\ldots,x_{n-1},0) & \leq & x_n & \leq & f_n(x_1,\ldots,x_{n-1},1)
\end{array}
\tag{4.41}
$$

identities, also makes $f_n(x_1,\ldots,x_n) = 1$ an identity. Thus, the system (4.41) defines the complete set of solutions for $f_n(x_1,\ldots,x_n) = 1$. This completes the proof. □

The name "successive elimination" for the technique developed in Theorem 4.3 refers to the fact that we derive the equation $f_{i-1}(x_1,\ldots,x_i) = 1$ via the elimination of x_i from $f_i(x_1,\ldots,x_i) = 1$. This is apparent in the progression from (4.31) to (4.37); the left-hand side of (4.37) is equal to the function $f_{i-1}(x_1,\ldots,x_{i-1})$.

The system (4.41) is a subsumptive general solution for $f_n(X) = 1$, for which (4.38) is the consistency condition. A particular solution is derived by selecting an a_1 in the range $f_1'(0) \leq x_1 \leq f_1(1)$, selecting an a_2 in the range $f_2'(a_1,0) \leq x_2 \leq f_2(a_1,1)$, and so on until we choose a_n in the range $f_n'(a_1,\ldots,a_{n-1},0) \leq x_n \leq f_n(a_1,\ldots,a_{n-1},1)$. We may develop a system such as (4.41) using any ordering of the X-arguments. The form of the general solution depends on the ordering of the X-arguments; however, the set of particular solutions is unique.

Procedure 4.1 produces a subsumptive general solution of a Boolean equation $f_n(X) = 1$ using successive elimination of variables. The procedure accepts a function $f_n(X)$ and an ordering of the X-arguments, and returns a set of intervals based on the X-argument order. The first X-argument is treated as x_n in (4.41), the second as x_{n-1}, and so on until the last argument in the X-argument order is handled as x_1. The consistency condition $EDIS(f_n(X), X) = 1$ is also returned as part of the result. Example 4.1 illustrates the application of Procedure 4.1.

Procedure 4.1 (Subsumptive General Solution - Successive Elimination): Given a Boolean function $f(x_1, \ldots, x_n)$ and an ordering $ARGS$ of the X-arguments, we develop a subsumptive general solution of $f(x_1, \ldots, x_n) = 1$ as follows:

Step 0. Initialize accumulators $f_{current}$ to $f(x_1, \ldots, x_n)$ and $SOLN$ to empty.

Step 1.

- If $ARGS$ is empty, then $SOLN$ contains a set of intervals such as (4.41) which represent a subsumptive general solution for $f(x_1, \ldots, x_n) = 1$. $f_{current} = 1$ is the consistency condition for $f(x_1, \ldots, x_n) = 1$. Return $SOLN$ and $f_{current}$.
- Otherwise, continue to Step 2.

Step 2.

1. Remove the first element from $ARGS$ and call it x.
2. Form $(f_{current}/x')'$. It is the lower bound in the interval for the argument x.
3. Form $(f_{current}/x)$. It is the upper bound in the interval for the argument x.
4. Form $EDIS(f_{current}, \{x\})$. Replace $f_{current}$ with $EDIS(f_{current}, \{x\})$.
5. Create a list consisting of x, $(f_{current}/x')'$, and $(f_{current}/x)$. This list represents the interval $(f_{current}/x')' \leq x \leq (f_{current}/x)$. Add this list to $SOLN$.
6. Return to Step 1.

Example 4.1: Given a function $f(x_1, x_2, x_3)$ represented by the formula

$$b'x_1x_2x_3' + a'b'x_1x_2 + abx_1'x_2 + ax_1'x_2'x_3' + b'x_1'x_2'x_3 + ab'x_2', \tag{4.42}$$

we develop a subsumptive general solution for $f(x_1, x_2, x_3) = 1$.

Suppose the ordering of the X-arguments is given as x_3, x_2, x_1. We form the following eliminants of f:

$$\begin{aligned} f_3(x_1, x_2, x_3) &= b'x_1x_2x_3' + a'b'x_1x_2 + abx_1'x_2 + ax_1'x_2'x_3' + b'x_1'x_2'x_3 + ab'x_2' \\ f_2(x_1, x_2) &= abx_1' + b'x_1'x_2' + ab'x_2' + b'x_1x_2 \\ f_1(x_1) &= b' + ax_1' \\ f_0 &= a + b'. \end{aligned} \tag{4.43}$$

Using these eliminants and (4.41), the system

$$\begin{array}{rcccl} & & 1 & \leq & a+b' \\ & a'b & \leq & x_1 & \leq & b' \\ & bx_1+a'x_1+a'b & \leq & x_2 & \leq & b'x_1+abx_1' \\ a'x_1'+a'x_2'+bx_1+b'x_1'x_2 & \leq & x_3 & \leq & b'x_1'x_2'+ab'x_2'+a'b'x_1x_2+abx_1'x_2 \end{array} \tag{4.44}$$

is developed. System (4.44) is a subsumptive general solution for $f(x_1, x_2, x_3) = 1$.

Just as in the single-variable case, we develop an extended range for the n-variable case. This idea is formalized by Theorem 4.4.

Theorem 4.4 (Subsumptive General Solution - Successive Elimination - Extended Range): *Let $f : \mathbf{B}^n \rightarrow \mathbf{B}$ be a Boolean function for which the equation $f(x_1, \ldots, x_n) = 1$ is consistent, and let*

$$f_i(x_1, \ldots, x_i) = EDIS(f(x_1, \ldots, x_n), \{x_{i+1}, \ldots, x_n\}) \quad (i = 1, \ldots, n). \tag{4.45}$$

Then, the set of solutions of

$$f(x_1, \ldots, x_n) = 1 \tag{4.46}$$

is described by

$$f_i'(x_1, \ldots, x_{i-1}, 0) \cdot f_i(x_1, \ldots, x_{i-1}, 1) \leq x_i \leq f_i'(x_1, \ldots, x_{i-1}, 0) + f_i(x_1, \ldots, x_{i-1}, 1) \quad (i = 1, \ldots, n). \tag{4.47}$$

Proof. In Theorem 4.3 it was shown that

$$f_i'(x_1, \ldots, x_{i-1}, 0) \leq x_i \leq f_i(x_1, \ldots, x_{i-1}, 1) \qquad (i = 1, \ldots, n), \tag{4.48}$$

is a set of solutions for $f(x_1, \ldots, x_n) = 1$. Then for each i, we form

$$f_i'(x_1, \ldots, x_{i-1}, 0) \cdot f_i'(x_1, \ldots, x_{i-1}, 1) = 0 \tag{4.49}$$

by the definition of the inclusion relation.

Using Boole's Expansion Theorem, we expand the function $f_i'(x_1, \ldots, x_{i-1}, 0)$ with respect to $f_i(x_1, \ldots, x_{i-1}, 1)$. Then $f_i'(x_1, \ldots, x_{i-1}, 0)$ is equal to

$$f_i'(x_1, \ldots, x_{i-1}, 1) \cdot f_i'(x_1, \ldots, x_{i-1}, 0) + f_i(x_1, \ldots, x_{i-1}, 1) \cdot f_i'(x_1, \ldots, x_{i-1}, 0). \tag{4.50}$$

Since (4.49) is an identity, (4.50) is equal to

$$f_i'(x_1, \ldots, x_{i-1}, 0) \cdot f_i(x_1, \ldots, x_{i-1}, 1). \tag{4.51}$$

Adding $f_i'(x_1, \ldots, x_{i-1}, 0) \cdot f_i'(x_1, \ldots, x_{i-1}, 1)$ to $f_i(x_1, \ldots, x_{i-1}, 1)$, and applying property (2.30) yields

$$f_i'(x_1, \ldots, x_{i-1}, 0) + f_i(x_1, \ldots, x_{i-1}, 1), \tag{4.52}$$

which is equal to $f_i(x_1, \ldots, x_{i-1}, 1)$.

In view of (4.51) and (4.52), we develop the range

$$f_i'(x_1,\ldots,x_{i-1},0)\cdot f_i(x_1,\ldots,x_{i-1},1) \leq x_i \leq f_i'(x_1,\ldots,x_{i-1},0)+f_i(x_1,\ldots,x_{i-1},1), \qquad (4.53)$$

$(i=1,\ldots,n)$, which is equivalent to (4.48). Hence, $f(x_1,\ldots,x_n)=1$ is equivalent to (4.53). This completes the proof. □

Procedure 4.2 is a modification of Procedure 4.1 which incorporates the extended range concept. Example 4.2 presents the solution of the equation $f(x_1,x_2,x_3)=1$ from Example 4.1 using the extended range.

Procedure 4.2 (Subsumptive General Solution - Successive Elimination - Extended Range): Given a Boolean function $f(x_1,\ldots,x_n)$ and an ordering $ARGS$ of the X-arguments, we develop a subsumptive general solution of $f(x_1,\ldots,x_n)=1$ as follows:

Step 0. Initialise accumulators $f_{current}$ to $f(x_1,\ldots,x_n)$ and $SOLN$ to empty.

Step 1.

- If $ARGS$ is empty, then $SOLN$ contains a set of intervals such as (4.41) which represent a subsumptive general solution for $f(x_1,\ldots,x_n)=1$. $f_{current}=1$ is the consistency condition for $f(x_1,\ldots,x_n)=1$. Return $SOLN$ and $f_{current}$.
- Otherwise, continue to Step 2.

Step 2.

1. Remove the first element from $ARGS$ and call it x.
2. Form the functions $(f_{current}/x')'$ and $(f_{current}/x)$.
3. Multiply $(f_{current}/x')'$ by $(f_{current}/x)$. It is the lower bound in the interval for the argument x.
4. Add $(f_{current}/x')'$ to $f_{current}/x$. It is the upper bound in the interval for the argument x.
5. Form $EDIS(f_{current},\{x\})$. Replace $f_{current}$ with $EDIS(f_{current},\{x\})$.
6. Create a list consisting of x, $(f_{current}/x')'\cdot(f_{current}/x)$, and $(f_{current}/x')'+(f_{current}/x)$. This list represents the interval $(f_{current}/x')'\cdot(f_{current}/x) \leq x \leq (f_{current}/x')'+(f_{current}/x)$. Add this list to $SOLN$.
7. Return to Step 1.

Example 4.2: Given the function $f(x_1, x_2, x_3)$ from Example 4.1, represented by the formula

$$b'x_1x_2x_3' + a'b'x_1x_2 + abx_1'x_2 + ax_1'x_2'x_3' + b'x_1'x_2'x_3 + ab'x_2', \tag{4.54}$$

we develop a subsumptive general solution for $f(x_1, x_2, x_3) = 1$. Suppose the ordering of the X-arguments is given as x_3, x_2, x_1. The eliminants are the same as in (4.43). Using these eliminants, we form the system

$$\begin{array}{rcccl} & & 1 & \leq & a + b' \\ 0 & \leq & x_1 & \leq & a' + b' \\ a'b'x_1 & \leq & x_2 & \leq & x_1 + b \\ a'b'x_1'x_2' & \leq & x_3 & \leq & a' + bx_2 + b'x_2' + x_1x_2' + x_1'x_2 \end{array} \tag{4.55}$$

using the extended range concept. System (4.55) is a subsumptive general solution for $f(x_1, x_2, x_3) = 1$. We observe that the upper bounds are significantly higher and the lower bounds much lower in (4.55) than in (4.44).

Solutions of Switching Equations. In this section we discuss solutions of Boolean equations of the form $f(X) = 1$ where the function $f(X)$ is a switching function. Solutions of switching equations are useful for developing minimal digital circuit designs.

Truth Equations. An equation of the form

$$f(X) = 1, \tag{4.56}$$

where $f(X)$ is a switching function, is called a ***truth equation*** (Rudea 74:346). All possible particular solutions for $f(X) = 1$ are found by expressing $f(X)$ in its minterm canonical form, $MCF(f)$, and determining by inspection the substitutions $A \in \mathbf{B}_2^n$ for X which make minterms in $MCF(f)$ equal to 1.

Example 4.3: Given the equation $f(X) = 1$, where

$$f(X) = xyz + x'z' + y'z'. \tag{4.57}$$

The minterm canonical form of $f(X)$ is

$$x'y'z' + x'yz' + xy'z' + xyz. \tag{4.58}$$

By inspection, solutions of the equation $f(X) = 1$ are

$$\{(0,0,0),(0,1,0),(1,0,0),(1,1,1)\}. \tag{4.59}$$

An equation typically will have several solutions. Constant vectors, A and B, are called *equivalent* with respect to f if $f(A) = f(B)$. Two equations are called "*equivalent* if they have the same set of solutions" (Rudea 74:50).

Solutions Over a Free Boolean Algebra. Given an $n+m$ variable switching function $f : \mathbf{B}_2^{n+m} \rightarrow \mathbf{B}_2$, an equation of the form

$$f(X, Z) = 1, \tag{4.60}$$

for which $X = (x_1, \ldots, x_n)$ and $Z = (z_1, \ldots, z_m)$, may be solved by determining the unknowns $z_1, \ldots, z_m$ as a set of n-variable switching functions on the variables $x_1, \ldots, x_n$. Another way of stating this is that $f(X, Z) = 1$ is solved with respect to unknowns $z_1, \ldots, z_m$ in the free Boolean algebra $FB(x_1, \ldots, x_n)$. A solution of (4.60) then takes the form

$$Z = \Phi(X) \tag{4.61}$$

where Φ is a vector of m switching functions $\phi_1, \ldots, \phi_m : \mathbf{B}_2^n \rightarrow \mathbf{B}_2$ of the variables $x_1, \ldots, x_n$. For a solution of the form (4.61), the equation

$$f(X, \Phi(X)) = 1 \tag{4.62}$$

is an identity for every vector $X \in \mathbf{B}_2^n$. (Rudea 74:351-352)

The unknown variables, $z_1, \ldots, z_m$, in $f(X, Z) = 1$ are called *dependent* variables, because they are functions of the X-variables in a solution of the form $Z = \Phi(X)$. The X-variables are called *independent* variables.

We may develop solutions such as (4.61) using the method of successive eliminations for forming subsumptive general solutions of a Boolean equation. Using this technique, we first generate a general solution of the form:

$$\begin{array}{rcccl}
 & & 1 & \leq & f_0(X) \\
f_1'(X,0) & \leq & z_1 & \leq & f_1(X,1) \\
f_2'(X,z_1,0) & \leq & z_2 & \leq & f_2(X,z_1,1) \\
f_3'(X,z_1,z_2,0) & \leq & z_3 & \leq & f_3(X,z_1,z_2,1) \\
 & & \vdots & & \\
f_m'(X,z_1,\ldots,z_{m-1},0) & \leq & z_m & \leq & f_m(X,z_1,\ldots,z_{m-1},1).
\end{array} \tag{4.63}$$

We then form a particular solution, i.e., a solution of the form $Z = \Phi(X)$, by

1. choosing $z_1 = \phi_1(X)$ in the interval $[f_1'(X,0), f_1(X,1)]$;
2. selecting $z_2 = \phi_2(X)$ in the interval $[f_2'(X, \phi_1(X), 0), f_2(X, \phi_1(X), 1)]$;
3. and so on, until we select $z_m = \phi_m(X)$ in the interval

$$[f_m'(X, \phi_1(X), \ldots, \phi_{m-1}(X), 0), f_m(X, \phi_1(X), \ldots, \phi_{m-1}(X), 1)].$$

The particular solution $Z = \Phi(X)$ is

$$\begin{aligned} z_1 &= \phi_1(X) \\ z_2 &= \phi_2(X) \\ &\vdots \\ z_m &= \phi_m(X). \end{aligned} \tag{4.64}$$

Example 4.4 demonstrates the use of the method of successive eliminations for solving with respect to unknowns $z_1, \ldots, z_m$ in the free Boolean algebra $FB(x_1, \ldots, x_n)$.

Example 4.4: Given the equation $f(X, Z) = 1$, where

$$f(X, Z) = x_1' z_1' z_2 + x_2 z_1 z_2' + x_1 z_1 z_2', \tag{4.65}$$

we use the method of successive eliminations to solve for z_1 and z_2 in the free Boolean algebra $FB(x_1, x_2)$. First, we develop a general solution

$$\begin{aligned} 1 &\leq 1 \\ x_1 \leq z_1 &\leq x_1 + x_2 \\ x_1' z_1' \leq z_2 &\leq z_1'. \end{aligned} \tag{4.66}$$

Second, we form a function $\phi_1(X)$, $z_1 = \phi_1(X)$, by selecting a z_1 in the interval $x_1 \leq z_1 \leq x_1 + x_2$. We pick $\phi_1(X) = x_1$. Then we form a function $\phi_2(X)$, $z_2 = \phi_2(X)$, by substituting x_1 for z_1 in the interval

$$x_1' z_1' \leq z_2 \leq z_1'. \tag{4.67}$$

After the substitution for z_1 and simplification of the upper and lower bounds of (4.67), the interval

$$x_1' \leq z_2 \leq x_1' \tag{4.68}$$

is derived. Thus, $z_2 = x_1'$. Hence, we develop a solution $Z = \Phi(X)$,

$$\begin{aligned} z_1 &= \phi_1(X) \\ z_2 &= \phi_2(X), \end{aligned} \tag{4.69}$$

where

$$\begin{aligned} \phi_1(X) &= x_1 \\ \phi_2(X) &= x_1'. \end{aligned} \tag{4.70}$$

Substituting for z_1 and z_2 in $f(X, Z) = 1$, we find that $f(x_1, x_2, \phi_1(x_1, x_2), \phi_2(x_1, x_2))$ is identically equal to 1. Thus, $f(X, \Phi(X)) = 1$ is an identity for every vector $X \in \mathbf{B}_2^2$.

Constrained Equations. In some situations a solution $Z = \Phi(X)$ may not exist such that $f(X, \Phi(X)) = 1$ is an identity for every vector $X \in \mathbf{B}_2^n$. The equation $f(X, Z) = 1$ is then said to have no solution in $FB(x_1, \ldots, x_n)$. However, there may exist an n-variable switching function $g(X)$—deducible from $f(X, Z) = 1$—such that if

$$g(X) = 1 \tag{4.71}$$

is satisfied, then $Z = \Phi(X)$ is a solution for $f(X, Z) = 1$. The equation $g(X) = 1$ is called a ***constraint*** on the solution. Such a solution is called a ***constrained solution***; $f(X, Z) = 1$ is called a ***constrained equation***.

Usually, but not always, the constraint (4.71) is the consistency condition

$$EDIS(f(X,Z),Z) = 1 \tag{4.72}$$

for $f(X,Z) = 1$ (Rudea 74:358). Then,

$$g(X) = EDIS(f(X,Z),Z). \tag{4.73}$$

Example 4.5 demonstrates a problem in which the solution of an equation is subject to a constraint.

Example 4.5: Given the equation $f(X,Z) = 1$, where

$$f(X,Z) = x_1x_2' + x_1'x_2z_1z_2 + x_1z_1'z_2', \tag{4.74}$$

we use the method of successive eliminations to solve for z_1 and z_2 in the free Boolean algebra $FB(x_1,x_2)$. First, we develop a general solution

$$\begin{array}{rcccl} & & 1 & \leq & x_1 + x_2 \\ x_1' & \leq & z_1 & \leq & x_1'x_2 + x_1x_2' \\ x_1' + x_2z_1 & \leq & z_2 & \leq & x_1x_2' + x_1'x_2z_1. \end{array} \tag{4.75}$$

Unlike Example 4.4, we cannot form a function $\phi_1(X)$ because $x_1' \not\leq x_1'x_2 + x_1x_2'$ in the interval

$$x_1' \leq z_1 \leq x_1'x_2 + x_1x_2'. \tag{4.76}$$

By the definition of the inclusion relation, $x_1' \leq x_1'x_2 + x_1x_2'$ if and only if

$$x_1' \cdot (x_1'x_2 + x_1x_2')' = 0. \tag{4.77}$$

When we compute the left-hand side of (4.77) we form $x_1'x_2' = 0$; this can be stated equivalently as $x_1 + x_2 = 1$. We note that this is the consistency condition of the general solution (4.75). Hence, we can only form a solution $Z = \Phi(X)$ for $f(X, Z) = 1$ if the constraint $x_1 + x_2 = 1$ is satisfied.

We can add $x_1'x_2'$ to the right side of $x_1' \leq x_1'x_2 + x_1x_2'$; then we form the interval

$$x_1' \leq z_1 \leq x_1' + x_2'. \tag{4.78}$$

A function $\phi_1(X)$ is formed, in which $z_1 = \phi_1(X)$, by selecting a z_1 in the interval $x_1' \leq z_1 \leq x_1' + x_2'$. We may pick $\phi_1(X) = x_1'$. Then we form a function $\phi_2(X)$, $z_2 = \phi_2(X)$, by substituting x_1' for z_1 in the interval

$$x_1' + x_2z_1 \leq z_2 \leq x_1x_2' + x_1'x_2z_1. \tag{4.79}$$

After substituting for z_1, adding $x_1'x_2'$ to the right-hand side of (4.79), and simplifying the upper and lower bounds of (4.79), the interval

$$x_1' \leq z_2 \leq x_1' + x_2' \tag{4.80}$$

is formed. We select $z_2 = x_1'$. Hence, we develop a solution $Z = \Phi(X)$,

$$\begin{aligned} z_1 &= \phi_1(X) \\ z_2 &= \phi_2(X), \end{aligned} \tag{4.81}$$

where

$$\begin{aligned} \phi_1(X) &= x_1' \\ \phi_2(X) &= x_1', \end{aligned} \tag{4.82}$$

subject to the constraint $x_1'x_2' = 0$.

Substituting for z_1 and z_2 in $f(X, Z) = 1$, we find that $f(x_1, x_2, \phi_1(x_1, x_2), \phi_2(x_1, x_2)) = x_1 + x_2$. $f(x_1, x_2, \phi_1(x_1, x_2), \phi_2(x_1, x_2)) = 1$ is not an identity when $x_1 = 0$ and $x_2 = 0$, the condition specified by the constraint $x_1'x_2' = 0$. $f(X, \Phi(X)) = 1$ is an identity for every other vector $X \in \mathbf{B}_2^2$.

Modeling of Circuits with Boolean Algebra

Basic Concepts. In a digital circuit a set of binary signals is applied to nodes which are called ***input nodes*** or ***inputs***. The circuit's response to the application of the signals to the inputs is binary signals which appear on the ***output nodes*** or ***outputs***. Each of the outputs is a function of the inputs. A digital circuit is represented abstractly in Figure 4.1. The inputs are represented by the n-variable input vector X; the outputs are denoted by the m-variable output vector Z. When $m = 1$, we say that a circuit is a ***single-output*** circuit. If m is greater than one, then the circuit is called a ***multiple-output*** circuit.

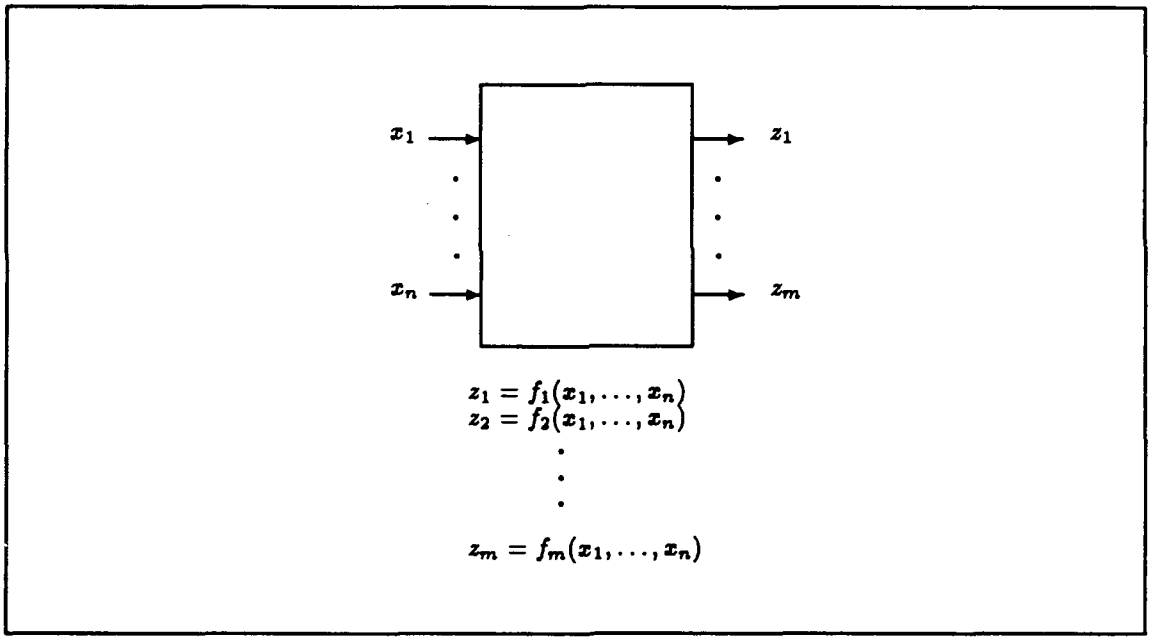


Figure 4.1. Representation of a Digital Circuit

If the value of the output signals Z at a given instant is dependent solely on the current values applied to the inputs X, then a digital circuit is called *combinational*. If a circuit has some form of memory, i.e., if the outputs depend on previous values of the inputs and/or outputs as well as the current values of the inputs, then we say that the circuit is ***sequential***. A representation of a

sequential circuit is given in Figure 4.2. A sequential circuit has two components: a combinational portion and a memory component. The memory component stores information regarding the history of the circuit.

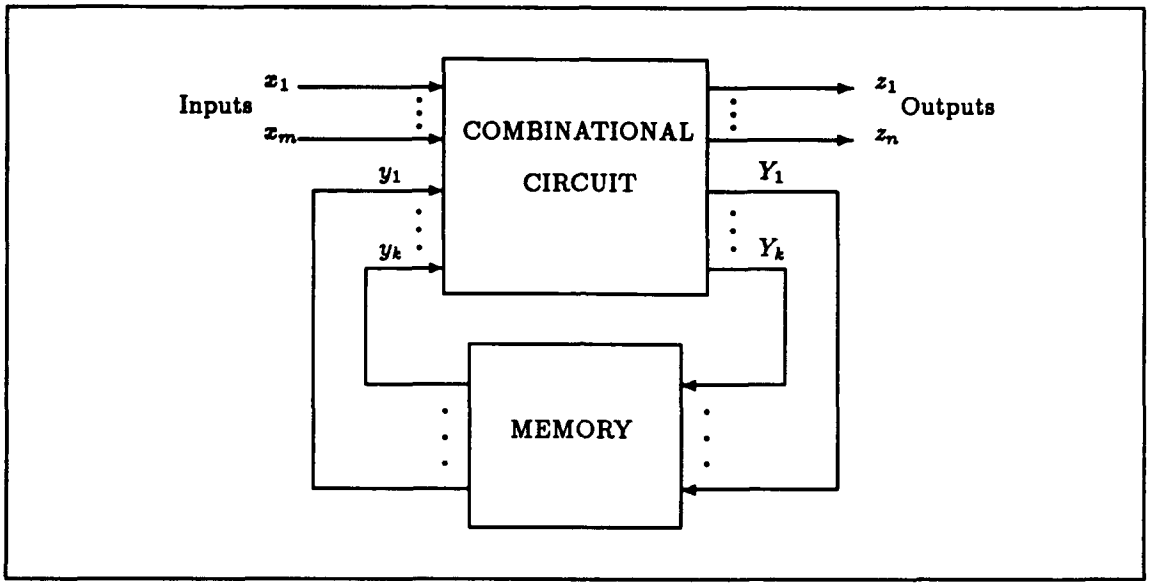


Figure 4.2. Representation of a Sequential Circuit

One of the advantages of digital circuits is that they may be described mathematically by Boolean functions of the two-element Boolean algebra, $\mathbf{B_2} = \{0, 1\}$. For historical reasons digital circuits often are called *switching circuits*. Hence, the two-element Boolean algebra is called *switching algebra*. Boolean functions in the switching algebra are called *switching functions*. Additionally, the terms *switching* and *logic* often are used interchangeably. The nodes of a circuit are depicted by Boolean variables; the gates of a circuit are modeled by Boolean operators. Every Boolean formula which represents a switching function has a corresponding switching-circuit implementation and vice versa, i.e., there is a correspondence between a formula and a circuit. A conjunction corresponds to an AND gate; a disjunction corresponds to an OR gate; and a complement is implemented by an inverter. The output value of a switching circuit for a particular

input combination is the same as the value of the corresponding switching function given the same assignment of values to its variables.

An example of a sum-of-products formula which represents a three-variable switching function, $f : \mathbf{B}_2^3 \rightarrow \mathbf{B}_2$, is

$$x'z + x'yz' + xy' + xyz. \tag{4.83}$$

The corresponding circuit for this formula is given in Figure 4.3. Each term of the formula is implemented by an AND gate; the disjunction of the terms of the formula is implemented by an OR gate. Because only two gates must be traversed between the circuit inputs and the circuit output, this circuit it is called a *two-level* or *two-stage* logic circuit; specifically, it is an AND-OR circuit. The AND gates form the first level; the OR gate forms the second level. The inverters are not said to form a level, because often the input signals and their complements are both available, eliminating the need for inverters. Other two-level logic circuits are NAND-NAND and NOR-NOR circuits. The number of levels of a circuit is defined as the maximum number of gates that must be traversed between the circuit inputs and circuit outputs, less inverters required to complement the input signals. In general, any circuit which has more than two levels is called a *multi-level* or *multi-stage* circuit.

Often when designing a circuit it is necessary to list the output values of the circuit for given combinations of input values; we may use a truth table for this purpose. Switching circuits and their corresponding switching functions have the same truth table. A truth table for the circuit of Figure 4.3 is shown in Table 4.1.

A switching circuit may be implemented by different combinations of components and still behave the same. Likewise, a given switching function can be represented by a variety of formulas. In either case, the number of realizations is infinite. Different formulas which represent the same

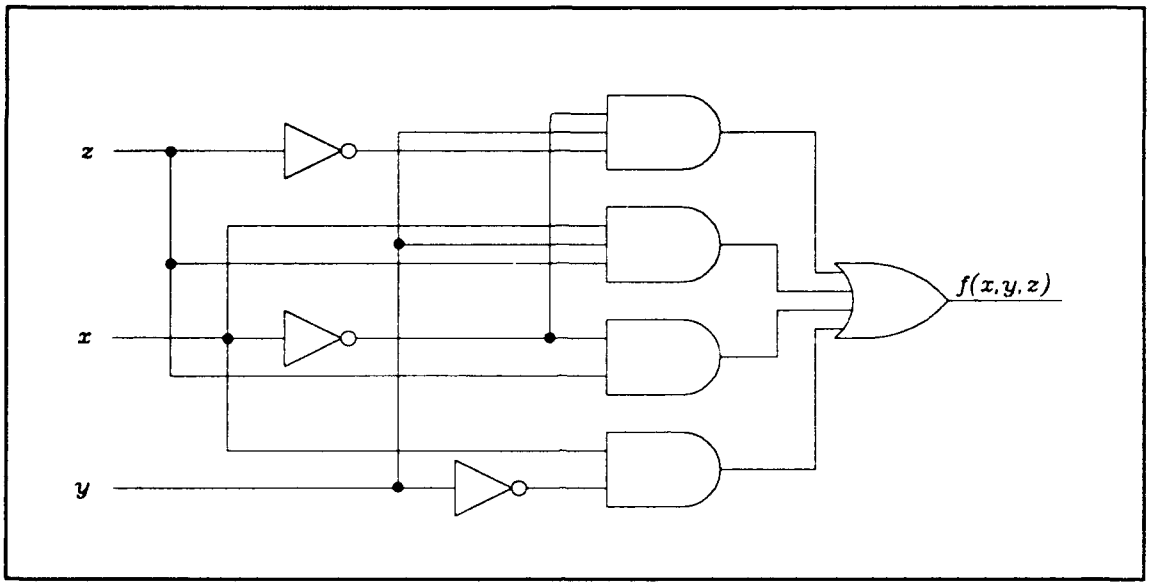


Figure 4.3. Circuit Implementation of $x'z + x'yz' + xy' + xyz$

x y z	$f(x,y,z)$
0 0 0	0
0 0 1	1
0 1 0	1
0 1 1	1
1 0 0	1
1 0 1	1
1 1 0	0
1 1 1	1

Table 4.1. Truth Table for $x'z + x'yz' + xy' + xyz$

function are called *equivalent formulas;* different switching circuits which realize the same function are called *equivalent circuits*. The primary goal when minimizing a circuit is to find a simplest formula—with respect to some criterion—which represents a given function. In some instances, we are given a formula which represents a function and then must use a minimization technique to produce a simpler equivalent formula. A simpler formula corresponds to a simpler switching circuit; however, the function remains the same. For example, a simple formula equivalent to (4.83) is

$$xy' + z + x'y. \tag{4.84}$$

A corresponding simple circuit, equivalent to that of Figure 4.3, is shown in Figure 4.4. The simplified circuit requires two fewer AND gates and one less inverter than the original circuit; additionally, a three-input OR gate is required versus a four-input gate. Hence, there is a substantial decrease in required hardware.

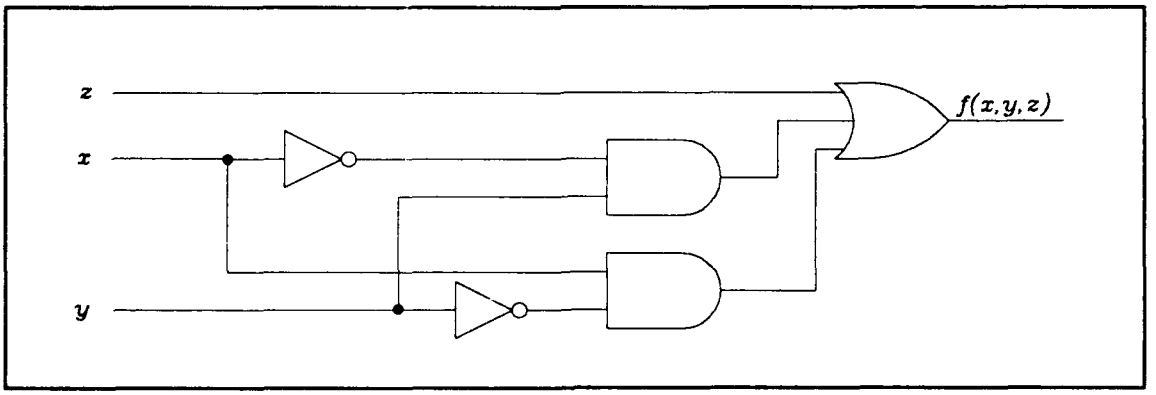


Figure 4.4. Circuit Implementation of $x'y + xy' + z$

Designs and Specifications.

Relationship Between Designs and Specifications. An implementation of a digital circuit is called a *design*. A design has a correspondence with a Boolean formula. During logic

synthesis a design is created to meet a specification. A *specification* is a desired relationship—a mapping—between input signals and output signals, i.e., it states how we would like the circuit to respond to input stimuli. Hence, a specification usually correlates to a switching function. A design corresponds to a formula which necessarily represents a completely-specified switching function. On the other hand, a specification is more general than a design. A specification may correspond to a switching function, although more typically it may be stated by an interval of switching functions.[2] A design which meets a specification stated by an interval of functions will correspond to a formula which represents a function in the range of possible functions given by the specification.

A design which is the cheapest possible circuit with respect to given cost criteria is called a *minimal* design. The goal of logic minimization is to find a minimal design among the infinite number of possible designs to meet a specification. The requirement to find a minimal design to satisfy a specification is called the *minimization problem;* more generally, we call this problem the *design problem.* The process of finding a minimal design is called *minimization.*

Since there is a correspondence between a design and a Boolean formula, an approach to minimisation is to use Boolean techniques to devise a minimal formula to represent a function. If we can find a minimal formula, we necessarily develop an economical circuit. If a specification is given by a function, then the object of minimization is simply to find a minimal formula to represent that function. For example, if an SOP formula is one which consists of the fewest possible terms of all the SOP formulas that could represent a function, then the corresponding two-level AND-OR design would contain the fewest possible AND gates. If a specification is given by an interval of functions, then the minimization problem is to find a least-cost formula with respect to given cost criteria among all the possible formulas which may represent the functions in the interval; the formula represents one of the completely-specified functions in the given range of functions.

[2] An interval which specifies an output is usually called incompletely-specified Boolean function (defined in Chapter 2).

In summary, given a circuit specification the goal of the design problem is to find a design that:

- meets the specification;
- is realizable; and
- is economical

A specification is stated by a Boolean function (interval) and a design is given by a Boolean formula. A design expressed by a Boolean formula meets a specification if it represents the corresponding function (represents a function in the interval). A design represented by a formula is realizable because every Boolean formula has a corresponding circuit and vice versa. Additionally, since techniques are available to develop economical Boolean formulas, forming an economical formula yields a corresponding circuit that also is economical. We now discuss criteria used to identify economical digital circuit designs.

Minimization Criteria. When devising a design to meet a specification, we would like to build the best circuit possible. However, to measure what we consider a "good" circuit, we must have a means of distinguishing a superior implementation from a poor one. Two criteria which have been devised to determine the goodness of a circuit are *cost* and *performance*. The cost of a circuit refers to the number of components that it takes to implement the circuit and/or the size of the circuit. We desire that the cost of a circuit be as low as possible. The performance of a circuit is the response-time, i.e., the time required for a result to appear on the outputs given an input stimulus. The response-time of a circuit is also called the *delay* of the circuit. A circuit with a short delay is typically preferred over one having a longer delay.

The delay of a circuit is dependent on the number of circuit components a signal must traverse in traveling from an input node to an output node. In general, the delay reflects the longest path, with respect to the number of gates, of all of the possible paths between one of the inputs and an

output dependent on that input. Hence, one way to measure delay during the design process is to count the maximum number of levels between the inputs and the outputs.

There exist various measures to judge the cost of a circuit. The most common metrics are:

- the number of gates,
- the number of gate-inputs, and
- the number of interconnections.

When designing using discrete components, the number of gates and gate-inputs are a direct reflection of the number of components required to implement a circuit. If a circuit is implemented using VLSI, then the gates, gate-inputs, and the number of interconnections reflect the area required to implement the circuit. In VLSI design we are primarily concerned with the area required to implement a circuit. The number of gate-inputs indirectly reflects the number of interconnections. If a given node is connected to several gates, then there is an interconnection among these gates. Hence, if the number of gate-inputs is large, then the number of interconnections also is likely to be substantial.

We calculate the cost of a circuit by examining the number of gates, gate-inputs, and interconnections of the circuit. Many of these measures can be determined from the formula which corresponds to the circuit. In particular, if the number of gates and/or gate-inputs in a two-level circuit is our primary concern, then we may determine the cost of the circuit from the corresponding sum-of-products formula. For example, the number of gates in a two-level circuit corresponds directly to the number of terms in the SOP formula; the number of literals in an SOP formula represents the number of gate-inputs in the corresponding circuit. Hence, a strategy to form an minimal two-level circuit is to devise an SOP formula which consists of the fewest possible terms and for each of the terms to have the fewest possible literals.

To develop an economical design for a two-level, single-output digital circuit, we endeavor to find an SOP formula which meets our primary cost criteria. If the number of gates in the two-

level AND-OR circuit implementation is our biggest concern, then we would attempt to develop a formula which contains the fewest terms. For example, if a formula consists of five terms, then the cost of the circuit is six gates (five AND gates plus one OR gate). If the total number of gate-inputs is the primary design consideration, then the approach would be to find a formula which contains the fewest literals. In this approach, the cost of the formula is calculated by:

- adding one to the sum for every term consisting of a single literal, and
- adding, for terms which contain more than one literal, the number of literals in each, plus one, to the sum.

Calculating the cost in this manner yields the total number of gate-inputs for each of the AND gates as well as for the OR gate in the corresponding two-level AND-OR circuit. For example, the formula

$$x' + y'z' + yz \tag{4.85}$$

represents a circuit which costs seven gate-inputs. The terms $y'z'$ and yz each cost three; the term x' costs one.

We can combine the preceding measures to find an SOP formula which consists of the fewest terms as the primary consideration and the fewest literals as a secondary concern. When evaluating the cost of a formula, the cost of each term is calculated as follows:

- a term consisting of a single literal has a cost of k, where k is a large constant, and
- the cost of all other terms is calculated by counting the number of literals in the term and adding k to the result.

The costs of terms contained in the formula are then summed to develop the cost of the formula. When calculating the cost of a formula in this manner, care must be taken to ensure that k is

properly scaled; k must be at least one order of magnitude greater than the number of terms in the resulting SOP formula. If $k = 100$, then the cost of formula (4.85) is 304. The terms $y'z'$ and yz each cost 102; the term x' costs 100. In examining the cost of formula (4.85), the number of hundreds corresponds to the number of terms. The remaining portion of the score, i.e., 4, represents the number of gate-inputs to the AND gates of the circuit.

There often is a trade-off between the cost of a circuit implementation and the delay of the circuit. Faster circuits generally require more components than a least-cost design. Conversely, the delay of a least-cost design is usually longer than that of the fastest possible implementation. When developing a design which meets a specification, we attempt to construct the cheapest circuit which meets constraints on circuit delay.

Specification Formats. At the outset of the design process, we must have a circuit specification from which we derive the design. In this section, specification formats for digital circuits are discussed.

Each of the outputs in a circuit may be treated as a separate function for which an output signal is generated for a given input combination. We will refer to the inputs by the vector $X = (x_1, \ldots, x_n)$; the output signals are represented by the vector $Z = (z_1, \ldots, z_m)$.[3] Typically, a design corresponds to a set of Boolean formulas consisting of X-variables, each of which represents a switching function. On the other hand, a specification for each of the outputs may be either a single switching function or an interval of functions. We normally say that the set of switching functions corresponding to each of the outputs forms a single specification for the circuit. We may state a specification in a number of ways. Formats include:

- Boolean formulas,
- truth tables, and

[3] For single-output circuits, i.e., when $m = 1$, we will refer to the output node simply by the symbol z.

- 1-normal forms.

In the following sections we will discuss each of these specification formats. Unless otherwise noted, the discourse is limited to combinational circuits.

Boolean Formulas. One form of circuit specification is a set of Boolean formulas which represents a set of m switching functions. The outputs z_j are related to associated functions by the equation

$$z_j = f_j(X), \qquad j = 1, \ldots, m. \tag{4.86}$$

We may state (4.86) more compactly as

$$Z = \underline{f}(X). \tag{4.87}$$

The minimisation problem entails finding a formula F_j to represent each of the associated functions f_j such that F_j is equivalent to the formula given as the specification for f_j and the set $\underline{F} = \{F_1, \ldots, F_j, \ldots, F_m\}$ of formulas representing the functions $\underline{f}(X)$ is minimal with respect to a given cost criterion.

In some instances, each of the output functions z_j may be specified by formulas representing lower and upper-bound functions of an interval. Each output z_j is related to an upper and lower bound by the statement

$$g_j(X) \leq z_j \leq h_j(X), \qquad j = 1, \ldots, m. \tag{4.88}$$

Given the formulas which represent the functions in the interval, we must find a set of minimal formulas from among all of the possible formulas which represent functions in the set of intervals

$[g_j(X), h_j(X)]$, $j = 1, \ldots, m$. Each of the resulting formulas represents a function belonging to the associated interval.

In two-level multiple-output circuits, a term may appear in SOP formulas representing functions corresponding to different output nodes. A term which appears in more than one formula is called a *shared* term, because the output of the AND gate which corresponds to the term is shared by each of the OR gates which combines terms for the formulas in which the shared term appears. It is advantageous that there exist many shared terms in a two-level design.

Truth Tables. The most common form of circuit specification is the truth table. A truth table specifies the binary signals that should appear on each output of a circuit given a stimulus to the input nodes, i.e., the mapping between possible input signals and the required output signals. A truth table is of great utility because it may be used to represent the specification of an output by a single function as well as an interval of functions. Moreover, a single truth table specifies all of the circuit outputs.

When each of the outputs z_j is specified by a single function f_j, a truth table states the mapping $\mathbf{B}_2^n \rightarrow \mathbf{B}^m$ for the 2^n possible input combinations. Table 4.2 depicts the two 3-variable switching functions: $z_1 = f_1(X)$ and $z_2 = f_2(X)$.

x_1	x_2	x_3	z_1	z_2
0	0	0	0	1
0	0	1	1	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	0
1	0	1	1	1
1	1	0	0	1
1	1	1	1	1

Table 4.2. Truth Table for a Multiple-Output Circuit

For an output z_j specified by an interval $[g_j, h_j]$, a truth table depicts the mapping of input combinations $A \in \mathbf{B}_2^n$ to 0, 1, or X. If an input $A \in \mathbf{B}_2^n$ is mapped to either 0 or 1 for an output, then the corresponding output must assume the value of 0 or 1, respectively, for the given input combination. If an input combination $A \in \mathbf{B}_2^n$ is mapped to the don't-care value—depicted as X—for a given output function, then we do not care whether the design associated with the respective output produces a 0 or 1 for that input stimulus. For an output specified in this manner, the corresponding lower-bound function $g_j(X)$ maps to 1 for input combinations $A \in \mathbf{B}_2^n$ which map to 1 in the truth table, and 0 otherwise. The corresponding upper-bound function $h_j(X)$ maps to 1 for input combinations $A \in \mathbf{B}_2^n$ which map to 1 or X in the truth table, and 0 otherwise. For a given input combination $A \in \mathbf{B}_2^n$, it is possible that one output may map to 0 or 1 while another output may map to X.

There exists another situation in circuit design that is referred to as a don't care condition (Barte 61). In some cases, it may be known *a priori* that a specific input combination $A \in \mathbf{B}_2^n$ will never occur on the input nodes. This condition is denoted in a truth table by a missing row, i.e., the row associated with the input combination $A \in \mathbf{B}_2^n$ is not listed. However, this information is useful in devising a minimal design to meet a specification. Table 4.3 is a truth table specifying a multiple-output circuit which contains both forms of don't care condition. The assignments $X = (0, 1, 0)$ and $X = (1, 1, 0)$ correspond to missing rows—input combinations which will not occur.

x_1	x_2	x_3	z_1	z_2
0	0	0	0	1
0	0	1	1	X
0	1	1	0	0
1	0	0	X	X
1	0	1	1	0
1	1	1	X	0

Table 4.3. Truth Table With Don't Cares Specifying a Multiple-Output Circuit

A design which meets a specification as given by Table 4.3 will correspond to a set of formulas representing a set of completely-specified functions which have every input combination $A \in \mathbf{B}_2^n$ mapped to 0 or 1. Table 4.4 corresponds to a set of functions which meet the specification of Table 4.3.

x_1	x_2	x_3	$f_1(X)$	$f_2(X)$
0	0	0	0	1
0	0	1	1	1
0	1	0	1	1
0	1	1	0	0
1	0	0	1	0
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0

Table 4.4. Truth Table Which Meets Specification of Table 4.3

1-Normal Form. A way of specifying a circuit which has received little attention by switching theorists is the 1-*normal form*. Using a 1-normal form, the circuit specification is given in the form of the single equation

$$\phi(X, Z) = 1. \tag{4.89}$$

The 1-normal form has several advantages relative to other specification formats:

- It provides a standard representation on which to base analysis and synthesis.
- The function ϕ corresponding to a given specification is unique.
- The function ϕ is directly related to the truth-table representation of a specification.
- The normal form provides a uniform way to represent and handle don't-care conditions. (Brown 90:215)

When specifying a circuit using a 1-normal form, the output signals Z are explicit in the function $\phi(X, Z)$. However, except for knowing which variables are associated with input nodes

and which variables correspond to outputs, we cannot differentiate the output variables from the input variables in $\phi(X, Z)$. Since we can control the signal applied to the inputs, we call the input variables in $\phi(X, Z)$ *independent* variables. Since the output signals are dependent on the stimulus applied to the input signals, the output variables are called *dependent* variables.

Other specification formats may be converted to and generated from a 1-normal form. A specification given by a set of functions (4.86) is converted to 1-normal form (4.89) using reduction:

$$\phi(X, Z) = \prod_{j=1}^{m} (z_j \odot f_j(X)). \tag{4.90}$$

Similarly, a set of intervals which specify a circuit can be converted to an equivalent 1-normal form. Given

$$g_j(X) \leq z_j \leq h_j(X), \quad j = 1, \ldots, m, \tag{4.91}$$

the definition of the inclusion relation, and (2.40), the equation

$$z_j' g(X) + z_j h'(X) = 0 \tag{4.92}$$

is formed. We complement both sides of (4.92) to form

$$z_j' g'(X) + z_j h(X) = 1. \tag{4.93}$$

The left-hand side of (4.93) is equal to $\phi_j(X, z_j)$. We form $\phi(X, Z)$ by (2.41):

$$\phi(X, Z) = \prod_{j=1}^{m} \phi_j(X, z_j). \tag{4.94}$$

Given a 1-normal form, the set of functions (intervals) which specify the outputs of the circuit may be extracted. Let Z_j be the set of variables associated with the output nodes less the output z_j. We use elimination to develop a function which involves only the input variables and the output z_j. Eliminating Z_j from $\phi(X, Z) = 1$ yields the resultant of elimination $EDIS(\phi(X, Z), Z_j) = 1$. The function $EDIS(\phi(X, Z), Z_j)$ involves only the input variables and the output z_j. Let us define $\tilde{\phi}_j(X, z_j) = EDIS(\phi(X, Z), Z_j)$. Then

$$\tilde{\phi}_j(X, z_j) = 1. \tag{4.95}$$

If we view $\tilde{\phi}_j(X, z_j)$ as a function consisting of the single variable z_j, then the equation $\tilde{\phi}_j(X, z_j) = 1$ may be solved for z_j using Lemma 4.1. Consequently, we develop the the interval

$$\tilde{\phi}'_j(X, 0) \leq z_j \leq \tilde{\phi}_j(X, 1). \tag{4.96}$$

If the original specification for the outputs was a set of functions, then

$$\tilde{\phi}'_j(X, 0) = \tilde{\phi}_j(X, 1) \tag{4.97}$$

and

$$f_j(X) = \tilde{\phi}_j(X, 1) \tag{4.98}$$

are valid. If the original specification for the outputs was a set of intervals, then

$$g_j(X) = \tilde{\phi}'_j(X, 0) \tag{4.99}$$

and

$$h_j(X) = \tilde{\phi}_j(X, 1). \tag{4.100}$$

Example 4.6 demonstrates the conversion of a set of intervals to 1-normal form and the extraction of the set of intervals from the 1-normal form.

Example 4.6 Suppose a circuit specification is given by a set of intervals

$$\begin{array}{lll} x_1x_2 \leq & z_1 \leq & x_1 + x_2 \\ x_1x_2 \leq & z_2 \leq & x_1' + x_2. \end{array} \tag{4.101}$$

Using (4.93), we develop the functions

$$\begin{array}{lcl} \phi_1(X, z_1) & = & x_1'z_1' + x_1'x_2 + x_1x_2' + x_1z_1 \\ \phi_2(X, z_2) & = & x_2'z_2' + x_1' + x_2z_2. \end{array} \tag{4.102}$$

We then form the function $\phi(X, Z)$:

$$\phi(X, Z) = \phi_1(X, z_1) \cdot \phi_2(X, z_2) = x_1x_2'z_2' + x_2z_1z_2 + x_1'x_2 + x_1'z_1'. \tag{4.103}$$

The equation $\phi(X, Z) = 1$ is the circuit specification in 1-normal form.

Since $\phi(X, Z) = 1$, eliminating z_2 yields $EDIS(\phi(X, Z), z_2) = 1$, for which

$$EDIS(\phi(X, Z), \{z_2\}) = x_1x_2' + x_2z_1 + x_1'x_2 + x_1'z_1'. \tag{4.104}$$

Similarly, we derive

$$EDIS(\phi(X, Z), \{z_1\}) = x_2'z_2' + x_2z_2 + x_1'. \tag{4.105}$$

From (4.104) and (4.105), we form the functions

$$\begin{aligned} \tilde{\phi}_1(X, z_1) &= x_1 x_2' + x_2 z_1 + x_1' x_2 + x_1' z_1' \\ \tilde{\phi}_2(X, z_2) &= x_2' z_2' + x_2 z_2 + x_1'. \end{aligned} \tag{4.106}$$

Using (4.96) we form the intervals $\tilde{\phi}_j'(X, 0) \leq z_j \leq \tilde{\phi}_j(X, 1)$:

$$\begin{aligned} x_1 x_2 \leq \quad & z_1 \leq \quad x_1 + x_2 \\ x_1 x_2 \leq \quad & z_2 \leq \quad x_1' + x_2. \end{aligned} \tag{4.107}$$

It is very easy to convert information given in a truth table to 1-normal form and vice versa. Expanding $\phi(X, Z)$ to its minterm canonical form with respect to the X-arguments, we develop a formula in which each term corresponds to a line in a truth table. For each term, the X-arguments correspond to a given input combination $A \in \mathbf{B}_2^n$ and the discriminant represents the value of each of the outputs z_j for that input combination. Each discriminant of $MCF(\phi(X, Z))$ with respect to the X-arguments is either a term consisting of Z-arguments (which we shall call a Z-term) or 0.[4] A discriminant corresponds to an entry in a truth table in the following manner:

- If a discriminant is a Z-term which contains the literal z_j', then $z_j = 0$ for the input combination $A \in \mathbf{B}_2^n$.
- If a discriminant is a Z-term which contains the literal z_j, then $z_j = 1$ for the input combination $A \in \mathbf{B}_2^n$.
- If a discriminant is a Z-term which does not contain the variable z_j, then $z_j = X$ for the input combination $A \in \mathbf{B}_2^n$.
- If a discriminant is 1, then $z_j = X$ for the input combination $A \in \mathbf{B}_2^n$ for each z_j.
- If a discriminant is 0, then the input combination $A \in \mathbf{B}_2^n$ does not appear in the truth table.

Example 4.7 demonstrates the relationship between a truth table specification and the 1-normal form.

[4] In the general case, the discriminant can be an SOP formula in which each term consists of Z-arguments. We assume for the moment that each discriminant is a Z-term.

Example 4.7: Suppose a specification is given by a truth table as stated by Table 4.5. Then, the 1-normal form which represents the same specification is given by $\phi(X, Z) = 1$, where $MCF(\phi(X, Z))$ is the formula

$$(x_1'x_2'x_3')\cdot z_1'z_2 + (x_1'x_2'x_3)\cdot z_1 + (x_1'x_2x_3)\cdot z_1'z_2' + (x_1x_2'x_3')\cdot 1 + (x_1x_2'x_3)\cdot z_1z_2 + (x_1x_2x_3)\cdot z_2'. \quad (4.108)$$

x_1	x_2	x_3	z_1	z_2
0	0	0	0	1
0	0	1	1	X
0	1	1	0	0
1	0	0	X	X
1	0	1	1	1
1	1	1	X	0

Table 4.5. Truth Table for Example 4.7

An important concept that is easily conveyed using the 1-normal form is the relationship between a design and a specification. Using the 1-normal form we formalize this relationship. A design meets a specification if it implies the specification. A design corresponds to a set $\underline{f}(X)$ of switching functions. Reducing the set of equations $Z = \underline{f}(X)$ to an equivalent 1-normal form, we form $\phi_D(X, Z) = 1$, where

$$\phi_D(X, Z) = \prod_{j=1}^{m} (z_j \odot f_j(X)). \quad (4.109)$$

If the 1-normal form of the specification is $\phi_S(X, Z) = 1$, then a design meets a specification if

$$\phi_D(X, Z) = 1 \Rightarrow \phi_S(X, Z) = 1. \quad (4.110)$$

By the Extended Verification Theorem, statement (4.110) is equivalent to the inclusion

$$\phi_D(X, Z) \leq \phi_S(X, Z) \tag{4.111}$$

given that $\phi_D(X, Z) = 1$ is consistent. The equation $\phi_D(X, Z) = 1$ is consistent since it is the 1-normal form which correspoonds to a design. Example 4.8 demonstrates this concept for a single-output function.

Example 4.8: Suppose we are given a specification stated by the truth table of Table 4.6 and a design which meets the specification which corresponds to the function of Table 4.7.

x_1	x_2	z
0	0	0
0	1	0
1	0	X
1	1	1

Table 4.6. Truth Table for Example 4.8 Specification

x_1	x_2	z
0	0	0
0	1	0
1	0	1
1	1	1

Table 4.7. Truth Table for Example 4.8 Design

The specification is equivalently stated by the range

$$x_1 x_2 \leq z \leq x_1. \tag{4.112}$$

The equivalent 1-normal form for (4.112) is $\phi_S(X, z) = 1$, where

$$\phi_S(X, z) = x_1 z + x_1' z' + x_2' z'. \tag{4.113}$$

The design is equivalently stated by $z = x_1$. Given this design, the function $\phi_D(X, z)$ is

$$\phi_D(X, z) = x_1 z + x_1' z'. \tag{4.114}$$

It is readily apparent that $\phi_D \leq \phi_S$; hence, the design implies the specification.

The don't-care condition associated with an input condition which will not occur imposes a constraint on the possible input combinations. We model this constraint by forming an equation in which the relevant input condition is set equal to 0. For example, if an input condition such as $x_1 = 1$ and $x_2 = 0$ never occurs in a two-input circuit, then the constraint equation

$$x_1 x_2' = 0 \tag{4.115}$$

would be formed. By (2.40), a single equation may be formed which represents all possible input combinations that will not occur. We form the equation

$$\phi_{DC}(X) = 0 \tag{4.116}$$

where the left-hand side of (4.116) is the sum of the constraints representing input combinations which do not occur. When testing to determine whether a design implies a specification, we must consider the constraints on the inputs prior to determining the validity of the statement $\phi_D \leq \phi_S$. One way to do this is to form the equation $\phi_{DC}'(X) = 1$ and then to combine by (2.41) the equations $\phi_{DC}'(X) = 1$ and $\phi_D(X, Z) = 1$. This enforces the constraints on the inputs of the design prior to

determining whether it implies the specification. To determine if a design, subject to constraints on the inputs, implies a specification, we test the validity of the statement

$$\phi_D(X,Z)\cdot\phi'_{DC}(X)\leq\phi_S(X,Z). \tag{4.117}$$

In example 4.9, we demonstrate the necessity of multiplying $\phi_D(X,Z)=1$ by $\phi'_{DC}(X)$ prior to testing whether a design meets a specification.

Example 4.9: Suppose we are given a specification such as the truth table in Table 4.8. In this table, a missing input combination forces the constraint $x'_1x_2=0$. Hence, $\phi_{DC}(X)=x'_1x_2$. The equivalent 1-normal form for the truth table is given by

$$\phi_S(X,z)=x'_1x'_2z'+x_1x'_2+x_1x_2z. \tag{4.118}$$

A truth table for $\phi_S(X,z)$ is given by Table 4.9.

x_1	x_2	z
0	0	0
1	0	X
1	1	1

Table 4.8. Specification for Example 4.9

x_1	x_2	z	$\phi_S(x_1,x_2,z)$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Table 4.9. Truth Table for $\phi_S(X,z)$

Using the right-hand side of equation (4.118), we may solve for z to determine possible ways the output z may be a function of the inputs x_1 and x_2. The interval

$$x_2 \leq z \leq x_1 \tag{4.119}$$

is formed subject to the consistency condition that $x_1' x_2 = 0$. We note that the consistency condition is the constraint forced by the missing input combination in the truth table. In view of Theorem 4.2, we use this condition to form an extended interval for z:

$$x_1 \cdot x_2 \leq z \leq x_1 + x_2. \tag{4.120}$$

Given (4.120) a suitable design is represented by the equation $z = x_1$. Reducing the design to 1-normal form, we develop the equation $\phi_D(X, z) = 1$, where

$$\phi_D(X, z) = x_1' z' + x_1 z. \tag{4.121}$$

A truth table for $\phi_D(X, z)$ is given by Table 4.10.

x_1	x_2	z	$\phi_D(x_1, x_2, z)$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Table 4.10. Truth Table for $\phi_D(X, z)$

Comparing truth tables for $\phi_D(X, z)$ and $\phi_S(X, z)$, we find that $\phi_D(X, z) \not\leq \phi_S(X, z)$. However, the constraint $\phi_{DC}(X) = 0$ was not imposed on the design. We form the equation $\phi_{DC}'(X) = 1$ and

combine it with $\phi_D(X, z) = 1$ to form a single equation. Since $\phi_{DC}(X) = x_1' x_2$, $\phi'_{DC}(X) = x_1 + x_2'$. We then form

$$\phi_D(X, z) \cdot \phi'_{DC}(X) = (x_1' z' + x_1 z) \cdot (x_1 + x_2') = x_1 z + x_1' x_2' z'. \tag{4.122}$$

The truth table for $\phi_D(X, z) \cdot \phi'_{DC}(X)$ is given by Table 4.11. By examining the truth tables, it is clear that $\phi_D(X, z) \cdot \phi'_{DC}(X) \leq \phi_S(X, z)$. Hence, the design—subject to the constraint on the inputs—meets the specification.

x_1	x_2	z	$\phi_D(X, z) \cdot \phi'_{DC}(X)$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Table 4.11. Truth Table for $\phi_D(X, z) \cdot \phi'_{DC}(X)$

Relationship of Equation-Solving and Minimization

Tabular Specifications. Given a circuit specification $\phi(X, Z) = 1$, a design is developed by finding a system

$$Z = \underline{f}(X) \tag{4.123}$$

such that $\phi(X, \underline{f}(X)) = 1$ is an identity. If such a system exists, then the specification is said to be ***consistent.*** We observe that (4.123) is a solution of $\phi(X, Z) = 1$ with respect to the unknowns $z_1, \ldots, z_m$ in the free Boolean algebra $FB(x_1, \ldots, x_n)$. Hence, finding a design which meets a specification entails finding a solution of the specification $\phi(X, Z) = 1$.

One way to approach the design problem is to use the method of successive eliminations to form a subsumptive general solution of the equation $\phi(X, Z) = 1$. We develop the general solution

$$\begin{array}{rcccl}
 & & 1 & \leq & \phi_0(X) \\
\phi_1'(X, 0) & \leq & z_1 & \leq & \phi_1(X, 1) \\
\phi_2'(X, z_1, 0) & \leq & z_2 & \leq & \phi_2(X, z_1, 1) \\
\phi_3'(X, z_1, z_2, 0) & \leq & z_3 & \leq & \phi_3(X, z_1, z_2, 1) \\
 & & \vdots & & \\
\phi_m'(X, z_1, \ldots, z_{m-1}, 0) & \leq & z_m & \leq & \phi_m(X, z_1, \ldots, z_{m-1}, 1)
\end{array} \tag{4.124}$$

from which we generate particular solutions in the form of (4.123). Because each successive subsumptive is dependent on previously-defined Z-variables, we call the solution (4.124) a ***recurrent system.*** System (4.124) is the most general way of portraying a general solution, since a general solution of this form may be developed given any 1-normal form $\phi(X, Z) = 1$. In many situations, however, we desire a general solution which is a ***non-recurrent system,*** i.e., one of the form

$$\begin{array}{rcccl} \alpha_1(X) & \leq & z_1 & \leq & \beta_1(X) \\ \alpha_2(X) & \leq & z_2 & \leq & \beta_2(X) \\ \alpha_3(X) & \leq & z_3 & \leq & \beta_3(X) \\ & & \vdots & & \\ \alpha_m(X) & \leq & z_m & \leq & \beta_m(X). \end{array} \tag{4.125}$$

If we can form a general solution (4.125) of $\phi(X, Z) = 1$ in which each z_j is stated as an independent subsumption, then we say that the specification $\phi(X, Z) = 1$ is *tabular*. (Brown 90:219)

It has been shown that a specification is tabular if and only if it can be expressed by a truth table. This is apparent in the following theorem:

> *A specification equivalent to $\phi(X, Z) = 1$ is tabular if and only if, for each $A \in \{0, 1\}^n$, the discriminant $\phi(A, Z)$ is either zero or reduces to a term on the Z-variables.* (Brown 90:220)

Our earlier discussion regarding the relationship of the 1-normal form and truth tables for circuit specifications was restricted to tabular specifications, i.e., ones for which each discriminant of $MCF(\phi(X, Z))$ with respect to the X-variables is either 0 or a term on the Z-variables. If a specification is *non-tabular*, there exists a discriminant $\phi(A, Z)$ which is an SOP formula of two or more terms on the Z-variables which cannot be represented by an equivalent formula consisting of a single term. Hence, for non-tabular specifications there exists no way to represent $\phi(X, Z) = 1$ with a conventional truth table. Conventional minimization techniques cannot be used to develop designs for circuits with non-tabular specifications.

In (4.91) through (4.94) we developed an approach for reducing a system such as (4.125) to an equivalent 1-normal form $\phi(X, Z) = 1$. The function $\phi(X, Z)$ is formed in the following manner:

$$\phi(X, Z) = \phi_1(X, z_1)\phi_2(X, z_2) \cdots \phi_m(X, z_m). \tag{4.126}$$

The functions on the right-hand side of (4.126) are not unique; however, the set of intervals (4.125) derived from the functions is unique (Brown 90:221). Statement (4.95) gives a process for forming the functions on the right-hand side of (4.126), where $\tilde{\phi}_j(X, z_j) \mapsto \phi_j(X, z_j)$. Then

$$\alpha_j(X) = \phi_j'(X, 0) \tag{4.127}$$

and

$$\beta_j(X) = \phi_j(X, 1). \tag{4.128}$$

Solution $\Leftrightarrow$ Design. In the last section we observed that from a circuit specification $\phi(X, Z) = 1$ a design is developed by finding a system

$$Z = \underline{f}(X) \tag{4.129}$$

such that $\phi(X, \underline{f}(X)) = 1$ is an identity. The design corresponds to the set $\underline{F}$ of formulas representing the functions $\underline{f}(X)$. The system $Z = \underline{f}(X)$ is a solution of $\phi(X, Z) = 1$ with respect to the unknowns $z_1, \ldots, z_m$ in the free Boolean algebra $FB(x_1, \ldots, x_n)$. Hence, finding a design which meets a specification entails finding a particular solution of the specification $\phi(X, Z) = 1$.

We display pictorially in Figure 4.5 the relationships among designs, solutions to equations, specifications, and equations. In Figure 4.5 we observe the statement

$$Z = \underline{f}(X) \quad \Longrightarrow \quad \phi(X, Z) = 1. \tag{4.130}$$

Statement (4.130) has two interpretations which correspond to one another. We say that $Z = \underline{f}(X)$, a particular solution of the unknowns $z_1, \ldots, z_m$ in the free Boolean algebra $FB(x_1, \ldots, x_n)$, is an antecedent, i.e., a solution, of the equation $\phi(X, Z) = 1$. Additionally, $Z = \underline{f}(X)$ corresponds

to a design which implies a 1-normal form specification $\phi(X, Z) = 1$. The difference between the interpretations is that when we form a particular solution $Z = \underline{f}(X)$ for the equation $\phi(X, Z) = 1$ our focus is on the functions $\underline{f}(X)$; on the other hand, when we develop designs which meet a specification our attention is placed on the formulas $\underline{F}$ which represent $\underline{f}(X)$.

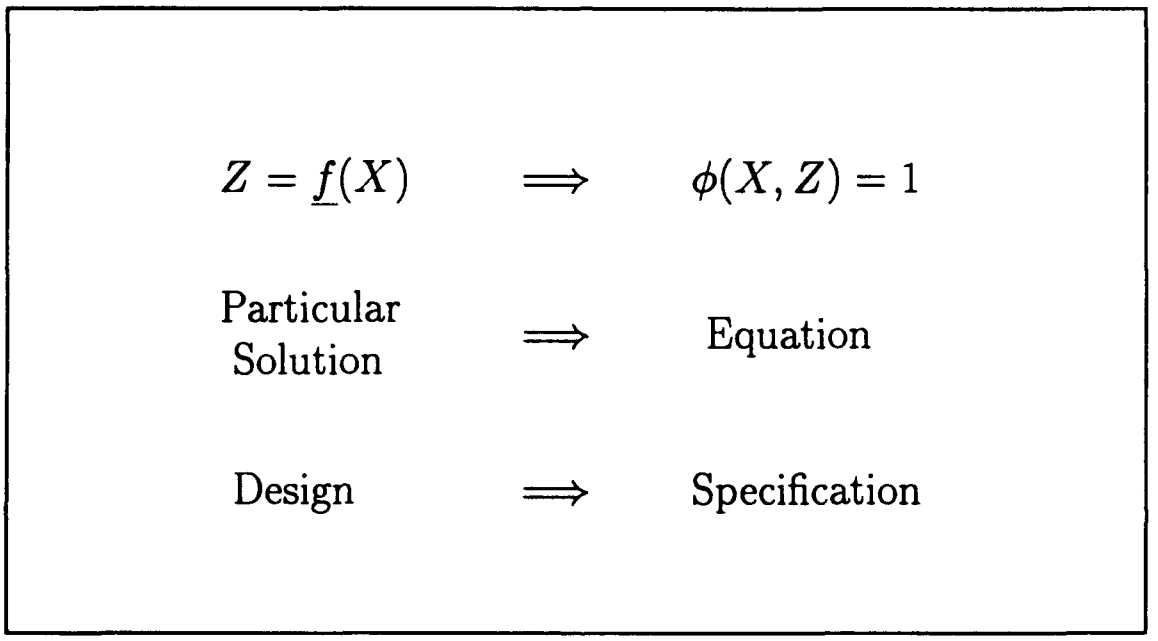


Figure 4.5. Relationships Among Designs, Solutions, Equations, and Specifications

We must necessarily develop a design which implies its specification, i.e., find a particular solution for $\phi(X, Z) = 1$. However, an arbitrary particular solution is not sufficient for our purposes. Rather, we would like to guarantee that functions $\underline{f}(X)$ in the particular solution $Z = \underline{f}(X)$ are represented by formulas which correspond to an economical design. In his book on Boolean equations, Rudeanu proposed the following problem which has specific relevance with respect to the minimisation problem:

> Given the solution of a Boolean equation over a finite free Boolean algebra, ..., determine the best (optimal) solution according to a given criterion. (Rudea 74:408)

In other words, given a general solution of a Boolean equation, find the best particular solution with respect to a given criterion. If we interpret a "good" solution as meaning one in which the formulas $\underline{F}$ representing $\underline{f}(X)$ are the most economical, then Rudeanu's problem of finding a good particular solution corresponds directly to the minimization problem.

Conventional Minimisation. In conventional circuit minimisation, the goal is to find a system $Z = \underline{f}(X)$ for which the set of SOP formulas which represent $\underline{f}(X)$ is minimal with respect to some measure of cost. To develop a system $Z = \underline{f}(X)$ from a tabular specification $\phi(X, Z) = 1$, we first develop a general solution of the form

$$\begin{array}{rcccl} \alpha_1(X) & \leq & z_1 & \leq & \beta_1(X) \\ \alpha_2(X) & \leq & z_2 & \leq & \beta_2(X) \\ \alpha_3(X) & \leq & z_3 & \leq & \beta_3(X) \\ & & \vdots & & \\ \alpha_m(X) & \leq & z_m & \leq & \beta_m(X) \end{array} \tag{4.131}$$

in which each output z_j is specified by an interval $[\alpha_j(X), \beta_j(X)]$.

Given the equation $\phi(X, Z) = 1$, we extract the set of functions which specify the outputs of the circuit. Defining the function $\tilde{\phi}_j(X, z_j) = EDIS(\phi(X, Z), Z_j)$, in which Z_j is the set of variables associated with the output nodes less the output z_j, we develop the system

$$\begin{array}{rcccl} \tilde{\phi}'_1(X, 0) & \leq & z_1 & \leq & \tilde{\phi}_1(X, 1) \\ \tilde{\phi}'_2(X, 0) & \leq & z_2 & \leq & \tilde{\phi}_2(X, 1) \\ \tilde{\phi}'_3(X, 0) & \leq & z_3 & \leq & \tilde{\phi}_3(X, 1) \\ & & \vdots & & \\ \tilde{\phi}'_m(X, 0) & \leq & z_m & \leq & \tilde{\phi}_m(X, 1) \end{array} \tag{4.132}$$

from which a particular solution $Z = \underline{f}(X)$ is developed. In this case,

$$\begin{aligned} \alpha_j(X) &= \tilde{\phi}'_j(X,0) \quad \text{and} \\ \beta_j(X) &= \tilde{\phi}_j(X,1). \end{aligned} \tag{4.133}$$

A general solution more useful than (4.132) is one based on the extended-range concept developed in Theorem 4.2. Using the extended range we produce the general solution

$$\begin{array}{rcccl} \tilde{\phi}'_1(X,0)\cdot\tilde{\phi}_1(X,1) & \leq & z_1 & \leq & \tilde{\phi}'_1(X,0)+\tilde{\phi}_1(X,1) \\ \tilde{\phi}'_2(X,0)\cdot\tilde{\phi}_2(X,1) & \leq & z_2 & \leq & \tilde{\phi}'_2(X,0)+\tilde{\phi}_2(X,1) \\ \tilde{\phi}'_3(X,0)\cdot\tilde{\phi}_3(X,1) & \leq & z_3 & \leq & \tilde{\phi}'_3(X,0)+\tilde{\phi}_3(X,1) \\ & & \vdots & & \\ \tilde{\phi}'_m(X,0)\cdot\tilde{\phi}_m(X,1) & \leq & z_m & \leq & \tilde{\phi}'_m(X,0)+\tilde{\phi}_m(X,1). \end{array} \tag{4.134}$$

System (4.134) must be used when there exists a constraint condition such as an input combination $A \in \mathbf{B}_2^n$ which does not occur (See Example 4.9).

Once we develop the general solution (4.134) for $\phi(X,Z) = 1$, we must apply a technique to find the cheapest set of formulas with respect to a given cost criterion among all of the possible formulas which may represent functions in the intervals

$$[\tilde{\phi}'_j(X,0)\cdot\tilde{\phi}_j(X,1), \tilde{\phi}'_j(X,0)+\tilde{\phi}_j(X,1)], \quad j = 1,\ldots,m.$$

The resulting formulas represent functions in the range; each corresponds to a design which implements a circuit for the respective output. Techniques to develop such formulas are discussed in Chapters 6 and 7.

Alternative Approach. In conventional circuit minimization, a system such as (4.132) is used as the basis for developing a design. This approach constrains the form of the resulting design

such that each output z_j is a function only of the input variables X. Using the method of successive eliminations to form a subsumptive general solution of the equation $\phi(X, Z) = 1$, we develop the recurrent system

$$\begin{array}{rcccl} & & 1 & \leq & \phi_0(X) \\ \phi_1'(X,0) & \leq & z_1 & \leq & \phi_1(X,1) \\ \phi_2'(X,z_1,0) & \leq & z_2 & \leq & \phi_2(X,z_1,1) \\ \phi_3'(X,z_1,z_2,0) & \leq & z_3 & \leq & \phi_3(X,z_1,z_2,1) \\ & & \vdots & & \\ \phi_m'(X,z_1,\ldots,z_{m-1},0) & \leq & z_m & \leq & \phi_m(X,z_1,\ldots,z_{m-1},1). \end{array} \tag{4.135}$$

In view of Theorem 4.4, a recurrent system based on the extended-range concept is developed:

$$\begin{array}{rcccl} & & 1 & \leq & \phi_0(X) \\ \phi_1'(X,0)\cdot\phi_1(X,1) & \leq & z_1 & \leq & \phi_1'(X,0)+\phi_1(X,1) \\ \phi_2'(X,z_1,0)\cdot\phi_2(X,z_1,1) & \leq & z_2 & \leq & \phi_2'(X,z_1,0)+\phi_2(X,z_1,1) \\ \phi_3'(X,z_1,z_2,0)\cdot\phi_3(X,z_1,z_2,1) & \leq & z_3 & \leq & \phi_3'(X,z_1,z_2,0)+\phi_3(X,z_1,z_2,1) \\ & & \vdots & & \\ \phi_m'(X,z_1,\ldots,z_{m-1},0) & \leq & z_m & \leq & \phi_m'(X,z_1,\ldots,z_{m-1},0) \\ \cdot\phi_m(X,z_1,\ldots,z_{m-1},1) & & & & +\phi_m(X,z_1,\ldots,z_{m-1},1). \end{array} \tag{4.136}$$

From the recurrent system (4.136) we derive a system

$$\begin{array}{rcl} z_1 & = & f_1(X) \\ z_2 & = & f_2(X,z_1) \\ z_3 & = & f_3(X,z_1,z_2) \\ & \vdots & \\ z_m & = & f_m(X,z_1,z_2,\ldots,z_{m-1}). \end{array} \tag{4.137}$$

We denote the system (4.137) by $Z = \underline{f}_m(X, Z)$ with the understanding that each z_j is dependent only on $z_1, \ldots, z_{j-1}$. We call a design represented by $Z = \underline{f}_m(X, Z)$ a *recurrent design*. The advantage of a recurrent system such as (4.137) is that a design may be developed in which we use

output signals as well as input signals to generate a given output signals. This allows us, in many instances, to develop more economical designs than in conventional approaches to minimization.

We may also use a recurrent system in the design process in ways that are not possible in conventional minimization. If a portion of a circuit already has been constructed, we may be able to use the existing subcircuit to develop a more economical design for the remainder of the circuit than would be possible using conventional techniques. This idea is illustrated in Example 4.10. We will discuss this approach in detail in Chapter 9.

Example 4.10: Suppose we are given the circuit specification denoted by Table 4.12. Furthermore, suppose the output z_1 has been chosen to be

$$z_1 = x_1 + x_2. \tag{4.138}$$

The function $\phi(X, Z)$ of the equivalent 1-normal form specification $\phi(X, Z) = 1$ for the truth table is given by

$$\phi(X, Z) = x_1'x_2'x_3'z_1'z_2 + x_2x_3'z_1z_2' + x_1x_3'z_1z_2' + x_1'x_2'x_3z_1'z_2' + x_2x_3z_1z_2 + x_1x_3z_1z_2. \tag{4.139}$$

Using the conventional approach to minimization we develop the general solution

$$\begin{array}{rcccl} x_1 + x_2 & \leq & z_1 & \leq & x_1 + x_2 \\ x_1x_3 + x_2x_3 + x_1'x_2'x_3' & \leq & z_2 & \leq & x_1x_3 + x_2x_3 + x_1'x_2'x_3'. \end{array} \tag{4.140}$$

We note that the upper and lower bounds for both z_1 and z_2 are equal, i.e., they are completely-specified. Hence, we form

$$z_1 = x_1 + x_2 \tag{4.141}$$
$$z_2 = x_1x_3 + x_2x_3 + x_1'x_2'x_3'. \tag{4.142}$$

The formula on the right-hand side of (4.142) represents the design for z_2.

Using the alternative approach, we develop the recurrent system

$$\begin{aligned} 1 &\leq 1 \\ x_1 + x_2 \leq\ & z_1 \leq x_1 + x_2 && (4.143) \\ x_1x_3z_1 + x_2x_3z_1 + x_1'x_2'x_3'z_1' \leq\ & z_2 \leq x_1x_3 + x_2x_3 + x_1'x_2'x_3' + x_3'z_1' && (4.144) \\ & \quad + x_3z_1 + x_1z_1' + x_1'x_2'z_1 + x_2z_1'. \end{aligned}$$

We have placed the upper bound of z_2 in (4.144) in Blake canonical form. A minimal set of prime implicants in the upper bound which covers the lower bound of z_2 is $\{x_3'z_1', x_3z_1\}$. Hence, a possible design for z_2 is denoted by

$$z_2 = x_3'z_1' + x_3z_1. \tag{4.145}$$

The right-hand side of (4.145) represents a cheaper design for z_2 than does the right-hand side of (4.142).

Summary

In this chapter, we have discussed solutions of Boolean equations of the form $f(X) = 1$, the modeling of digital circuits with Boolean algebra, and the correspondence of the digital design

x_1	x_2	x_3	z_1	z_2
0	0	0	0	1
0	0	1	0	0
0	1	0	1	0
0	1	1	1	1
1	0	0	1	0
1	0	1	1	1
1	1	0	1	0
1	1	1	1	1

Table 4.12. Truth Table for Example 4.10

problem with that of developing solutions of a Boolean equation. Given the ideas developed in this chapter, we may now outline the general design process that is used in this work:

1. Given a specification for a digital circuit, form an equivalent 1-normal form specification $\phi(X, Z) = 1$ for the circuit.
2. Develop a general solution for $\phi(X, Z) = 1$. If each output of the resulting circuit is to be a function only of the inputs, then form the general solution (4.134). If the outputs may be formed as functions of the outputs as well as other inputs, then form the general solution (4.136).
3. Given the general solution, form a particular solution which meets cost criteria. The formulas representing the outputs correspond to circuit designs.

We discussed in this chapter how to form a 1-normal form specification and how to develop a general solution for $\phi(X, Z) = 1$. In the remainder of this dissertation, we develop means for developing particular solutions corresponding to digital circuit designs which are minimal or near-minimal with respect to a given cost criterion.

V. Formation of All Irredundant Formulas

The following methodology for finding a minimal sum-of-products (SOP) formula has been pursued by many researchers: first, develop the set of all irredundant formulas which may represent a function; then select a formula from the set which is minimal with respect to a given cost criterion. Forming a minimal irredundant SOP formula to represent a function is equivalent to finding an economical design for a two-level digital circuit. The formation of all irredundant formulas to represent either a single function or an interval of functions is discussed in this chapter. We will present techniques for deriving a single minimal SOP formula in Chapter 6.

We present three different methods for forming all irredundant SOP formulas. In each method, after generating all irredundant formulas, a least-cost formula F is then selected. F corresponds to a minimal two-level design. All three approaches are based on the determination of implication relations as developed in Chapter 3. Our third method—which we call the *Multiplication Method*—is similar to many techniques found in the literature. Before presenting an implementation of the Multiplication Method, we give an overview of several of the significant results from the past.

A comparison of computational results for each of the presented techniques is given in the last section of the chapter. We begin with a discussion of the initial circuit specification.

Initial Specification

We assume that at the outset of the design process we are given a specification in 1-normal form specification, i.e., $\phi(X, z) = 1$. If not, then we convert the given specification to an equivalent 1-normal form. Viewing $\phi(X, z)$ as a function consisting of the single variable z, we use Theorem 4.2 to solve $\phi(X, z) = 1$ for the variable z over the free Boolean algebra $FB(x_1, \ldots, x_n)$. We may thus represent $\phi(X, z) = 1$ by the equivalent statement

$$g(X) \leq z \leq h(X), \tag{5.1}$$

for which

$$g(X) = \phi'(X,0) \cdot \phi(X,1) \tag{5.2}$$

and

$$h(X) = \phi'(X,0) + \phi(X,1). \tag{5.3}$$

It is a well-known result that a minimal SOP formula F which represents a function $f(X)$ belonging to the interval $[g(X), h(X)]$ is an irredundant sum of prime implicants of $h(X)$ which covers $g(X)$. For the remainder of this chapter we discuss the development of all irredundant formulas which represent functions $f(X)$ belonging to the interval $[g(X), h(X)]$. We assume for the moment that the function $EDIS(\phi(X,z),\{z\})$ in the consistency condition for $\phi(X,z) = 1$ is identically equal to 1.

Brown's Method

As established in the previous section, z is a solution of $\phi(X,z) = 1$ if and only if z belongs to the interval $[g(X), h(X)]$. We apply Procedure 3.4 to form all irredundant formulas which represent functions belonging to the interval $[g(X), h(X)]$. All possible irredundant formulas are found by ascertaining all possible ways the function $g(X)$ is included in minimal subsets of the set of prime implicants of $h(X)$. Procedure 3.4 may be applied to determine all possible irredundant formulas, if the following associations are made with the parameters of the procedure:

- function $g(X)$ is the function f_0, and
- the set of prime implicants of $h(X)$ forms the set of functions $\{f_1, f_2, \ldots, f_k\}$.

Each function of the set $\{f_1, f_2, \ldots, f_k\}$ is a prime implicant; f_0 is a function represented by a SOP formula.

Let us define the vector $A = (G_0, P_1, P_2, \ldots, P_k)$ of labels in which G_0 is the label associated with $g(X)$ and $P_1, P_2, \ldots, P_k$ are labels associated with the prime implicants of $h(X)$. If the function h is identically equal to 1, then $BCF(h(X))$ consists of a single prime implicant—the term 1. Since $BCF(h(X))$ is the term 1, we derive a system of the form

$$\begin{aligned} G_0 &\leq g(X) \\ 1 &\leq P_1 \end{aligned} \tag{5.4}$$

which is equivalent to the system

$$\begin{aligned} G_0 \cdot g'(X) &= 0 \\ P_1' &= 0. \end{aligned} \tag{5.5}$$

System (5.5) is equivalent to the equation $f(A, X) = 0$, for which

$$f(A, X) = G_0 \cdot g'(X) + P_1'. \tag{5.6}$$

If $g(X) \neq 0$, then the only A-consequent deducible from $f(A, X) = 0$ is $P_1' = 0$. Hence, the only irredundant formula which covers the function $g(X)$ is the formula which represents the term 1.

If the function $h(X)$ consists of a sum of prime implicants and $h(X) \neq 1$, no subset of the prime implicants is normal. Therefore, all terms $p(A)$ of prime A-consequents $p(A) = 0$ of $f(A, X) = 0$ take the form $G_0 P_1' \cdots P_s'$. We begin by formulating the system

$$\begin{aligned} G_0 &\leq g(X) \\ p_1(X) &\leq P_1 \\ p_2(X) &\leq P_2 \\ &\vdots \\ p_k(X) &\leq P_k. \end{aligned} \tag{5.7}$$

This system is reduced to the form $f(A, X) = 0$, for which

$$f(A, X) = G_0 g'(X) + p_1(X)P_1' + p_2(X)P_2' + \cdots + p_k P_k'. \tag{5.8}$$

Prime consequent terms $G_0 P_1' \cdots P_s'$ are generated by forming $BCF(ECON(f(A, X), X))$. Using goal-directed elimination to eliminate X-arguments from $f(A, X) = 0$, we derive a formula $G(A)$ to represent $g(A) = ECON(f(A, X), X)$ which consists of all prime A-consequent terms denoting irredundant formulas for $f(X)$.

We now present an algorithm for determining all irredundant formulas which represent Boolean functions belonging to the interval $[g(X), h(X)]$ developed from $\phi(X, z) = 1$. This procedure is a modified version of Procedure 3.4 designed specifically for the present problem. A theoretically similar method is presented in (Brown 90:148). Hence, we call this algorithm *Brown's Method.*

Algorithm 5.1 (Brown's Method): Given a 1-normal form specification $\phi(X, z) = 1$, all irredundant formulas which represent Boolean functions f belonging to the interval $[g(X), h(X)]$ developed from $\phi(X, z) = 1$ are formed in the following manner:

Step 1.

1. Form $g(X) = \phi'(X, 0) \cdot \phi(X, 1)$.
2. Form $h(X) = \phi'(X, 0) + \phi(X, 1)$.

Step 2.

1. Form a simplified formula to represent $g(X)$ using Procedure 2.15 (Simplification). Call the simplified formula G.
2. Develop the Blake canonical form for function $h(X)$ using Procedure 2.20 (Blake canonical form). Denote the set of prime implicants of $h(X)$ by P.

Step 3.

1. Generate a label G_0.
2. Complement function $g(X)$ using Procedure 2.7 (Complementation).
3. Prefix each term of the formula which represents $g'(X)$ with the literal G_0.

The resulting formula represents $G_0 \cdot g'(X)$.

Step 4. For each prime $p_i(X) \in P$:

1. Generate an associated label P_i.
2. Append the complemented literal P_i' to the prime $p_i(X)$.

The resulting term represents $p_i(X) \cdot P_i'$.

Step 5. Append together the formula formed in Step 3 and each of the terms formed in Step 4. The resulting formula represents $f(A, X)$.

Step 6. Using the quick method for forming the conjunctive eliminant (Procedure 3.1), form $ECON(f(A, X), X)$. The resulting function is equal to $g(A)$; the formula which represents $g(A)$ is $BCF(g(A))$. Each term of $BCF(g(A))$ denotes an irredundant formula for the function $f(X)$.

Step 7. Given a term of $BCF(g(A))$, each complemented literal in the term $G_0 P_1' \cdots P_s'$ is associated with a prime implicant in an irredundant formula which represents a function in the interval $[g(X), h(X)]$. Generate an irredundant formula representing a function in the interval $[g(X), h(X)]$ by forming disjunctions consisting of prime implicants associated with the complemented literals P_i'. Formulate all irredundant formulas by repeating this process for each term of $BCF(g(A))$.

Example 5.1 demonstrates the application of Algorithm 5.1 to generate all irredundant formulas which represent a Boolean function.

Example 5.1: Given the 1-normal form specification $\phi(X, z) = 1$, for which

$$\phi(X, z) = x_1' x_2 z' + x_1' x_2 x_3 + x_1' x_2' z + x_1 x_2 x_3' + x_1 x_2 z + x_1 x_2' z', \tag{5.9}$$

and $X = (x_1, x_2, x_3)$, we apply Algorithm 5.1 to determine all irredundant formulas which may represent a function $f(X)$ in the interval $[g(X), h(X)]$.

Step 1. To develop the interval $[g(X), h(X)]$, we first generate the functions

$$\begin{aligned} \phi'(X, 0) &= x_1 x_2 x_3 + x_1' x_2' \\ \phi(X, 1) &= x_1' x_2' + x_1 x_2 + x_2 x_3 + x_1' x_3. \end{aligned} \tag{5.10}$$

We then form $g(X) = \phi'(X,0) \cdot \phi(X,1)$ and $h(X) = \phi'(X,0) + \phi(X,1)$, for which

$$g(X) = x_1'x_2' + x_1x_2x_3 \tag{5.11}$$
$$h(X) = x_1'x_2' + x_1x_2 + x_2x_3 + x_1'x_3. \tag{5.12}$$

Step 2. A simplified formula to represent $g(X)$ is given by the right-hand side of (5.11). Moreover, the right-hand side of (5.12) is $BCF(h(X))$.

Steps 3-5. Having $g(X)$ and $BCF(h(X))$, we form the system

$$\begin{aligned} G_0 &\leq x_1'x_2' + x_1x_2x_3 \\ x_1'x_2' &\leq P_1 \\ x_1x_2 &\leq P_2 \\ x_2x_3 &\leq P_3 \\ x_1'x_3 &\leq P_4, \end{aligned} \tag{5.13}$$

where the following associations are made between labels P_i and prime implicants of $h(X)$:

- P_1 with $x_1'x_2'$,
- P_2 with x_1x_2,
- P_3 with x_2x_3, and
- P_4 with $x_1'x_3$.

Given the complement of function $g(X)$,

$$g'(x_1,x_2,x_3) = x_1x_2' + x_1x_3' + x_1'x_2, \tag{5.14}$$

the equation $f(A,X) = 0$ is formed, for which $f(A,X)$ is defined by the equation

$$f(A,X) = G_0x_1x_2' + G_0x_1x_3' + G_0x_1'x_2 + P_1'x_1'x_2' + P_2'x_1x_2 + P_3'x_2x_3 + P_4'x_1'x_3. \tag{5.15}$$

Step 6. Eliminating the X-arguments x_1, x_2, and x_3 from $f(A,X) = 0$ using goal-directed elimination, we develop an equation $g(A) = 0$, for which

$$BCF(g(A)) = G_0P_1'P_2' + G_0P_1'P_3'. \tag{5.16}$$

Step 7. Prime implicant $G_0P_1'P_2'$ denotes the irredundant formula $x_1'x_2' + x_1x_2$; prime implicant $G_0P_1'P_3'$ denotes the irredundant formula $x_1'x_2' + x_2x_3$. We observe that the prime implicant $x_1'x_2'$ is an essential prime implicant of h because it appears in every irredundant formula which represents a function in the interval $[g(X), h(X)]$.

Algorithm 5.1 produces all irredundant formulas which represent functions $f(X)$ belonging to the interval $[g(X), h(X)]$. We can use the set of formulas returned by Algorithm 5.1 to develop an economical design for a single-output digital circuit. The set of irredundant formulas may be

examined to determine a least-cost formula with respect to a cost criterion to represent a design. Hence, after finding all irredundant formulas, we determine the cost of each formula based on the given evaluation criterion. Cost criteria mentioned in Chapter 4 include:

- the fewest terms (gates);
- the fewest literals (gate inputs); and
- a combination of the fewest terms and fewest literals (gates and gate inputs).

After each formula is graded, a least-cost formula is chosen to represent the design.

As the number of terms of the formula which represents $g(X)$—and the number of prime implicants of $h(X)$—increase, the time and space required to execute Algorithm 5.1 grow very rapidly. Therefore, we seek methods to reduce the computational complexity of the method. Such techniques are discussed in the next section.

Modified Brown's Method

A way to decrease the computational complexity of Algorithm 5.1 is to reduce the number of prime implicants of $h(X)$ involved in the partial label-and-reduce process. Hence, there is a corresponding decrease in the number of $p_i(X) \leq P_i$ expressions in (5.7). We present two techniques for identifying prime implicants of $h(X)$ which may be deleted from consideration prior to these processes.

Certain prime implicants of $h(X)$ are not "useful" for covering the function $g(X)$, i.e., they are "useless". Useless prime implicants can be identified and removed from consideration prior to the label-and-reduce process. Thus, the first method we introduce for reducing the number of prime implicants identifies and deletes such prime implicants.

The second technique we employ is to partition the set of prime implicants of $h(X)$ into essential, inessential, and conditionally-eliminable prime implicants prior to the label-and-reduce

process. Essential prime implicants (EPIs) are prime implicants which appear in every irredundant formula which represents a function $f(X)$ belonging to the interval $[g(X), h(X)]$; each EPI is required because it is the only prime implicant of $h(X)$ which covers some portion of the lower bound $g(X)$. Inessential prime implicants (IPIs) are prime implicants which are included in the function $h_{sss}(X)$ formed by summing all essential prime implicants; inessential PIs do not appear in any irredundant formula. Therefore, the only difference among irredundant formulas is the conditionally-eliminable prime implicants (CEPIs) which appear in each one. Hence, the only prime implicants that must be involved in the label-and-reduce and elimination processes are CEPIs.

Once the useful, conditionally-eliminable prime implicants of $h(X)$ are identified, the relative simplification technique as applied in Chapter 3 may be used to remove particular literals from the remaining prime implicants. This technique decreases the number of operations required in the elimination process.

Before presenting a revised algorithm for generating all irredundant formulas, we discuss the identification of useless prime implicants of $h(X)$. We then present a methodology for partitioning the prime implicants of $h(X)$.

Useless Prime Implicants. A prime implicant $p(X)$ of $h(X)$ is *useless* with respect to $g(X)$ if and only if the statement

$$p(X) \leq h(X) - g(X) \tag{5.17}$$

is an identity. Prime implicants which are not useless with respect to $g(X)$ are called *useful* prime implicants (Tison 67:452). A way to determine whether a prime implicant is useless is to determine the validity of the statement $p(X) \cdot g(X) = 0$; we demonstrate this by Theorem 5.1.

Theorem 5.1 (Useless Prime Implicants): *Given a lower bound function $g(X)$ and a prime implicant $p(X)$ of an upper bound function $h(X)$, the prime implicant $p(X)$ is a useless prime implicant if and only if the equation*

$$p(X) \cdot g(X) = 0 \tag{5.18}$$

is an identity.

Proof. A prime implicant p of h is useless if and only if $p \leq h - g$. By the definition of subtraction, the function $h - g$ is equal to hg'. Because of the property

$$a \leq b \iff a - b = 0, \tag{5.19}$$

we may determine whether a prime implicant p is included in hg' by subtracting hg' from p and testing whether the result is identically equal to 0. Thus,

$$p \leq h - g \iff p - hg' = 0. \tag{5.20}$$

Then, the following statements are equivalent:

$$\begin{aligned} p - hg' &= 0 && (5.21)\\ p(hg')' &= 0 && (5.22)\\ ph' + pg &= 0. && (5.23) \end{aligned}$$

Equation (5.23) is equivalent to the set

$$\begin{aligned} ph' &= 0 && (5.24)\\ pg &= 0 && (5.25) \end{aligned}$$

of equations. By the definition of the inclusion relation, $ph' = 0 \Leftrightarrow p \leq h$. The statement $p \leq h$ is an identity because p is a prime implicant of h. Hence, the statement $p \leq h - g$ is true if and only if $pg = 0$. This completes the proof. □

Applying Theorem 5.1, we develop a test to determine the usefulness a prime implicant p by forming the function $p \cdot g$ and examining the result. If $p \cdot g \neq 0$, then a prime implicant is useful; otherwise, it is useless. We use this test in Procedure 5.1 which returns the elements in a set P of prime implicants which are useful with respect to a function $g(X)$. In Procedure 5.1, we determine whether a prime implicant p is useful by comparing it to each term in the formula G which represents $g(X)$. If one term t in G is found for which $p \cdot t \neq 0$, then p is useful. Otherwise, the prime implicant is useless.

Procedure 5.1 (Useful Prime Implicants): Given a formula G which represents a function $g(x)$, and a set P of prime implicants, we determine the prime implicants of P which are useful in the following manner:

Step 0. Initialise an accumulator P_{useful} to the empty set $\emptyset$.

Step 1.

- If the set P of prime implicants is empty, then return P_{useful}. It is the set of useful prime implicants.
- Otherwise, remove the first term from P and call it $\hat{p}$. Initialize an accumulator G_{ACC} by placing in it the contents of G. Continue to Step 2.

Step 2.

- If G_{ACC} is empty, then the prime implicant $\hat{p}$ is useless. Return to Step 1.
- Otherwise, remove the first term from G_{ACC} and call it t. Continue to Step 3.

Step 3.

- If t and $\hat{p}$ have any opposed literals, then $t \cdot \hat{p} = 0$. The prime implicant $\hat{p}$ must be compared to another term of G to determine usefulness. Return to Step 2.
- Otherwise, $\hat{p}$ is a useful prime implicant. Place $\hat{p}$ in P_{useful} and return to Step 1.

Partitioning of Prime Implicants. We now introduce a methodology for partitioning the prime implicants of $h(X)$. In this methodology, the partitioning of the prime implicants is performed in two steps:

1. The essential prime implicants of $h(X)$ with respect to $g(X)$ are identified.
2. The inessential prime implicants of $h(X)$ are determined.

After the sets of essential and inessential prime implicants of $h(X)$ with respect to $g(X)$ have been identified, the remaining prime implicants of $h(X)$ form the set of conditionally-eliminable prime implicants.

The first step in our methodology for partitioning the prime implicants is to identify the essential prime implicants of $h(X)$ with respect to $g(X)$. A number of techniques for developing a minimal formula F found in the literature also identify the essential prime implicants at the outset of developing a minimal F. These methods test each prime implicant to determine if it essential. However, it is possible to identify a subset of the prime implicants of $h(X)$ which are potential essential prime implicants with respect to $g(X)$, such that all other prime implicants of $h(X)$ are not essential. We use the terms in the formula G which represents $g(X)$ to identify potential essential prime implicants.

Each term t in G has at least one prime implicant p of $h(X)$ which completely contains it, i.e., $t \leq p$. The prime implicant p is essential if and only if it necessary to cover t. If $\widetilde{P}$ is formed by removing prime implicant p from $BCF(h(X))$, then p is essential if and only if

$$t \not\leq \widetilde{P} \tag{5.26}$$

is valid. If $t \leq \widetilde{P}$, then p is not necessary to cover t, i.e., p is not essential. We only need to test a subset of the prime implicants of $h(X)$ such that each prime implicant tested completely contains a term in G. A way to develop such a subset is to find for each term t a prime implicant p in

which it is included and then test p to determine if it is essential. The test for inclusion of a term t in a prime implicant p is simple, because t is included in p if the literals contained in t are a superset of the literals contained in p. As a result of the manner in which we determine potential essential prime implicants, we test at most as many prime implicants as there are terms in G. In many cases, the number of prime implicants that are tested is only a fraction of the total number of prime implicants.

Procedure 5.2 is a method for identifying all essential prime implicants of an interval $[g(X), h(X)]$. It is assumed that the formula representing $h(X)$ is $BCF(h(X))$ and the formula G is a simplified formula for $g(X)$. A key aspect of Procedure 5.2 is that all elements of $BCF(h(X))$ do not have to be tested to determine if they are essential, which greatly improves the efficiency of detecting essential prime implicants. Procedure 5.2 returns both the set of essential prime implicants of $h(X)$ with respect to $g(X)$ and the terms in G which were included in the essential prime implicants, i.e., the terms in G used to identify the essential PIs. Procedure 5.2 may also be used to identify essential prime implicants in a completely-specified Boolean function $f(X)$; $BCF(f(X))$ is substituted for $BCF(h(X))$, and a simplified formula which represents $f(X)$ is used in place of G.

Procedure 5.2 (Essential Prime Implicants): Given an interval $[g(X), h(X)]$ in which $g(X)$ is represented by a simplified formula G and $h(X)$ is represented by $BCF(h(X))$, the set of essential prime implicants of $h(X)$ with respect to $g(X)$ is determined in the following manner:

Step 0.

1. Initialise an accumulator P_{essen} to the empty set $\emptyset$.
2. Initialise an accumulator $G_{covered}$ to the empty set $\emptyset$.

Step 1.

- If G is empty, then P_{essen} contains all essential prime implicants and $G_{covered}$ contains terms of G used to identify the essential prime implicants. Return P_{essen} and $G_{covered}$.
- Otherwise, remove the first term from G and call it t. Initialize an accumulator $\widehat{P}$ by placing the contents of $BCF(h(X))$ into it. Continue to Step 2.

Step 2. Remove the first term from $\widehat{P}$ and call it p.

- If t consists of a superset of the literals in p, then p will be tested to determine if it is essential. Continue to Step 3.

- Otherwise, t is not included in p. Repeat Step 2. (Note: We do not have to be concerned with exhausting $\widehat{P}$ because t is included in at least one prime implicant in $\widehat{P}$.)

Step 3.

- If p is a member of P_{essen}, then we previously identified p as being an essential prime implicant. Additionally, term t is covered by an essential prime implicant. Add t to $G_{covered}$. Return to Step 1.
- Otherwise, continue to Step 4.

Step 4. Form a formula $\widetilde{P}$ by copying all terms in $BCF(h(X))$ into $\widetilde{P}$ except the prime implicant p. Using Procedure 2.25, determine if $t \leq \widetilde{P}$.

- If $t \leq \widetilde{P}$, then p is not an essential prime implicant. Return to Step 1.
- If $t \not\leq \widetilde{P}$, then p is an essential prime implicant. Add p to P_{essen} and add t to $G_{covered}$. Return to Step 1.

After identifying all essential prime implicants via Procedure 5.2, the set of inessential prime implicants must be determined. Subsequently, a function $h_{ess}(X)$ is formed which consists of all essential prime implicants. A generalized technique for finding all terms in a formula F which are included in a function g is given by Procedure 5.3. Thus, Procedure 5.3 may be used to identify the inessential prime implicants in which the function $h_{ess}(X)$ corresponds to the function g in Procedure 5.3, and the formula F is $BCF(h(X))$ less all essential prime implicants. Procedure 5.3 then returns all inessential prime implicants. After identifying the inessential prime implicants, we also know the set of conditionally-eliminable prime implicants since the CEPIs are the elements of $BCF(h(X))$ which are neither essential nor inessential.

Procedure 5.3 (Covered Terms): Given a formula F and a function g, terms in F which are included in g are determined as follows:

Step 0. Initialise an accumulator $F_{covered}$ to the empty set $\emptyset$.

Step 1.

- If F is empty, then $F_{covered}$ contains all terms in the original formula F which are included in g. Return $F_{covered}$.
- Otherwise, continue to Step 2.

Step 2. Remove the first term from F and call it t. Use Procedure 2.25 to determine if $t \leq g$.

- If $t \leq g$, then t is covered by g. Add t to $F_{covered}$ and return to Step 1.
- If $t \not\leq g$, then t is not covered by g. Return to Step 1.

A Revised Algorithm. Let G_{cov} denote the sum of the terms in the formula G which are returned with the essential prime implicants of $h(X)$ in Procedure 5.2, i.e., the terms in G covered by a single essential PI and used to determine that the prime implicant was essential. Terms in G_{cov} may be removed from G to form a new formula $(G - G_{cov})$. Moreover, the function $h_{ess}(X)$ may be subtracted from the function represented by $(G - G_{cov})$ to develop a function $\hat{g}(X)$, i.e.,

$$\hat{g}(X) = (g(X) - g_{cov}(X)) - h_{ess}(X), \tag{5.27}$$

which must be covered by the conditionally-eliminable prime implicants.[1] Thus, effort is reduced by forming only those implication relations in which the portion of the lower bound which is not covered by essential prime implicants is covered by sums of CEPIs.

After forming $\hat{g}(X)$, Procedure 5.1 (Useful Prime Implicants) is used to determine the useful PIs of the set of conditionally-eliminable prime implicants with respect to $\hat{g}(X)$. After identifying the useful conditionally-eliminable prime implicants, we may then apply the ideas illustrated by Theorem 3.7 to simplify each of the useful CEPIs relative to the formula $\widehat{G}'$ which represents the function $\hat{g}'(X)$. This technique reduces the number of literals in each prime implicant prior to the label-and-reduce process. Subsets of terms resulting from the relative simplification process which cover $\hat{g}(X)$ correspond to subsets of useful conditionally-eliminable prime implicants which cover $\hat{g}(X)$. However, because the terms resulting from the relative simplification process consist of fewer literals, the label-and-reduce and elimination processes require less memory and fewer computations.

Finally, all irredundant formulas which cover the function $\hat{g}(X)$ are generated. These formulas are composed of useful conditionally-eliminable prime implicants.

[1] Mathematically, we can also say that $\hat{g}(X) = g(X) - h_{ess}(X)$. Since $g_{cov} \leq h_{ess}$ is true, we know that $g_{cov} \cdot h'_{ess} = 0$. The right-hand side of (5.27) may be written as $g \cdot g'_{cov} \cdot h'_{ess}$, which, in view of $g_{cov} \cdot h'_{ess} = 0$, may be rewritten as $g \cdot h'_{ess}$.

Algorithm 5.2 is a revised version of Algorithm 5.1 which incorporates the aforementioned techniques. Each subset of the useful conditionally-eliminable prime implicants which covers the function $\hat{g}(X)$ is added to the essential prime implicants to form an irredundant formula F representing a function $f(X)$ belonging to the interval $[g(X), h(X)]$. Since this algorithm is an improvement of Brown's Method, we call this technique the *Modified Brown's Method.*

Algorithm 5.2 (Modified Brown's Method): Given a 1-normal form specification $\phi(X, z) = 1$, all irredundant formulas which represent Boolean functions belonging to the interval $[g(X), h(X)]$ developed from $\phi(X, z) = 1$ are formed in the following manner:

Step 1.

1. Form $g(X) = \phi'(X, 0) \cdot \phi(X, 1)$.
2. Form $h(X) = \phi'(X, 0) + \phi(X, 1)$.

Step 2.

1. Form a simplified formula to represent $g(X)$ using Procedure 2.15 (Simplification). Call the simplified formula G.
2. Develop the Blake canonical form for function $h(X)$ using Procedure 2.20 (Blake canonical form).

Step 3. Using Procedure 5.2 (Essential Prime Implicants), G, and $BCF(h(X))$, determine the essential prime implicants of $h(X)$.

1. Denote the set of essential prime implicants by $H_{ess}(X)$ and the function formed by the disjunction of the essential prime implicants by $h_{ess}(X)$.
2. Call the set of terms of G used to identify essential prime implicants in Procedure 5.2—terms covered by the essential prime implicants—$G_{covered}$.

Step 4.

1. Form a set $\widehat{H}$ of prime implicants consisting of all prime implicants of $h(X)$ except the essential prime implicants.
2. Use Procedure 5.3 (Covered Terms), $\widehat{H}$, and $h_{ess}(X)$ to determine the terms of $\widehat{H}$ covered by $h_{ess}(X)$. These terms comprise the set of inessential prime implicants of $h(X)$; denote this set of terms by $H_{inessen}$.

Step 5.

1. Remove from G the terms in $G_{covered}$; denote the resulting formula by $G - G_{covered}$.
2. Using Procedure 2.10 (Subtraction), subtract the function $h_{ess}(X)$ from the function represented by $G - G_{covered}$.
3. Call the resulting formula $\widehat{G}$ and the function which it represents $\hat{g}(X)$.

Step 6.

1. Form the set H_{ce} of conditionally-eliminable prime implicants by removing the prime implicants in $H_{inessen}$ from $\hat{H}$.
2. Using Procedure 5.1 (Useful Prime Implicants), determine which prime implicants in set H_{ce} are useful with respect to $\hat{g}(X)$. Call the set of useful prime implicants H_{useful}.

Step 7. Complement function $\hat{g}(X)$ using Procedure 2.7 (Complementation). For each prime implicant p_i in H_{useful}, simplify the prime implicant relative to $\widehat{G}'$, i.e.,

$$SIMPREL(p_i, \widehat{G}'). \tag{5.28}$$

Form a set T_{useful} consisting of the terms resulting from the relative simplification process.

Step 8.

1. Generate a label G_0.
2. Prefix each term of the formula $\widehat{G}'$ which represents $\hat{g}'(X)$ with the literal G_0.

The resulting formula represents $G_0 \cdot \hat{g}'(X)$.

Step 9. For each term $t_i \in T_{useful}$:

1. Generate an associated label P_i.
2. Append the complemented literal P_i' to the term t_i.

The resulting term represents $t_i \cdot P_i'$.

Step 10. Append together the formula formed in Step 8 and each of the terms formed in Step 9. The resulting formula represents $f(A, X)$.

Step 11. Using the quick method for forming the conjunctive eliminant (Procedure 3.1), form $ECON(f(A, X), X)$. The resulting function is equal to $g(A)$; the formula which represents $g(A)$ is $BCF(g(A))$.

Step 12.

1. Each term of $BCF(g(A))$ consisting only of complemented literals represents a minimal normal subset; append the literal G_0 to each of these terms. (Do nothing to the remaining terms.) Call the resulting formula $\widetilde{G}$.
2. Form the equivalent absorptive formula, $ABS(\widetilde{G})$, for $\widetilde{G}$. Each terms of $ABS(\widetilde{G})$ denotes the portion of an irredundant formula for the function $f(X)$ consisting of useful conditionally-eliminable prime implicants.

Step 13. Given an arbitrary term of $ABS(\widetilde{G})$, each complemented literal in the term $G_0 P_1' \cdots P_i'$ is associated with a conditionally-eliminable prime implicant in an irredundant formula F which represents a function $f(X)$ belonging to the interval $[g(X), h(X)]$. Generate an irredundant formula representing a function $f(X)$ in $[g(X), h(X)]$ by forming disjunctions of prime implicants consisting of the essential prime implicants and the terms associated with the complemented literals P_i'. Formulate all irredundant formulas by repeating this process for each term of $ABS(\widetilde{G})$.

Just as when using Algorithm 5.1, after generating all irredundant formulas we must determine the cost of each irredundant formula based on a given evaluation criterion. Once each formula is assigned a cost, a least-cost formula F is selected to represent a minimal design for a two-level single-output digital circuit. Example 5.2 demonstrates the application of Algorithm 5.2 to generate all irredundant formulas which represent Boolean functions f belonging to the interval $[g, h]$.

Example 5.2: Given a 1-normal form specification $\phi(X, z) = 1$, from which we have formed the functions

$$g(X) = u'v'wxy'z + u'v'w'x'yz + u'vwx'y \tag{5.29}$$

$$h(X) = u'v'w'yz' + u'v'w'x'z' + u'v'w'x'y + uw'x'y'z' + uvw'x'y' + v'w'x'y'z' + v'wx'y'z + u'v'wxy' + u'v'wy'z + u'vwyz' + vwx'yz + u'vwx'y \tag{5.30}$$

in the interval $[g(X), h(X)]$, we apply Algorithm 5.2 to determine all IDFs representing functions $f(X)$ belonging to $[g(X), h(X)]$.

Step 1. This step was performed by forming $g(X)$ and $h(X)$ from the 1-normal form $\phi(X, z) = 1$.

Step 2. A simplified formula to represent $g(X)$ is given by the right-hand side of (5.29). Moreover, the right-hand side of (5.30) is $BCF(h(X))$.

Step 3. Using Procedure 5.2 to determine essential prime implicants, we find that the prime implicant $u'v'w'x'y$ of $h(X)$ is an essential prime implicant. Hence,

$$H_{ess} = \{u'v'w'x'y\}. \tag{5.31}$$

Additionally,

$$G_{covered} = \{u'v'w'x'yz\}. \tag{5.32}$$

Step 4. After determining the essential prime implicants, the set $\widehat{H}$ is formed consisting of all the prime implicants of $h(X)$ except the essential prime implicants:

$$\widehat{H} = \{u'v'w'yz', u'v'w'x'z', uw'x'y'z', uvw'x'y', v'w'x'y'z', v'wx'y'z, u'v'wxy', u'v'wy'z, u'vwyz', vwx'yz, u'vwx'y\}. \tag{5.33}$$

Using Procedure 5.3 (Covered Terms), $\widehat{H}$, and $h_{ess}(X)$, the terms of $\widehat{H}$ covered by $h_{ess}(X)$ are identified. These terms constitute the set of inessential prime implicants of $h(X)$. In this example,

$$H_{inessen} = \emptyset. \tag{5.34}$$

Step 5. Removing the terms in $G_{covered}$ from G, we form $G - G_{covered}$. Subtracting $h_{ess}(X)$ from the function represented by $G - G_{covered}$, the function $\hat{g}(X)$ is thus formed:

$$\hat{g}(X) = u'v'wxy'z + u'vwx'y. \tag{5.35}$$

Step 6. The set H_{ce} of conditionally-eliminable prime implicants is defined by removing the prime implicants in $H_{inessen}$ from $\widehat{H}$. Since $H_{inessen}$ is equal to the empty set, H_{ce} is equal to $\widehat{H}$. Using Procedure 5.1 (Useful Prime Implicants), $\hat{g}(X)$, and H_{ce}, we determine the prime implicants in H_{ce} which are useful with respect to $\hat{g}(X)$. Five conditionally-eliminable prime implicants are useful; hence,

$$H_{useful} = \{u'v'wxy', u'v'wy'z, u'vwyz', vwx'yz, u'vwx'y\}. \tag{5.36}$$

Step 7. Complementing the function $\hat{g}(X)$, we generate $\widehat{G}'$, a formula which represents $\hat{g}'(X)$:

$$u + w' + vy' + vx + v'z' + v'y + v'x'. \tag{5.37}$$

The set H_{useful} of prime implicants is then simplified relative to $\widehat{G}'$. We thus develop the set of terms

$$T_{useful} = \{xy', y'z, z', vz, v\}. \tag{5.38}$$

Steps 8-10. Having $\hat{g}(X)$ and T_{useful}, we form the system

$$\begin{aligned} G_0 &\leq u'v'wxy'z + u'vwx'y \\ xy' &\leq P_1 \\ y'z &\leq P_2 \\ z' &\leq P_3 \\ vz &\leq P_4 \\ v &\leq P_5 \end{aligned} \tag{5.39}$$

where the following associations are made between labels P_i and prime implicants in H_{useful}:

- P_1 with $u'v'wxy'$,
- P_2 with $u'v'wy'z$,
- P_3 with $u'vwyz'$,
- P_4 with $vwx'yz$, and
- P_5 with $u'vwx'y$.

Given the complement of function $\hat{g}(X)$ represented by $\widehat{G}'$, the equation $f(A, X) = 0$ is formed. A formula representing $f(A, X)$ is

$$\begin{aligned} &G_0u + G_0w' + G_0vy' + G_0vx + G_0v'z' + G_0v'y + G_0v'x' \\ &+ P_1'xy' + P_2'y'z + P_3'z' + P_4'vz + P_5'v. \end{aligned} \tag{5.40}$$

Step 11. Eliminating the X-arguments from $f(A, X) = 0$ using goal-directed elimination, we derive an equation $g(A) = 0$, for which

$$BCF(g(A)) = G_0P_1'P_5' + G_0P_2'P_5' + G_0P_1'P_3'P_4' + G_0P_2'P_3'P_4'. \tag{5.41}$$

Step 12. No actions required.

Step 13. Since $BCF(g(A))$ consists of four terms, there are four irredundant formulas which represent functions $f(X)$ in $[g(X), h(X)]$. The essential prime implicant $u'v'w'x'y$ appears in each irredundant formula. In the first irredundant formula we add the prime implicants associated with P_1 and P_5—the prime implicants $u'v'wxy'$ and $u'vwx'y$, respectively. The remaining irredundant formulas are similarly developed. We thus generate the irredundant formulas which represent $f(X)$:

$$\begin{aligned}
&u'v'w'x'y + u'v'wxy' + u'vwx'y \\
&u'v'w'x'y + u'v'wy'z + u'vwx'y \\
&u'v'w'x'y + u'v'wxy' + u'vwyz' + vwx'yz \\
&u'v'w'x'y + u'v'wy'z + u'vwyz' + vwx'yz.
\end{aligned} \tag{5.42}$$

By any cost criterion, the first two formulas of (5.42) correspond to the cheapest possible implementations of a two-level digital circuit which meets the given specification.

Algorithm 5.2 is more efficient than Algorithm 5.1 for problems of moderate complexity. Nevertheless, as the number of terms of the formula which represents $\hat{g}(X)$ and the number of useful conditionally-eliminable prime implicants of $h(X)$ increase, the time and space required to execute the algorithm become prohibitive. Thus, we seek other means for generating all irredundant formulas which represent a function. Such a technique is described in the next section.

Multiplication Method

A variation of the technique for forming all irredundant formulas discussed in the previous section is to generate sets of implication relations and then perform a multiplicative process. Rather than forming implication relations to represent coverage of a lower bound $g(X)$ of a function with prime implicants of the upper bound $h(X)$, implication relations are formed to represent coverage of each term of $g(X)$ by the prime implicants of $h(X)$. After implications relations are formed for all terms of $g(X)$, a multiplicative process generates the implication relations representing coverage of the function $g(X)$ by sets of prime implicants of $h(X)$.

Suppose a formula G which represents $g(X)$ consists of two terms t_1 and t_2, and $BCF(h(X))$ consists of six prime implicants $p_1, \ldots, p_6$. Also, let us assume that the following irredundant implication relations (IIRs) denote the coverage of terms t_1 and t_2 by subsets of the set $P = \{p_1, \ldots, p_6\}$ of prime implicants:

$$\begin{array}{rcl} t_1 & \leq & p_4 \\ t_1 & \leq & p_1 + p_2 \\ t_1 & \leq & p_1 + p_3 \\ t_2 & \leq & p_1 \\ t_2 & \leq & p_2 + p_4 \\ t_2 & \leq & p_3 + p_5 + p_6. \end{array} \tag{5.43}$$

Since the formula G is the disjunction of terms t_1 and t_2, we can state that

$$g = t_1 + t_2. \tag{5.44}$$

Theorem 2.1 allows us to form implication relations denoting the coverage of $g(X)$ by subsets of P. All possible implication relations representing coverage of $g(X)$ are formed by constructing all possible ways that the sums of t_1 and t_2 are covered by the prime implicants.

An upper bound on the number of irredundant implication relations representing the coverage of $g(X)$ is the product of the number of IIRs for each of its component terms. Since terms t_1 and t_2 each have three irredundant implication relations, an upper bound on the number of IIRs for the function $g(X)$ is three times three or nine. The implication relations for $g(X)$ are formed as follows:

$$\begin{aligned}
t_1 + t_2 &\leq (p_4) + (p_1) \\
t_1 + t_2 &\leq (p_4) + (p_2 + p_4) \\
t_1 + t_2 &\leq (p_4) + (p_3 + p_5 + p_6) \\
t_1 + t_2 &\leq (p_1 + p_2) + (p_1) \\
t_1 + t_2 &\leq (p_1 + p_2) + (p_2 + p_4) \\
t_1 + t_2 &\leq (p_1 + p_2) + (p_3 + p_5 + p_6) \\
t_1 + t_2 &\leq (p_1 + p_3) + (p_1) \\
t_1 + t_2 &\leq (p_1 + p_3) + (p_2 + p_4) \\
t_1 + t_2 &\leq (p_1 + p_3) + (p_3 + p_5 + p_6).
\end{aligned} \tag{5.45}$$

Deleting duplicate prime implicants and ordering terms on the right-hand side of each statement of (5.45), and substituting the symbol g for the sum $t_1 + t_2$ on the left-hand side, we develop the implication relations

$$\begin{aligned}
g &\leq p_1 + p_4 \\
g &\leq p_2 + p_4 \\
g &\leq p_3 + p_4 + p_5 + p_6 \\
g &\leq p_1 + p_2 \\
g &\leq p_1 + p_2 + p_4 \\
g &\leq p_1 + p_2 + p_3 + p_5 + p_6 \\
g &\leq p_1 + p_3 \\
g &\leq p_1 + p_2 + p_3 + p_4 \\
g &\leq p_1 + p_3 + p_5 + p_6.
\end{aligned} \tag{5.46}$$

After forming implication relations for $g(X)$, we observe that not all of the implication relations in (5.46) are irredundant. Deleting the redundant relations, the set of irredundant implication relations representing the coverage of $g(X)$ by subsets of P is

$$\begin{aligned} g &\leq p_1 + p_4 \\ g &\leq p_2 + p_4 \\ g &\leq p_3 + p_4 + p_5 + p_6 \\ g &\leq p_1 + p_2 \\ g &\leq p_1 + p_3. \end{aligned} \tag{5.47}$$

To form the implication relations in (5.47), the techniques discussed in Chapter 3 are used to generate A-consequent terms representing coverage of terms t_1 and t_2 of $g(X)$ by subsets of P. If we use the symbols T_1, T_2 and $P_1, \ldots, P_6$ to denote the labels representing the terms of $g(X)$ and the prime implicants of $h(X)$, respectively, the disjunction of A-consequent terms representing the coverage of t_1 as given by (5.43) is

$$T_1 P_4' + T_1 P_1' P_2' + T_1 P_1' P_3'. \tag{5.48}$$

Similarly, a disjunction of the A-consequent terms representing the coverage of t_2 by prime implicants of $h(X)$ is

$$T_2 P_1' + T_2 P_2' P_4' + T_2 P_3' P_5' P_6'. \tag{5.49}$$

If the literals T_1 and T_2 are divided out of (5.48) and (5.49), respectively, we derive the formulas

$$P_4' + P_1' P_2' + P_1' P_3' \tag{5.50}$$

$$P_1' + P_2' P_4' + P_3' P_5' P_6'. \tag{5.51}$$

Using the unate cross-product operation (Procedure 2.?), the formulas (5.50) and (5.51) are multiplied to derive the formula

$$P_1'P_4' + P_2'P_4' + P_3'P_4'P_5'P_6' + P_1'P_2' + P_1'P_2'P_4' \\ + P_1'P_2'P_3'P_5'P_6' + P_1'P_3' + P_1'P_2'P_3'P_4' + P_1'P_3'P_5'P_6'. \tag{5.52}$$

Terms of (5.52) represent coverage of $g(X)$ as given by the implication relations (5.46). Forming the equivalent absorptive formula for (5.52), we develop the formula

$$P_1'P_4' + P_2'P_4' + P_3'P_4'P_5'P_6' + P_1'P_2' + P_1'P_3'. \tag{5.53}$$

If we appended the literal g to each term of (5.53), we would have A-consequent terms representing the irredundant implication relations (5.47). Hence, after forming A-consequent terms to denote coverage of each term of $g(X)$ by prime implicants of $h(X)$, product and absorption operations are performed to form the IIRs representing coverage of $g(X)$ by the prime implicants of $h(X)$.

Previous Work. A methodology similar to the one described above has been applied in various algorithms in the literature developed for the purpose of developing all irredundant formulas for a function. The primary differences among the algorithms are the form of the formula G which represents the function $g(X)$ and the method for determining the coverage of terms in G by subsets of the prime implicants. For example, one form for G is the minterm expansion, $MCF(g(X))$. Petrick used this form to develop for each minterm of $g(X)$ a formula, similar to (5.50), which is an alterm denoting the prime implicants of $h(X)$ which cover the minterm (Petri 56). The conjunction of the set of alterms—one for each minterm—is called a *Petrick function*. The set of terms of G is called the *base* of interval $[g(X), h(X)]$. Hence, the base used by Petrick is the minterm canonical form, $MCF(g(X))$, of the function. We call a formula, such as (5.50), denoting the coverage of a

term of the base by subsets of prime implicants an *inclusion formula*, since the associated term is included in the subsets of the prime implicants denoted by each term in the formula.

We summarise the steps followed by most methods for forming all irredundant formulas which may represent a function:

1. form the set of prime implicants of h;
2. develop a base for $[g, h]$;
3. develop inclusion formulas representing coverage of the terms of the base by prime implicants of h; and
4. form the product of the inclusion formulas.

Once the product of the inclusion formulas is formed, all absorbed terms are deleted. Each term of the resulting formula denotes an irredundant formula F which represents a function f in the interval $[g, h]$.

A key problem in minimization theory is to devise a base for $[g(X), h(X)]$ and a corresponding method for forming inclusion formulas that is efficient. In addition to the minterm canonical form used by Petrick, other bases have been used in minimization theory. The Blake canonical form of a function was used in (Ghaza 57), (Mott 60), (Gaine 64), and (Tison 67). Chang and Mott (Chang 65) employed an irredundant disjunctive form of a function as a base. Reusch showed that an arbitrary disjunctive form may be used as a base for a function (Reusc 75). A subset of the minterm canonical form of a function called the *abridged minterm base* was devised by Cutler (Cutle 80). Hong used a subset of the abridged minterm base that he called the "epi-eliminated" minterm base (Hong 91); the epi-eliminated minterm base contains only the minterms of the abridged minterm base which are not covered by essential prime implicants of the function. We now present a discussion of several of these methods.

One of the first techniques for generating inclusion formulas was developed by Ghazala for use in an algorithm for developing all irredundant formulas representing a completely-specified

function f (Ghasa 57). Ghazala used the Blake canonical form of a function as the base and generated formulas denoting the coverage of each term by subsets of the prime implicants. He called the conjunction of the inclusion formulas the *presence function* of f. The formula derived by computing the conjunction and forming the equivalent absorptive formula denotes all of the irredundant SOP formulas which represent a completely-specified function f. For example, if the formula

$$P_1 \cdot (P_2 + P_3P_4) \cdot (P_3 + P_1P_4) \cdot P_4 \tag{5.54}$$

is the presence function, then computing the conjunction yields the formula

$$P_1P_2P_3P_4 + P_1P_3P_4 + P_1P_2P_4 + P_1P_3P_4. \tag{5.55}$$

The equivalent absorptive formula for (5.55) is

$$P_1P_3P_4 + P_1P_2P_4. \tag{5.56}$$

Hence, there are two irredundant formulas which represent the function.

As a vehicle for generating inclusion formulas, Ghazala developed a matrix that he called a *ϕ-chart*. The rows in the ϕ-chart are associated with members of the base; the columns are associated with prime implicants used to cover terms in the base. Since the base is composed of the n prime implicants of the function, there are n rows and n columns in the ϕ-chart. Entries in the chart are derived by forming the *ratio* ϕ_i/ϕ_j, in which ϕ_i and ϕ_j are prime implicants of the function.[2] Hence, a ϕ-chart is formed as shown in Table 5.1. Ghazala did not fill in the main diagonal, in which each entry is equal to 1.

[2]When prime implicants are both members of a base, i.e., terms to be covered, as well as terms which do the covering, we shall refer to the prime implicants of the base by the j subscript and the prime implicants which do the covering by the i subscript.

	ϕ_1	ϕ_2	$\cdots$	ϕ_i	$\cdots$	ϕ_n
ϕ_1	-	ϕ_2/ϕ_1	$\cdots$	ϕ_i/ϕ_1	$\cdots$	ϕ_n/ϕ_1
ϕ_2	ϕ_1/ϕ_2	-	$\cdots$	ϕ_i/ϕ_2	$\cdots$	ϕ_n/ϕ_2
$\vdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
ϕ_j	ϕ_1/ϕ_j	ϕ_2/ϕ_j	$\cdots$	ϕ_i/ϕ_j	$\cdots$	ϕ_n/ϕ_j
$\vdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
ϕ_n	ϕ_1/ϕ_n	ϕ_2/ϕ_n	$\cdots$	ϕ_i/ϕ_n	$\cdots$	-

Table 5.1. Formation of a ϕ-Chart

If the entries in the j-th row of the ϕ-chart do not sum to 1, then prime implicant ϕ_j is an essential prime implicant. Sum-to-one subsets of the entries in the j-th row denote coverage of prime implicant ϕ_j by subsets of prime implicants other than ϕ_j (ϕ_j always covers itself). Ghazala presented a method called *cracking* for determining such subsets from the ϕ-chart. Example 5.3 presents an example taken from (Ghaza 57) which illustrates a ϕ-chart for a function and the inclusion formulas developed from it.

Example 5.3: Given a function f, for which $BCF(f)$ is defined by the equation

$$BCF(f) = d'e + cde' + a'cd + a'ce + ab'd + ab'e + b'cd + b'ce, \tag{5.57}$$

the ϕ-chart for f is given by Table 5.2.

The entries in the rows corresponding to prime implicants $d'e, cde'$, and $ab'd$ do not sum to one, hence, these terms are essential prime implicants of f. The inclusion formulas denoting coverage of the essential prime implicants are P_1, P_2, and P_5, respectively.

In the row corresponding to $a'cd$, the entries in columns two and four sum to one, i.e., $e' + e = 1$. Thus, the prime implicants from columns two and four, cde' and $a'ce$, combine to cover $a'cd$. No other sum-to-one combinations can be formed for the row. Denoting $a'cd$ by the label P_3, the inclusion formula thus representing the coverage of $a'cd$ is $P_3 + P_2P_4$.

Similarly, the following inclusion formulas represent the coverage of the remaining prime implicants:

- $a'ce$: $P_4 + P_1P_3$;
- $ab'e$: $P_6 + P_1P_5$;
- $b'cd$: $P_7 + P_3P_5 + P_2P_8 + P_2P_3P_6 + P_2P_4P_5 + P_2P_4P_6$; and
- $b'ce$: $P_8 + P_1P_7 + P_4P_6 + P_1P_3P_5 + P_1P_4P_5 + P_1P_3P_6$.

	$d'e$	cde'	$a'cd$	$a'ce$	$ab'd$	$ab'e$	$b'cd$	$b'ce$
$d'e$	-	0	0	$a'c$	0	ab'	0	$b'c$
cde'	0	-	a'	0	ab'	0	b'	0
$a'cd$	0	e'	-	e	0	0	b'	$b'e$
$a'ce$	d'	0	d	-	0	0	$b'd$	b'
$ab'd$	0	ce'	0	0	-	e	c	ce
$ab'e$	d'	0	0	0	d	-	cd	c
$b'cd$	0	e'	a'	$a'e$	a	ae	-	e
$b'ce$	d'	0	$a'd$	a'	ad	a	d	-

Table 5.2. ϕ-Chart for Example 5.3

Tison presented a consensus-based technique for forming inclusion formulas (Tison 67). Given the Blake canonical form for a function, labels are affixed to each prime implicant to create a formula. Then for each binate variable in the formula, all possible consensus terms are created and added to the formula. If a new term is absorbed, then it is deleted. After consensus operations are performed for all binate variables, the resulting formula yields the inclusion formulas. Example 5.4 demonstrates Tison's method with an example taken from (Tison 67).

Example 5.4: Given a function f, where $BCF(f)$ is defined by the equation

$$BCF(f) = ax' + ay + bx' + by + bz + x'y' + x'z' + xy + yz', \tag{5.58}$$

we associate labels with the prime implicants of the functions as follows:

- ax' : P_1;
- ay : P_2;
- bx' : P_3;
- by : P_4;
- bz : P_5;
- $x'y'$: P_6;
- $x'z'$: P_7;
- xy : P_8; and
- yz' : P_9.

We thus generate the formula

$$ax'P_1 + ayP_2 + bx'P_3 + byP_4 + bzP_5 + x'y'P_6 + x'z'P_7 + xyF_8 + yz'P_9. \tag{5.59}$$

Since x, y, and z are the binate variables in (5.59), consensus operations are performed with respect to these variables. First, all consensus terms are formed with respect to x:

- ayP_1P_8 is the consensus of $ax'P_1$ and xyP_8;
- byP_3P_8 is the consensus of $bx'P_3$ and xyP_8; and
- $yz'P_7P_8$ is the consensus of $x'z'P_7$ and xyP_8.

We add the new terms to (5.59) to form

$$\begin{aligned} &ax'P_1 + ayP_2 + bx'P_3 + byP_4 + bzP_5 + x'y'P_6 + x'z'P_7 \\ &+ xyP_8 + yz'P_9 + ayP_1P_8 + byP_3P_8 + yz'P_7P_8. \end{aligned} \tag{5.60}$$

All consensus terms are then formed with respect to y:

- $ax'P_2P_6$ is the consensus of ayP_2 and $x'y'P_6$;
- $bx'P_4P_6$ is the consensus of byP_4 and $x'y'P_6$;
- $x'z'P_6P_9$ is the consensus of $x'y'P_6$ and $yz'P_9$;
- $ax'P_1P_6P_8$ is the consensus of ayP_1P_8 and $x'y'P_6$;
- $bx'P_3P_6P_8$ is the consensus of byP_3P_8 and $x'y'P_6$; and
- $x'z'P_6P_7P_8$ is the consensus of $x'y'P_6$ and $yz'P_7P_8$.

When the new consensus terms are added to (5.60), we find that terms $ax'P_1P_6P_8$, $bx'P_3P_6P_8$, and $x'z'P_6P_7P_8$ are absorbed by other terms. Hence, these terms are deleted. The formula which results is

$$\begin{aligned} &ax'P_1 + ayP_2 + bx'P_3 + byP_4 + bzP_5 + x'y'P_6 + x'z'P_7 + xyP_8 + yz'P_9 \\ &+ ayP_1P_8 + byP_3P_8 + yz'P_7P_8 + ax'P_2P_6 + bx'P_4P_6 + x'z'P_6P_9. \end{aligned} \tag{5.61}$$

All consensus terms are then formed with respect to z. The resulting terms are $bx'P_5P_7$, byP_5P_9, $byP_5P_7P_8$, and $bx'P_5P_6P_9$. These terms are added to (5.61) to generate the formula which yields the inclusion formulas:

$$\begin{aligned} &ax'P_1 + ayP_2 + bx'P_3 + byP_4 + bzP_5 + x'y'P_6 + x'z'P_7 \\ &+ xyP_8 + yz'P_9 + ayP_1P_8 + byP_3P_8 + yz'P_7P_8 + ax'P_2P_6 \\ &+ bx'P_4P_6 + x'z'P_6P_9 + bx'P_5P_7 + byP_5P_9 + byP_5P_7P_8 + bx'P_5P_6P_9. \end{aligned} \tag{5.62}$$

To form the inclusion formula for prime implicant ax', the labels in the terms in (5.62) which contain the literals ax' are summed. Since terms $ax'P_1$ and $ax'P_2P_6$ contain ax', the inclusion formula for ax' is $P_1 + P_2P_6$. Inclusion formulas for the prime implicants are as follows:

- $ax' : P_1 + P_2P_6$;
- $ay : P_2 + P_1P_8$;

- bx' : $P_3 + P_4P_6 + P_5P_7 + P_5P_6P_9$;
- by : $P_4 + P_3P_8 + P_5P_9 + P_5P_7P_8$;
- bz : P_5;
- $x'y'$: P_6;
- $x'z'$: $P_7 + P_6P_9$;
- xy : P_8; and
- yz' : $P_9 + P_7P_8$.

We observe that the formula from which the inclusion formulas are generated, e.g., (5.62), is nothing more than the Blake canonical form of the function represented by the formula (5.59) derived by affixing labels to the prime implicants of the original function. The successive consensus operations are an organized way of performing the technique of iterated consensus to develop a Blake canonical form.

Reusch showed that any disjunctive form which represents a function may be used as a base for the development of inclusion formulas (Reusc 75). He also introduced a modified version of Ghasala's ϕ-chart in which the rows are associated with n prime implicants of the function and the columns are associated with m terms in the base. Entries in the chart are derived by forming p_i/t_j, the division of prime implicant p_i by the term t_j (Table 5.3). Sum-to-one subsets of the entries in the j- th column correspond to subsets of the prime implicants which cover the j-th term of the base. We will use Reusch's form of the ϕ-chart in the remainder of this work.

	t_1	t_2	$\cdots$	t_j	$\cdots$	t_m
p_1	p_1/t_1	p_1/t_2	$\cdots$	p_1/t_j	$\cdots$	p_1/t_m
p_2	p_2/t_1	p_2/t_2	$\cdots$	p_2/t_j	$\cdots$	p_2/t_m
$\vdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
p_i	p_i/t_1	p_i/t_2	$\cdots$	p_i/t_j	$\cdots$	p_i/t_m
$\vdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
p_n	p_n/t_1	p_n/t_2	$\cdots$	p_n/t_j	$\cdots$	p_n/t_m

Table 5.3. Formation of Reusch's ϕ-Chart

The emphasis of the foregoing discussion has been placed on the development of inclusion formulas for bases corresponding to single functions f. However, several of the processes either handle intervals $[g, h]$ or may be extended to do so. We now examine Tison's extension for intervals.

Tison's method for developing inclusion formulas for intervals adds several steps to the process outlined for functions. Similar to his method for functions, labels are affixed to the prime implicants of the upper-bound function h and a Blake canonical form is generated. However, prior to this process, all prime implicants which are useless with respect to the lower-bound function g are deleted. The formula which results is used in combination with the terms of the formula G which represents g to form inclusion formulas denoting the coverage of terms in G by subsets of the prime implicants of h. The conjunction of the inclusion formulas derived for each term in G yields the set of irredundant formulas which represent functions belonging to the interval. We demonstrate Tison's method with an example taken from (Tison 67).

Example 5.5: Suppose we are given an interval $[g, h]$,

$$g = abd' + a'bd + a'b'c + a'cd \tag{5.63}$$
$$h = ab' + ac + ad' + a'bc' + a'bd + b'c + b'd' + cd + c'd'. \tag{5.64}$$

The right-hand side of (5.64) is the Blake canonical form for h.

The prime implicant ab' of h is useless, since $g \cdot ab' = 0$. All other prime implicants are useful.

Using the remaining prime implicants of h, a formula

$$acP_1 + ad'P_2 + a'bc'P_3 + a'bdP_4 + b'cP_5 + b'd'P_6 + cdP_7 + c'd'P_8 \tag{5.65}$$

similar to (5.59) is developed. Using (5.65), we develop the Blake canonical form:

$$\begin{aligned} &acP_1 + ad'P_2 + a'bc'P_3 + a'bdP_4 + b'cP_5 + b'd'P_6 + cdP_7 + c'd'P_8 \\ &+bcdP_1P_4 + bc'd'P_2P_3 + a'c'd'P_3P_6 + a'cdP_4P_5 + cdP_1P_4P_5 + c'd'P_2P_3P_6 \\ &+ad'P_1P_8 + b'd'P_5P_8 + a'bdP_3P_7 + a'bc'P_4P_8 + acP_2P_7 + b'cP_6P_7. \end{aligned} \tag{5.66}$$

Terms on the right-hand side of (5.63) are used in combination with (5.66) to form inclusion formulas for the terms in G. Inclusion formulas are developed as follows:

- abd' : Only terms $ad'P_2$ and $ad'P_1P_8$ in (5.66) contain a subset of the literals of abd'. Thus, an inclusion formula denoting the coverage of term abd' is $P_2 + P_1P_8$.

- $a'bd$: Terms $a'bdP_4$ and $a'bdP_3P_7$ contain a subset of the literals of $a'bd$. An inclusion formula denoting the coverage of term $a'bd$ is $P_4 + P_3P_7$.
- $a'b'c$: Terms $b'cP_5$ and $b'cP_6P_7$ contain a subset of the literals of $a'b'c$. An inclusion formula denoting the coverage of term $a'b'c$ is $P_5 + P_6P_7$.
- $a'cd$: Terms cdP_7 and $cdP_1P_4P_5$ contain a subset of the literals of $a'cd$. Additionally, term $a'cdP_4P_5$ contains a subset of literals of $a'cd$. Hence, an inclusion formula denoting the coverage of the term is $P_7 + P_1P_4P_5 + P_4P_5$. However, since term $P_1P_4P_5$ is absorbed by P_4P_5, we simplify the formula to form $P_7 + P_4P_5$.

Computing the conjunction

$$(P_2 + P_1P_8)(P_4 + P_3P_7)(P_5 + P_6P_7)(P_7 + P_4P_5) \tag{5.57}$$

of the inclusion formulas yields a formula which denotes the set of irredundant formulas F which represent a function f in the interval $[g, h]$.

Multiplication Algorithm. In the last section we surveyed a number of algorithms found in the literature. A number of different bases as well as techniques for developing inclusion formulas are used in these methods. In this section we present a new base for a interval $[g, h]$. Procedure 3.6 in Chapter 3 is then used to develop inclusion formulas denoting coverage of terms in our base by subsets of prime implicants of h.

We desire a base which consists of a small number of terms. This is important because for each term in the base, we develop an inclusion formula which denotes the coverage of each term in the base by subsets of the PIs of h. All inclusion formulas are then multiplied together to derive a formula similar to (5.52). After absorption, a formula is derived which denotes the irredundant formulas representing functions f belonging to the interval $[g, h]$. The fewer the terms in the base, the fewer the inclusion formulas that have to be multiplied together to derive the expression representing irredundant formulas. Thus, a base consisting of a small number of terms requires less work than a base with a large number of terms to develop the set of irredundant formulas representing functions in the interval $[g, h]$.

A base that we propose is a simplified formula $\widehat{G}$ representing the portion of the lower bound $g(X)$ of the interval $[g(X), h(X)]$ that is not covered by essential prime implicants of $h(X)$, i.e, the function $\hat{g}(X)$ defined by the equation

$$\hat{g}(X) = g(X) - h_{ess}(X). \tag{5.68}$$

Using this base, inclusion formulas are generated to indicate the coverage of terms in $\widehat{G}$ by useful, conditionally-eliminable prime implicants of $h(X)$. A multiplicative process then is used to develop subsets denoting the coverage of $\hat{g}(X)$ by sets of the CEPIs. The addition of the essential prime implicants to each irredundant set of conditionally-eliminable prime implicants which covers $\hat{g}(X)$ yields a set of PIs the sum of which is an irredundant formula representing a function f belonging to the interval $[g(X), h(X)]$.

An algorithm is now presented for generating all irredundant formulas representing functions in $[g, h]$ based on the use of a multiplicative process. When developing the inclusion formula for each t in $\widehat{G}$, we must use the same labels P_i for corresponding prime implicants of $h(X)$. After all inclusion formulas are formed, the formulas are multiplied together to generate a formula denoting the coverage of $\hat{g}(X)$ by subsets of the set of CEPIs. Since the inclusion formulas are collectively unate, i.e., they consist of the same variables all of which are unate, the most efficient technique for multiplying the formulas together is the unate cross-product operation implemented by Procedure 2.3. Furthermore, after each product operation the resulting formula is made absorptive to eliminate absorbed terms. Because the algorithm is multiplicative, we it the *Multiplication Method.*

Algorithm 5.3 (Multiplication Method): Given a 1-normal form specification $\phi(X, z) = 1$, all irredundant formulas which represent Boolean functions $f(X)$ belonging to the interval $[g(X), h(X)]$ developed from $\phi(X, z) = 1$ are formed in the following manner:

Step 1.

1. Form $g(X) = \phi'(X, 0) \cdot \phi(X, 1)$.
2. Form $h(X) = \phi'(X, 0) + \phi(X, 1)$.

Step 2.

1. Form a simplified formula to represent $g(X)$ using Procedure 2.15 (Simplification). Call the simplified formula G.
2. Develop the Blake canonical form for function $h(X)$ using Procedure 2.20 (Blake canonical form).

Step 3. Using Procedure 5.2 (Essential Prime Implicants), G, and $BCF(h(X))$, determine the essential prime implicants of $h(X)$.

1. Denote the set of essential prime implicants by $H_{ess}(X)$ and the function formed by the disjunction of the essential prime implicants by $h_{ess}(X)$.
2. Denote the set of terms in G used to identify essential prime implicants in Procedure 5.2—terms covered by the essential prime implicants—by $G_{covered}$.

Step 4.

1. Form a set $\widehat{H}$ of prime implicants consisting of all prime implicants of $h(X)$ except the essential prime implicants.
2. Use Procedure 5.3 (Covered Terms), $\widehat{H}$, and $h_{ess}(X)$ to determine the terms in $\widehat{H}$ covered by $h_{ess}(X)$. These terms comprise the set of inessential prime implicants of $h(X)$; call this set of terms $H_{inessen}$.

Step 5.

1. Remove from G the terms in $G_{covered}$; call the resulting formula $G - G_{covered}$.
2. Using Procedure 2.10 (Subtraction), subtract the function $h_{ess}(X)$ from the function represented by $G - G_{covered}$.
3. Call the resulting formula $\widehat{G}$ and the function which it represents $\hat{g}(X)$.

Step 6. Form the set H_{ce} of conditionally-eliminable prime implicants by removing the prime implicants in $H_{inessen}$ from $\widehat{H}$.

Step 7. For each term $t(X)$ in $\widehat{G}$:

1. Using Procedure 5.1 (Useful Prime Implicants), determine which prime implicants in set H_{ce} are useful with respect to $t(X)$. Call the set of useful prime implicants H_{useful}.
2. Using Procedure 3.6 (Coverage of a Term), t, and H_{useful}, determine minimal subsets of H_{useful} which cover t. The resulting formula denotes coverage of t by minimal subsets of H_{useful}.

Notes:

- The same labels $\{P_1, \ldots, P_k\}$ must be associated with prime implicants in H_{ce} for every term t in $\widehat{G}$.
- Step 9 of Procedure 3.6 is not executed; we use the formula resulting after Step 8 of the procedure.

Call the set of formulas which result from this step P.

Step 8. Initialize an accumulator ACC to be equal to the term 1.

Step 9.

- If P is empty, then all formulas generated in Step 7 have been multiplied together; the resulting formula is contained in ACC. Continue to Step 11.
- Otherwise, continue to Step 10.

Step 10.

1. Remove the first formula from P and call it Q.
2. Using Procedure 2.3 (Unate Cross-Product), form the product of formulas Q and ACC.
3. Make absorptive the formula resulting from $Q \times ACC$.
4. Replace the contents of ACC with $ABS(Q \times ACC)$.
5. Return to Step 9.

Step 11. Formula ACC indicates the coverage of $\hat{g}(X)$ by minimal subsets of the set H_{ce} of conditionally-eliminable prime implicants of $h(X)$. Irredundant formulas which represent functions $f(X)$ belonging to the interval $[g(X), h(X)]$ are formed by combining each minimal subset of the set H_{ce} which covers $\hat{g}(X)$ with the set H_{essen} of essential prime implicants of $h(X)$.

Once Procedure 3.6 is used in Step 7 of Algorithm 5.3 to determine minimal subsets of the conditionally-eliminable prime implicants of $h(X)$ which cover terms in $\widehat{G}$, we develop an upper bound on the number of irredundant formulas for functions $f(X)$ belonging to the interval $[g(X), h(X)]$. This upper bound is calculated by:

1. determining for each term t in $\widehat{G}$ the number of minimal subsets of the prime implicants of $h(X)$ which cover the term, i.e., the number of terms in the associated inclusion formula; and
2. forming the product of the resulting numbers.

The number of irredundant formulas which may represent a function often is only a fraction of the upper bound. In some cases, however, the number of irredundant formulas is very close to the upper bound.

Example 5.6 demonstrates the application of Algorithm 5.3 to generate all irredundant formulas for the interval of Example 5.2.

Example 5.6: Given a 1-normal form specification $\phi(X, z) = 1$, from which we have formed the functions

$$g(X) = u'v'wxy'z + u'v'w'x'yz + u'vwx'y \tag{5.69}$$

$$h(X) = u'v'w'yz' + u'v'w'x'z' + u'v'w'x'y + uw'x'y'z' + uvw'x'y' + v'w'x'y'z' + v'wx'y'z + u'v'wxy' + u'v'wy'z + u'vwyz' + vwx'yz + u'vwx'y, \tag{5.70}$$

in the interval $[g(X), h(X)]$, we apply Algorithm 5.3 to determine all IDFs which represent functions $f(X)$ belonging to the interval $[g(X), h(X)]$.

Step 1. This step was performed by forming $g(X)$ and $h(X)$ from the 1-normal form $\phi(X, z) = 1$.

Step 2. A simplified formula to represent $g(X)$ is given by the right-hand side of (5.69). Moreover, the right-hand side of (5.70) is $BCF(h(X))$.

Step 3. Using Procedure 5.2 to determine essential prime implicants, we find that the prime implicant $u'v'w'x'y$ of $h(X)$ is an essential prime implicant. Hence,

$$H_{ess} = \{u'v'w'x'y\}. \tag{5.71}$$

Additionally,

$$G_{covered} = \{u'v'w'x'yz\}. \tag{5.72}$$

Step 4. After determining the essential prime implicants, the set $\widehat{H}$ of prime implicants is formed which consists of all the prime implicants of $h(X)$ except the essential prime implicants. The set $\widehat{H}$ is defined as follows:

$$\widehat{H} = \{u'v'w'yz', u'v'w'x'z', uw'x'y'z', uvw'x'y', v'w'x'y'z', v'wx'y'z, u'v'wxy', u'v'wy'z, u'vwyz', vwx'yz, u'vwx'y\}. \tag{5.73}$$

Using Procedure 5.3 (Covered Terms), $\widehat{H}$, and $h_{ess}(X)$, the terms of $\widehat{H}$ covered by $h_{ess}(X)$ are identified. These terms comprise the set of inessential prime implicants of $h(X)$. In this example,

$$H_{inessen} = \emptyset. \tag{5.74}$$

Step 5. Removing the terms in $G_{covered}$ from G, we form $G - G_{covered}$. Subtracting $h_{ess}(X)$ from the function represented by $G - G_{covered}$, the function $\hat{g}(X)$ is thus formed:

$$\hat{g}(X) = u'v'wxy'z + u'vwx'y. \tag{5.75}$$

Step 6. The set of conditionally-eliminable prime implicants, H_{ce}, is defined by removing the prime implicants in $H_{inessen}$ from $\widehat{H}$. Since $H_{inessen}$ is equal to empty set, H_{ce} is equal to $\widehat{H}$, i.e.,

$$H_{ce} = \{u'v'w'yz', u'v'w'x'z', uw'x'y'z', uvw'x'y', v'w'x'y'z', v'wx'y'z, u'v'wxy', u'v'wy'z, u'vwyz', vwx'yz, u'vwx'y\}. \tag{5.76}$$

Step 7. For each term of the base, $\widehat{G}$, we use Procedure 3.6 (Coverage of a Term) to develop a formula which indicates the coverage of the term by subsets of H_{ce}. Prior to applying Procedure 3.6, we determine for each term the elements of H_{ce} which are useful with respect to the term.

For term $u'v'wxy'z$, the useful conditionally-eliminable prime implicants are

$$\{u'v'wxy', u'v'wy'z\}. \tag{5.77}$$

The formula returned by Procedure 3.6 denoting the coverage of $u'v'wxy'z$ by subsets of (5.77) is

$$P_7 + P_8. \tag{5.78}$$

The subscripts of the terms in (5.78) denote the respective prime implicants in (5.76) which cover $u'v'wxy'z$. Hence, the term may be covered either by the seventh or by the eighth element of (5.76), i.e.,

$$\begin{aligned} u'v'wxy'z &\leq u'v'wxy' \\ u'v'wxy'z &\leq u'v'wy'z. \end{aligned} \tag{5.79}$$

Similarly, for term $u'vwx'y$, the useful conditionally-eliminable prime implicants are:

$$\{u'vwyz', vwx'yz, u'vwx'y\}. \tag{5.80}$$

The formula returned by Procedure 3.6 denoting the coverage of $u'v'wxy'z$ by subsets of (5.80) is

$$P_{11} + P_9P_{10}. \tag{5.81}$$

Hence, the term $u'vwx'y$ may be covered either by the eleventh element of (5.76) or by a combination of the ninth and tenth elements, i.e.,

$$\begin{aligned} u'vwx'y &\leq u'vwx'y \\ u'vwx'y &\leq u'vwyz' + vwx'yz. \end{aligned} \tag{5.82}$$

Steps 8-10. Since there are two terms in each of the formulas (5.78) and (5.81), an upper bound on the number of irredundant formulas for $f(X)$ is four. We develop a formula which indicates the conditionally-eliminable prime implicants in each irredundant formula for $f(X)$ by multiplying (5.78) and (5.81). The resulting formula is

$$P_7P_{11} + P_7P_9P_{10} + P_8P_{11} + P_8P_9P_{10}. \tag{5.83}$$

Step 11. Since (5.83) consists of four terms, there are four irredundant formulas for $f(X)$. Each formula is formed by adding the conditionally-eliminable prime implicants denoted by each term of (5.83) to the essential prime implicant $u'v'w'x'y$. In the first irredundant formula we add the prime implicants associated with P_7 and P_{11}—the prime implicants $u'v'wxy'$ and $u'vwx'y$, respectively. The remaining irredundant formulas are similarly developed. We thus generate the irredundant formulas which represent functions in $[g(X), h(X)]$:

$$
\begin{aligned}
&u'v'w'x'y + u'v'wxy' + u'vwx'y \\
&u'v'w'x'y + u'v'wxy' + u'vwyz' + vwx'yz \\
&u'v'w'x'y + u'v'wy'z + u'vwx'y \\
&u'v'w'x'y + u'v'wy'z + u'vwyz' + vwx'yz.
\end{aligned} \tag{5.84}
$$

Computational Results

In this section we compare the execution times for Algorithms 5.1, 5.2, and 5.3 on several sets of example functions and intervals. Data on each set of functions is listed in Appendix B. Each algorithm is programmed in the Scheme dialect of the LISP programming language. The implementation of Scheme used is PC Scheme, a version which runs on IBM-compatible computers. The computer used to produce the results was a 20 MHz, 80386-based, IBM-compatible computer. PC Scheme was run as a task in the Microsoft Windows environment. Of particular significance is the fact that PC Scheme uses only 640K of computer memory; an implementation of Scheme hosted on a workstation or a minicomputer should yield results in some cases where an implementation on a personal computer will not.

Table 5.4 contains the execution time for each algorithm on Data Set B, a set of completely-specified functions. The times listed in the tables are given in the format:

`minutes:seconds.hundredths of a second.`

The times listed for each algorithm include the time required to generate the Blake canonical form for each function as well as to develop a simplified formula to represent the function (step 1 of each algorithm). We include as a separate entry the time required to perform these calculations in order to illuminate the portion of computational effort devoted to these tasks. The number of terms in $\widehat{G}$ is pertinent to both Algorithms 5.2 and 5.3. The "upper bound" entry, generated by Algorithm 5.3, is the projected number of irredundant formulas based on the number of terms in the inclusion formulas. The "number IDFs" column lists the actual number of irredundant formulas produced

by the algorithms. An entry of M denotes that the procedure ran out of memory for the respective function. An entry of - denotes that we were unable to attain a result for the given item.

Function Identifier	Time BCF/Simp	Alg 5.1	Alg 5.2	Alg 5.3	No Terms $\widehat{G}$	Upper Bound	Number IDFs
B1	0.11	0.66	0.44	1.04	0	1	1
B2	0.27	1.38	0.61	1.21	0	1	1
B3	0.66	4.06	1.32	1.92	0	1	1
B4	2.09	15.11	5.49	5.77	1	3	3
B5	5.55	M	3:21.19	49.21	7	1152	192
B6	22.25	M	M	M	43	1.95×10^{19}	-
B7	1:02.41	M	M	M	54	9.91×10^{23}	-

Table 5.4. Data Set B (Results)

Algorithms 5.2 and 5.3 were able to produce a result for function B5, a function for which Algorithm 5.1 ran out of memory. Algorithms 5.2 and 5.3 produced results more quickly than Algorithm 5.1 for all functions except the simplest one (B1). Algorithm 5.2 produced a result more quickly than Algorithm 5.3 for simple functions; Algorithm 5.2 was better for B5 which has many irredundant formulas. In cases where the upper bound on the number of formulas is in the millions or greater, none of the algorithms is able to produce a result without exhausting memory.

Table 5.5 contains the results for each algorithm for Data Set C, a set of completely-specified functions. The results are similar to those produced using Data Set B.

Function Identifier	Time BCF/Simp	Alg 5.1	Alg 5.2	Alg 5.3	No Terms $\widehat{G}$	Upper Bound	Number IDFs
C1	2.74	40.43	4.78	6.32	0	1	1
C2	7.85	2:38.78	23.46	21.75	1	2	2
C3	21.19	9:18.10	1:10.47	1:08.82	0	1	1
C4	34.71	M	2:16.27	2:06.99	3	8	4
C5	2:33.52	M	M	M	17	2.18×10^{15}	-

Table 5.5. Data Set C (Results)

Table 5.6 contains the results for each algorithm for Data Set D, a set of completely-specified functions. Algorithm 5.1 produced a result only for functions D1 and D2; for other functions, Algorithm 5.1 exhausted memory before yielding a result. Algorithm 5.2 and 5.3 work about equally as well when the respective function is represented by a single irredundant formula. In functions which have only one irredundant formula, the associated formula consists only of essential prime implicants. All other prime implicants are inessential.

Function Identifier	Time BCF/Simp	Alg 5.1	Alg 5.2	Alg 5.3	No Terms $\widehat{G}$	Upper Bound	Number IDFs
D1	3.73	55.14	4.78	5.77	0	1	1
D2	12.30	7:42.53	18.95	20.00	0	1	1
D3	30.00	M	51.57	51.85	0	1	1
D4	35.92	M	1:19.37	1:16.56	0	1	1
D5	1:17.18	M	2:52.79	2:48.67	0	1	1
D6	1:40.68	M	4:44.29	4:34.47	0	1	1
D7	2:07.86	M	6:24.53	6:04.27	0	1	1
D8	3:47.01	M	12:18.96	2:02.11	0	1	1
D9	11:36.78	M	M	M	2	4	4
D10	7:07.93	M	29:45.84	28:31.75	0	1	1
D11	10:51.59	M	M	M	1	3	3
D12	11:52.99	M	M	M	0	1	1

Table 5.6. Data Set D (Results)

Table 5.7 contains the results for each algorithm for Data Set IC, a set of intervals. Typically, Algorithms 5.2 and 5.3 compute a result faster than Algorithm 5.1. Algorithm 5.3 is able to produce a result for IC10, an interval for which the other procedures exhausted memory. Additionally, Algorithm 5.3 exhausts memory during the process of multiplying the inclusion formulas representing coverage of the terms of $\widehat{G}$; hence, an upper bound on the number of irredundant formulas is calculated prior to exhausting memory.

In general, Algorithms 5.2 and 5.3 execute more quickly than Algorithm 5.1 and are able to produce results in many cases where Algorithm 5.1 exhausts memory. Algorithm 5.2 seems to

Function Identifier	Time BCF/Simp	Alg 5.1	Alg 5.2	Alg 5.3	No Terms $\widehat{G}$	Upper Bound	Number IDFs
IC1	0.28	1.49	0.71	1.27	0	1	1
IC2	0.44	2.36	1.93	2.14	2	4	4
IC3	0.61	2.75	2.03	2.15	1	3	3
IC4	1.43	6.92	3.52	3.84	1	2	2
IC5	1.81	27.62	19.28	12.36	7	1200	48
IC6	2.48	31.47	12.64	8.74	4	54	24
IC7	3.85	30.76	12.25	13.41	2	4	4
IC8	7.75	1:27.61	29.77	26.04	3	12	12
IC9	11.64	M	M	M	23	2.37×10^{11}	-
IC10	11.54	M	M	28:38.89	10	2304	1728
IC11	24.06	M	M	M	19	5.97×10^{8}	-
IC12	40.37	M	M	M	60	2.11×10^{33}	-

Table 5.7. Data Set IC (Results)

be the best procedure to use for relatively simple functions. Algorithm 5.3 computes results more quickly than Algorithm 5.2 for moderately complex functions.

Summary

None of the algorithms presented in this chapter is able to produce a result in cases where an upper bound on the number of irredundant formulas is more than about two thousand. In these situations, we must be satisfied with finding a single irredundant formula which is minimal with respect to a given cost criterion.

We summarise the new ideas presented in this chapter:

- A methodology for partitioning of the prime implicants was introduced which allows concentration of effort on determining the useful conditionally-eliminable prime implicants which will appear in each irredundant formula.
- Two new algorithms, Modified Brown's Method and the Multiplication Method, were presented for determining all irredundant formulas which may represent a function. The partitioning of the prime implicants and the deletion of useless PIs were incorporated in both techniques for efficiency purposes. Additionally, we introduced a base—$\widehat{G}$—used in the Multiplication Method which in general contains fewer terms than bases found in the literature.

VI. Formation of a Single Minimal Formula

The formation of all irredundant sum-of-products formulas which represent a switching function is only possible when the number of irredundant disjunctive forms (IDFs) is relatively small. For many functions of moderate complexity, the number of IDFs may be in the millions or greater. Hence, we often must limit our effort to developing a single minimal SOP formula with respect to a given cost criterion, or at the very least produce a near-minimal formula which closely approximates the cost of a minimal formula.

The procedures presented in Chapter 2 for developing sub-minimal formulas (Procedures 2.31 and 2.33) yield relatively good irredundant formulas. In many cases, however, we desire a minimal SOP formula to represent a function. In this chapter we present a set of algorithms for developing minimal SOP formulas. These algorithms are similar in that each requires the formation of inclusion formulas representing the coverage of prime implicants (PIs) of a function by subsets of the prime implicants. After all inclusion formulas are formed, a reduction step is applied which decreases the number of inclusion formulas while identifying prime implicants to be placed in the resulting irredundant formula. In some cases, a minimal irredundant formula which represents a function results after the reduction step. In other instances, there remains a set of inclusion formulas; a search process is then required to judiciously select prime implicants for the final minimal formula. The search process is discussed in Chapter 9.

Basic Methodology

We begin the process of forming a single minimal SOP formula F by using a 1-normal form specification $\phi(X, z) = 1$ to develop an interval $[g, h]$. A base is then developed for $[g, h]$. The base is used as a vehicle for developing inclusion formulas denoting the coverage of each term of the base by subsets of the prime implicants of h. The inclusion formulas are used to identify the set of prime implicants of h which compose F.

A good base to use to develop a minimal formula F is one composed of prime implicants of h. The rationale for this choice is that if prime implicants are used as the base, then the inclusion formulas yield information which allows us to identify prime implicants to include in a minimal formula as well as prime implicants to discard from consideration. A set of rules, which we call ***reduction rules***, facilitates this process. The reduction rules are so called because identifying prime implicants to keep as well as to discard also facilitates a reduction of the number of terms and literals in each of the inclusion formulas. The reduction rules are applied iteratively until they can no longer be applied. Once this occurs, a search process must be used to identify the remaining prime implicants to include in F. We summarize the basic steps in the process of forming a minimal formula F:

1. derive a 1-normal form specification $\phi(X, z) = 1$ if not already formed;
2. construct a general solution of $\phi(X, z) = 1$ for z, in the form of an interval $g(X) \leq z \leq h(X)$, i.e., $z \in [g(X), h(X)]$;
3. develop the set of all prime implicants of h;
4. develop a base for $[g, h]$;
5. develop inclusion formulas representing coverage of the terms of the base by prime implicants of h;
6. reduce the inclusion formulas using reduction rules—identifying prime implicants of h to include in F as well as to discard from consideration; and
7. use a search process to determine the remaining prime implicants to include in F.

For most simple and some moderately complex functions, the first six steps may identify all of the prime implicants to place in F without having to perform the last step, i.e., search. Whether this is possible is dependent on the function itself as well as the base used in the process. For highly complex functions, search is generally required to form an irredundant formula. The first two steps were discussed in Chapters 4 and 5. The search process is discussed in Chapter 9. We present in turn the third through sixth steps in the remainder of this chapter. Step 3 is discussed in the next section.

Three Bases for a Function

An important issue in the foregoing methodology is the choice of a useful base for $[g, h]$. Bases discussed in the literature include the Blake canonical form of h, an IDF which represents a function f in $[g, h]$, and any disjunctive formula which represents an f in $[g, h]$. We propose three new bases for $[g, h]$. A distinguishing characteristic of each of these bases is that each is a subset of the conditionally-eliminable prime implicants of h. The bases differ in the way they are formed as well as in the number of terms they comprise. The selection of one base over another should depend on the available computational resources, e.g., memory space, the complexity of the interval $[g, h]$, and the time available for computation. We use a large base only if there is enough memory to support its use.

Base #1 - All Useful CEPIs. The first base that we propose is the set of all useful, conditionally-eliminable prime implicants of h. This base is similar is some respects to the Blake canonical form base used in (Ghaza 57), (Mott 60), (Gaine 64), and (Tison 67). However, since we know that essential prime implicants will be contained in any minimal irredundant formula F and inessential prime implicants will not be contained in F, we need only focus our effort on determining the coverage of the CEPIs of a function by the prime implicants of the function. This, however, necessitates the partitioning *a priori* of the prime implicants of a function into essential, inessential, and conditionally-eliminable categories—a technique used in Algorithms 5.2 and 5.3—at the outset of the process. To limit the size of the base, as well as for a reason to be explained later, we perform this partitioning at the beginning of each minimization algorithm presented in this work. After the partitioning of the PIs is performed, the useful CEPIs are identified and the useless CEPIs are discarded.

This base is used when an implementation of an algorithm which utilizes the base is hosted on a computer which has sufficient memory capacity given the specification for which a minimal SOP formula F is being developed. It is the base with the most terms; hence, the number of inclusion

formulas representing coverage of each term of the base by subsets of the prime implicants of the function is large. Consequently, a large memory is required to form the inclusion formulas and then to apply the reduction rules which identify prime implicants to place in F as well as to remove from consideration.

However, it is the most desirable base because it is the least likely to require an auxiliary search process.

Base #2 - CEPIs of an IDF. The second base that we present is a set of conditionally-eliminable prime implicants contained in an irredundant disjunctive form which represents a function f in $[g, h]$. Chang and Mott (Chang 65) showed that an IDF of a function is a sufficient base. However, we remove the essential prime implicants because they are contained in all minimal SOP formulas.

After partitioning the prime implicants of f into essential, inessential, and conditionally-eliminable categories, we form an IDF using Procedure 2.32 for functions or Procedure 2.34 for intervals. If the set of essential prime implicants is first identified, then these two procedures are more efficient for forming IDFs than is a general method for forming an IDF (e.g., Procedures 2.31 and 2.33). After an IDF is formed, the essential prime implicants are removed from the formula leaving the terms of the base.

This base is the least desirable of the three bases presented because the formation of an IDF is a very computationally intensive operation. However, it may be necessary to employ an algorithm which uses this base if the memory capacity of the host computer is somewhat limited, because in many cases this base contains the fewest terms of the three bases presented. Specifically, this base tends to contain the fewest terms when the number of essential prime implicants of a function is small relative to the total number of prime implicants. Hence, the number of inclusion formulas which must be developed and stored in memory is generally smaller than when other bases are used.

Base #3 - CEPIs Covering $\hat{g}$. In the multiplication method for finding all irredundant formulas which cover a function, it suffices to form inclusion formulas denoting the coverage by the CEPIs of each term in $\widehat{G}$, the formula representing $\hat{g}$, for which $\hat{g}$ is defined by the equation $\hat{g} = g - h_{sss}$. On the other hand, we desire a base comprising prime implicants in order to take advantage of reduction rules which allow us to identify, using the inclusion formulas, prime implicants to place in a minimal formula F as well as prime implicants to discard from consideration. Fortuitously, we can use $\widehat{G}$ to identify particular conditionally-eliminable prime implicants to include in a base. Specifically, we select CEPIs which alone cover at least one term in $\widehat{G}$.

Example 6.1: Function C4 contains three terms in $\widehat{G}$ (see Table 5.5), i.e.,

$$\widehat{G} = abcde'f'g'hik'l' + a'bcdef'i'jk'l + a'bcdef'gi'jk'. \tag{6.1}$$

Using the prime implicants of C4, we develop inclusion formulas denoting the coverage of each term in $\widehat{G}$ by the CEPIs:

$$\begin{aligned} &P_{96} + P_{85}P_{133} \\ &P_{105} + P_{107}P_{124} \\ &P_{105} + P_{107}P_{124}P_{93} \end{aligned} \tag{6.2}$$

The formulas (6.2) are the inclusion formulas for the terms in (6.1), respectively.

Prime implicant $P_{96} = abcde'f'g'hi$ contains term $abcde'f'g'hik'l'$. Similarly, prime implicant $P_{105} = a'bcdef'i'jk'$ contains both terms $a'bcdef'i'jk'l$ and $a'bcdef'gi'jk'$. Hence, prime implicants P_{96} and P_{105} are sufficient to form a base, because the two together cover the function $\hat{g}$.

In Example 6.1 we used inclusion formulas to illustrate the coverage of terms in $\widehat{G}$ by the conditionally-eliminable prime implicants. However, inclusion formulas do not have to be formed to identify the prime implicants which completely cover a term in $\widehat{G}$. We simply determine for each term in $\widehat{G}$ the CEPIs which are subsets with respect to literals of the term. Every term in $\widehat{G}$ has at least one prime implicant which alone contains it.

We develop a list of conditionally-eliminable prime implicants which completely cover terms in $\widehat{G}$. While forming this list, we also keep track of the terms in $\widehat{G}$ that each prime implicant covers.

Then the set of prime implicants to include in the base is determined by following a greedy method of selection, i.e.,

1. choose the prime implicant which covers the most terms in $\widehat{G}$;
2. choose the prime implicant which covers the most terms in $\widehat{G}$ that have yet to be covered; and
3. continue in the same fashion until all terms in $\widehat{G}$ are covered by some member of the base.

Procedure 6.1 implements the foregoing method for forming a base. Example 6.2 demonstrates the application of Procedure 6.1. In Procedure 6.1, a structure called an *association list* is used which affiliates an element with one or more items. For example, terms are associated with prime implicants which cover them in the following manner:

```
((t1 p1 p2 p3) (t2 p3) (t3 p5 p6) (t4 p6 p7) (t5 p8)).
```

An association list is a structure used in LISP programming, typically affiliating an element with one item; we use it in a more general fashion.

Procedure 6.1 (Base #3 - CEPIs Completely Covering $\hat{g}$): Given the formula $\widehat{G}$ which represents $\hat{g}$ and the set H_{ce} of conditionally-eliminable prime implicants, we form a base in the following manner:

Step 0.

1. Initialise an accumulator $BASE$ to the empty set $\emptyset$.
2. Initialise an accumulator $\widehat{G}_{covered}$ to the empty set $\emptyset$. $\widehat{G}_{covered}$ will serve as an association list in which each element is list containing a term in $\widehat{G}$ along with the prime implicants which completely cover the term. $\widehat{G}_{covered}$ is completely formed in Steps 1 through 4.
3. Initialise an accumulator P_{covers} to the empty set $\emptyset$. P_{covers} will serve as an association list in which each element is list containing a prime implicant which completely covers at least one term in $\widehat{G}$ along with the terms that it completely covers. P_{covers} is completely formed in Steps 5 through 8.

Step 1.

- If $\widehat{G}$ is empty, then we have determined for each term in $\widehat{G}$ the set of prime implicants that completely cover it. Continue to Step 5.
- Otherwise, continue to Step 2.

Step 2.

1. Remove the first term from $\widehat{G}$ and call it t.
2. Initialize an accumulator PI by copying into it the contents of H_{ce}.
3. Also, initialize a list T_{assoc} by placing t into it.

Step 3.

- If PI is empty, then we have determined the prime implicants in H_{ce} which completely cover t. Place the resulting list T_{assoc} in $\widehat{G}_{covered}$, and return to Step 1.
- Otherwise, continue to Step 4.

Step 4. Remove the first term from PI and determine whether it completely contains t.

- If the prime implicant completely contains t, then append the prime implicant to the list T_{assoc}.
- Otherwise, do nothing.

Return to Step 3.

Step 5. Initialize an accumulator $\widehat{G}_{temp}$ by placing into it the contents of $\widehat{G}_{covered}$.

Step 6.

- If $\widehat{G}_{temp}$ is empty, then we have formed the association list P_{covers} in which each element is a prime implicant associated with the terms in $\widehat{G}$ that it completely covers. Continue to Step 9.
- Otherwise, remove the first element from $\widehat{G}_{temp}$ and call it T_{assoc}. Continue to Step 7.

Step 7.

- If all of the prime implicants have been removed from T_{assoc}, then return to Step 6.
- Otherwise, continue to Step 8.

Step 8. Remove the first prime implicant p from T_{assoc}.

- If p does not have a corresponding element in the association list P_{covers}, then create one by forming a list containing p and the term t from the list T_{assoc}.
- Otherwise, append the term t to the list P_{assoc} corresponding to p in P_{covers}.

Return to Step 7.

Step 9. Sort P_{covers} such that the prime implicant which covers the most terms is first, the one which covers the second most terms is second, and so on.

Step 10. Remove the first element P_{assoc} from P_{covers} and add the prime implicant p in P_{assoc} to $BASE$.

Step 11. Determine the terms t that the prime implicant p in P_{assoc} completely covers (stored in P_{assoc}). For each term t in P_{assoc}:

1. Remove the corresponding list T_{assoc} from $\widehat{G}_{covered}$.
2. For the prime implicants p in T_{assoc}, remove the term t from their corresponding lists P_{assoc} in P_{covers}.

Step 12. Remove lists P_{assoc} from P_{covers} which no longer contain any terms.

Step 13.

- If P_{covers} is empty, then we have formed the base. Return $BASE$.
- Otherwise, re-sort P_{covers} such that the prime implicant which covers the most remaining terms is first, the one which covers the second most terms is second, and so on. Perform this sort in a manner such that a prime implicant p which covers fewer terms than previously precedes the prime implicants which previously covered the same number of terms that p now covers, e.g., if p used to cover three terms, but now covers only one, then place it before prime implicants that used to cover one term. Return to Step 10.

Example 6.2: We demonstrate the application of Procedure 6.1 to develop a base for function C5. C5 has 93 conditionally-eliminable prime implicants; the formula $\widehat{G}$ for C5 consists of 17 terms. In this example, we will abstractly denote terms in $\widehat{G}$ by the labels $T_1, T_2, \ldots, T_{17}$. Prime implicants in H_{cs} will be denoted by the labels $P_1, P_2, \ldots, P_{93}$.

Step 0.

1. $BASE = \emptyset$.
2. $\widehat{G}_{covered} = \emptyset$.
3. $P_{covers} = \emptyset$.

Steps 1-4. In these steps, the association list $\widehat{G}_{covered}$ is filled. After execution of these steps, $\widehat{G}_{covered}$ contains the set of lists:

```
( (T1  P63 P82) (T2  P58) (T3  P63) (T4 P70 P7) (T5 P21)
  (T6  P84 P24) (T7  P73) (T8  P84) (T9 P84) (T10 P73)
  (T11 P70 P87) (T12 P35) (T13 P63 P82 P76) (T14 P65)
  (T15 P63 P39) (T16 P63 P75) (T17 P65) )
```

Steps 5-8. In these steps, the association list P_{covers} is filled. After execution of these steps, P_{covers} contains the set of lists:

```
( (P63 T1 T3 T13 T15 T16) (P82 T1 T13) (P58 T2) (P70 T4 T11)
  (P7 T4) (P21 T5) (P84 T6 T8 T9) (P24 T6) (P73 T7 T10) (P87 T11)
  (P35 T12) (P76 T13) (P65 T14 T17) (P39 T15) (P75 P16) )
```

Step 9. We sort the association list P_{covers} in this step. The revised list is:

```
( (P63 T1 T3 T13 T15 T16) (P84 T6 T8 T9) (P82 T1 T13)
  (P70 T4 T11) (P73 T7 T10) (P65 T14 T17) (P58 T2) (P7 T4)
  (P21 T5) (P24 T6) (P87 T11) (P35 T12) (P76 T13) (P39 T15)
  (P75 P16) )
```

Step 10. Element (P63 T1 T3 T13 T15 T16) is removed from P_{covers} and P63 is added to $BASE$.

Step 11. We compute this step for terms T1,T3,T13,T15, and T16 in (P63 T1 T3 T13 T15 T16). At the completion of this step, P_{covers} contains the lists:

```
( (P63) (P84 T6 T8 T9) (P82) (P70 T4 T11) (P73 T7 T10)
  (P65 T14 T17) (P58 T2) (P7 T4) (P21 T5) (P24 T6)
  (P87 T11) (P35 T12) (P76) (P39) (P75) )
```

Steps 12-13. Lists which no longer contain terms are removed from P_{covers}. P_{covers} is then sorted. A revised list is:

```
( (P84 T6 T8 T9) (P70 T4 T11) (P73 T7 T10) (P65 T14 T17)
  (P58 T2) (P7 T4) (P21 T5) (P24 T6) (P87 T11) (P35 T12) )
```

Step 10. Element (P84 T6 T8 T9) is removed from P_{covers} and P84 is added to $BASE$. $BASE$ then contains (P63 P84).

Steps 11-13. After (P84 T6 T8 T9) is added to the base, P_{covers} is again revised and sorted. A revised list is:

```
( (P70 T4 T11) (P73 T7 T10) (P65 T14 T17)
  (P58 T2) (P7 T4) (P21 T5) (P87 T11) (P35 T12) )
```

Step 10. Element (P70 T4 T11) is removed from P_{covers} and P70 is added to $BASE$. $BASE$ then contains (P63 P84 P70).

Steps 11-13. After P70 is added to the base, P_{covers} is again revised and sorted. A revised list is:

```
( (P73 T7 T10) (P65 T14 T17) (P58 T2) (P21 T5) (P35 T12) )
```

Steps 10-13. In the remaining iterations, all of the remaining elements of P_{covers} are added to the base. The list returned by the procedure is:

```
(P63 P84 P70 P73 P65 P58 P21 P35)
```

The base thus consists of the set of conditionally-eliminable prime implicants:

$$\{P_{63}, P_{84}, P_{70}, P_{73}, P_{65}, P_{58}, P_{21}, P_{35}\}. \tag{6.3}$$

If we are not able to use Base #1 due to memory constraints, this base may provide a useful alternative. It is relatively simple to form—not nearly as computationally intensive as forming Base #2. Moreover, in many functions the number of terms in this base is about the same as in Base #2. Furthermore, we attain the added benefit that the prime implicants chosen to form Base #3 are very likely to be included in the final irredundant formula F. This information is useful during the search process, if search is required.

Comparison of Bases. We now compare the number of prime implicants contained in the three bases for several sets of example functions. Tables 6.1, 6.2, and 6.3 contain the number of terms in the bases for Data Sets B, C, and IC, respectively. For all three data sets, the number of terms in Base #1 is significantly greater than the number of terms in Bases #2 and #3. For these data sets, the number of terms in Bases #2 and #3 is about the same.

Function Identifier	Number PIs	Essen PIs	Base #1 No Terms	No Terms IDF	Base #2 No Terms	No Terms $\widehat{G}$	Base #3 No Terms
B1	3	3	0	3	0	0	0
B2	5	4	0	4	0	0	0
B3	11	10	0	10	0	0	0
B4	25	19	3	20	1	1	1
B5	48	26	19	33	7	7	7
B6	127	33	87	68	35	43	43
B7	206	72	123	127	55	54	52
B8	525	112	400	274	162	187	183

Table 6.1. Data Set B (Bases)

Function Identifier	Number PIs	Essen PIs	Base #1 No Terms	No Terms IDF	Base #2 No Terms	No Terms $\widehat{G}$	Base #3 No Terms
C1	16	10	0	10	0	0	0
C2	51	19	3	20	1	1	1
C3	113	29	0	29	0	0	0
C4	149	37	7	39	2	3	2
C5	321	42	93	49	7	17	8
C6	407	37	235	59	22	46	27
C7	446	44	174	61	17	31	18

Table 6.2. Data Set C (Bases)

Formation of Inclusion Formulas for Base Terms

Once the base for $[g, h]$ is developed, inclusion formulas denoting the coverage of terms of the base by subsets of the prime implicants of h are formed. An overview of a number of previous approaches to this problem was presented in Chapter 5. First, we will discuss Cutler's methods

Function Identifier	Number PIs	Essen PIs	Base #1 No Terms	No Terms IDF	Base #2 No Terms	No Terms $\widehat{G}$	Base #3 No Terms
IC1	6	3	3	3	0	0	0
IC2	12	1	11	3	2	2	2
IC3	14	2	12	3	1	1	1
IC4	23	7	16	8	1	1	1
IC5	25	4	21	9	5	7	4
IC6	34	10	24	15	5	4	4
IC7	38	18	19	21	3	2	2
IC8	61	24	32	27	3	3	3
IC9	96	8	92	32	24	23	21
IC10	81	24	54	35	11	10	10
IC11	136	31	98	52	21	19	19
IC12	183	14	168	61	47	60	53
IC13	206	45	156	79	34	32	32
IC14	295	38	255	106	68	77	68
IC15	398	23	373	123	100	113	101

Table 6.3. Data Set IC (Bases)

for forming inclusion formulas. We distinguish Cutler's work from the methods discussed earlier because he was one of the first to adopt the use of search to derive a single minimal formula F rather than multiplying out the set of inclusion formulas to form all irredundant formulas. We then present several new techniques which simplify the development of inclusion formulas for both functions and intervals.

Previous Work. A technique for generating inclusion formulas for functions developed by Cutler (Cutle 80, Cutle 87) combines both Ghazala's and Tison's approach to the problem. In his method, coverage of a term t of the base by the set $S = \{p_1, p_2, \ldots, p_n\}$ of prime implicants of a function is determined as follows:

1. Divide each member of S by t to form the set $\{p_1/t, p_2/t, \ldots, p_n/t\}$. Elements p_i/t which are equal to 0 are dropped, thus leaving a set of terms.
2. Affix labels $P_1, P_2, \ldots, P_n$ to each member of $\{p_1/t, p_2/t, \ldots, p_n/t\}$ to form the set

$$G = \{P_1(p_1/t), P_2(p_2/t), \ldots, P_n(p_n/t)\}, \tag{6.4}$$

in which each $P_i(p_i/t)$ is a term.

3. For each binate variable x in the resulting set, perform in turn the following steps:
   (a) form all possible consensus terms;
   (b) add the consensus terms to the set; and
   (c) delete all terms which contain either x or x'.
4. If a variable x is unate in the disjunction of terms of the set at any point in the process, then delete all terms which contain x.
5. After all terms which contain unate X-variables are deleted and consensus operations with respect to all binate variables are performed, the resulting terms represent the subsets of the prime implicants which cover term t.

This method is essentially the same as we presented in Procedure 3.6 for determining the coverage of a term by subsets of a set of terms. However, terms in the set $S = \{p_1, p_2, \ldots, p_n\}$ which are unrelated to t are not deleted in the process. Furthermore, Cutler did not discuss the theoretical underpinning of the process, i.e., the elimination of the X-arguments to derive a formula denoting the coverage of t.

Whereas we call the disjunction of terms in the final set the "inclusion formula", Cutler, Tison, and others have called it the *inclusion function*. We prefer our terminology because we are not concerned with the function that the formula represents, but rather with what the formula symbolises—coverage of a term by subsets of a set of terms. When the Blake canonical form is used as the base, Cutler refers to the conjunction of the inclusion formulas—called the *presence function* by Ghasala—as the *Tison function*. Moreover, if an irredundant disjunctive form is used as a base, he calls the presence function the *"abbreviated" Tison function*. (Chang and Mott (Chang 65) originally demonstrated that an IDF is a suitable base; they referred to corresponding presence function as the *"abbreviated" presence function*.)

Cutler also developed a method for using the prime implicants of h as a base for an interval $[g, h]$. Cutler's focus was on forming inclusion formulas for the useful prime implicants of the function so that reduction rules could be used to reduce the inclusion formulas while identifying prime implicants to place in a single minimal formula. His procedure for forming an inclusion formula for a useful PI p_j is as follows:

1. Compute the product $p_j \cdot g$.
2. For each term t of $p_j \cdot g$, develop a formula denoting the coverage of t by the useful prime implicants of h. (Procedure 3.6 may be used to develop such a formula.)
3. Form the product of the formulas denoting the coverage of each term t. The equivalent absorptive formula for the result denotes the coverage of prime implicant p_j by the useful prime implicants of h.

The goal of this method is to cover only the portion of prime implicant p_j which is required to cover g. The portion of p_j which covers g' is "filtered out" by the product operation $p_j \cdot g$. We demonstrate this method in Example 6.3.

Example 6.3: Suppose we are given an interval $[g, h]$, defined by

$$g = bcd + ab'cd' \tag{6.5}$$
$$h = ac + bd + b'c + cd. \tag{6.6}$$

The right-hand side of (6.6) is the Blake canonical form for h. Each prime implicant is useful.

For prime implicant ac, the product $ac \cdot g$ is formed:

$$ac \cdot g = abcd + ab'cd'. \tag{6.7}$$

Formulas denoting the coverage of terms $abcd$ and $ab'cd'$ by the prime implicants of h are then developed. Denoting the prime implicants of h by the labels P_1, P_2, P_3, and P_4, respectively, the formulas developed are

- $abcd$: $P_1 + P_2 + P_4$ and
- $ab'cd'$: $P_1 + P_3$.

The product of $P_1 + P_2 + P_4$ and $P_1 + P_3$,

$$P_1 + P_2P_3 + P_3P_4, \tag{6.8}$$

is the inclusion formula for prime implicant ac. Inclusion formulas are similarly developed for all prime implicants of h:

- ac : $P_1 + P_2P_3 + P_3P_4$;
- bd : $P_2 + P_4$;
- $b'c$: $P_3 + P_1$; and
- cd : $P_4 + P_2$.

The methods that we have discussed for developing inclusion formulas have evolved as the result of research over the past four decades. In an effort to continue this progression, we now present several new techniques which make the process of developing inclusion formulas more efficient.

Formation of Inclusion Formulas for Bases #1 and #2. We introduce two techniques for making the process of developing inclusion formulas more efficient in cases in which Bases #1 and #2 are used. First, we present a technique in which the identification of the essential prime implicants of a function allows us to use the EPIs to simplify the development of inclusion formulas denoting coverage of terms of the base. We achieve two benefits by using this technique:

1. the generation of inclusion formulas is more efficient; and
2. the resulting formulas contain fewer terms and literals.

In the second section, we introduce an approach which simplifies the generation of inclusion formulas for intervals.

Essential PI Constraint. A term of the base is typically covered by a number of combinations of prime implicants. In Example 5.3, the inclusion formula denoting the coverage of prime implicant $b'ce$ is the formula

$$P_8 + P_1P_7 + P_4P_6 + P_1P_3P_5 + P_1P_4P_5 + P_1P_3P_6. \tag{6.9}$$

Hence, $b'ce$ is covered by the prime implicant associated with label P_8—itself in this case—as well as the combination of the prime implicants associated with labels P_1 and P_7, etc. We also know in this example that the prime implicants associated with the labels P_1, P_2, and P_5 are essential prime implicants, which means that they must appear in the final minimal formula. We can thus treat essential prime implicants as a "given". It follows that if we identify the essential prime implicants at the outset, we can then take them as given with respect to the remaining prime implicants. For example, if we knew beforehand the essential prime implicants in Example 5.3, we then could develop the formula

$$P_8 + P_7 + P_3 + P_4. \tag{6.10}$$

to represent the coverage of $b'ce$ rather than (6.9). (We form (6.10) by setting the labels associated with the essential PIs in (6.9) to 1 and deleting absorbed terms.) The second term of (6.9) denotes the coverage of $b'ce$ by a combination of P_1 and P_7; however, since P_1 is an essential PI, only P_7 is required to complete the coverage of the term. Hence, the second term of (6.10) is the single-literal term P_7.

To develop a formula such as (6.10) rather than (6.9), we restate the problem of forming inclusion formulas denoting the coverage of a term by the prime implicants in the following manner:

> Given the essential prime implicants of a function, form an inclusion formula denoting the coverage by the conditionally-eliminable prime implicants of the portion of the term not covered by h_{ess}.

Theorem 6.1 facilitates the solution to this problem. We first state the theorem and then discuss its ramifications.

Theorem 6.1: *Given a set $F = \{f_1, f_2, \ldots, f_i, f_{i+1}, \ldots, f_k\}$ of functions, and a function $\tilde{f}$ formed by summing the first i elements of F,*

$$\tilde{f} = f_1 + f_2 + \cdots + f_i, \tag{6.11}$$

for which $\tilde{f} = 0$ is consistent, the set F sums to one, i.e.,

$$f_1 + f_2 + \cdots + f_i + f_{i+1} + \cdots + f_k = 1, \tag{6.12}$$

if and only if

$$\tilde{f} = 0 \implies f_{i+1} + \cdots + f_k = 1. \tag{6.13}$$

Proof. In view of (6.11), we restate equation (6.12) as follows:

$$\tilde{f} + f_{i+1} + \cdots + f_k = 1. \tag{6.14}$$

By the definition of the inclusion relation, $a + b = 1 \Leftrightarrow a' \leq b$. Applying this equivalence, (6.14) is equivalent to the statement

$$\tilde{f}' \leq f_{i+1} + \cdots + f_k. \tag{6.15}$$

By the Extended Verification Theorem, (6.15) is equivalent to

$$\tilde{f} = 0 \implies f_{i+1} + \cdots + f_k = 1, \tag{6.16}$$

provided that $\tilde{f} = 0$ is consistent. This completes the proof. □

In Chapter 3 we discussed the coverage of a term t by subsets of a set $T = \{t_1, t_2, \ldots, t_k\}$ of terms. Procedure 3.6 implements a technique for determining these subsets. In Procedure 3.6, each member of T is first divided by t to form the set $\tilde{T} = \{\tilde{t_1}, \tilde{t_2}, \ldots, \tilde{t_k}\}$. Then, sum-to-one subsets of $\tilde{T}$ correspond to subsets of T which cover t. This is the same basic approach employed by Ghazala, Reusch, Cutler, and others. For example, the following correspondences may be made between Reusch's ϕ-chart (Table 6.4) and the terminology used in Chapter 3:

- each term t_j corresponds to the term t to be covered;

- the set $\{p_1, p_2, \ldots, p_n\}$ of prime implicants corresponds to the set T; and
- terms in a column of the table compose the members of set $\tilde{T}$.

In many cases, a sum-to-one subset of terms in a column of Table 6.4 includes terms p_i/t_j corresponding to essential prime implicants p_i.

	t_1	t_2	$\cdots$	t_j	$\cdots$	t_m
p_1	p_1/t_1	p_1/t_2	$\cdots$	p_1/t_j	$\cdots$	p_1/t_m
p_2	p_2/t_1	p_2/t_2	$\cdots$	p_2/t_j	$\cdots$	p_2/t_m
$\vdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
p_i	p_i/t_1	p_i/t_2	$\cdots$	p_i/t_j	$\cdots$	p_i/t_m
$\vdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
p_n	p_n/t_1	p_n/t_2	$\cdots$	p_n/t_j	$\cdots$	p_n/t_m

Table 6.4. Reusch's ϕ-Chart

Theorem 6.1 enables the determination of subsets of the conditionally-eliminable prime implicants which cover the portion of a term t_j not covered by the essential PIs. If a number of essential prime implicants combine with other PIs to cover t_j, then all corresponding terms p_i/t_j form a sum-to-one subset. If we view the sum of the terms p_i/t_j which correspond to essential PIs as $\tilde{f}$ in Theorem 6.1, then the terms p_i/t_j corresponding to CEPIs only need to cover what $\tilde{f}$ does not cover, i.e. $\tilde{f}'$. This concept is demonstrated by statement (6.15). Statement (6.16) states that if we enforce the condition that $\tilde{f} = 0$, then the terms not contained in $\tilde{f}$ which combined with terms of $\tilde{f}$ to form a sum-to-one subset will alone form a sum-to-one subset. Thus, enforcing the constraint $\tilde{f} = 0$ enables us to determine the subsets of the conditionally-eliminable prime implicants which cover the portion of a term not covered by the essential prime implicants.

Let us define a function, $\mathcal{EPI}_j$, to be the sum of all terms p_i/t_j in column j corresponding to essential prime implicants. We call the equation

$$\mathcal{EPI}_j = 0 \tag{6.17}$$

the ***essential prime implicant constraint*** with respect to term t_j. We shall refer to the essential prime implicant constraint as the EPI-constraint.

We apply the methodology presented in Chapter 3 for determining the sum-to-one subsets among a set of terms to determine such subsets in the j-th column of Table 6.4. Using this methodology, we first form a system

$$\begin{array}{rcl} p_1(X)/t_j(X) & \leq & P_1 \\ p_2(X)/t_j(X) & \leq & P_2 \\ & \vdots & \\ p_k(X)/t_j(X) & \leq & P_k, \end{array} \tag{6.18}$$

which is in turn reduced to an equation $f_j(A, X) = 0$. The function $f_j(A, X)$ is defined by the equation

$$f_j(A, X) = \sum_{i=1}^{k} (p_i(X)/t_j(X) \cdot P_i'). \tag{6.19}$$

In this case, we define the A-vector as the vector of labels associated with prime implicants, i.e., $A = (P_1, P_2, \ldots, P_k)$. As in Procedure 3.6, terms $p_i(X)/t_j(X)$ which contain unate variables or unrelated variables with respect to t_j are not used to form system (6.18). We must consider all terms $p_i(X)/t_j(X)$ in a column of Table 6.4 in making this assessment.

Once the equation $f_j(A, X) = 0$ is developed, property (2.40) allows us then to form

$$f_j(A, X) + \mathcal{EPI}_j(X) = 0. \tag{6.20}$$

By combining the equations $f_j(A, X) = 0$ and $\mathcal{EPI}_j(X) = 0$, we thus enforce the condition that the terms p_i/t_j corresponding to essential prime implicants be set equal to 0. We apply goal-directed

elimination to eliminate the X-arguments in (6.20) to form an equation $g_j(A) = 0$, for which $g_j(A)$ is defined by the equation

$$g_j(A) = ECON(f_j(A, X) + \mathcal{EPI}_j(X), X). \tag{6.21}$$

After elimination, $g_j(A)$ is represented by its Blake canonical form. Terms of $BCF(g_j(A))$ denote the coverage of the portion of a term t_j not covered by the essential prime implicants by subsets of the conditionally-eliminable prime implicants.

Prior to performing the elimination process, we take several steps which reduce the computations and memory space required to eliminate the X-arguments. First, we represent $\mathcal{EPI}_j(X)$ by its Blake canonical form. We then use relative absorption and relative simplification to simplify $F_j(A, X)$ relative to $BCF(\mathcal{EPI}_j(X))$. We denote the resulting formula by the notation $\tilde{F}_j(A, X)$ and the function it represents by the notation $\tilde{f}_j(A, X)$. By the Relative Simplification Theorem (2.123), the functions $f + g$ and $\tilde{f} + g$ are equal. It then follows that equation (6.21) is equivalent to the equation

$$g_j(A) = ECON(\tilde{f}_j(A, X) + \mathcal{EPI}_j(X), X). \tag{6.22}$$

Hence, we may eliminate the X-arguments from $\tilde{f}_j(A, X) + \mathcal{EPI}_j(X)$ rather than $f_j(A, X) + \mathcal{EPI}_j(X)$ to form $g_j(A)$.

Because no terms in the formula $BCF(\mathcal{EPI}_j(X))$ contain any A-arguments and every term in the formula $F_j(A, X)$ does, terms in $BCF(\mathcal{EPI}_j(X))$ may absorb terms in $F_j(A, X)$. We use the relative absorption operation to remove terms in $F_j(A, X)$ absorbed by terms in $BCF(\mathcal{EPI}_j(X))$. We thus form $\dot{F}_j(A, X)$, for which $\dot{F}_j(A, X)$ is defined by the equation

$$\dot{F}_j(A, X) = ABSREL(F_j(A, X), BCF(\mathcal{EPI}_j(X))). \tag{6.23}$$

One aspect of this operation is that all terms $p_i(X)/t_j(X) \cdot P_i'$ corresponding to essential prime implicants in $F_j(A, X)$ will be absorbed by terms in $BCF(\mathcal{EPI}_j(X))$. Consequently, terms $p_i(X)/t_j(X)$ corresponding to essential prime implicants are not used at the outset to form system (6.18). Entries corresponding to inessential PIs are not involved because we know that they are covered by the essential prime implicants. Hence, only the conditionally-eliminable prime implicants are used to cover the term t_j.

After relative absorption, we simplify the terms in $\dot{F}_j(A, X)$ relative to $BCF(\mathcal{EPI}_j(X))$ to form $\bar{F}_j(A, X)$, i.e.,

$$\bar{F}_j(A, X) = SIMPREL(\dot{F}_j(A, X), BCF(\mathcal{EPI}_j(X))). \tag{6.24}$$

The effect of this process is to reduce the number of literals contained in terms of $\dot{F}_j(A, X)$. Fewer computations are thus performed during goal-directed elimination. We now present an example which illustrates the foregoing process.

Example 6.4: We develop the inclusion formulas for the function f from Example 5.3, for which $BCF(f)$ is defined by the equation

$$BCF(f) = d'e + cde' + a'cd + a'ce + ab'd + ab'e + b'cd + b'ce. \tag{6.25}$$

The ϕ-chart for f is given by Table 6.5. In this example, $X = (a, b, c, d, e)$ and $A = (P_1, P_2, \ldots, P_8)$. The inclusion formulas derived in Example 5.3 for each of the prime implicants are:

- $d'e$: P_1;
- cde' : P_2;
- $a'cd$: $P_3 + P_2P_4$;
- $a'ce$: $P_4 + P_1P_3$;
- $ab'd$: P_5;
- $ab'e$: $P_6 + P_1P_5$;
- $b'cd$: $P_7 + P_3P_5 + P_2P_8 + P_2P_3P_6 + P_2P_4P_5 + P_2P_4P_6$; and
- $b'ce$: $P_8 + P_1P_7 + P_4P_6 + P_1P_3P_5 + P_1P_4P_5 + P_1P_3P_6$.

We conclude from analysis of the prime implicants that $d'e, cde'$, and $ab'd$ are essential prime implicants of f. Moreover, prime implicant $ab'e$ is covered by the essential prime implicants, i.e.,

		T_1 $d'e$	T_2 cde'	T_3 $a'cd$	T_4 $a'ce$	T_5 $ab'd$	T_6 $ab'e$	T_7 $b'cd$	T_8 $b'ce$
P_1	$d'e$	1	0	0	d'	0	d'	0	d'
P_2	cde'	0	1	e'	0	ce'	0	e'	0
P_3	$a'cd$	0	a'	1	d	0	0	a'	$a'd$
P_4	$a'ce$	$a'c$	0	e	1	0	0	$a'e$	a'
P_5	$ab'd$	0	ab'	0	0	1	d	a	ad
P_6	$ab'e$	ab'	0	0	0	e	1	ae	a
P_7	$b'cd$	0	b'	b'	$b'd$	c	cd	1	d
P_8	$b'ce$	$b'c$	0	$b'e$	b'	ce	c	e	1

Table 6.5. ϕ-Chart for Example 6.4

		T_3 $a'cd$	T_4 $a'ce$	T_7 $b'cd$	T_8 $b'ce$
P_1	$d'e$	0	d'	0	d'
P_2	cde'	e'	0	e'	0
P_3	$a'cd$	1	d	a'	$a'd$
P_4	$a'ce$	e	1	$a'e$	a'
P_5	$ab'd$	0	0	a	ad
P_6	$ab'e$	0	0	ae	a
P_7	$b'cd$	b'	$b'd$	1	d
P_8	$b'ce$	$b'e$	b'	e	1

Table 6.6. Revised ϕ-Chart for Example 6.4

it is inessential. Hence we may delete the columns corresponding to $d'e, cde', ab'd$, and $ab'e$ from the ϕ-chart to form a revised chart (Table 6.6).

For the column of Table 6.6 associated with T_8, we form $F_8(A, X)$:

$$F_8(A, X) = d'P_1' + a'dP_3' + a'P_4' + adP_5' + aP_6' + dP_7' + P_8'. \tag{6.26}$$

The function $\mathcal{EPI}_8(X)$ is formed by summing the entries in the rows corresponding to the essential PIs, i.e., rows 1, 2, and 5:

$$\mathcal{EPI}_8(X) = d' + ad. \tag{6.27}$$

The Blake canonical form of $\mathcal{EPI}_8(X)$ is the formula $a + d'$.

Applying relative absorption, any term on the right-hand side of (6.26) which is absorbed by a term in $a + d'$ is deleted. We thus form

$$\dot{F}_8(A, X) = a'dP_3' + a'P_4' + dP_7' + P_8'. \tag{6.28}$$

We simplify the right-hand side of (6.28) relative to $a + d'$ to form $\tilde{F}_8(A, X)$. The term a in $BCF(\mathcal{EPI}_8(X))$ forms the consensus terms dP_3' and P_4' with the first and second terms of the right-hand side of (6.28), respectively. Since these terms absorb their parent terms, they replace their parent terms in the formula. The right-hand side of (6.28) thus becomes

$$dP_3' + P_4' + dP_7' + P_8'. \tag{6.29}$$

The new formula is then simplified with respect to the term d' in $BCF(\mathcal{EPI}_8(X))$. We thus form $\tilde{F}_8(A, X)$:

$$\tilde{F}_8(A, X) = P_3' + P_4' + P_7' + P_8'. \tag{6.30}$$

Typically, we would then form $g_8(A)$ by eliminating the X-arguments from the equation

$$\tilde{f}_8(A, X) + \mathcal{EPI}_8(X) = 0. \tag{6.31}$$

Since $\tilde{f}_8(A, X) + \mathcal{EPI}_8(X)$ is represented by the formula

$$P_3' + P_4' + P_7' + P_8' + a + d', \tag{6.32}$$

we observe by inspection that the elimination of the X-arguments a and d yields the formula $P_3' + P_4' + P_7' + P_8'$. Hence, $P_3' + P_4' + P_7' + P_8'$ is the inclusion formula representing the coverage of the term $b'ce$ by the conditionally-eliminable prime implicants.

Similarly, we form the following inclusion formulas for the remaining terms:

- $a'cd$: $P_3' + P_4'$;
- $a'ce$: $P_4' + P_3'$; and
- $b'cd$: $P_7' + P_3' + P_8' + P_4'$.

We observe in Example 6.4 that the inclusion formula $P_3' + P_4' + P_7' + P_8'$ representing the coverage of the term $b'ce$ by the conditionally-eliminable prime implicants is the same as (6.10).[1] Hence, the methodology that we have developed accomplishes the stated objective of forming an inclusion formula denoting the coverage by the conditionally-eliminable prime implicants of the portion of the term not covered by essential prime implicants. The resulting inclusion formula has fewer terms and literals than would otherwise be the case. Moreover, the relative absorption and relative simplification operations allow us to reduce the number of terms and literals in the formula developed prior to the elimination process. Hence, less work is required during the elimination process.

The removal of essential prime implicants from the set of prime implicants for placement in a minimal SOP formula is not in itself a novel idea. This idea is used in the heuristic minimization program ESPRESSO (Brayt 84). To our knowledge, however, the introduction of a constraint to simplify the formation of inclusion formulas is a new technique. We will shortly present a procedure which implements this method. We first introduce a constraint for intervals $[g, h]$ which simplifies the development of inclusion formulas for a prime implicant base.

Don't-Care Constraint for Intervals. Cutler (Cutle 80) developed a technique for deriving inclusion formulas for prime implicants of an interval $[g, h]$. To form an inclusion formula for a PI p_j using his method, the following operations are performed:

1. the product $p_j \cdot g$ is computed;
2. an inclusion formula is developed for each term t of the formula representing $p_j \cdot g$;
3. the product of the resulting formulas is formed; and
4. the equivalent absorptive formula for the formula resulting from the third step is developed.

[1] The formulas are the same with the exception that the labels associated with the prime implicants in our formulas are complemented whereas they are not in other formulas. We use the complemented form in order to maintain theoretical consistency with our development in Chapter 3.

The goal of this method is to cover only the portion of prime implicant p_j which is included in the lower bound function g; the portion of p_j which covers g' is removed by the product operation $p_j \cdot g$. This method is demonstrated in Example 6.3.

However, rather than placing emphasis on the product $p_j \cdot g$, a more direct approach to this problem is to enforce validity of the condition

$$p_j \cdot g' = 0. \tag{6.33}$$

In the same vein as the essential prime implicant constraint, equation (6.33) is a constraint that we may employ to force the portion of the prime implicant p_j that is included in the function g' not to be covered. The effect of enforcing this condition is that rather than having to form a set of inclusion formulas—one for each term of the formula representing $p_j \cdot g$—and then performing product and absorption operations, we achieve the same result in the process of forming a single inclusion formula.

In view of (2.85), we can make the following statement:

$$f = 0 \;\Rightarrow\; f/t = 0. \tag{6.34}$$

Hence, equation (6.33) implies the equation

$$(p_j \cdot g')/p_j = 0. \tag{6.35}$$

Additionally, rather than complementing function g and multiplying by p_j, the function $p_j \cdot g'$ may be rewritten as $p_j - g$ by the definition of subtraction. Hence, to form the left-hand side of (6.35), we subtract g from p_j and divide the result by the term p_j. Equation (6.35) may be used in the same manner as the EPI-constraint.

The don't-care set of a function is defined by minterms of the function $h-g$. By the definition of subtraction, $h-g$ may be rewritten as $h \cdot g'$. The function $p_j \cdot g'$ is included in the function $h \cdot g'$, since p_j is a prime implicant of h. It follows that minterms of $p_j \cdot g'$ are members of the don't-care set of the function. We thus denote the function $(p_j - g)/p_j$ by the notation $\mathcal{DC}_j$ and call the equation

$$\mathcal{DC}_j = 0 \tag{6.36}$$

the *don't-care constraint* with respect to prime implicant p_j. We also refer to the don't-care constraint as the DC-constraint.

The DC-constraint is handled in conjunction with the EPI-constraint introduced in the previous section. A system

$$\begin{array}{rcl} p_1(X)/t_j(X) & \leq & P_1 \\ p_2(X)/t_j(X) & \leq & P_2 \\ & \vdots & \\ p_k(X)/t_j(X) & \leq & P_k \end{array} \tag{6.37}$$

is formed which is in turn reduced to an equation $f_j(A,X) = 0$. The function $f_j(A,X)$ is defined by the equation

$$f_j(A,X) = \sum_{i=1}^{k} (p_i(X)/t_j(X) \cdot P_i'), \tag{6.38}$$

for which $A = (P_1, P_2, \ldots, P_k)$. We apply Property (2.40) to form the equation

$$f_j(A,X) + \mathcal{EPI}_j(X) + \mathcal{DC}_j(X) = 0, \tag{6.39}$$

which enforces both the essential prime implicant and don't-care constraints.

Terms $p_i(X)/t_j(X)$ which contain unate variables are not used to form system (6.37). However, terms of the formula $BCF(\mathcal{EPI}_j(X) + \mathcal{DC}_j(X))$ must be added to the terms $p_i(X)/t_j(X)$ in a column of Table 6.4 to identify the terms which contain unate variables. In some cases, a variable which is unate in the disjunction of terms in a column of Table 6.4 will not be unate in the formula

$$\sum_{i=1}^{k}(p_i(X)/t_j(X)) + BCF(\mathcal{EPI}_j(X) + \mathcal{DC}_j(X)). \tag{6.40}$$

Entries corresponding to terms $p_i(X)/t_j(X)$ which contain variables which are unate in (6.40) are not used to form system (6.37).

Goal-directed elimination of the X-arguments in (6.39) yields the equation $g_j(A) = 0$, for which $g_j(A)$ is defined by the equation

$$g_j(A) = ECON(f_j(A, X) + \mathcal{EPI}_j(X) + \mathcal{DC}_j(X), X). \tag{6.41}$$

After elimination, $g_j(A)$ is represented by its Blake canonical form. Terms of $BCF(g_j(A))$ denote the coverage by subsets of the CEPIs of the portion of a prime implicant p_j which is:

- not covered by the essential prime implicants, and
- not a part of the don't-care set.

Prior to elimination, we may again use relative absorption and relative simplification to simplify the formula $F_j(A, X)$ which represents $f_j(A, X)$. We first represent the function $\mathcal{EPI}_j(X) + \mathcal{DC}_j(X)$ by its Blake canonical form. Terms in $BCF(\mathcal{EPI}_j(X) + \mathcal{DC}_j(X))$ may absorb terms in $F_j(A, X)$. We thus perform the relative absorption operation

$$ABSREL(F_j(A, X), BCF(\mathcal{EPI}_j(X) + \mathcal{DC}_j(X))) \tag{6.42}$$

to remove terms in $F_j(A,X)$ absorbed by terms in $BCF(\mathcal{EPI}_j(X)+\mathcal{DC}_j(X))$. The formula which results from the relative absorption operation is denoted by $\dot{F}_j(A,X)$. Terms in $\dot{F}_j(A,X)$ may then be simplified relative to the terms in $BCF(\mathcal{EPI}_j(X)+\mathcal{DC}_j(X))$ to form $\tilde{F}_j(A,X)$, i.e.,

$$\tilde{F}_j(A,X) = SIMPREL(\dot{F}_j(A,X), BCF(\mathcal{EPI}_j(X)+\mathcal{DC}_j(X))). \tag{6.43}$$

$\tilde{F}_j(A,X)$ represents the function $\tilde{f}_j(A,X)$.

In view of the Relative Simplification Theorem (2.123), equation (6.41) may be restated by the equation

$$g_j(A) = ECON(\tilde{f}_j(A,X)+\mathcal{EPI}_j(X)+\mathcal{DC}_j(X),X). \tag{6.44}$$

Replacing $f_j(A,X)$ by $\tilde{f}_j(A,X)$ facilitates a reduction of computations and memory space required to eliminate the X-arguments. Example 6.5 demonstrates the use of the DC-constraint.

Example 6.5: In Example 6.3 we demonstrated Cutler's method for forming inclusion formulas for an interval $[g,h]$:

$$g = bcd + ab'cd' \tag{6.45}$$
$$h = ac + bd + b'c + cd. \tag{6.46}$$

The right-hand side of (6.46) is the Blake canonical form for h. Each prime implicant is a useful, CEPI. The ϕ-chart for the function is given by Table 6.7. In this case, $X=(a,b,c,d)$ and $A=(P_1,P_2,P_3,P_4)$.

		T_1 ac	T_2 bd	T_3 $b'c$	T_4 cd
P_1	ac	1	ac	a	a
P_2	bd	bd	1	0	b
P_3	$b'c$	b'	0	1	b'
P_4	cd	d	c	d	1

Table 6.7. ϕ-Chart for Example 6.5

To develop an inclusion formula for PI ac using Cutler's method, the product $ac \cdot g$ is formed. The function $ac \cdot g$ is represented by the formula $abcd + ab'cd'$. Inclusion formulas for each term in

$abcd + ab'cd'$ are developed. The formulas, $P_1 + P_2 + P_4$ and $P_1 + P_3$ are then multiplied together to form the inclusion formula $P_1 + P_2P_3 + P_3P_4$ for prime implicant ac.

The function $\mathcal{DC}_1(X)$ associated with prime implicant ac is defined by the equation

$$\mathcal{DC}_1(X) = (ac - (bcd + ab'cd'))/ac \tag{6.47}$$

Computing the right-hand side of (6.47) yields

$$\mathcal{DC}_1(X) = (abcd' + ab'cd)/ac = bd' + b'd. \tag{6.48}$$

Hence, the DC-constraint for ac is

$$bd' + b'd = 0. \tag{6.49}$$

For the column of Table 6.7 associated with T_1, we form $F_1(A, X)$:

$$F_1(A, X) = P_1' + bdP_2' + b'P_3' + dP_4'. \tag{6.50}$$

Since there are no essential prime implicants, the function $\mathcal{EPI}_1(X)$ is identically equal to zero. The formula $bd' + b'd$ representing $\mathcal{DC}_1(X)$ is its Blake canonical form. We are not able to absorb any term of (6.50) with either term of $BCF(\mathcal{DC}_1(X))$.

We then simplify the right-hand side of (6.50) relative to $bd' + b'd$ to form $\tilde{F}_1(A, X)$. The term bP_2' is formed by the consensus of bd' in $BCF(\mathcal{DC}_1(X))$ with the second term of the right-hand side of (6.50). Since this term absorbs its parent term in (6.50) it replaces its parent term in the formula. We thus form $\tilde{F}_1(A, X)$ from the right-hand side of (6.50), i.e.,

$$\tilde{F}_1(A, X) = P_1' + bP_2' + b'P_3' + dP_4'. \tag{6.51}$$

We form $g_1(A)$ by eliminating the X-arguments from the equation

$$\tilde{f}_1(A, X) + \mathcal{DC}_1(X) = 0, \tag{6.52}$$

i.e.,

$$P_1' + bP_2' + b'P_3' + dP_4' + b'd + bd' = 0. \tag{6.53}$$

Eliminating arguments b and d yields the equation

$$P_1' + P_2'P_3' + P_3'P_4' = 0. \tag{6.54}$$

The left-hand side of (6.54) is the same inclusion formula, denoting the coverage of ac by the conditionally-eliminable prime implicants, as formed in Example 6.3. The remaining inclusion formulas are similarly formed.

Example 6.5 demonstrates the advantage of using the DC-constraint versus Cutler's approach to the problem. Rather than having to form several inclusion formulas, multiplying them together,

and performing absorption, a single inclusion formula is generated to denote coverage of a prime implicant.

In some cases, we find that a prime implicant p_j is covered by the function $\mathcal{EPI}_j(X)+\mathcal{DC}_j(X)$. This condition is identified by the combined EPI- and DC-constraint, because in such a circumstance the function $\mathcal{EPI}_j(X) + \mathcal{DC}_j(X)$ is identically equal to one. Such a prime implicant is discarded from consideration for inclusion in a minimal formula.

Inclusion-Formula Procedure for a General PI Base. We now present a procedure which incorporates the EPI- and DC-constraints for generating inclusion formulas in cases where Bases #1 and #2 are used. This procedure is equally applicable to both functions and intervals. If a function f rather than an interval $[g, h]$ is developed from the initial specification $\phi(X, z) = 1$, then the function $\mathcal{DC}_j(X)$ is identically equal to zero and thus has no effect on the process. We assume in this procedure that all prime implicants are related to the PI p_j for which the inclusion formula is being developed; hence, we do not include a check for unrelated terms.[2]

Procedure 6.2 (Formation of an Inclusion Formula): Given a prime implicant p_j, the set $H_{ce} = \{p_1, \ldots, p_k\}$ of conditionally-eliminable prime implicants, a set $LABS = \{P'_1, \ldots, P'_k\}$ of labels associated with each member of H_{ce}, the set $H_{ess} = \{p_{k+1}, \ldots, p_l\}$ of essential prime implicants, and the lower-bound function g, an inclusion formula denoting the coverage of p_j by the conditionally-eliminable prime implicants is formed in the following manner:

Step 0.

1. Form the function $p_j - g$ using Procedure 2.10 (Subtraction).
2. Derive the function $\mathcal{DC}_j(X) = (p_j - g)/p_j$ using Procedure 2.4 (Boolean Division).

Step 1. Using Procedure 2.4 (Boolean Division):

1. Divide each term in H_{ce} by p_j and replace the contents of H_{ce} by the results.
2. Divide each term in H_{ess} by p_j and replace the contents of H_{ess} by the results.
3. Remove from each set elements which are equal to 0.

Step 2.

1. Form $\mathcal{EPI}_j(X)$, the disjunction of terms in H_{ess}.

[2]The definition of a *related* prime implicant is given in Chapter 3. See Step 4 in Procedure 3.6.

2. Form the Blake canonical form of $\mathcal{EPI}_j(X) + \mathcal{DC}_j(X)$ using Procedure 2.20.

Step 3. Determine the set V of X-variables which are unate in the formula formed by the disjunction of terms in H_{ce} and $BCF(\mathcal{EPI}_j(X) + \mathcal{DC}_j(X))$.

Step 4. Remove from V the variables contained in the term p_j.

- If $V = \emptyset$, then continue to Step 5.
- Otherwise, remove from H_{ce} terms containing any variable in V and return to Step 3.

Step 5. Form the set $\tilde{H}_{ce} = \{p_1 \cdot P_1', \ldots, p_k \cdot P_k'\}$ by affixing elements of $LABS$ to the corresponding elements of H_{ce}.

Step 6. Form $F_j(A, X)$, the disjunction of terms in $\tilde{H}_{ce}$.

Step 7.

1. Form the formula $\dot{F}_j(A, X)$, defined by

$$\dot{F}_j(A, X) = ABSREL(F_j(A, X), BCF(\mathcal{EPI}_j(X) + \mathcal{DC}_j(X))).$$

2. Form the formula $\tilde{F}_j(A, X)$, defined by

$$\tilde{F}_j(A, X) = SIMPREL(\dot{F}_j(A, X), BCF(\mathcal{EPI}_j(X) + \mathcal{DC}_j(X))).$$

Step 8. Using the quick method for forming the conjunctive eliminant (Procedure 3.1), form

$$ECON(\tilde{f}_j(A, X) + \mathcal{EPI}_j(X) + \mathcal{DC}_j(X), X).$$

The function $\tilde{f}_j(A, X) + \mathcal{EPI}_j(X) + \mathcal{DC}_j(X)$ is represented by $\tilde{F}_j(A, X) + BCF(\mathcal{EPI}_j(X) + \mathcal{DC}_j(X))$.

Step 9. The resulting eliminant is equal to $g_j(A)$ which is represented by $BCF(g_j(A))$. The resulting formula IF_j is the inclusion formula for p_j. Return IF_j as the result.

Formation of Inclusion Formulas for Base #3. In the previous section we discussed a method for forming inclusion formulas in cases in which Bases #1 and #2 are used. In this section we present a technique for developing inclusion formulas for terms in Base #3, a set of conditionally-eliminable prime implicants which covers the function $\hat{g}$.

Using the same methodology as in the DC-constraint, we would like to enforce the condition that the inclusion formula for p_j denotes coverage of the portion of $\hat{g}$ that p_j covers and nothing more. We therefore enforce the condition

$$p_j \cdot \hat{g}' = 0 \tag{6.55}$$

from which follows the constraint

$$(p_j - \hat{g})/p_j = 0. \tag{6.56}$$

We denote the function $(p_j - \hat{g})/p_j$ by the notation $\widehat{\mathcal{G}}_j$ and call the equation

$$\widehat{\mathcal{G}}_j = 0 \tag{6.57}$$

the $\widehat{G}$-*constraint* with respect to prime implicant p_j.

The $\widehat{G}$-constraint is actually a combination of the EPI- and DC-constraints discussed earlier. The function $\hat{g}$ is defined by the equation $\hat{g} = g \cdot h'_{ess}$. Substituting for $\hat{g}$ in (6.55), we develop the equation

$$p_j \cdot (g \cdot h'_{ess})' = 0. \tag{6.58}$$

Equation (6.58) is equivalent to the system

$$\begin{aligned} p_j \cdot g' &= 0 \\ p_j \cdot h_{ess} &= 0. \end{aligned} \tag{6.59}$$

The first equation of (6.59) is identical to (6.33), the condition we enforce in the DC-constraint. The second equation means that the portion of p_j covered by the essential prime implicants should not be covered, which is in essence the ЬPI-constraint.

We use the $\widehat{G}$-constraint in the same way as the EPI- and DC-constraints. We first form the system

$$\begin{array}{rcl} p_1(X)/t_j(X) & \leq & P_1 \\ p_2(X)/t_j(X) & \leq & P_2 \\ & \vdots & \\ p_k(X)/t_j(X) & \leq & P_k \end{array} \tag{6.60}$$

in which each prime implicant $p_i(X)$ is a conditionally-eliminable prime implicant. The system is then reduced to an equation $f_j(A, X) = 0$, for which $f_j(A, X)$ is defined by the equation

$$f_j(A, X) = \sum_{i=1}^{k} (p_i(X)/t_j(X) \cdot P_i'). \tag{6.61}$$

We then form the equation

$$f_j(A, X) + \widehat{\mathcal{G}}_j(X) = 0, \tag{6.62}$$

which, by combining the equations $f_j(A, X) = 0$ and $\widehat{\mathcal{G}}_j(X) = 0$, enforces the $\widehat{G}$-constraint.

Before forming system (6.60), terms $p_i(X)/t_j(X)$ which contain unate variables or unrelated variables with respect to t_j are identified so that such terms are not used to form system (6.60). We first represent $\widehat{\mathcal{G}}_j(X)$ by its Blake canonical form. $BCF(\widehat{\mathcal{G}}_j(X))$ must be added to the terms $p_i(X)/t_j(X)$ in column j of Table 6.4 when determining terms $p_i(X)/t_j(X)$ which contain unate variables or unrelated variables with respect to t_j. In some cases, a variable which is unate in the disjunction of terms in a column of Table 6.4 will not be unate in the formula

$$\sum_{i=1}^{k} (p_i(X)/t_j(X)) + BCF(\widehat{\mathcal{G}}_j(X)). \tag{6.63}$$

Entries corresponding to terms $p_i(X)/t_j(X)$ which contain variables which are unate in (6.63) are not involved in system (6.60).

Prior to elimination, we again use relative absorption and relative simplification. Terms in $BCF(\widehat{\mathcal{G}}_j(X))$ may absorb terms in $F_j(A,X)$. We thus perform the relative absorption operation to develop the formula $\dot{F}_j(A,X)$, defined by

$$\dot{F}_j(A,X) = ABSREL(F_j(A,X), BCF(\widehat{\mathcal{G}}_j(X))). \tag{6.64}$$

Terms in $\dot{F}_j(A,X)$ are then simplified relative to the terms in $BCF(\widehat{\mathcal{G}}_j(X))$ to form $\tilde{F}_j(A,X)$, i.e.,

$$\tilde{F}_j(A,X) = SIMPREL(\dot{F}_j(A,X), BCF(\widehat{\mathcal{G}}_j(X))). \tag{6.65}$$

Elimination of the X-arguments from the equation

$$\tilde{f}_j(A,X) + \widehat{\mathcal{G}}_j(X) = 0 \tag{6.66}$$

yields the resultant of elimination $g_j(A) = 0$. The function $g_j(A)$ is defined by the equation

$$g_j(A) = ECON(\tilde{f}_j(A,X) + \widehat{\mathcal{G}}_j(X), X). \tag{6.67}$$

After goal-directed elimination, $g_j(A)$ is represented by $BCF(g_j(A))$. Terms of $BCF(g_j(A))$ denote the coverage by subsets of the CEPIs of the portion of a prime implicant p_j which is:

- not covered by the essential prime implicants, and
- not a part of the don't-care set.

Formation of an inclusion formula using the $\widehat{G}$-constraint is performed using Procedure 6.3. Example 6.6 demonstrates the application of Procedure 6.3.

Procedure 6.3 (Formation of an Inclusion Formula): Given a prime implicant p_j, the set $H_{ce} = \{p_1, \ldots, p_k\}$ of conditionally-eliminable prime implicants, a set $LABS = \{P'_1, \ldots, P'_k\}$ of labels associated with each member of H_{ce}, and the lower-bound function $\hat{g}$, an inclusion formula denoting the coverage of p_j by the conditionally-eliminable prime implicants is formed in the following manner:

Step 0.

1. Form the function $p_j - \hat{g}$ using Procedure 2.10 (Subtraction).
2. Derive the function $\widehat{\mathcal{G}}_j(X) = (p_j - \hat{g})/p_j$ using Procedure 2.4 (Boolean Division).

Step 1. Using Procedure 2.4 (Boolean Division):

1. Divide each term in H_{ce} by p_j and replace the contents of H_{ce} by the results.
2. Remove from each set elements which are equal to 0.

Step 2. Form the Blake canonical form of $\widehat{\mathcal{G}}_j(X)$ using Procedure 2.20.

Step 3. Determine the set V of X-variables which are unate in the formula formed by the disjunction of terms in H_{ce} and $BCF(\widehat{\mathcal{G}}_j(X))$.

Step 4. Remove from V variables contained in the term p_j.

- If $V = \emptyset$, then continue to Step 5.
- Otherwise, remove from H_{ce} terms containing any variable in V and return to Step 3.

Step 5. Form the set $\tilde{H}_{ce} = \{p_1 \cdot P'_1, \ldots, p_k \cdot P'_k\}$ by affixing elements of $LABS$ to the corresponding elements of H_{ce}.

Step 6. Form $F_j(A, X)$, the disjunction of terms in $\tilde{H}_{ce}$.

Step 7.

1. Form the formula $\dot{F}_j(A, X)$:

$$\dot{F}_j(A, X) = ABSREL(F_j(A, X), BCF(\widehat{\mathcal{G}}_j(X))).$$

2. Form the formula $\tilde{F}_j(A, X)$:

$$\tilde{F}_j(A, X) = SIMPREL(\dot{F}_j(A, X), BCF(\widehat{\mathcal{G}}_j(X))).$$

Step 8. Using the quick method for forming the conjunctive eliminant (Procedure 3.1), form

$$ECON(\tilde{f}_j(A, X) + \widehat{\mathcal{G}}_j(X), X).$$

The function $\tilde{f}_j(A, X) + \widehat{\mathcal{G}}_j(X)$ is represented by $\tilde{F}_j(A, X) + BCF(\widehat{\mathcal{G}}_j(X))$.

Step 9. The resulting eliminant is equal to $g_j(A)$ which is represented by $BCF(g_j(A))$. The resulting formula is the inclusion formula IF_j for p_j. Return IF_j as the result.

Example 6.6: The interval IC5 is defined by lower and upper-bound functions g and h as follows:

$$\begin{aligned} g &= a'bd'e'f'g + a'b'c'df'g + a'b'cde'fg' + a'b'de'f'g + ab'de'f'g' + ab'c'de'f' \\ &\quad + abcde'g + abd'e'fg' + abcd'e'f + abcdf'g + abcef'g + bcd'e'fg' + bcdef'g \end{aligned} \tag{6.68}$$

$$\begin{aligned} h &= a'b'c'd'e'f'g' + ab'de'f' + b'de'f'g + acd'ef' + abcef' + acef'g + cdef'g \\ &\quad + bcef'g + a'bcd'f'g + b'cdf'g + acdf'g + b'de'fg' + ab'c'd'efg + a'b'c'dfg' \\ &\quad + bce'fg' + abde'f + a'ce'fg' + cde'fg' + ade'fg' + ab'de'g' + abcde'g \\ &\quad + abce'f + abe'fg' + a'b'df'g + a'bd'e'f'g \end{aligned} \tag{6.69}$$

The right-hand side of (6.69) is the Blake canonical form for h. Of the 25 prime implicants, the first 21 are conditionally-eliminable and the last four are essential prime implicants. The function $\hat{g}(X)$ is defined by the equation

$$\hat{g}(X) = abcdf'g + abcef'g + bcdef'g + a'bcd'e'fg' + a'b'cde'fg' + ab'de'f'g' + ab'c'de'f' \tag{6.70}$$

Using Procedure 6.1 we use the conditionally-eliminable prime implicants and $\widehat{G}$ to develop the following base for the interval:

$$\{ab'de'f', bcef'g, acdf'g, a'ce'fg'\}. \tag{6.71}$$

The ϕ-chart for the function is given by Table 6.8. The terms of the base form the columns, the conditionally-eliminable prime implicants form the rows.

We present the development of the inclusion formula for prime implicant $acdf'g$ which is associated with the label P_{11}.

Step 0. The function $\widehat{\mathcal{G}}_{11}(X)$ associated with the $\widehat{G}$-constraint for the prime implicant is defined by the equation

$$\widehat{\mathcal{G}}_{11} = (acdf'g - \hat{g}(X))/acdf'g = b'. \tag{6.72}$$

Hence, the $\widehat{G}$-constraint for $acdf'g$ is

$$b' = 0. \tag{6.73}$$

Step 1. The division of members of H_{ce} by $acdf'g$ is depicted by the column associated with $acdf'g$ in Table 6.8.

Step 2. b' is the Blake canonical form for $\widehat{\mathcal{G}}_{11}$.

Steps 3-4. Variables b and e are not unate; hence, no actions are taken.

Steps 5-6. From the column of Table 6.8 associated with P_{11}, we form $F_{11}(A, X)$:

$$F_{11}(A, X) = b'e'P'_2 + b'e'P'_3 + beP'_5 + eP'_6 + eP'_7 + beP'_8 + b'P'_{10} + P'_{11} + be'P'_{21}. \tag{6.74}$$

		P_2	P_8	P_{11}	P_{17}
		$ab'de'f'$	$bcef'g$	$acdf'g$	$a'ce'fg'$
$\widehat{G}_j$		cg	$a'd'$	b'	$b'd' + bd$
P_1	$a'b'c'd'e'f'g'$	0	0	0	0
P_2	$ab'de'f'$	1	0	$b'e'$	0
P_3	$b'de'f'g$	g	0	$b'e'$	0
P_4	$acd'ef'$	0	ad'	0	0
P_5	$abce f'$	0	a	be	0
P_6	$c\ ef'g$	0	a	e	0
P_7	$cdef'g$	0	d	e	0
P_8	$bcef'g$	0	1	be	0
P_9	$a'bcd'f'g$	0	$a'd'$	0	0
P_{10}	$b'cdf'g$	cg	0	b'	0
P_{11}	$acdf'g$	cg	ad	1	0
P_{12}	$b'de'fg'$	0	0	0	$b'd$
P_{13}	$ab'c'd'efg$	0	0	0	0
P_{14}	$a'b'c'dfg'$	0	0	0	0
P_{15}	$bce'fg'$	0	0	0	b
P_{16}	$abde'f$	0	0	0	0
P_{17}	$a'ce'fg'$	0	0	0	1
P_{18}	$cde'fg'$	0	0	0	d
P_{19}	$ade'fg'$	0	0	0	0
P_{20}	$ab'de'g'$	g'	0	0	0
P_{21}	$abcde'g$	0	0	be'	0

Table 6.8. ϕ-Chart for Example 6.6

Step 7. The formula b' representing $\widehat{\mathcal{G}}_{11}$ absorbs terms $b'e'P'_2$, $b'e'P'_3$, and $b'P'_{10}$ in $F_{11}(A,X)$. We thus form

$$\dot{F}_{11}(A,X) = beP'_5 + eP'_6 + eP'_7 + beP'_8 + P'_{11} + be'P'_{21}. \tag{6.75}$$

We then simplify the right-hand side of (6.75) relative to b' to form $\ddot{F}_{11}(A,X)$. We are able to delete the literal b from the terms beP'_5, beP'_8, and $be'P'_{21}$. The formula $\ddot{F}_{11}(A,X)$ is defined as follows:

$$\ddot{F}_{11}(A,X) = eP'_5 + eP'_6 + eP'_7 + eP'_8 + P'_{11} + e'P'_{21}. \tag{6.76}$$

Step 8. Given $\ddot{F}_{11}(A,X)$ and $\widehat{\mathcal{G}}_{11}$, we form the equation $\ddot{F}_{11}(A,X) + \widehat{\mathcal{G}}_{11} = 0$:

$$eP'_5 + eP'_6 + eP'_7 + eP'_8 + P'_{11} + e'P'_{21} + b' = 0 \tag{6.77}$$

Eliminating the X-argument b and e from equation (6.77) yields the resultant

$$P'_{11} + P'_5P'_{21} + P'_6P'_{21} + P'_7P'_{21} + P'_8P'_{21} = 0. \tag{6.78}$$

Step 9. The left-hand side of (6.78) is the inclusion formula for prime implicant $acdf'g$.

The inclusion formulas for the remaining prime implicants are:

- $ab'de'f'$: $P'_2 + P'_3P'_{20}$;
- $bcef'g$: $P'_8 + P'_4P'_7 + P'_5P'_7 + P'_6P'_7$; *and*
- $a'ce'fg'$: $P'_{17} + P'_{12}P'_{15} + P'_{15}P'_{18}$.

The $\widehat{G}$-constraint is a single constraint that encompasses both the EPI- and DC-constraints. A benefit of this constraint is that the handling of functions as well as intervals is dealt with implicitly. The resulting inclusion formulas are very useful for the subsequent rule reduction and search processes due to the fact that many prime implicants in Base #3 are likely to be in the final minimal formula.

An additional benefit to the inclusion formulas for Base #3 is that the set of inclusion formulas gives us a better estimate on the upper bound on the number of irredundant SOP formulas for a given function. The number of elements of this base is usually smaller than the $\widehat{G}$ base used in the Multiplication Method for forming all irredundant formulas. Hence, the method we used to estimate an upper bound in the Multiplication Method, i.e., forming the product of the number of terms in each inclusion formula, will be closer to the actual number of formulas. For example, for

function C5 the number of terms in the $\widehat{G}$ is 17, whereas it is 8 in Base #3. An upper bound on the number of irredundant formulas for C5 is shown in Table 5.5 to be 2.18×10^{15}. Using Base #3 and the inclusion formulas developed using Procedure 6.3, we calculate an upper bound of 1.09×10^{8}.

We have now discussed the formation of a base and the development of inclusion formulas for members of the base. The next step in the overall methodology is to reduce the inclusion formulas using reduction rules. Given a prime implicant base and the corresponding inclusion formulas, the reduction rules allow us to identify prime implicants to include in a minimal formula as well as prime implicants to discard from consideration. However, to be able to judge when to discard one prime implicant and when to keep another, we often have to determine whether one PI is better than another. To do this, we associate a value, or cost, with each prime implicant. In the next section, we discuss how these costs are determined.

Assignment of Costs to Terms

Common metrics for judging the cost of a circuit include the number of gates and the number of gate inputs. In the case of a two-level circuit, it is possible to assign a cost to each AND-gate in the circuit with respect to typical cost criterion such that the global cost of the circuit may be determined by summing the individual costs of the AND-gates. For example, if the cost criterion is the fewest gates, then each AND-gate is assigned a cost of 1. Summing the costs of the AND-gates yields a circuit cost equal to the number of AND-gates. If the cost criterion is the fewest gate-inputs (AND-gates only), then each AND-gate is assigned a cost equal to the number of its inputs. In this case, summing the costs of the AND-gates yields a circuit cost equal to the AND-gate inputs in the circuit. Because the cost of a two-level circuit may be determined by summing the cost of each AND gate, the cost of the circuit may be determined by studying each term in the corresponding SOP formula rather than the formula as a whole. A term in an SOP formula has a one-to-one correspondence with an AND gate in the corresponding two-level AND-OR circuit.

To form a minimal two-level AND-OR circuit, we derive a minimal SOP formula to represent the given specification. We desire a set of terms for which the sum of the costs of the corresponding AND gates is minimal. In the course of deriving a minimal formula, we often will have a choice among several terms when only one is required. We choose the term for which the cost of the corresponding AND gate is the smallest. Since each AND gate has an associated cost, we in turn associate this cost with the term which corresponds to the gate. We call this the *cost* of the term.

We now discuss how costs are assigned to terms for three different cost criteria: number of gates, number of literals, and number of gates and literals.

Criterion #1 - Fewest Gates: To implement a two-level AND-OR circuit with the minimum number of gates, we develop an SOP formula which contains the fewest terms. In this situation each term is assigned a cost of 1 since one term is as good as another.

Criterion #2 - Fewest Gate Inputs: If the total number of gate inputs is the primary design consideration, then the approach would be to find a formula which contains the fewest literals. In this approach the cost of each term is assigned in the following manner:

- a term consisting of a single literal is assigned a cost of 1, and
- a term which contains more than one literal is assigned a cost equaling the number of contained literals, plus one.

Prime implicants are terms which contain the fewest literals of the terms which may appear in an SOP formula. Hence, all terms in a minimal SOP formula developed to minimize gate inputs must be prime implicants.

Criterion #3 - Fewest Gates (Primary); Fewest Gate Inputs (Secondary): When we require a circuit with the fewest gates and also demand that the number of gate inputs be minimised, then we must find an SOP formula which consists of the fewest terms as the primary consideration and the fewest literals as a secondary concern. We evaluate the cost of each term as follows:

- a term consisting of a single literal is assigned a cost of k, where k is a large constant, and
- a term consisting of l literals, $l > 1$, is assigned a cost of $k + l$.

Care must be taken to ensure that k is properly scaled. k must be at least one order of magnitude greater than the number of terms in the resulting SOP formula.

Given a set $T = \{t_1, \ldots, t_k\}$ of terms, a set $A = \{P_1, \ldots, P_k\}$ of labels associated with the terms, and the cost criterion used to assign the cost, Procedure 6.4 associates a cost with each term in T. An association list (defined earlier) is returned in which the label P_i corresponding to the term t_i is paired with the term's cost c_i. Example 6.7 illustrates the lists returned by Procedure 6.4 for the three cost criteria.

Procedure 6.4 (Assignment of Costs to Terms): Given a set $T = \{t_1, \ldots, t_k\}$ of terms, a set $A = \{P_1, \ldots, P_k\}$ of labels associated with the terms, and a cost criterion $CRITERION$, we develop a list associating terms (labels) with costs in the following manner:

Step 0. Initialize an association list T_{assoc} to the empty set $\emptyset$.

Step 1.

- If T is empty, then return T_{assoc}. T_{assoc} is the list associating terms (labels) with their costs.
- Otherwise, continue to Step 2..

Step 2. Remove the first term t_i from T and calculate the cost c_i of t_i. Remove the first label P_i

- If $CRITERION$ = Fewest Gates, then the cost c_i of t_i is 1.
- If $CRITERION$ = Fewest Gate Inputs, then the cost c_i of t_i is calculated as follows:
  - if t_i consists of a single literal, then $c_i = 1$; or
  - if t_i contains l literals, $l > 1$, then $c_i = l + 1$.
- If $CRITERION$ = Fewest Gates/Fewest Gate Inputs, then the cost c_i of t_i is calculated as follows:
  - if t_i consists of a single literal, then $c_i = k$; or
  - if t_i contains l literals, $l > 1$, then $c_i = k + l$.

  Note: We assume that k is a large, pre-defined constant.

Form a list $(P_i \ c_i)$ and append it to T_{assoc}. Return to Step 1.

Example 6.7: Given the set PI of prime implicants of interval IC3,

$$PI = \{a'b'cdef', a'cde'f, a'bcd'ef, a'bc'df', a'bc'ef', a'b'c'd'e'f', ac'ef \\ abcde'f', abc'de, abdef, ab'c'df, ab'cd'e', ab'cd'f', bc'def'\},$$

and a set $P = \{P_1, P_2, \ldots, P_{14}\}$ of labels associated with elements of PI, we use Procedure 6.4 to associate costs with each prime implicant.

Using the fewest-gates criterion, Procedure 6.4 returns the following list:

```
( (P1 1) (P2 1) (P3  1) (P4  1) (P5  1) (P6  1) (P7  1)
  (P8 1) (P9 1) (P10 1) (P11 1) (P12 1) (P13 1) (P14 1) ).
```

When the fewest-gate-inputs criterion is used, the following list is returned:

```
( (P1 6) (P2 6) (P3  7) (P4  6) (P5  6) (P6  7) (P7  5)
  (P8 7) (P9 6) (P10 6) (P11 6) (P12 6) (P13 6) (P14 6) ).
```

Given that $k = 100$, the list

```
( (P1 105) (P2 105) (P3  106) (P4  105) (P5  105) (P6  106) (P7  104)
  (P8 106) (P9 105) (P10 105) (P11 105) (P12 105) (P13 105) (P14 105) )
```

is returned when the third cost criterion is used.

The assignment of costs to prime implicants of a function is particularly important for judging the benefit of one prime implicant over another. In the next section, reduction rules are presented which allow us in many situations to select prime implicants to place in a minimal SOP formula as well as to discard from consideration. In many cases, these decisions are based on the cost of a given prime implicant.

Reduction Rules

The use of prime implicants as a base and the assignment of costs to prime implicants are done to facilitate the use of reduction rules. Reduction rules allow us to identify prime implicants to place in a minimal formula F as well as prime implicants to discard from consideration. Furthermore, these rules provide a mechanism for reducing the number of terms and literals in each of the inclusion formulas. Thus, reduction rules are very important in reducing the complexity of the problem of developing a minimal formula.

One of the key ideas on which reduction rules are based is the concept of domination as illustrated in prime implicant tables. We first present an overview of prime implicant charts and related terminology. We then introduce reduction rules, originally presented in (Gaine 64) and (Chang 65), which are applicable when Base #1 is used. Finally, we propose a revised set of rules which is applied when Bases #2 and #3 are used.

Prime-Implicant Tables. A *prime-implicant table* is a table which conveys the coverage of minterms by prime implicants of a function (Quine 52). The minterms of the function g form the columns, the useful prime implicants of h make up the rows. The prime-implicant table for the function

$$f(a,b,c,d,e) = d'e + cde' + a'cd + a'ce + ab'd + ab'e + b'cd + b'ce \tag{6.79}$$

of Example 5.3, for example, is given by Table 6.9. Typically, the prime implicants are ordered such that the PIs with the least cost appear in the highest rows and the PIs with the highest cost appear in the lowest rows. The minterms appear in order of their decimal notation. The rows of Table 6.9 correspond respectively to the prime implicants on the right-hand side of (6.79), the Blake canonical form for f.

	1	5	6	7	9	13	14	15	17	18	19	21	22	23	25	29	30
P_1	×	×			×	×			×			×			×	×	
P_2			×				×						×				×
P_3			×	×			×	×									
P_4		×		×		×		×									
P_5										×	×		×	×			
P_6									×		×	×		×			
P_7			×	×									×	×			
P_8		×		×								×		×			

Table 6.9. Prime-Implicant Table for Example 5.3

If a single × appears in a column, the minterm associated with the column is covered by a single prime implicant. The prime implicant in the corresponding row is thus an essential prime implicant. In Table 6.9, for example, minterm m_{30} is covered only by the prime implicant corresponding to P_2. Thus, the PI corresponding to P_2 is an essential prime implicant. PIs corresponding to P_1 and P_5 are also essential prime implicants. Essential prime implicants are placed in a ***partial sum, PS***, which contains PIs selected for inclusion in a minimal formula.

A ***reduced*** table is formed by deleting all columns covered by the essential prime implicants as well as the rows corresponding to the essential PIs. A reduced table derived from Table 6.9 is Table 6.10. Prime implicants whose corresponding rows in a reduced table contain no ×'s are inessential. Thus, the prime implicant corresponding to P_6 is an inessential PI. Such rows are also deleted from the table (we left P_6 in Table 6.10 for illustration purposes). Prime implicants which remain after the first table-reduction are conditionally-eliminable prime implicants.

	m_7	m_{15}
P_3	×	×
P_4	×	×
P_6		
P_7	×	
P_8	×	

Table 6.10. Reduced PI Table for Example 5.3

In a reduced PI table, a row P_m is said to *dominate* a row P_n if P_m has $\times$'s in every column that P_n does and the cost c_m of the corresponding prime implicant is less than or equal to the cost c_n. Row P_n is said to be *dominated;* row P_m is said to be *dominating.* This is a property called *row dominance.* In Table 6.10, rows P_3 and P_4 dominate all other rows (including each other). All dominated rows are removed from the PI table. The table is then examined to determine if a remaining row is the only row which contains an $\times$ for some column. If for some column a remaining row is the only row containing an $\times$, the prime implicant corresponding to the row must be selected for inclusion in the current partial sum. Such a prime implicant is called a *secondary essential* prime implicant. For example, if we denote P_3 in Table 6.10 as the dominating row, then rows P_8, P_7, and P_4 are deleted. Once row P_4 is deleted, P_3 becomes a secondary essential PI. Columns covered by the new secondary essential PI, as well as the corresponding row, are then removed from the table to form a new reduced table.

Example 6.8: The prime-implicant table for the interval $[g, h]$, defined by

$$g = bcd + ab'cd' \tag{6.80}$$
$$h = ac + bd + b'c + cd, \tag{6.81}$$

from Example 5.9 is given by Table 6.11. Row P_1 dominates row P_3; hence, row P_3 is deleted. Prime implicant ac corresponding to row P_1 then become a secondary essential PI because it is then the only PI which covers m_{10}. We then delete the columns covered by P_1. The result is Table 6.12.

In the revised table, rows P_2 and P_4 dominate each other. We arbitrarily choose one for inclusion in a minimal formula. If we choose the PI corresponding to P_2, i.e., bd, then a minimal formula is $ac + bd$.

	m_7	m_{10}	m_{15}
P_1		$\times$	$\times$
P_2	$\times$		$\times$
P_3		$\times$	
P_4	$\times$		$\times$

Table 6.11. PI Table for Example 6.8

After all potential reduction is performed using row dominance, it is sometimes possible to further reduce a PI table using a property called *column dominance.* In a reduced PI table, a

	m_7
P_2	×
P_4	×

Table 6.12. Revised Table for Example 6.8

column m_j is said to *dominate* a column m_k if m_j has ×'s in every row that m_k does. Column m_k is said to be *dominated;* column m_j is said to be *dominating.* Column m_7 dominates column m_{15} in Table 6.10. Dominating columns may be deleted to further reduce a PI table.

The order in which rows and columns are deleted using row and column dominance may change the prime implicants that appear in the resulting formula. However, the cost of the final formula will not change. (Murog 79:171)

After all potential reduction of a PI table is carried out, it may occur that the table may not be further reduced. Such a table is called *cyclic.* All columns contain at least two ×'s in a cyclic table (although not all tables with this property are cyclic). Table 6.13 depicts a cyclic prime implicant table. Petrick formulated a method for determining the remaining PIs to include in a minimal formula (Petri 59):

1. For each column j, form an alterm depicting the prime implicants which cover the associated minterm m_j.
2. Form the conjunction of the alterms and compute the product while deleting absorbed terms in the process.
3. Each term in the resulting formula P denotes a set of prime implicants which completes an irredundant formula. (Moreover, each term of the formula is itself a prime implicant of the function represented by P.) Choose the least-cost set of prime implicants to complete a minimal formula.

The conjunction of alterms for Table 6.13 is the formula

$$(P_1 + P_4) \cdot (P_1 + P_2) \cdot (P_2 + P_3) \cdot (P_3 + P_4). \tag{6.82}$$

	m_a	m_b	m_c	m_d
P_1	×	×		
P_2		×	×	
P_3			×	×
P_4	×			×

Table 6.13. A Cyclic Prime Implicant Table

Formula (6.82) is often referred to as a *Petrick function*. Computing the conjunction and simplifying we develop the formula

$$P_1P_3 + P_2P_4. \tag{6.83}$$

Assuming the each prime implicant costs the same, either the set of PIs corresponding to P_1 and P_3 or the set corresponding to P_2 and P_4 may be chosen to complete the formation of a minimal formula.

Now that we have presented prime implicant tables and the concepts of row and column dominance, we present reduction rules applicable to the inclusion formulas used in this work. We will relate the reduction rules to the aforementioned concept of row dominance. The first set of rules is applied when Base #1 is used.

Reduction Rules for Base #1. Gaines (Gaine 64) introduced a set of reduction rules which he applied to implication relations denoting the coverage of a Blake canonical form base by the prime implicants of a function. The implication relations used by Gaines are theoretically the same as the inclusion formulas we use in this work. For example, if we form the inclusion formula

$$P_1' + P_2'P_3' + P_2'P_4' \tag{6.84}$$

to denote the coverage of prime implicant P_1 by the prime implicants of a function, then Gaines would form the set of implication relations denoting the coverage of P_1:

$$\begin{aligned} P_1 &\leq P_2 + P_3 \\ P_1 &\leq P_2 + P_4. \end{aligned} \tag{6.85}$$

It is obvious that P_1 also covers itself. Gaines described techniques for forming implication relations from both prime implicant tables as well as a Blake canonical form.

Gaines's rules are now restated in a fashion that applies to our inclusion formulas. The set $IF = \{IF_1, IF_2, \ldots, IF_m\}$ of inclusion formulas denotes coverage of terms of the base. We assume at the outset that labels denoting essential prime implicants of the original function do not appear in inclusion formulas. Additionally, a partial sum PS is initialized prior to executing reduction rules by placing the essential prime implicants into PS. Each rule consists of three elements:

1. a condition which may occur in an inclusion formula IF_j;
2. the significance of the condition; and
3. actions to be taken due to the occurrence of the condition.

Theorems justifying these rules are found in (Gaine 64).

Rule I. If the term 1 appears in the inclusion formula IF_j associated with prime implicant P_j, then P_j is covered by prime implicants in PS.

1. Remove IF_j from IF.
2. Delete every term in which the variable P_j appears in the remaining inclusion formulas IF_k, $k \neq j$.

Rule II. If the term P_j' is the only term in inclusion formula IF_j, then P_j is a secondary essential prime implicant.

1. Remove IF_j from IF.
2. Add P_j to the partial sum PS.

3. Remove the literal P_j' from terms in the remaining inclusion formulas IF_k, $k \neq j$. (If P_j' is the only literal, then the removing P_j' leaves the term 1.)
4. Delete newly-absorbed terms, if any, in all formulas IF_k.

Rule III. If the literal P_i' appears as a single term in the inclusion formula IF_j associated with prime implicant P_j, and the cost c_i associated with P_i is less than or equal to the cost c_j associated with P_j, then P_i dominates P_j.

1. Remove IF_j from IF.
2. Delete every term in which the variable P_j appears in the remaining inclusion formulas IF_k, $k \neq j$.

In all cases, Rules I and II must be applied until they can no longer be applied, prior to using Rule III. After Rule III is used one time, Rules I and II must be applied exhaustively before again using Rule III. Rules I and III combined are equivalent to removing a dominated row from a prime implicant table. (Gaine 64:179)

Since Gaines used the Blake canonical form as a base, he structured the rules so that inessential and essential prime implicants are immediately identified. After all inessential and essential prime implicants are dealt with, row domination is invoked by Rule III to eliminate a conditionally-eliminable prime implicant. Typically, the first rule that will be invoked for our inclusion formulas is Rule III, since we use a CEPI base and form inclusion formulas denoting coverage of terms of the base by CEPIs. However, on occasion a conditionally-eliminable prime implicant of the function h in an interval $[g, h]$ may be covered by the combination of the essential prime implicants and the don't-care function $h - g$. Rule I would identify such a prime implicant.

After all three rules have been applied exhaustively, a subset of IF remains. Formulas contained in IF after rule applications generally contain fewer terms and remaining terms consist of fewer literals than formulas which exist in IF prior to rule applications. Gaines presented a heuristic rule to facilitate the choice of a set of prime implicants from the remaining inclusion

formulas. However, this rule did not guarantee the minimality of the resulting formula. We will present a search process for determining the remaining prime implicants to include in a minimal SOP formula. Additionally, the reduction rules are applied throughout the search process.

Cutler (Cutle 80, Cutle 87) essentially implemented Gaines's rules in his work. Additionally, Cutler added the consideration of cost of a prime implicant in Rule III as well as the use of a search process to select prime implicants from the final inclusion formulas. Gaines treated all prime implicants as having equal cost.

Prior to presenting a procedure which implements Gaines's rules, we discuss a separate rule attributable to Chang and Mott (Chang 65) which identifies conditionally-eliminable prime implicants which will never appear in an irredundant disjunctive form. In Chang and Mott's terminology, an essential prime implicant is a PI which appears in every IDF. An *absolutely-dispensible* prime implicant is a PI which appears in no IDF. All of the prime implicants that we classify as inessential are absolutely dispensible. Moreover, certain conditionally-eliminable PIs are absolutely dispensible; such PIs are identified by Chang and Mott's rule. Chang and Mott called prime implicants which appear in at least one IDF *conditionally-eliminable* prime implicants.[3]

Chang and Mott's rule is used to identify conditionally-eliminable prime implicants (by our definition) which are absolutely dispensible. If a prime implicant is identified as absolutely dispensible, then the inclusion formulas are modified accordingly to discard the prime implicant from consideration. Chang and Mott's rule is stated as follows:

Chang and Mott's Rule. Given a term $P_1' \cdots P_l'$ other than P_j' in the inclusion formula IF_j, P_j is absolutely dispensible if and only if the literal P_j' does not appear in the inclusion formulas $IF_1, \ldots, IF_l$ corresponding to the prime implicants denoted by the labels in the term. If P_j is absolutely dispensible, then take the following actions:

[3] We use the same terminology in a somewhat broader sense; some PIs that we call conditionally-eliminable prime implicants are denoted absolutely dispensible by Chang and Mott.

1. Remove IF_j from IF.
2. Delete every term in which the variable P_j appears in the remaining inclusion formulas IF_k, $k \neq j$.

By analysing this rule we observe that for a prime implicant P_j not to be absolutely dispensible there must be a "circular" relationship such that P_j works to cover all prime implicants that in turn cover it. However, there are a number of drawbacks to using this rule:

- Applying the rule takes significant processing time because the circularity of the relationship must be analyzed for virtually every term of every inclusion formula.
- Inclusion formulas must exist for every prime implicant, i.e., we must use Base #1 at the minimum to be able to check for circularity.
- In the course of applying Gaines's rules, most if not all of the prime implicants which are absolutely dispensible are identified.

We have not endeavored to prove that the exhaustive application of Gaines's rules leads to the deletion of all absolutely dispensible prime implicants that are identified using Chang and Mott's rule.

The flowchart shown in Figure 6.1 depicts the manner in which Gaines's rules are applied. Integrated into the flow is Chang and Mott's rule. However, this rule may be bypassed and can be activated after Gaines's rules have been exhaustively applied.

Procedures 6.5–6.10 implement Rules I-III and Chang and Mott's Rule. Procedure 6.5 is a helping procedure which deletes all terms which contain the literal P'_k in each formula in a set IF of formulas. It is used by procedures which implement Rules I, III, and Chang and Mott's Rule. Procedure 6.6 is a helping procedure which removes literals from terms which contain the literal P'_k in each formula in a set IF of formulas. Procedures 6.7-6.9 implement Rules I-III, respectively. Procedure 6.10 implements Chang and Mott's Rule.

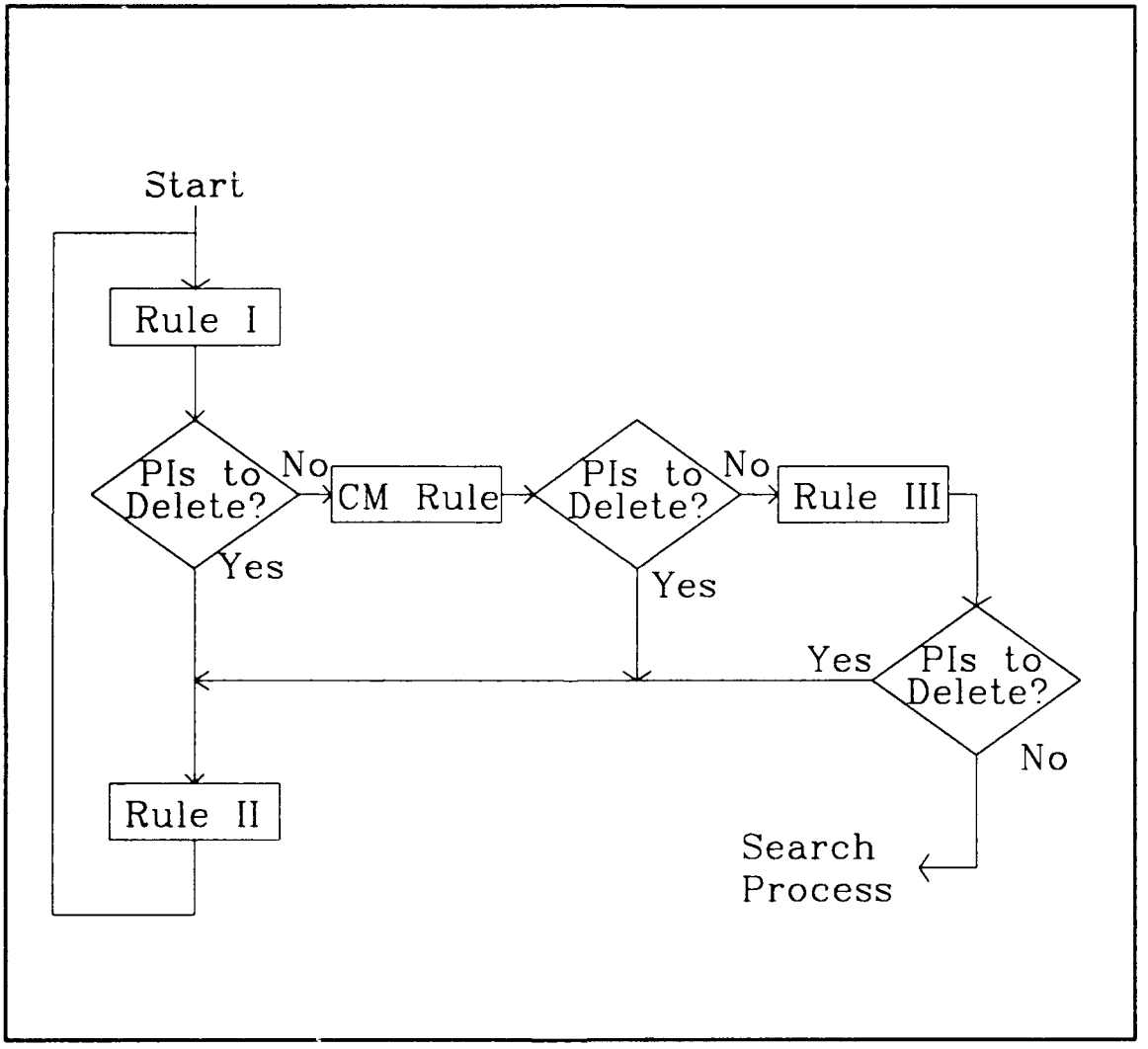


Figure 6.1. Flowchart of Reduction-Rule Application

Procedure 6.5 (Deletion of Terms): Given a set $\widehat{IF} = \{IF_1, IF_2, \ldots, IF_s\}$ of inclusion formulas, and a label P_k denoting a prime implicant, all terms containing the literal P_k' are removed from formulas in $\widehat{IF}$ in the following manner:

Step 0. Initialise an accumulator IF_{acc} to the empty set $\emptyset$.

Step 1.

- If $\widehat{IF} = \emptyset$, then IF_{acc} is the revised set of inclusion formulas. Return IF_{acc}.
- Otherwise, continue to Step 2.

Step 2.

1. Remove the first formula from $\widehat{IF}$ and call it F_{old}.
2. Initialize an accumulator F_{new} to the empty set $\emptyset$.

Step 3.

- If F_{old} is empty, then F_{new} is a revised inclusion formula. Append F_{new} to IF_{acc}, and return to Step 1.
- Otherwise, continue to Step 4.

Step 4. Remove the first term t from F_{old} and determine whether P_k' is contained in the term.

- If P_k' is contained in t, then do nothing.
- Otherwise, place t in F_{new}.

Return to Step 3.

Procedure 6.6 (Removal of Literals): Given a set $\widehat{IF} = \{IF_1, IF_2, \ldots, IF_s\}$ of inclusion formulas, and a label P_k denoting a prime implicant, the literal P_k' is removed from all terms containing the literal P_k' in $\widehat{IF}$ in the following manner:

Step 0. Initialise an accumulator IF_{acc} to the empty set $\emptyset$.

Step 1.

- If $\widehat{IF} = \emptyset$, then IF_{acc} is the revised set of inclusion formulas. Return IF_{acc}.
- Otherwise, continue to Step 2.

Step 2.

1. Remove the first formula from $\widehat{IF}$ and call it F_{old}.
2. Initialise an accumulator F_{new} to the empty set $\emptyset$.

Step 3.

- If F_{old} is empty, then F_{new} is a revised inclusion formula. Form $ABS(F_{new})$ and append it to IF_{acc}. Return to Step 2.
- Otherwise, continue to Step 4.

Step 4. Remove the first term t from F_{old} and determine whether P'_k is contained in the term.

- If P'_k is contained in t, then remove P'_k from t.
- Otherwise, do nothing.

Place t in F_{new} and return to Step 3.

Procedure 6.7 (Rule I): Given a set $IF = \{IF_1, IF_2, \ldots, IF_m\}$ of inclusion formulas, Rule I is implemented as follows:

Step 0.

1. Initialise an accumulator $P_{discard}$ to the empty set $\emptyset$.
2. Initialise an accumulator $IF_{checked}$ to the empty set $\emptyset$.

Step 1.

- If $IF = \emptyset$, then $IF_{checked}$ is the revised set of inclusion formulas. Return $IF_{checked}$ and $P_{discard}$.
- Otherwise, remove the first inclusion formula from IF and denote it as IF_j.

Step 2.

- If $IF_j \equiv 1$, then the prime implicant denoted by P_j is completely covered by prime implicants in the current partial sum. Place P_j in $P_{discard}$ and go to Step 3.
- Otherwise, append IF_j to $IF_{checked}$ and return to Step 1.

Step 3. Using Procedure 6.5, delete terms in each formula in $IF_{checked}$ which contain the variable P_j. Replace $IF_{checked}$ with the set of formulas returned by Procedure 6.5.

Step 4. Using Procedure 6.5, delete terms in each formula in IF which contain the variable P_j.

1. Replace IF with the set of formulas returned by Procedure 6.5.
2. Return to Step 1.

Procedure 6.8 (Rule II): Given a set $IF = \{IF_1, IF_2, \ldots, IF_m\}$ of inclusion formulas, Rule II is implemented as follows:

Step 0.

1. Initialise an accumulator $\widehat{PS}$ to the empty set $\emptyset$.
2. Initialise an accumulator $IF_{checked}$ to the empty set $\emptyset$.

Step 1.

- If $IF = \emptyset$, then $IF_{checked}$ is the revised set of inclusion formulas. Return $IF_{checked}$ and $\widehat{PS}$.
- Otherwise, remove the first inclusion formula from IF and denote it IF_j.

Step 2.

- If $IF_j \equiv P'_j$, i.e., IF_j consists of the single term P'_j, then the prime implicant denoted by P_j is a secondary essential prime implicant. Place P_j in $\widehat{PS}$ and continue to Step 3.
- Otherwise, append IF_j to $IF_{checked}$ and return to Step 1.

Step 3. Using Procedure 6.6, remove the literal P'_j from terms which contains it in each formula in $IF_{checked}$. Replace $IF_{checked}$ with the set of formulas returned by Procedure 6.6.

Step 4. Using Procedure 6.6, remove the literal P'_j from terms which contains it in each formula in IF.

1. Replace IF with the set of formulas returned by Procedure 6.6.
2. Return to Step 1.

Procedure 6.9 (Rule III - Version #1): Given a set $IF = \{IF_1, IF_2, \ldots, IF_m\}$ of inclusion formulas and a cost list C : $((P_1\ c_1)\ (P_2\ c_2)\ \cdots\ (P_m\ c_m))$, Rule III is implemented as follows:

Step 0.

1. Initialize an accumulator $IF_{checked}$ to the empty set $\emptyset$.
2. Initialize an accumulator $P_{discard}$ to the empty set $\emptyset$.

Step 1.

- If $IF = \emptyset$, then no dominated variables were found. Return $IF_{checked}$ and $P_{discard}$.
- Otherwise, continue to Step 2.

Step 2.

1. Remove the first inclusion formula from IF and denote it as IF_j.
2. Initialize an accumulator IF_{acc} to the empty set $\emptyset$.

Step 3.

- If IF_j is empty, we have checked all of the terms in IF_j. Place IF_{acc} in $IF_{checked}$ and return to Step 1.
- Otherwise, continue to Step 4.

Step 4. Remove the first term t from IF_j and determine if t consists of a single literal P'_l in which $l \neq j$.

- If t is a single literal P'_l, then continue to Step 5.
- Otherwise, place t in IF_{acc} and return to Step 3.

Step 5. Using the association list C, determine if $c_l \leq c_j$.

- If $c_l \leq c_j$, then place P_j in $P_{discard}$ and continue to Step 6.
- Otherwise, place t in IF_{acc} and return to Step 3.

Step 6. Using Procedure 6.5, delete terms in each formula in $IF_{checked}$ which contain the variable P_j. Replace $IF_{checked}$ with the set of formulas returned by Procedure 6.5.

Step 7. Using Procedure 6.5, delete terms in each formula in IF which contain the variable P_j. Replace IF with the set of formulas returned by Procedure 6.5.

Step 8.

1. Append $IF_{checked}$ to IF. IF is the revised set of inclusion formulas.
2. Return IF and $P_{discard}$.

Procedure 6.10 (Chang & Mott's Rule): Given a set $IF = \{IF_1, IF_2, \ldots, IF_m\}$ of inclusion formulas, Chang & Mott's Rule is implemented as follows:

Step 0.

1. Initialize an accumulator $IF_{checked}$ to the empty set $\emptyset$.
2. Initialize an accumulator PC_{list} to the empty set $\emptyset$.
3. Initialize an accumulator IF_{acc} by copying into it the contents of IF.
4. Initialize an accumulator $P_{discard}$ to the empty set $\emptyset$.

Step 1.

- If $IF_{acc} = \emptyset$, then PC_{list} contains a set of lists—one for each prime implicant P_j—in which each denotes the prime implicants that can be used to cover P_j. Go to Step 5.
- Otherwise, continue to Step 2.

Step 2.

1. Remove the first inclusion formula from IF_{acc} and denote it as IF_j.
2. Initialize a list PC_j to the empty set $\emptyset$.

Step 3.

- If IF_j is empty, we have checked all of the terms in IF_j. Append PC_j to PC_{list} and return to Step 1.
- Otherwise, continue to Step 4.

Step 4. Remove the first term t from IF_j:

- If t is a single literal P_j', then do nothing.
- Otherwise, if any of the literals in t are not members of PC_j, then add them to PC_j.

Return to Step 3.

Step 5.

- If $IF = \emptyset$, then $IF_{checked}$ is the revised set of inclusion formulas. Return $IF_{checked}$ and $P_{discard}$.
- Otherwise, continue to Step 6.

Step 6.

1. Remove the first inclusion formula from IF and denote it as IF_j.
2. Initialize IF_{acc} to the empty set $\emptyset$.

Step 7.

- If IF_j is empty, we have checked all of the terms in IF_j. Place IF_{acc} in $IF_{checked}$ and return to Step 5.
- Otherwise, continue to Step 8.

Step 8. Remove the first term t from IF_j.

- If t is the single literal P_j', then place t in IF_{acc} and return to Step 7.
- Otherwise, initialize an accumulator T_{acc} by copying into it the contents of t. Continue to Step 9.

Step 9.

- If T_{acc} is empty, then P_j is absolutely dispensible. Place P_j in $P_{discard}$ and go to Step 11.
- Otherwise, continue to Step 10.

Step 10. Remove the first literal P_l' from T_{acc} and determine if P_j appears in PC_l in PC_{list}.

- If P_j appears in PC_l, we must check the next term of IF_j to determine whether P_j is absolutely dispensible. Place t in IF_{acc} and return to Step 7.
- If P_j does not appear in PC_l, then return to Step 9.

Step 11. Using Procedure 6.5, delete terms in each formula in $IF_{checked}$ which contain the variable P_j. Replace $IF_{checked}$ with the set of formulas returned by Procedure 6.5.

Step 12. Using Procedure 6.5, delete terms in each formula in IF which contain the variable P_j.

1. Replace IF with the set of formulas returned by Procedure 6.5.
2. Return to Step 5.

Procedure 6.11 combines the rules to implement the process depicted in Figure 6.1 for Base #1. Chang and Mott's Rule is included as Step 3, but may skipped. Example 6.9 demonstrates the application of the reduction rules on a set of inclusion formulas.

Procedure 6.11 (Reduction Rules - Set #1): Given a set $IF = \{IF_1, IF_2, \ldots, IF_m\}$ of inclusion formulas and a cost list C : $((P_1\ c_1)\ (P_2\ c_2)\ \cdots\ (P_m\ c_m))$, the reduction rules are implemented as follows:

Step 0.

1. Initialise an accumulator $P_{discard}$ to the emptyset $\emptyset$.
2. Initialise an accumulator PS to the emptyset $\emptyset$.

Step 1. Given IF, use Procedure 6.7 to apply Rule I. Procedure 6.7 returns a set $\widehat{P}_{discard}$ of variables and a set $IF_{revised}$ of inclusion formulas.

- If $\widehat{P}_{discard}$ is empty, then no variables were found to be discarded. Skip to Step 3.
- Otherwise, append $\widehat{P}_{discard}$ to $P_{discard}$ and replace IF with $IF_{revised}$. Continue to Step 2.

Step 2. Given IF, use Procedure 6.8 to apply Rule II. Procedure 6.8 returns a set $\widehat{PS}$ of variables and a set $IF_{revised}$ of inclusion formulas.

- If $\widehat{PS}$ is empty, then no variables were found to be placed in the partial sum. Continue to Step 3.
- Otherwise, append $\widehat{PS}$ to PS and replace IF with $IF_{revised}$. Return to Step 1.

Step 3. Given IF, use Procedure 6.10 to apply Chang & Mott's Rule. Procedure 6.10 returns a set $\widehat{P}_{discard}$ of variables and a set $IF_{revised}$ of inclusion formulas.

- If $\widehat{P}_{discard}$ is empty, then no variables were found to be discarded. Continue to Step 4.
- Otherwise, append $\widehat{P}_{discard}$ to $P_{discard}$ and replace IF with $IF_{revised}$. Return to Step 2.

Note: This step is optional; if not used continue to Step 4.

Step 4. Given IF and the cost list C, use Procedure 6.9 to apply Rule III. Procedure 6.9 returns a variable $\widehat{P}_{discard}$ and a set $IF_{revised}$ of inclusion formulas.

- If $\widehat{P}_{discard}$ is empty, then no variables were found to be discarded. Continue to Step 5.
- Otherwise, place $\widehat{P}_{discard}$ in $P_{discard}$ and replace IF with $IF_{revised}$. Return to Step 2.

Step 5. No further reduction can take place. Return the inclusion formulas IF, the partial sum PS, and the set $P_{discard}$ of variables to discard.

Example 6.9: Using Base #1 and Procedure 6.2, the following inclusion formulas IF are developed for the interval IC5:

$$
\begin{aligned}
IF_2 &= P_2' + P_3'P_{20}' \\
IF_3 &= P_3' + P_2' \\
IF_4 &= P_4' + P_5' + P_6' + P_8' \\
IF_5 &= P_5' + P_6' + P_8' + P_4'P_7' + P_4'P_{11}' \\
IF_6 &= P_6' + P_5' + P_8' + P_4'P_7' + P_4'P_{11}' \\
IF_7 &= P_7' + P_8' \\
IF_8 &= P_8' + P_4'P_7' + P_5'P_7' + P_6'P_7' \\
IF_{11} &= P_{11}' + P_5'P_{21}' + P_6'P_{21}' + P_7'P_{21}' + P_8'P_{21}' \\
IF_{12} &= P_{12}' + P_{17}' + P_{18}' \\
IF_{15} &= P_{15}' + P_{17}' \\
IF_{17} &= P_{17}' + P_{12}'P_{15}' + P_{15}'P_{18}' \\
IF_{18} &= P_{18}' + P_{12}' + P_{17}' \\
IF_{20} &= P_{20}' + P_2' \\
IF_{21} &= P_{21}' + P_{11}'.
\end{aligned}
$$

Additionally, the inclusion formula associated with the prime implicant corresponding P_{16} is the term 1. We use Procedure 6.11 to reduce the set of inclusion formulas. We assume that the cost criterion is fewest gates; thus, the cost of each PI is equal to 1.

Step 0. Initialize $P_{discard} = \emptyset$ and $PS = \emptyset$.

Step 1. Since IF_{16} is the term 1, we delete P_{16} by Rule I. P_{16} is then added to $P_{discard}$.

Step 2. No PIs are found to place in the partial sum using Rule II.

Step 3. Using Chang & Mott's Rule, no PIs are found to discard.

Step 4. The first inclusion formula in which a dominated PI is found is the formula IF_3. Hence, IF_3 is deleted as well as any terms in other formulas which contain P_3'. $P_{discard} = \{P_{16}, P_3\}$. The revised set IF is:

$$
\begin{aligned}
IF_2 &= P_2' \\
IF_4 &= P_4' + P_5' + P_6' + P_8' \\
IF_5 &= P_5' + P_6' + P_8' + P_4'P_7' + P_4'P_{11}' \\
IF_6 &= P_6' + P_5' + P_8' + P_4'P_7' + P_4'P_{11}' \\
IF_7 &= P_7' + P_8' \\
IF_8 &= P_8' + P_4'P_7' + P_5'P_7' + P_6'P_7' \\
IF_{11} &= P_{11}' + P_5'P_{21}' + P_6'P_{21}' + P_7'P_{21}' + P_8'P_{21}' \\
IF_{12} &= P_{12}' + P_{17}' + P_{18}' \\
IF_{15} &= P_{15}' + P_{17}' \\
IF_{17} &= P_{17}' + P_{12}'P_{15}' + P_{15}'P_{18}' \\
IF_{18} &= P_{18}' + P_{12}' + P_{17}' \\
IF_{20} &= P_{20}' + P_2' \\
IF_{21} &= P_{21}' + P_{11}'.
\end{aligned}
$$

Step 2. Since $IF_2 = P_2'$, P_2 is added to partial sum PS. IF_2 is then removed from the set IF. In addition, IF_{20} is then represented by the term 1.

Step 1. Since IF_{20} is the term 1, we delete P_{20} by Rule I. $P_{discard} = \{P_{16}, P_3, P_{20}\}$.

Steps 2,1,3. No actions taken.

Step 4. P_5 dominates P_4 in IF_4. Hence, IF_4 is deleted as well as any terms in other formulas which contain P_4'. $P_{discard} = \{P_{16}, P_3, P_{20}, P_4\}$. The revised set IF is:

$$
\begin{aligned}
IF_5 &= P_5' + P_6' + P_8' \\
IF_6 &= P_6' + P_5' + P_8' \\
IF_7 &= P_7' + P_8' \\
IF_8 &= P_8' + P_5'P_7' + P_6'P_7' \\
IF_{11} &= P_{11}' + P_5'P_{21}' + P_6'P_{21}' + P_7'P_{21}' + P_8'P_{21}' \\
IF_{12} &= P_{12}' + P_{17}' + P_{18}' \\
IF_{15} &= P_{15}' + P_{17}' \\
IF_{17} &= P_{17}' + P_{12}'P_{15}' + P_{15}'P_{18}'
\end{aligned}
$$

$$
\begin{aligned}
IF_{18} &= P'_{18} + P'_{12} + P'_{17} \\
IF_{21} &= P'_{21} + P'_{11}.
\end{aligned}
$$

Steps 2,1,3. No actions taken.

Step 4. P_6 dominates P_5 in IF_5. Hence, IF_5 is deleted as well as any terms in other formulas which contain P'_5. $P_{discard} = \{P_{16}, P_3, P_{20}, P_4, P_5\}$. The revised set IF is:

$$
\begin{aligned}
IF_6 &= P'_6 + P'_8 \\
IF_7 &= P'_7 + P'_8 \\
IF_8 &= P'_8 + P'_6 P'_7 \\
IF_{11} &= P'_{11} + P'_6 P'_{21} + P'_7 P'_{21} + P'_8 P'_{21} \\
IF_{12} &= P'_{12} + P'_{17} + P'_{18} \\
IF_{15} &= P'_{15} + P'_{17} \\
IF_{17} &= P'_{17} + P'_{12} P'_{15} + P'_{15} P'_{18} \\
IF_{18} &= P'_{18} + P'_{12} + P'_{17} \\
IF_{21} &= P'_{21} + P'_{11}.
\end{aligned}
$$

Steps 2,1,3. No actions taken.

Step 4. P_8 dominates P_6 in IF_6. IF_6 is deleted as well as any terms in other formulas which contain P'_6. $P_{discard} = \{P_{16}, P_3, P_{20}, P_4, P_5, P_6\}$. The revised set IF is:

$$
\begin{aligned}
IF_7 &= P'_7 + P'_8 \\
IF_8 &= P'_8 \\
IF_{11} &= P'_{11} + P'_7 P'_{21} + P'_8 P'_{21} \\
IF_{12} &= P'_{12} + P'_{17} + P'_{18} \\
IF_{15} &= P'_{15} + P'_{17} \\
IF_{17} &= P'_{17} + P'_{12} P'_{15} + P'_{15} P'_{18} \\
IF_{18} &= P'_{18} + P'_{12} + P'_{17} \\
IF_{21} &= P'_{21} + P'_{11}.
\end{aligned}
$$

Step 2. Since $IF_8 = P'_8$, P_8 is added to the partial sum by Rule II. IF_8 is removed from the set IF. $PS = \{P_2, P_8\}$. Additionally, after applying this rule, $IF_7 \equiv 1$ and $IF_{11} = P'_{11} + P'_{21}$.

Step 1. Since $IF_7 \equiv 1$, we delete P_7 by Rule I. $P_{discard} = \{P_{16}, P_3, P_{20}, P_4, P_5, P_6, P_7\}$.

Steps 2,1,3. No actions taken.

Step 4. P_{21} dominates P_{11} in IF_{11}. IF_{11} is deleted as well as terms containing P_{11}. The revised set IF is:

$$
\begin{aligned}
IF_{12} &= P'_{12} + P'_{17} + P'_{18} \\
IF_{15} &= P'_{15} + P'_{17} \\
IF_{17} &= P'_{17} + P'_{12} P'_{15} + P'_{15} P'_{18} \\
IF_{18} &= P'_{18} + P'_{12} + P'_{17} \\
IF_{21} &= P'_{21}.
\end{aligned}
$$

Step 2. Since $IF_{21} = P'_{21}$, P_{21} is added to the partial sum by Rule II. IF_{21} is removed from the set IF. $PS = \{P_2, P_8, P_{21}\}$.

Steps 1,3. No actions taken.

Step 4. P_{17} dominates P_{12} in IF_{12}. IF_{12} is deleted as well as terms containing P_{12}. The revised set IF is:

$$\begin{aligned} IF_{15} &= P'_{15} + P'_{17} \\ IF_{17} &= P'_{17} + P'_{15}P'_{18} \\ IF_{18} &= P'_{18} + P'_{17}. \end{aligned}$$

Steps 2,1,3. No actions taken.

Step 4. P_{17} dominates P_{15} in IF_{15}. IF_{15} is deleted as well as terms containing P_{15}. The revised set IF is:

$$\begin{aligned} IF_{17} &= P'_{17} \\ IF_{18} &= P'_{18} + P'_{17}. \end{aligned}$$

Step 2. Since $IF_{17} = P'_{17}$, P_{17} is added to the partial sum by Rule II. IF_{17} is removed from the set IF. $PS = \{P_2, P_8, P_{21}, P_{17}\}$. Additionally, after applying this rule, $IF_{18} \equiv 1$.

Step 1. Since $IF_{18} \equiv 1$, we delete P_{18} by Rule I. Then $P_{discard}$ is the set

$$\{P_{16}, P_3, P_{20}, P_4, P_5, P_6, P_7, P_{11}, P_{12}, P_{15}, P_{18}\}.$$

Steps 2-4. No actions taken.

Step 5. Since $IF = \emptyset$, the reduction rules have determined a minimal set of terms to place in a minimal SOP formula. $PS = \{P_2, P_8, P_{21}, P_{17}\}$ is returned with $P_{discard}$.

Example 6.9 demonstrates a function for which the reduction rules are able to identify a complete set of conditionally-eliminable prime implicants to place in a minimal SOP formula. The essential prime implicants form the remaining terms to place in the formula.

Given available memory space, we would like to use Base #1 and the corresponding reduction rules because this combination yields a set of prime implicants which may be placed in the partial sum which parallels the set of prime implicants that is derived from a prime implicant table up to the point at which the table becomes cyclic. Similarly, the set of inclusion formulas which result after the application of the reduction rules corresponds to the information contained in a cyclic prime implicant table. Hence, a maximal number of prime implicants is both identified for membership in a minimal sum and discarded from consideration through the use of reduction rules. Moreover, the

resulting inclusion formulas are reduced significantly. For examples in which a reduced PI table is not cyclic, such as interval IC5, Base #1 and the corresponding reduction rules will identify a set of prime implicants to place in a minimal sum. If inclusion formulas exist after exhaustive application of these rules, then search is required to identify the remaining prime implicants.

Reduction Rules for Bases #2 and #3. In many situations, either the computational resources are limited or the functions with which we are dealing are very complex. Bases #2 or #3 may be used in these cases. However, no one has proposed reduction rules which can be applied if a subset of the conditionally-eliminable PIs is used as a base. A new set of rules is now presented which is a variant of the foregoing rules. Although these rules do not guarantee the reduction which occurs using Gaines's rules, they do significantly reduce a set of inclusion formulas while identifying some PIs to keep or discard.

There are several differences between the proposed revised rules and the aforementioned set. A revised Rule III is described which allows us to delete certain prime implicants from consideration because of dominance. Additionally, Chang & Mott's Rule is not used in the revised set. We must have inclusion formulas corresponding to all prime implicants whose labels appear in the set IF to be able to test for circularity among the prime implicants. Since we develop only a subset of these formulas when Bases #2 and #3 are used, we cannot apply Chang & Mott's Rule. Rules I is the same in both the revised set of rules and the former set, whereas Rule II differs slightly. We first discuss the revised Rule III.

In Gaines's Rule III, if the literal P_l' appears as a single term in the inclusion formula IF_j associated with the prime implicant denoted by P_j, and the cost c_l associated with P_l is less than or equal to the cost c_j associated with P_j, then P_l dominates P_j. This property holds true for any set of inclusion formulas. The use of the property, however, constitutes the difference between Gaines's Rule III and our revised Rule III. In the same vein as Chang and Mott, Gaines assumes that inclusion formulas corresponding to all prime implicants whose labels appear in the set IF

are formed. Therefore, when a dominated prime implicant P_j is found, the inclusion formula IF_j is simply deleted, as are terms in which P_j' appears in all other inclusion formulas. Since the dominating prime implicant denoted by P_l has an associated inclusion formula IF_l, we defer judgement on P_l.

Using a subset of the conditionally-eliminable prime implicants, we cannot in general take the same actions as in the former Rule III. We now examine the different possibilities that may occur and suggest methods for handling the varying situations. It is assumed in every case that the prime implicant denoted by P_l dominates the PI denoted by P_j in the inclusion formula IF_j. For every inclusion formula IF_j, we attempt to apply the following cases in order. Specifically, we would rather use Case #1 than Case #3 if possible.

Case #1: The dominating prime implicant P_l has an associated inclusion formula IF_l. We take the following actions:

1. Remove IF_j from IF.
2. Delete every term in which the variable P_j appears in the remaining inclusion formulas IF_k, $k \neq j$.

Note: This case is the same as Gaines's Rule III.

Case #2: The inclusion formula IF_j is of the form

$$P_j' + P_l', \tag{6.86}$$

and the dominating prime implicant P_l does not have an associated inclusion formula IF_l. In this situation, P_l is a secondary essential prime implicant. One of the two prime implicants P_j or P_l must be used to form a minimal formula F. Since P_l dominates P_j, choosing P_l guarantees the minimality of F. We take the following actions with respect to P_j:

1. Remove IF_j from IF.
2. Delete every term in which the variable P_j appears in the remaining inclusion formulas IF_k, $k \neq j$.

We then take the following actions with respect to P_l:

1. Add P_l to the partial sum PS.
2. Delete the literal P_l' in the remaining inclusion formulas IF_k, $k \neq l$.
3. Delete newly-absorbed terms, if any, in all formulas IF_k.

Case #3: The inclusion formula IF_j is of the form

$$P_j' + P_l' + \widetilde{IF}_j, \tag{6.87}$$

the dominating prime implicant denoted by P_l does not have an associated inclusion formula IF_l, and $\widetilde{IF}_j$ is a formula consisting of one of more terms each of which meets one of the following conditions:

- it contains more than one literal;
- it contains a single literal P_m' and the prime implicant denoted by P_m does not dominate P_j; or
- it contains a single literal P_m', the prime implicant denoted by P_m dominates P_j, and P_m does not have an associated inclusion formula IF_m.

Terms which contain more than one literal have no bearing on the domination of P_j. If the PI denoted by P_m does not dominate the one denoted by P_j, then it is of no concern to us. In the third condition, if P_m has an associated inclusion formula IF_m, then we would use Case #1 to process the domination of P_j by P_m. If IF_m does not exist, then P_l serves the same purpose as P_m in identifying the domination of P_j.

Because the dominating prime implicant denoted by P_l does not have an associated inclusion formula IF_l, and other possibilities exist for covering P_j, the only action we may take is to

rule out the possibility that P_j appears in a minimal formula F. Hence, we take the following actions with respect to P_j:

1. Delete the term P_j' in IF_j.
2. Delete every term in which the variable P_j appears in the remaining inclusion formulas IF_k, $k \neq j$.

There is one caveat to Case #3. Once we have identified that the prime implicant denoted by P_j is dominated in its associated inclusion formula IF_j, we do not check IF_j again for this condition. In other words, Case #3 is applied only once to each IF_j. A simple way to determine if we may apply this rule is to determine if P_j' is a term in IF_j. If it is a term, then way may apply the rule; otherwise, we may not.

Rule II differs only slightly from Gaines's Rule II. Gaines's Rule II is applied when a term P_j' is the only term in IF_j. We generalize the rule to be applied when any P_l' is the only term in IF_j. When $l = j$, this rule is the same as Gaines's Rule II.

Revised Rule II. If the term P_l' is the only term in the inclusion formula IF_j (l may or may not be equal to j), then the prime implicant denoted by P_l is a secondary essential prime implicant.

1. Remove IF_j from IF.
2. Remove IF_l (if it exists) from IF.
3. Add P_l to the partial sum PS.
4. Delete the literal P_l' in the remaining inclusion formulas IF_k, $k \neq j$.
5. Delete newly-absorbed terms, if any, in all formulas IF_k.

Procedures 6.12 and 6.13 implement the revised Rules II and III, respectively. Procedure 6.14 implements the revised set of rules.

Procedure 6.12 (Revised Rule II): Given a set $IF = \{IF_1, IF_2, \ldots, IF_m\}$ of inclusion formulas, Rule II is implemented as follows:

Step 0.

1. Initialise an accumulator $\widehat{PS}$ to the empty set $\emptyset$.
2. Initialise an accumulator $IF_{checked}$ to the empty set $\emptyset$.

Step 1.

- If $IF = \emptyset$, then $IF_{checked}$ is the revised set of inclusion formulas. Return $IF_{checked}$ and $\widehat{PS}$.
- Otherwise, remove the first inclusion formula from IF and denote it IF_j.

Step 2.

- If $IF_j \equiv P_l'$, i.e., IF_j consists of the single term P_l', then the prime implicant P_l is a secondary essential prime implicant.
  1. If $l \neq j$ and P_l has an associated inclusion formula IF_l in either $IF_{checked}$ or IF, then remove it.
  2. Place P_l in $\widehat{PS}$ and continue to Step 3.
- Otherwise, append IF_j to $IF_{checked}$ and return to Step 1.

Step 3. Using Procedure 6.6, remove the literal P_l' from terms which contains it in each formula in $IF_{checked}$. Replace $IF_{checked}$ with the set of formulas returned by Procedure 6.6.

Step 4. Using Procedure 6.6, remove the literal P_l' from terms which contains it in each formula in IF.

1. Replace IF with the set of formulas returned by Procedure 6.6.
2. Return to Step 1.

Procedure 6.13 (Revised Rule III): Given a set $IF = \{IF_1, IF_2, \ldots, IF_m\}$ of inclusion formulas and a cost list $C : ((P_1\ c_1)\ (P_2\ c_2)\ \cdots\ (P_m\ c_m))$, Rule III is implemented as follows:

Step 0.

1. Initialise an accumulator $IF_{checked}$ to the empty set $\emptyset$.
2. Initialise an accumulator $P_{discard}$ to the empty set $\emptyset$.
3. Initialise an accumulator PS to the empty set $\emptyset$.

Step 1.

- If $IF = \emptyset$, then no dominated variables were found. Return $IF_{checked}$, $P_{discard}$, and PS.
- Otherwise, continue to Step 2.

Step 2.

1. Remove the first inclusion formula from IF and denote it as IF_j.
2. Initialise an accumulator P_{single} by placing into it all single-literal terms P'_k in IF_j.
3. Initialise an accumulator P_{rest} by placing into it terms in IF_j which consist of two or more literals (the terms not placed in P_{single}).

Step 3.

- If P_{single} does not contain the term P'_j, then place IF_j in $IF_{checked}$ and return to Step 1.
- Otherwise, continue to Step 4.

Step 4.

- If a member P_l of P_{single}, $l \neq j$, has an associated inclusion formula IF_l and $c_l \leq c_j$, then place P_j in $P_{discard}$ and continue to Step 5.
- If P_{single} consists of only two elements, P_j and P_l, $c_l \leq c_j$, and $P_{rest} = \emptyset$, then place P_j in $P_{discard}$, place P_l in PS, and continue to Step 8.
- If one of the conditions
  - $P_{rest} \neq \emptyset$, P_{single} consists of only two elements, P_j and P_l, and $c_l \leq c_j$, or
  - P_{single} consists of more than two elements and a P_l exists such that $l \neq j$ and $c_l \leq c_j$

  holds, then take the following actions:
  1. Remove P'_j from IF_j.
  2. Place IF_j in $IF_{checked}$.
  3. Place P_j in $P_{discard}$ and continue to Step 5.
- Otherwise, no prime implicant dominates P_j. Place IF_j in $IF_{checked}$ and return to Step 1.

Step 5. Using Procedure 6.5, delete terms in each formula in $IF_{checked}$ which contain the variable P_j. Replace $IF_{checked}$ with the set of formulas returned by Procedure 6.5.

Step 6. Using Procedure 6.5, delete terms in each formula in IF which contain the variable P_j. Replace IF with the set of formulas returned by Procedure 6.5.

Step 7.

1. Append $IF_{checked}$ to IF. IF is the revised set of inclusion formulas.
2. Return IF, $P_{discard}$, and PS.

Step 8. Using Procedure 6.6, remove the literal P'_l from terms which contains it in each formula in $IF_{checked}$. Replace $IF_{checked}$ with the set of formulas returned by Procedure 6.6.

Step 9. Using Procedure 6.6, remove the literal P'_l from terms which contains it in each formula in IF.

1. Replace IF with the set of formulas returned by Procedure 6.6.
2. Return to Step 5.

Procedure 6.14 (Reduction Rules - Set #2): Given a set $IF = \{IF_1, IF_2, \ldots, IF_m\}$ of inclusion formulas and a cost list $C : ((P_1\ c_1)\ (P_2\ c_2)\ \cdots\ (P_m\ c_m))$, and a set $P_{discard}$ which contains the set of prime implicants which have been discarded from consideration, the reduction rules are implemented as follows:

Step 0. Initialise an accumulator PS to the emptyset $\emptyset$.

Step 1. Given IF, use Procedure 6.7 to apply Rule I. Procedure 6.7 returns a set $\widehat{P}_{discard}$ of variables and a set $IF_{revised}$ of inclusion formulas.

- If $\widehat{P}_{discard}$ is empty, then no variables were found to be discarded. Skip to Step 3.
- Otherwise, append $\widehat{P}_{discard}$ to $P_{discard}$ and replace IF with $IF_{revised}$. Continue to Step 2.

Step 2. Given IF, use Procedure 6.12 to apply Rule II. Procedure 6.12 returns a set $\widehat{PS}$ of variables and a set $IF_{revised}$ of inclusion formulas.

- If $\widehat{PS}$ is empty, then no variables were found to be placed in the partial sum. Continue to Step 3.
- Otherwise, append $\widehat{PS}$ to PS and replace IF with $IF_{revised}$. Return to Step 1.

Step 3. Given IF and the cost list C, use Procedure 6.13 to apply Rule III. Procedure 6.13 returns a variable $\widehat{P}_{discard}$ and a set $IF_{revised}$ of inclusion formulas.

- If $\widehat{P}_{discard}$ is empty, then no variables were found to be discarded. Continue to Step 4.
- Otherwise, place $\widehat{P}_{discard}$ in $P_{discard}$ and replace IF with $IF_{revised}$. Return to Step 2.

Step 4. No further reduction can take place. Return the inclusion formulas IF, the partial sum PS, and the set $P_{discard}$ of variables to discard.

Example 6.10: Using Base #2 and Procedure 6.2, the set IF of inclusion formulas is developed for the function B5:

$$\begin{aligned}
IF_4 &= P_4' + P_1' \\
IF_6 &= P_6' + P_3'P_5' \\
IF_9 &= P_9' + P_2'P_8' + P_8'P_{10}' \\
IF_{15} &= P_{15}' + P_{12}'P_{14}' + P_{14}'P_{18}' \\
IF_{16} &= P_{16}' + P_{11}' + P_{13}' + P_{14}' \\
IF_{17} &= P_{17}' + P_{11}' \\
IF_{19} &= P_{19}' + P_7'.
\end{aligned}$$

We use Procedure 6.12 to reduce IF. We assume that the cost criterion is fewest gates; hence, the cost of each PI is equal to 1. $P_{discard}$ is initialized to the empty set $\emptyset$.

Step 0. Initialise $PS = \emptyset$.

Steps 1-2. No actions are taken.

Step 3. The first inclusion formula in which a dominated PI is found is the formula IF_4. Case #2 of Rule III applies in this situation. PI P_1 is a secondary essential prime implicant. Since P_4' and P_1' appear only in IF_4, we delete IF_4. Then $P_{discard} = \{P_4\}$ and $PS = \{P_1\}$.

Steps 2,1. No actions are taken.

Step 3. The next inclusion formula in which a dominated PI is found is the formula IF_{16}. Case #3 of Rule III applies in this situation. Since P_{16}' appears only in IF_{16}, we delete the term P_{16}' in IF_{16}. Then $P_{discard} = \{P_4, P_{16}\}$ and $PS = \{P_1\}$. At this point, the revised set IF is:

$$\begin{aligned} IF_6 &= P_6' + P_3'P_5' \\ IF_9 &= P_9' + P_2'P_8' + P_8'P_{10}' \\ IF_{15} &= P_{15}' + P_{12}'P_{14}' + P_{14}'P_{18}' \\ IF_{16} &= P_{11}' + P_{13}' + P_{14}' \\ IF_{17} &= P_{17}' + P_{11}' \\ IF_{19} &= P_{19}' + P_7'. \end{aligned}$$

Steps 2,1. No actions are taken.

Step 3. The next inclusion formula in which a dominated PI is found is the formula IF_{17}. Case #2 of Rule III applies in this situation. PI P_{11} is a secondary essential prime implicant. Since P_{17}' appears only in IF_{17}, we delete IF_{17}. Additionally, the literal P_{11}' is deleted in terms in which it appears in all other inclusion formulas and absorption is performed. The revised set IF then is:

$$\begin{aligned} IF_6 &= P_6' + P_3'P_5' \\ IF_9 &= P_9' + P_2'P_8' + P_8'P_{10}' \\ IF_{15} &= P_{15}' + P_{12}'P_{14}' + P_{14}'P_{18}' \\ IF_{16} &= 1 \\ IF_{19} &= P_{19}' + P_7'. \end{aligned}$$

Then $P_{discard} = \{P_4, P_{16}, P_{17}\}$ and $PS = \{P_1, P_{11}\}$.

Step 2. No actions are taken.

Step 1. Since IF_{16} is the term 1, we delete P_{16} by Rule I. P_{16} is already in $P_{discard}$

Step 2. No actions are taken.

Step 3. The next inclusion formula in which a dominated PI is found is the formula IF_{19}. Case #2 of Rule III applies in this situation. PI P_7 is a secondary essential prime implicant. Since P_{19}' and P_7' appear only in IF_{19}, we delete IF_{19}. Then $P_{discard} = \{P_4, P_{16}, P_{17}, P_{19}\}$ and $PS = \{P_1, P_{11}, P_7\}$. The revised set IF is:

$$\begin{aligned} IF_6 &= P_6' + P_3'P_5' \\ IF_9 &= P_9' + P_2'P_8' + P_8'P_{10}' \\ IF_{15} &= P_{15}' + P_{12}'P_{14}' + P_{14}'P_{18}'. \end{aligned}$$

Step 2. No actions are taken.

Steps 1,3. No actions are taken.

Step 4. At this point, no further reduction can occur. We thus return $IF = \{IF_6, IF_9, IF_{15}\}$, $PS = \{P_1, P_{11}, P_7\}$ and $P_{discard} = \{P_4, P_{16}, P_{17}, P_{19}\}$.

After applying the rules, three inclusion formulas remain from the original seven. Moreover, three prime implicants were identified for placement in F and four were discarded from consideration. Prime implicants P_6, P_9, and P_{15} may be chosen by inspection from the remaining inclusion formulas to complete the formation of F. These prime implicants would be chosen by a follow-up search process.

Using Base #1 and Gaines's rules, six prime implicants are identified for placement in F and $IF = \emptyset$ at the completion of rule reduction. In contrast, the revised reduction rules do not identify all of the prime implicants to place in a minimal formula and do not completely reduce the inclusion formulas. The greatest possible reduction requires more information than what is available when Bases #2 and #3 are used. This is the cost associated with the benefit of generating a smaller base and the subsequent smaller subset of the inclusion formulas.

In cases in which the reduction rules do not totally reduce the set of inclusion formulas, the next step in the process of forming a minimal formula F is to execute a search process to determine the remain prime implicants to place in F. We discuss the search process in Chapter 9. We now introduce three algorithms which integrate the steps up to the point at which search is required in the process of developing a minimal F.

Minimisation Algorithms

We have discussed the first six steps of the methodology for forming a minimal sum-of-products formula F with respect to the given cost criterion to represent a function f belonging to the interval $[g, h]$. These steps are:

1. derive a 1-normal form specification $\phi(X, z) = 1$ if not already formed;
2. construct a general solution of $\phi(X, z) = 1$ for z, in the form of an interval $g(X) \leq z \leq h(X)$, i.e., $z \in [g(X), h(X)]$;
3. develop the set of all prime implicants of h;
4. develop a base for $[g, h]$;
5. develop inclusion formulas representing coverage of the terms of the base by prime implicants of h;
6. reduce the inclusion formulas using reduction rules—identifying prime implicants of h to include in F as well as to discard from consideration.

If a minimal formula is not formed after these steps, then a search process must be used to determine the remaining prime implicants to include in F. We now present three different algorithms for performing these steps. After introducing the algorithms, we will compare and contrast them in an ensuing section.

Algorithm Using Base #1. The first algorithm for forming a minimal SOP F uses the set of all useful, conditionally-eliminable prime implicants of a function as a base. A synopsis of this algorithm is:

1. derive a 1-normal form specification $\phi(X, z) = 1$ if not already formed;
2. construct a general solution of $\phi(X, z) = 1$ for z, in the form of an interval $g(X) \leq z \leq h(X)$, i.e., $z \in [g(X), h(X)]$;
3. develop the set of all prime implicants of h;
4. form a base for $[g, h]$ consisting of the set of useful CEPIs;
5. develop an inclusion formula for every term of the base using Procedure 6.2;
6. use Reduction Rule Set #1 to reduce the set of inclusion formulas; and
7. use a search process to determine the remaining terms.

Algorithm 6.1 implements the first six steps of the aforementioned process. The search process is introduced in Chapter 9.

Algorithm 6.1 (Minimisation Algorithm #1): Given a 1-normal form specification $\phi(X, z) = 1$ and a cost criterion *CRITERION*, a minimal formula F which represents a function f belonging to the interval $[g(X), h(X)]$ developed from $\phi(X, z) = 1$ is generated in the following manner:

Step 0. Initialise a partial sum PS to the empty set $\emptyset$.

Step 1.

1. Form $g(X) = \phi'(X, 0) \cdot \phi(X, 1)$.
2. Form $h(X) = \phi'(X, 0) + \phi(X, 1)$.

Step 2.

1. Form a simplified formula to represent $g(X)$ using Procedure 2.15 (Simplification). Call the simplified formula G.
2. Develop the Blake canonical form for function $h(X)$ using Procedure 2.20 (Blake canonical form).

Step 3. Using Procedure 5.2 (Essential Prime Implicants), G, and $BCF(h(X))$, determine the essential prime implicants of $h(X)$.

1. Denote the set of essential prime implicants by $H_{ess}(X)$ and the function formed by the disjunction of the essential prime implicants by $h_{ess}(X)$.
2. Denote the set of terms in G used to identify essential prime implicants in Procedure 5.2—terms covered by the essential prime implicants—by $G_{covered}$.

Step 4.

1. Form a set $\widehat{H}$ of prime implicants consisting of all prime implicants of $h(X)$ less the essential prime implicants.
2. Use Procedure 5.3 (Covered Terms), $\widehat{H}$, and $h_{ess}(X)$ to determine the terms in $\widehat{H}$ covered by $h_{ess}(X)$. These terms constitute the set of inessential prime implicants of $h(X)$; call this set of terms $H_{inessen}$.

Step 5. Form the set H_{ce} of conditionally-eliminable prime implicants by removing the prime implicants in $H_{inessen}$ from $\widehat{H}$

Step 6.

1. Remove from G the terms in $G_{covered}$; denote the resulting formula by $G - G_{covered}$.
2. Use Procedure 5.3 (Covered Terms), $G - G_{covered}$, and $h_{ess}(X)$ to determine the terms in $G - G_{covered}$ covered by $h_{ess}(X)$. Call the resulting terms G_{ess}.
3. Remove from $G - G_{covered}$ the terms in G_{ess}. Denote the resulting formula $\widetilde{G}$; it represents the function $\tilde{g}(X)$.

Step 7.

1. Using Procedure 5.1 (Useful Prime Implicants), determine the prime implicants in H_{ce} that are useful with respect to $\tilde{G}$. Call the set of useful prime implicants H_{base}. H_{base} is the base.
2. Place in $H_{useless}$ the terms in H_{ce} which are not in H_{base}.

Step 8. Form a set $LABS = \{P'_1, \ldots, P'_k\}$ of labels which will be used to denote the prime implicants in H_{base}.

Step 9. Using Procedure 6.2 (Formation of an Inclusion Formula), H_{base}, $LABS$, H_{ess}, and $\tilde{g}$, generate an inclusion formula IF_j for each prime implicant in H_{base}. Denote the set of inclusion formulas by IF.

Step 10. Using Procedure 6.4 (Assignment of Cost to Terms), H_{base}, $LABS$, and $CRITERION$, develop an association list affiliating a term with a cost. Each cost is paired with the label in $LABS$ with denotes a corresponding PI in H_{base}. Call the resulting list $LAB/COSTS$.

Step 11. Using Procedure 6.11 (Reduction Rules - Set #1), the set IF of inclusion formulas, and the cost list $LAB/COSTS$, apply rule reduction to the set IF. Procedure 6.11 returns a revised set IF_{rev} of inclusion formulas and a set PS_{new} of variables identified for placement in F.

1. Replace IF with IF_{rev}.
2. Append PS_{new} to PS.

Step 12.

- If $IF = \emptyset$, then a minimal formula F has been formed.
  1. Replace the labels in PS with their associated prime implicants from H_{base}.
  2. Append the contents of $H_{ess}(X)$ to PS. The resulting set of terms constitutes F.
  3. Return F.
- Otherwise, a search process must be used to complete F.
  1. Return the current inclusion formulas IF, PS, H_{base}, and $H_{ess}(X)$. (PS is a set of labels; H_{base} and $H_{ess}(X)$ are sets of prime implicants.)
  2. Also return $LAB/COSTS$ and $LABS$ for use in the search process.

Algorithm 6.1 is different from other algorithms found in the literature in the following ways:

- the minimisation process begins with a 1-normal form specification $\phi(X, z) = 1$;
- the partitioning of the prime implicants allows concentration of effort on the useful CEPIs;
- the base consists of useful, conditionally-eliminable PIs; and
- formation of inclusion formulas is simplified through the use of the EPI- and DC-constraints.

Algorithm Using Base #2. We now present an algorithm for forming a minimal SOP F which uses the set of all useful, conditionally-eliminable prime implicants which are contained in an irredundant disjunctive form, i.e. Base #2. A synopsis of this algorithm is:

1. derive a 1-normal form specification $\phi(X, z) = 1$ if not already formed;
2. construct a general solution of $\phi(X, z) = 1$ for z, in the form of an interval $g(X) \leq z \leq h(X)$, i.e., $z \in [g(X), h(X)]$;
3. develop the set of all prime implicants of h;
4. form an IDF for $[g, h]$;
5. form a base for $[g, h]$ consisting of the set of useful CEPIs in the IDF;
6. develop an inclusion formula for every term of the base using Procedure 6.2;
7. use Reduction Rule Set #2 to reduce the set of inclusion formulas; and
8. use a search process to determine the remaining terms.

Algorithm 6.2 implements the first seven steps of the foregoing process. The search process is introduced in Chapter 9.

Algorithm 6.2 (Minimization Algorithm #2): Given a 1-normal form specification $\phi(X, z) = 1$ and a cost criterion $CRITERION$, a minimal formula F which represents a function f belonging to the interval $[g(X), h(X)]$ developed from $\phi(X, z) = 1$ is generated in the following manner:

Step 0. Initialise a partial sum PS to the empty set $\emptyset$.

Step 1.

1. Form $g(X) = \phi'(X, 0) \cdot \phi(X, 1)$.
2. Form $h(X) = \phi'(X, 0) + \phi(X, 1)$.

Step 2.

1. Form a simplified formula to represent $g(X)$ using Procedure 2.15 (Simplification). Call the simplified formula G.
2. Develop the Blake canonical form for function $h(X)$ using Procedure 2.20 (Blake canonical form).

Step 3. Using Procedure 5.2 (Essential Prime Implicants), G, and $BCF(h(X))$, determine the essential prime implicants of $h(X)$.

1. Denote the set of essential prime implicants by $H_{ess}(X)$ and the function formed by the disjunction of the essential prime implicants by $h_{ess}(X)$.
2. Denote the set of terms in G used to identify essential prime implicants in Procedure 5.2—terms covered by the essential prime implicants—by $G_{covered}$.

Step 4.

1. Form a set $\widehat{H}$ of prime implicants consisting of all prime implicants of $h(X)$ less the essential prime implicants.
2. Use Procedure 5.3 (Covered Terms), $\widehat{H}$, and $h_{ess}(X)$ to determine the terms in $\widehat{H}$ covered by $h_{ess}(X)$. These terms constitute the set of inessential prime implicants of $h(X)$; call this set of terms $H_{inessen}$.

Step 5. Form the set H_{ce} of conditionally-eliminable prime implicants by removing the prime implicants in $H_{inessen}$ from $\widehat{H}$.

Step 6.

1. Remove from G the terms in $G_{covered}$; denote the resulting formula by $G - G_{covered}$.
2. Use Procedure 5.3 (Covered Terms), $G - G_{covered}$, and $h_{ess}(X)$ to determine the terms in $G - G_{covered}$ covered by $h_{ess}(X)$. Call the resulting terms G_{ess}.
3. Remove from $G - G_{covered}$ the terms in G_{ess}. Denote the resulting formula $\tilde{G}$; it represents the function $\tilde{g}(X)$.

Step 7.

1. Using Procedure 5.1 (Useful Prime Implicants), determine the prime implicants in H_{ce} that are useful with respect to $\tilde{G}$. Call the set of useful prime implicants H_{useful}.
2. Place in $H_{useless}$ the terms in H_{ce} which are not in H_{useful}.

Step 8.

1. Using Procedure 2.34 (Irredundant Formula), H_{useful}, $H_{ess}(X)$, and $\tilde{g}$ form an IDF for the function.
2. Remove the set $H_{ess}(X)$ of essential PIs from the IDF and call the remaining PIs H_{base}. H_{base} is the base.

Step 9. Form a set $LABS = \{P'_1, \ldots, P'_k\}$ of labels which will be used to denote the prime implicants in H_{useful}.

Step 10. Using Procedure 6.2 (Formation of an Inclusion Formula), H_{useful}, $LABS$, H_{ess}, and j, generate an inclusion formula IF_j for each prime implicant in H_{base}. Denote the set of inclusion formulas by IF.

Step 11. Using Procedure 6.4 (Assignment of Cost to Terms), H_{useful}, $LABS$, and $CRITERION$, develop an association list affiliating a term with a cost. Each cost is paired with the label in $LABS$ with denotes a corresponding PI in H_{useful}. Call the resulting list $LAB/COSTS$.

Step 12. Using Procedure 6.14 (Reduction Rules - Set #2), the set IF of inclusion formulas, and the cost list $LAB/COSTS$, apply rule reduction to the set IF. Procedure 6.14 returns a revised set IF_{rev} of inclusion formulas and a set PS_{new} of variables identified for placement in F.

1. Replace IF with IF_{rev}.
2. Append PS_{new} to PS.

Step 13.

- If $IF = \emptyset$, then a minimal formula F has been formed.
  1. Replace the labels in PS with their associated prime implicants from H_{useful}.
  2. Append the contents of $H_{sss}(X)$ to PS. The resulting set of terms constitutes F.
  3. Return F.
- Otherwise, a search process must be used to complete F.
  1. Return the current inclusion formulas IF, PS, H_{useful}, and $H_{sss}(X)$. (PS is a set of labels; H_{useful}, and $H_{sss}(X)$ are sets of prime implicants.)
  2. Also return $LAB/COSTS$ and $LABS$ for use in the search process.

Algorithm 6.2 is unique in the following ways:

- the minimisation process begins with a 1-normal form specification $\phi(X, z) = 1$;
- the partitioning of the prime implicants allows concentration of effort on the useful CEPIs;
- the base consists of useful, conditionally-eliminable prime implicants of an irredundant disjunctive form;
- formation of inclusion formulas is simplified through the use of the EPI- and DC-constraints; and
- rule reduction is facilitated through the use of a revised set of reduction rules.

Algorithm Using Base #3. An algorithm is now presented for forming a minimal SOP F which uses a set of useful, conditionally-eliminable prime implicants which covers the function $\hat{g}$ as a base. We summarise the algorithm as follows:

1. derive a 1-normal form specification $\phi(X, z) = 1$ if not already formed;
2. construct a general solution of $\phi(X, z) = 1$ for z, in the form of an interval $g(X) \leq z \leq h(X)$, i.e., $z \in [g(X), h(X)]$;
3. develop the set of all prime implicants of h;
4. using Procedure 6.1, form a base for $[g, h]$ consisting of the set of useful CEPIs which completely cover the function $\hat{g}$;
5. develop an inclusion formula for every term of the base using Procedure 6.3;
6. use Reduction Rule Set #2 to reduce the set of inclusion formulas; and
7. use a search process to determine the remaining terms.

The first six steps of the foregoing process are implemented by Algorithm 6.3. The search process is introduced in Chapter 9.

Algorithm 6.3 (Minimisation Algorithm #3): Given a 1-normal form specification $\phi(X, z) = 1$ and a cost criterion *CRITERION*, a minimal formula F which represents a function f belonging to the interval $[g(X), h(X)]$ developed from $\phi(X, z) = 1$ is generated in the following manner:

Step 0. Initialise a partial sum PS to the empty set $\emptyset$.

Step 1.

1. Form $g(X) = \phi'(X, 0) \cdot \phi(X, 1)$.
2. Form $h(X) = \phi'(X, 0) + \phi(X, 1)$.

Step 2.

1. Form a simplified formula to represent $g(X)$ using Procedure 2.15 (Simplification). Call the simplified formula G.
2. Develop the Blake canonical form for function $h(X)$ using Procedure 2.20 (Blake canonical form).

Step 3. Using Procedure 5.2 (Essential Prime Implicants), G, and $BCF(h(X))$, determine the essential prime implicants of $h(X)$.

1. Denote the set of essential prime implicants by $H_{ess}(X)$ and the function formed by the disjunction of the essential prime implicants by $h_{ess}(X)$.
2. Denote the set of terms in G used to identify essential prime implicants in Procedure 5.2—terms covered by the essential prime implicants—by $G_{covered}$.

Step 4.

1. Form a set $\widehat{H}$ of prime implicants consisting of all prime implicants of $h(X)$ less the essential prime implicants.
2. Use Procedure 5.3 (Covered Terms), $\widehat{H}$, and $h_{ess}(X)$ to determine the terms in $\widehat{H}$ covered by $h_{ess}(X)$. These terms constitute the set of inessential prime implicants of $h(X)$; call this set of terms $H_{inessen}$.

Step 5. Form the set H_{ce} of conditionally-eliminable prime implicants by removing the prime implicants in $H_{inessen}$ from $\widehat{H}$.

Step 6.

1. Remove from G the terms in $G_{covered}$; call the resulting formula $G - G_{covered}$.
2. Using Procedure 2.10 (Subtraction), subtract the function $h_{ess}(X)$ from the function represented by $G - G_{covered}$.
3. Call the resulting formula $\widehat{G}$ and the function which it represents $\hat{g}(X)$.

Step 7.

1. Using Procedure 5.1 (Useful Prime Implicants), determine the prime implicants in H_{ce} are useful with respect to $\widehat{G}$. Call the set of useful prime implicants H_{useful}.

2. Place in $H_{useless}$ the terms in H_{ce} which are not in H_{useful}.

Step 8. Using Procedure 6.1 (CEPIs Completely Covering $\hat{g}$), H_{useful}, and $\hat{g}$, form a base H_{base} for the function.

Step 9. Form a set $LABS = \{P'_1, \ldots, P'_k\}$ of labels which will be used to denote the prime implicants in H_{useful}.

Step 10. Using Procedure 6.3 (Formation of an Inclusion Formula), H_{useful}, $LABS$, and $\hat{g}$, generate an inclusion formula IF_j for each prime implicant in H_{base}. Denote the set of inclusion formulas by IF.

Step 11. Using Procedure 6.4 (Assignment of Cost to Terms), H_{useful}, $LABS$, and $CRITERION$, develop an association list affiliating a term with a cost. Each cost is paired with the label in $LABS$ with denotes a corresponding PI in H_{useful}. Call the resulting list $LAB/COSTS$.

Step 12. Using Procedure 6.14 (Reduction Rules - Set #2), the set IF of inclusion formulas, and the cost list $LAB/COSTS$, apply rule reduction to the set IF. Procedure 6.14 returns a revised set IF_{rev} of inclusion formulas and a set PS_{new} of variables identified for placement in F.

1. Replace IF with IF_{rev}.
2. Append PS_{new} to PS.

Step 13.

- If $IF = \emptyset$, then a minimal formula F has been formed.
  1. Replace the labels in PS with their associated prime implicants from H_{useful}.
  2. Append the contents of $H_{ess}(X)$ to PS. The resulting set of terms constitutes F.
  3. Return F.
- Otherwise, a search process must be used to complete F.
  1. Return the current inclusion formulas IF, PS, H_{useful}, and $H_{ess}(X)$. (PS is a set of labels; H_{useful}, and $H_{ess}(X)$ are sets of prime implicants.)
  2. Also return $LAB/COSTS$ and $LABS$ for use in the search process.

The following aspects of Algorithm 6.3 are novel:

- the minimization process begins with a 1-normal form specification $\phi(X, z) = 1$;
- the partitioning of the prime implicants allows concentration of effort on the useful CEPIs;
- the base consists of a subset of the useful CEPIs derived using the lower-bound function $\hat{g}$;
- formation of inclusion formulas is simplified through the use of the $\widehat{G}$-constraint; and
- rule reduction is facilitated through the use of a revised set of reduction rules.

Comparison of the Algorithms. The utility of each of the foregoing algorithms is dependent on the available computational resources as well as on the complexity of the functions for which we are attempting to form a minimal formula F. When sufficient memory is available, Algorithm 6.1 is the best technique to use since the resulting inclusion formulas give us the most information about prime implicants to place in F as well as those to discard from consideration. Hence, the least amount of work must be performed by the ensuing search process when Base #1 is used. However, if we are constrained with respect to resources, then either Algorithm 6.2 or 6.3 should be used.

Algorithm 6.1 has the largest base and consequently produces the largest set of inclusion formulas. With the largest set of formulas, we are able to identify the largest set of prime implicants for F. Moreover, the inclusion formulas are reduced to the greatest extent. Thus, if search is required, it will not involve as much effort. However, this method uses a considerable amount of memory because of the number of prime implicants and corresponding inclusion formulas. With many inclusion formulas, the process of applying reduction rules is memory intensive. This can be a problem when each prime implicant of a function is about as good as the next, e.g., as in symmetric functions.

When less memory is available, we must resort to using a smaller base. Fewer inclusion formulas are generated when a smaller base is used. Hence, the demands on memory resources are not as severe as when we use Base #1. However, the cost of developing fewer inclusion formulas is that we cannot in general reduce the set of inclusion formulas as much as when a larger base is used. As a result, effort is shifted to the search process. Additionally, Base #2 is formed from an IDF. The generation of an IDF requires a considerable computational effort.

In Appendix C, we discuss the computational results of applying Algorithms 6.1 and 6.2 to several sets of examples.

Summary

In this chapter, we have presented methods for taking an initial specification and developing a single minimal SOP formula F up to the point at which the use of search is necessary. The search process is discussed in Chapter 9. We have introduced a number of new ideas in this chapter:

- A set of algorithms for three different prime implicant bases was presented. We thus have available an algorithm appropriate for available computing resources as well as the complexity and size of function which corresponds to the circuit specification.
- We begin each algorithm with a 1-normal form specification $\phi(X, z) = 1$ for which we construct a general solution for z in the form of an interval $z \in [g(X), h(X)]$. This is done to emphasise that developing a design corresponds to solving an equation. (This technique is of additional significance in the minimisation of multiple-output circuits.)
- A methodology for partitioning of the prime implicants was presented which allows concentration of effort on the useful, conditionally-eliminable prime implicants.
- An equation-based approach to the generation of inclusion formulas which incorporates the use of constraints was formulated. This approach provides a theoretically-sound foundation for the reasoning process utilised to generate inclusion formulas, something that has been lacking in previous work. The use of constraints makes the process of generating inclusion formulas more efficient.
- A modified set of reduction rules was developed for situations in which only a subset of the set of conditionally-eliminable prime implicants is included in the base. These rules facilitate a moderate reduction in the inclusion formulas prior to the use of search.

VII. Minimization of Multiple-Output Functions

In Chapter 6 we presented techniques for developing a minimal sum-of-products formula F to represent a function f belonging to the interval $[g, h]$. The formula F corresponds to a minimal single-output design which meets a 1-normal form specification $\phi(X, z) = 1$. In the general case, however, we endeavor to form a multiple-output circuit specified by a 1-normal form

$$\phi(X, Z) = 1, \tag{7.1}$$

for which $X = (x_1, x_2, \ldots, x_n)$ and $Z = (z_1, z_2, \ldots, z_m)$. Assuming that the 1-normal form specification (7.1) is tabular, we develop a system of the form

$$\begin{array}{rcccl} g_1(X) & \leq & z_1 & \leq & h_1(X) \\ g_2(X) & \leq & z_2 & \leq & h_2(X) \\ & & \vdots & & \\ g_m(X) & \leq & z_m & \leq & h_m(X), \end{array} \tag{7.2}$$

which is equivalent to (7.1). By applying a similar methodology as in Chapter 6, we then derive a design by forming a system

$$\begin{array}{rcl} z_1 & = & f_1(X) \\ z_2 & = & f_2(X) \\ & \vdots & \\ z_m & = & f_m(X), \end{array} \tag{7.3}$$

such that a design corresponds to the vector $\underline{F}$ of formulas which represents the functions $\underline{f}(X)$. Each function $f_j(X)$ is a member of the interval $[g_j(X), h_j(X)]$.

In this chapter, we extend the methods discussed in Chapter 6 to facilitate development of designs for multiple-output combinational circuits. We assume that all specifications $\phi(X, Z) = 1$

considered in this chapter are tabular. Additionally, all designs produced using techniques presented in this chapter correspond to non-recurrent designs, i.e., designs in which each output is a function only of the input nodes. Non-recurrent designs are those which are produced using conventional minimisation systems. For algorithmic efficiency, we restrict the cost criteria used to judge multiple-output circuits to the fewest-gates criterion.

A vector $\underline{F}$ of formulas is said to be *minimal* if the formulas contained in $\underline{F}$ are collectively minimal with respect to a given cost criterion. For example, if the cost criterion is the fewest gates, then $\underline{F}$ is said to be minimal only if the number of distinct terms in the formulas contained in $\underline{F}$ is minimal. Moreover, $\underline{F}$ is said to be *irredundant* if each formula F_j in $\underline{F}$ is irredundant.

We follow a similar approach in developing a minimal, irredundant vector $\underline{F}$ of SOP formulas to represent the functions $\underline{f}(X)$ as applied in the formation of a minimal SOP formula F. The steps in this process are:

1. derive a 1-normal form specification $\phi(X, Z) = 1$, if not already formed;
2. form a general solution of $\phi(X, Z) = 1$ for Z to develop a set (7.2) of intervals;
3. form the set of all *multiple-output prime implicants* of the upper-bound functions $\underline{h}(X)$;
4. develop a base for $[\underline{g}(X), \underline{h}(X)]$;
5. develop inclusion formulas representing coverage of the terms of the base by the multiple-output prime implicants;
6. reduce the inclusion formulas using reduction rules—identifying multiple-output prime implicants to include in formulas in $\underline{F}$ as well as to discard from consideration; and
7. use a search process to determine the remaining multiple-output prime implicants to include in formulas in $\underline{F}$.

A number of the steps in this process vary somewhat from the steps used in the single-output case. With the exception of the last step—search—we will discuss each step in turn in the course of this chapter. Differences between the foregoing steps and steps in the single-output case will be highlighted in the course of the presentation. As in Chapter 6, we present algorithms which can be used to form a minimal design.

Initial Specification

A specification for a digital circuit may be stated by a single Boolean equation in 1-normal form, i.e.,

$$\phi(X, Z) = 1. \tag{7.4}$$

If we are given a specification in some other medium, e.g., a truth table, we may use the methods described in Chapter 4 to form an equation of the form (7.4). Given the equation $\phi(X, Z) = 1$, we develop a solution of the form (7.2) in which each output z_j is defined by an interval $[g_j(X), h_j(X)]$.

Let us define the function $\tilde{\phi}_j(X, z_j) = EDIS(\phi(X, Z), Z_j)$, in which Z_j is the set Z of variables less the variable z_j. Viewing $\tilde{\phi}_j(X, z_j)$ as a function of the single variable z_j, we form a general solution based on the extended-range concept developed in Theorem 4.2. Using the extended range, we produce the general solution

$$\begin{array}{lll} g_1(X) & \leq z_1 \leq & h_1(X) \\ g_2(X) & \leq z_2 \leq & h_2(X) \\ & \vdots & \\ g_m(X) & \leq z_m \leq & h_m(X), \end{array} \tag{7.5}$$

of $\phi(X, Z) = 1$ for Z, in which

$$g_j(X) = \tilde{\phi}'_j(X, 0) \cdot \tilde{\phi}_j(X, 1) \tag{7.6}$$

and

$$h_j(X) = \tilde{\phi}'_j(X, 0) + \tilde{\phi}_j(X, 1). \tag{7.7}$$

We must use an extended range to form system (7.5) in case there exists a don't care condition such as an input combination in $\mathbf{B}_2^n$ which does not occur (see Example 4.9).

Hence, system (7.5) may be used to develop a design corresponding to the vector $\underline{F}$ of formulas representing the functions $\underline{f}(X)$ in the system

$$\begin{aligned} z_1 &= f_1(X) \\ z_2 &= f_2(X) \\ &\vdots \\ z_m &= f_m(X). \end{aligned} \tag{7.8}$$

In the remainder of the chapter we present an approach for developing a vector $\underline{F}$ which is minimal with respect to a given cost criterion. We discuss the formation of multiple-output prime implicants for the upper-bound functions $\underline{h}(X)$ in (7.5) in the ensuing section.

Multiple-Output Prime Implicants (MOPIs)

A minimal vector $\underline{F}$ of SOP formulas corresponds to a minimal two-level design. In the single-output case, a minimal formula F consists of an irredundant set of prime implicants of $h(X)$ which covers $g(X)$. However, in multiple-output circuits the prime implicants of each function $h_j(X)$ do not generally suffice to compose a minimal formula F_j to cover the corresponding function $g_j(X)$. A special form of prime implicant, called *multiple-output prime implicants* (MOPIs), must be used to construct a minimal $\underline{F}$. We demonstrate the reason in Example 7.1.

Example 7.1: Suppose we are given a 1-normal form specification $\phi(X, Z) = 1$, in which

$$\phi(X, Z) = x_1'x_3'z_1z_2' + x_1'x_2'z_1z_2' + x_1'x_2x_3z_1z_2 + x_1x_2'z_1z_2 + x_1x_2x_3'z_1'z_2' + x_1x_2x_3z_1'z_2. \tag{7.9}$$

We develop a system that is equivalent to $\phi(X, Z) = 1$, such as (7.5), forming a range $[g_j, h_j]$ of possible functions for each output z_j:

$$\begin{aligned} x_1' + x_2' &\leq z_1 \leq x_1' + x_2' && (7.10) \\ x_1x_2' + x_2x_3 &\leq z_2 \leq x_1x_2' + x_2x_3 + x_1x_3. && (7.11) \end{aligned}$$

The right-hand side of (7.10) is $BCF(h_1)$; the right-hand side of (7.11) is $BCF(h_2)$. Using the prime implicants of h_1 to form $f_1(X)$ and the prime implicants of h_2 to form $f_2(X)$, the best system $Z = \underline{f}(X)$ we can form with respect to the number of distinct terms in $\underline{F}$ is

$$\begin{aligned} z_1 &= x_1' + x_2' \\ z_2 &= x_1x_2' + x_2x_3. \end{aligned} \qquad (7.12)$$

The number of distinct terms in $\underline{F}$ is four in this case. However, we may replace the term x_2' in F_1 with the term x_1x_2'--which is not a prime implicant of h_1—and still represent the function $f_1(X)$. We thus form the system

$$\begin{aligned} z_1 &= x_1' + x_1x_2' \\ z_2 &= x_1x_2' + x_2x_3 \end{aligned} \qquad (7.13)$$

in which the number of distinct terms in $\underline{F}$ is three. The vector $\underline{F}$ in system (7.13) has fewer distinct terms than the $\underline{F}$ in (7.12) developed using the prime implicants of the upper-bound functions $\underline{h}$. The term x_1x_2', which is shared between F_1 and F_2 in (7.13), is a special term called a multiple-output prime implicant.

The use of multiple-output prime implicants is required to form a minimal $\underline{F}$ to represent $\underline{f}(X)$. The specific utility of MOPIs, as demonstrated in Example 7.1, is to facilitate the sharing of terms among the formulas F_j in $\underline{F}$, thus making $\underline{F}$ cheaper. It is well-known that a minimal

design for a multiple-output function consists of multiple-output prime implicants. We now define the set of MOPIs for a set $[\underline{g}, \underline{h}]$ of intervals:

> A multiple-output prime implicant (MOPI) for a set $[\underline{g}, \underline{h}]$ of intervals is a product of literals which is either:
>
> 1. A prime implicant of one of the functions h_j, $j = 1, 2, \ldots, m$; or
> 2. A prime implicant of one of the product functions
>
> $$h_i \cdot h_j \cdots h_k,$$
>
> in which
>
> - $i, j, k = 1, 2, \ldots, m$, and
> - $i \neq j \neq \cdots \neq k$.
>
> (Givon 70:179-180)

Thus, a multiple-output prime implicant can be a prime implicant of any of the m functions in $\underline{h}$, a prime implicant of the product of two of the functions $h_i \cdot h_j$, up to and including a prime implicant of the product of all m functions $h_i \cdot h_j \cdots h_m$.

Although the set of all multiple-output prime implicants is defined as the prime implicants of the functions $\underline{h}$ as well as the prime implicants of all possible product functions formed from the functions in $\underline{h}$, we do not have to form the product functions to develop the set of all MOPIs. We can use procedures which generate the Blake canonical form of a function to develop the set of all MOPIs, providing that we use a specialized formula $\Phi_H(X, Z)$ derived from the 1-normal form specification $\phi(X, Z) = 1$. Before presenting how $\Phi_H(X, Z)$ is formed, however, we discuss the assignment of costs to MOPIs.

Assignment of Costs to MOPIs. Similar to the cost criteria presented for developing a minimal single-output design, there exist cost criteria that may be used to develop a multiple-output digital design. Typical cost criteria include fewest gates, fewest gate inputs, and variations thereof.

The most common cost criterion is the fewest gates in a two-level circuit. This criterion is generally used when a two-level circuit is to be implemented with a Programmable Logic Array (PLA), the most common method of implementing a two-level circuit. Since the size of a PLA varies in proportion to the number of gates, our main concern is to minimize the number of distinct terms in the corresponding vector $\underline{F}$ of SOP formulas.

Other cost criteria generally involve the number of gate inputs in some fashion. One cost criterion is the fewest gate inputs for the AND gates in the circuit. Using this criterion, the cost of the circuit is measured by the total number of literals contained in distinct terms in $\underline{F}$. A variation of this criterion is the fewest gate inputs with the fewest interconnections. Using this criterion, the cost of $\underline{F}$ is determined by adding the total gate inputs to a sum calculated by adding for each distinct term the number of formulas in $\underline{F}$ in which the term appears; hence, the total number of gate inputs and interconnections is minimized. It is possible that different designs will result for a given specification depending on the cost criterion used to measure minimality of a circuit.

For the remainder of this chapter, we presume that the fewest-gates cost criterion will be used. We take this approach for a number of reasons. First, the most common implementation for two-level circuits is a PLA, in which we are not as concerned with the number of gate inputs as we are with the number of gates. Moreover, heuristic techniques exist which reduce the number of gate inputs after a design with a minimal number of gates is developed. Additionally, most modern algorithms for two-level design, such as MINI (Hong 74), ESPRESSO (Brayt 84), and ESPRESSO-EXACT (Rudel 89), consider only the number of gates. Finally, techniques which are used to form a design with the fewest gates are simpler than those which are used to develop a design with the fewest gate inputs. We thus restrict the available cost criteria in order to reduce the complexity of the resulting algorithms. The techniques presented in the following sections can be easily extended to accommodate additional cost criteria.

Since we are assuming that the fewest-gates cost criterion is used, each multiple-output prime implicant is assigned a cost of 1. The cost of using a MOPI is 1, whether it is contained in a single formula F_j or a number of formulas in $\underline{F}$. Thus, if a MOPI must be contained in one formula F_j, then its use is free in other formulas. An application of this idea is that if a MOPI p_1 is essential for the function f_1 and is not for f_2—but may be used to form F_2—then we will automatically use it in F_2 since it does not cost anything extra to do so with respect to our chosen cost criterion. In some cases, this will cause a formula F_j contained in a minimal $\underline{F}$ to contain redundant terms. Many of these redundant terms are absorbed by other terms in F_j; hence, a way to quickly reduce the number of redundant terms after $\underline{F}$ is formed is to replace each formula F_j with an equivalent absorptive formula at the end of the minimization process.

When using the fewest-gates cost criterion, each MOPI p is treated as a single entity. However, when employing cost criteria other than the fewest gates, we sometimes must create copies of p which must be handled distinctly. One copy must be considered for every possible combinations of formulas in which p may appear. In fact, if p can appear in n formulas in $\underline{F}$, then $2^n - 1$ copies of p may have to be created (Barte 61:28). This requirement adds to algorithmic complexity and increases memory usage. Hence, the handling of MOPIs is much simpler if using the fewest gates cost criterion. We now discuss the formation of the formula $\Phi_H(X, Z)$ used to form the set of all MOPIs.

Formation of $\Phi_H(X, Z)$. In Chapter 4 we discussed the correspondence between a 1-normal form specification and a truth table. Forming $MCF(\phi(X, Z))$ with respect to the X-arguments, a discriminant corresponds to an entry in a truth table in the following manner:

- If a discriminant is a Z-term which contains the literal z_j', then $[g_j(A), h_j(A)] = [0, 0]$[1] for the input combination $A \in \mathbf{B}_2^n$.

[1] The interval $[0, 0]$ is normally denoted as 0; $[1\ 1]$ is denoted as 1; and $[0, 1]$ is denoted as X.

- If a discriminant is a Z-term which contains the literal z_j, then $[g_j(A), h_j(A)] = [1, 1]$ for the input combination $A \in \mathbf{B}_2^n$.
- If a discriminant is a Z-term which does not contain the variable z_j, then $[g_j(A), h_j(A)] = [0, 1]$ for the input combination $A \in \mathbf{B}_2^n$.
- If a discriminant is 1, then $[g_j(A), h_j(A)] = [0, 1]$ for the input combination $A \in \mathbf{B}_2^n$ for each $[g_j(X), h_j(X)]$.
- If a discriminant is 0, then the input combination $A \in \mathbf{B}_2^n$ does not appear in the truth table.

This correspondence may be generalized for the case in which $\phi(X, Z)$ is not represented by its minterm canonical form.

The formula which represents $\phi(X, Z)$ is denoted by $\Phi(X, Z)$. Let us call the portion of a term $t(X, Z)$ in $\Phi(X, Z)$ which consists of X-variables the *X-part*, denoted by $u(X)$. Let us call the portion of $t(X, Z)$ containing Z-variables the *Z-part*, denoted by $v(Z)$. If $t(X, Z)$ contains no Z-variables, then $v(Z) = 1$. It follows that $t(X, Z) = u(X) \cdot v(Z)$. Furthermore, we denote a minterm with respect to X-arguments of $\phi(X, Z)$ by the notation $m(X)$. A substitution $A \in \mathbf{B}_2^n$ which makes the statement $m(A) = 1$ valid is a solution of $m(X) = 1$. Then, for a given term $t(X, Z)$ of $\Phi(X, Z)$, if $v(Z)$ contains the literal z_j', then $[g_j(A), h_j(A)] = [0, 0]$ for all input combinations $A \in \mathbf{B}_2^n$ such that A is a solution of all equations $m(X) = 1$ for which $m(X) \leq u(X)$. We make similar correspondences for the remaining cases.

Bartee (Barte 61) showed that a truth table may be used to develop a formula $\widehat{\Phi}$—in minterm canonical form with respect to the X-variables—in which the equivalent Blake canonical form depicts the set of all MOPIs which can be developed from the functions $\underline{h}$. A truth table depicts the value $[g_j(A), h_j(A)]$ for a given input combination $A \in \mathbf{B}_2^n$. If an $A \in \mathbf{B}_2^n$ exists such that $m(A) = 1$ is valid and $[g_j(A), h_j(A)] = [0, 0]$, then the discriminant $v(Z)$ associated with $m(X)$ in $\widehat{\Phi}$ contains the literal z_j'. However, if $[g_j(A), h_j(A)]$ is equal to $[1, 1]$ or $[0, 1]$, then the discriminant $v(Z)$ associated with $m(X)$ contains neither z_j nor z_j'. Additionally, if there exists a don't care condition corresponding to a missing row in a truth table, i.e., situations in which $[g_j(A), h_j(A)]$ is undefined for a substitution $A \in \mathbf{B}_2^n$, then a minterm $m(X)$ is created for such a row in which the associated discriminant $v(Z)$ is equal to 1.

Using Bartee's methodology, we develop a formula $\Phi_H(X, Z)$ using $\Phi(X, Z)$ for which the equivalent Blake canonical form depicts the set of all MOPIs. $\Phi_H(X, Z)$ is formed in a two-step process:

1. For each term t in $\Phi(X, Z)$, create a term $\hat{t}$ to place in $\Phi_H(X, Z)$ such that $\hat{t}$ contains the literals that t contains with the exception of uncomplemented Z-variables.
2. Terms are created to cover the minterms $m(X)$ corresponding to missing rows in a truth table, i.e., the don't care condition in which $[g_j(X), h_j(X)]$ is undefined for an input combination $A \in \mathbf{B}_2^n$.

The manner in which terms are created in Step 1 is obvious; we now discuss how terms are created in the second step.

In Chapter 4 we defined an equation $\phi_{DC}(X) = 0$ which represents a constraint on $\phi(X, Z) = 1$ due to input combinations which do not occur. This constraint denotes the conditions in which $\phi(X, Z) = 1$ has no solution $Z = \underline{f}(X)$. Another way of viewing this problem is to examine the conditions in which a solution $Z = \underline{f}(X)$ exists for $\phi(X, Z) = 1$, i.e., the conditions in which $\phi(X, \underline{f}(X)) = 1$ is an identity. Such a solution exists if and only if the condition

$$EDIS(\phi(X, Z), Z) = 1 \tag{7.14}$$

is satisfied. Consequently, the equation $EDIS(\phi(X, Z), Z) = 1$ expresses the conditions for which $[g_j(X), h_j(X)]$ is defined. We can derive the conditions in which $[g_j(X), h_j(X)]$ is not defined by forming $(EDIS(\phi(X, Z), Z))' = 0$; whence,

$$\phi_{DC}(X) = (EDIS(\phi(X, Z), Z))'. \tag{7.15}$$

Substitutions $A \in \mathbf{B}_2^n$ such that A is a solution of equations $m(X) = 1$, for which $m(X)$ is a minterm included in terms of the formula representing $\phi_{DC}(X)$, are the substitutions for which

$[g_j(X), h_j(X)]$ is undefined. Hence, the terms in the formula representing $\phi_{DC}(X)$, if any exist, are used to complete the formation of $\Phi_H(X, Z)$.

Certain terms in $\Phi(X, Z)$ may be discarded from consideration while forming $\Phi_H(X, Z)$; these terms may be identified in the process of forming terms to place in $\Phi_H(X, Z)$. A term $t(X, Z)$ in $\Phi(X, Z)$ which contains a Z-part $v(Z)$ of the form $z_1' z_2' \cdots z_m'$, i.e., all Z-variables appear and each is in complemented form, is of no use in forming the MOPIs. In this situation, substitutions $A \in \mathbf{B}_2^n$ such that A is a solution of equations $m(X) = 1$, for which $m(X)$ is a minterm included in corresponding X-part $u(X)$, are substitutions such that $[g_j(A), h_j(A)] = [0, 0]$ for $j = 1, 2, \ldots, m$. Since we are only concerned with substitutions A for which $h_j(A) = 1$, all terms $t(X, Z)$ in $\Phi(X, Z)$ in which $v(Z) = z_1' z_2' \cdots z_m'$ are not placed in the formula $\Phi_H(X, Z)$. Placing such terms in $\Phi_H(X, Z)$ will not affect the formation of the set of MOPIs; however, leaving them out is desirable for efficiency purposes.

The formula $\Phi_H(X, Z)$ is used to generate the set of all MOPIs which may be used to form the vector $\underline{F}$ of formulas. After forming $\Phi_H(X, Z)$, a procedure such as Procedure 2.20 is used to develop the equivalent Blake canonical form. Terms in $BCF(\phi_H(X, Z))$ which contain a Z-part of the form $z_1' z_2' \cdots z_m'$, i.e., terms in which all Z-variables appear and each such variable is in complemented form, may be deleted from $BCF(\phi_H(X, Z))$. Alternatively, we may place the term $z_1' z_2' \cdots z_m'$ in $\Phi_H(X, Z)$ prior to forming $BCF(\Phi_H(X, Z))$ so that terms containing the Z-part $z_1' z_2' \cdots z_m'$ are inhibited from appearing in $BCF(\Phi_H(X, Z))$, i.e., such terms are absorbed by the term $z_1' z_2' \cdots z_m'$.

We now present Procedure 7.1 which develops the formula $\Phi_H(X, Z)$ from $\Phi(X, Z)$. Example 7.2 demonstrates the application of Procedure 7.1.

Procedure 7.1 (Formation of $\Phi_H(X, Z)$): Given a formula $\Phi(X, Z)$, and a set $Z = \{z_1, \ldots, z_m\}$ of Z-variables, the formula $\Phi_H(X, Z)$ is formed in the following manner:

Step 0. Initialise Φ_H to the empty set $\emptyset$.

Step 1.

- If $\Phi(X, Z)$ is empty, then go to Step 3.
- Otherwise, continue to Step 2.

Step 2. Remove the first term $t(X, Z)$ from $\Phi(X, Z)$ and take one of the following actions:

- If $t(X, Z)$ contains a Z-part of the form $z_1' z_2' \cdots z_m'$, i.e., all Z-variables appear and each is in complemented form, then $t(X, Z)$ will not be placed in Φ_H.
- Otherwise, remove any uncomplemented Z-variables from the Z-part of $t(X, Z)$ and place the resulting term in Φ_H.

Return to Step 1.

Step 3. Form $\phi_{DC} = (EDIS(\phi(X, Z), Z))'$ and place the terms in the formula representing $\phi_{DC}(X)$ in Φ_H.

Step 4. Add the term $z_1' z_2' \cdots z_m'$ to $\Phi_H(X, Z)$.

Step 5. $\Phi_H(X, Z)$ has been formed. Return $\Phi_H(X, Z)$.

Example 7.2: Given a 1-normal form specification $\phi(X, Z) = 1$, for which

$$\Phi(X, Z) = a'b'c'd'z_1z_2 + a'b'cd'z_1' + a'b'cdz_1'z_2' + a'bc'd'z_2' + a'bc'dz_1'z_2' \quad (7.16)$$
$$+ a'bcd'z_1z_2 + a'bcdz_1'z_2' + ab'c'd'z_1'z_2' + ab'c'dz_1'z_2' + ab'cd'z_1 \quad (7.17)$$
$$+ ab'cdz_2' + abc'd'z_1'z_2' + abc'dz_1z_2 + abcd'z_2' + abcdz_1z_2,$$

$X = (a, b, c, d)$, and $Z = (z_1, z_2)$, we develop a formula $\Phi_H(X, Z)$. The 1-normal form specification corresponds to the truth table given by Table 7.1. Table 7.2 defines the functions $\underline{h}$.

a	b	c	d	$[g_1(X), h_1(X)]$	$[g_2(X), h_2(X)]$
0	0	0	0	1	1
0	0	1	0	0	X
0	0	1	1	0	0
0	1	0	0	X	0
0	1	0	1	0	0
0	1	1	0	1	1
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	1	X
1	0	1	1	X	0
1	1	0	0	0	0
1	1	0	1	1	1
1	1	1	0	X	0
1	1	1	1	1	1

Table 7.1. Truth Table Corresponding to $\phi(X, Z)$ (Example 7.2)

a	b	c	d	$h_1(X)$	$h_2(X)$
0	0	0	0	1	1
0	0	0	1	1	1
0	0	1	0	0	1
0	0	1	1	0	0
0	1	0	0	1	0
0	1	0	1	0	0
0	1	1	0	1	1
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	1	1
1	0	1	1	1	0
1	1	0	0	0	0
1	1	0	1	1	1
1	1	1	0	1	0
1	1	1	1	1	1

Table 7.2. Truth Table Which Defines $\underline{h}(X)$ (Example 7.2)

Step 0. $\Phi_H = \emptyset$.

Steps 1-2. We iterate through Steps 2 and 3 to form the terms in Φ_H corresponding to terms in $\Phi(X, Z)$. We present the different actions that are taken in these steps.

- Terms $a'b'cdz_1'z_2'$, $a'bc'dz_1'z_2'$, $a'bcdz_1'z_2'$, $ab'c'd'z_1'z_2'$, $ab'c'dz_1'z_2'$, and $abc'd'z_1'z_2'$ in $\Phi(X, Z)$ (7.17) contain a Z-part of the form $z_1'z_2'$. These terms are not placed in Φ_H.
- Uncomplemented Z-literals are removed from the terms $a'b'c'd'z_1z_2$, $a'bcd'z_1z_2$, $ab'cd'z_1$, $abc'dz_1z_2$, and $abcdz_1z_2$ to form the terms

$$a'b'c'd', a'bcd', ab'cd', abc'd, abcd,$$

  which are placed in Φ_H.
- The remaining terms—$a'b'cd'z_1'$, $a'bc'd'z_2'$, $ab'cdz_2'$, and $abcd'z_2'$—do not contain any uncomplemented Z-variables. These terms are simply placed in Φ_H.

Step 3. The function ϕ_{DC} is formed in which $\phi_{DC} = (EDIS(\phi(X, Z), Z))'$. In this case,

$$\phi_{DC}(X) = a'b'c'd.$$

The term $a'b'c'd$ is added to Φ_H.

Step 4. The term $z_1'z_2'$ is added to Φ_H.

Step 5. The formula Φ_H returned by the procedure is:

$$\begin{aligned} &a'b'c'd' + a'b'c'd + a'b'cd'z_1' + a'bc'd'z_2' + a'bcd' \\ &+ ab'cd' + ab'cdz_2' + abc'd + abcd'z_2' + abcd + z_1'z_2'. \end{aligned} \tag{7.18}$$

With the exception of term $z_1'z_2'$, terms in the formula correspond to rows in Table 7.2 in which at least one of the two statements $h_1 \neq 0$ or $h_2 \neq 0$ is true.

Generation of MOPIs. After developing the formula $\Phi_H(X, Z)$ using Procedure 7.1, a procedure such as Procedure 2.20 is used to develop the equivalent Blake canonical form. After forming $BCF(\phi_H(X, Z))$, the term $z_1'z_2' \cdots z_m'$ is deleted from $BCF(\phi_H(X, Z))$. We denote the formula which results after this step as M_{all}.

Each term $t(X, Z)$ in M_{all} contains a single multiple-output prime implicant. The X-part $u(X)$ of $t(X, Z)$ is the actual multiple-output prime implicant that may be used to constitute formulas in $\underline{F}$. The formulas F_j that may be constructed using a given MOPI are determined from the Z-part $v(Z)$ of $t(X, Z)$. Specifically, if a literal z_j' does not appear in $v(Z)$, then $u(X)$ is a MOPI that may be used to form F_j. For example, if $x_1'x_2x_3z_2'$ is a term of M_{all}, then the MOPI $x_1'x_2x_3$ may be used to derive the formula F_1, but not the formula F_2. Thus, each term $t(X, Z)$ of M_{all} contains a MOPI, $u(X)$, as well as an identifier, $v(Z)$, which indicates the formulas F_j that the MOPI may be used to form.

Example 7.3 demonstrates the use of the formula $\Phi_H(X, Z)$ developed in Example 7.2 to generate a formula M_{all}.

Example 7.3: Given the formula $\Phi_H(X, Z)$ developed in Example 7.2, we apply Procedure 2.20 to form M_{all}. We first restate $\Phi_H(X, Z)$:

$$\begin{aligned} &a'b'c'd' + a'b'c'd + a'b'cd'z_1' + a'bc'd'z_2' + a'bcd' \\ &+ ab'cd' + ab'cdz_2' + abc'd + abcd'z_2' + abcd + z_1'z_2'. \end{aligned} \tag{7.19}$$

Using Procedure 2.20, we form $BCF(\Phi_H(X, Z))$. The formula which results is:

$$\begin{aligned} &a'c'd'z_2' + a'cd'z_1' + a'bcd' + a'bd'z_2' + a'b'c' + a'b'd'z_1' \\ &+ acz_2' + abd + ab'cd' + b'cd'z_1' + bcd'z_2' + z_1'z_2'. \end{aligned} \tag{7.20}$$

The term $z_1' z_2'$ is removed from (7.20) to form M_{all}. Using M_{all} we determine the MOPIs which may be used to form F_1. Such MOPIs are contained in terms in M_{all} which do not contain the complemented literal z_1'. We thus derive the set

$$\{a'c'd', a'bcd', a'bd', a'b'c', ac, abd, ab'cd', bcd'\}$$

of MOPIs associated with $[g_1, h_1]$. Similarly, we derive the set

$$\{a'cd', a'bcd', a'b'c', a'b'd', abd, ab'cd', b'cd'\}$$

of MOPIs which may be used to form F_2. We can verify these sets by forming $BCF(h_1)$, $BCF(h_2)$ and $BCF(h_1 \cdot h_2)$ using the functions $\underline{h}$ as defined by Table 7.2. We find that these formulas are defined as follows:

$$\begin{aligned} BCF(h_1) &= a'c'd' + a'bd' + a'b'c' + ac + abd + bcd' \\ BCF(h_2) &= a'b'd' + a'cd' + a'b'c' + abd + b'cd' \\ BCF(h_1 \cdot h_2) &= a'b'c' + a'bcd' + ab'cd' + abd. \end{aligned}$$

Terms in $BCF(h_1)$ and $BCF(h_1 \cdot h_2)$ combine to form the set of MOPIs which may form F_1. Moreover, terms in $BCF(h_2)$ and $BCF(h_1 \cdot h_2)$ combine to form the set of MOPIs which may form F_2. The resulting sets of te ms correspond with the sets developed from $BCF(\Phi_H(X, Z))$.

Useful MOPIs. After forming the set of all MOPIs, we then identify the MOPIs associated with each interval $[g_j, h_j]$ that are useless with respect to g_j for forming a formula F_j. A MOPI p is ***useless*** with respect to an interval $[g_j, h_j]$ if and only if the condition

$$p \leq h_j - g_j \tag{7.21}$$

is an identity. Using this definition, we test a MOPI p for uselessness by determining the validity of the condition

$$p \cdot g_j = 0. \tag{7.22}$$

Procedure 7.2 determines the useless, CE MOPIs with respect to each $[g_j, h_j]$. Once a MOPI p is determined to be useless with respect to $[g_j, h_j]$, the literal z_j' is inserted in the Z-part of the associated term in M_{all}. The association of the literal z_j' with a MOPI denotes that the MOPI

should not be used to develop the formula F_j to represent a function f_j in the interval $[g_j, h_j]$. If at the conclusion of Procedure 7.2 any terms exist in M_{all} which contain the Z-part of the form $z_1' z_2' \cdots z_m'$, i.e., all Z-variables appear and each is in complemented form, then such terms are deleted from M_{all} since they are inessential for all intervals in $[\underline{g}, \underline{h}]$.

Procedure 7.2 (Useless MOPIs): Given the set $\{g_1, g_2, \ldots, g_m\}$ of functions, the set M_{all} which represents the set of all multiple-output prime implicants, and the set $Z = \{z_1, z_2, \ldots, z_m\}$ of Z-variables, the useless prime implicants of M_{all} are identified in the following manner:

Step 0. Initialise the set $\widehat{Z}$ by copying into it the contents of Z.

Step 1. Remove the first literal z_j from $\widehat{Z}$.

Step 2. Form M^j which is the set of all terms in M_{all} which contain neither z_j' nor z_j.

Step 3. Using Procedure 5.1 (Useless Prime Implicants), g_j, and M^j, determine the set of MOPIs in $M^j(X)$ which are useless with respect to interval $[g_j, h_j]$. For each useless MOPI that is identified, insert the literal z_j' in the Z-part of the associated term in M_{all}.

Step 4.

- If $\widehat{Z}$ is empty, then we have completed the identification of useless MOPIs. Continue to Step 5.
- Otherwise, return to Step 2.

Step 5.

1. Any terms in M_{all} which contain the Z-part of the form $z_1' z_2' \cdots z_m'$, i.e., all Z-variables appear and each is in complemented form, are useless and inessential for all intervals in $[\underline{g}, \underline{h}]$. Delete all such terms from M_{all}.
2. Return M_{all}.

Partitioning of MOPIs. After identifying the useless prime implicants, we follow a methodology similar to that used in the single-output case and partition the MOPIs formed from $\underline{h}$ into essential, inessential, and conditionally-eliminable categories with respect to functions in $\underline{g}$. However, this task is considerably more complicated in the multiple-output case due to the fact that a given MOPI may be used to form a number of different formulas F_j in $\underline{F}$. Hence, the bookkeeping required in this process is more involved than in the single-output case.

Given the set M_{all}, the set M^j contains the set of MOPIs in M_{all} which may be used to constitute the formula F_j, i.e., the terms in M_{all} which do not contain a complemented literal z_j'. Let $M^j(X)$ denote the terms in M^j in which the Z-parts have been omitted. The partitioning of

the elements of $M^j(X)$ with respect to $g_j(X)$ is performed in turn for each $j = 1, 2, \ldots, m$. A key aspect of this procedure is that once a MOPI p is identified as essential for a function, it will be treated thereafter as an essential MOPI with respect to each $[g_j, h_j]$ such that p may appear in F_j. Let us denote $M^j_{ess}(X)$ the MOPIs considered as essential for an interval $[g_j, h_j]$. Then, since we use $g_j(X)$ in the process of identifying essential MOPIs (see Procedure 5.2), we may use $M^j_{ess}(X)$ to form a revised function $\tilde{g}_j(X) = g_j(X) - m^j_{ess}(X)$ which is used in place of $g_j(X)$ in identifying the remaining MOPIs in $M^j(X)$ to place in $M^j_{ess}(X)$.

Procedure 7.3 is used to identify essential MOPIs. As a notational device, as essential MOPIs are identified, the Z-part in the associated terms in M_{all} are filled with uncomplemented literals z_j for which no literal z'_j appears, $j = 1, 2, \ldots, m$. Thus, after identifying the essential MOPIs for all functions, terms in M_{all} will contain a Z-part which denotes whether a MOPI is not associated with a function (denoted by a z'_j), is essential for a function (denoted by a z_j), or associated with a function but not essential (denoted by the absence of z_j and z'_j).

Procedure 7.3 (Essential MOPIs): Given the set $\{g_1, g_2, \ldots, g_m\}$ of functions, the set M_{all} which represents the set of all multiple-output prime implicants, and the set $Z = \{z_1, z_2, \ldots, z_m\}$ of Z-variables, the essential MOPIs in M_{all} are identified in the following manner:

Step 0. Initialise the set $\widehat{Z}$ by copying into it the contents of Z.

Step 1.

- If $\widehat{Z}$ is empty, then we have completed the identification of essential MOPIs. Return M_{all} and the set $\{\tilde{g}_1, \tilde{g}_2, \ldots, \tilde{g}_m\}$.
- Otherwise, continue to Step 2.

Step 2. Remove the first literal z_j from $\widehat{Z}$.

Step 3. Form M^j which is the set of all terms in M_{all} which contain neither z'_j nor z_j.

Step 4. Form M^j_{ess} which is the set of all terms in M_{all} which contain the literal z_j.

Step 5. Remove the Z-parts from terms in M^j and M^j_{ess} to form $M^j(X)$ and $M^j_{ess}(X)$, respectively.

Step 6. Form the function $\tilde{g}_j(X) = g_j(X) - m^j_{ess}(X)$.

Step 7. Using Procedure 5.2 (Essential Prime Implicants), the formula $\widetilde{G}_j(X)$, and $M^j(X)$, determine the set of essential MOPIs in $M^j(X)$ with respect to $\tilde{g}_j(X)$.

1. Remove the terms in $\widetilde{G}_j(X)$ used to identify the new essential MOPIs from $\widetilde{G}_j(X)$; these terms are returned by Procedure 5.2.

2. For each new essential MOPI, fill the Z-part of the associated term in M_{all} with uncomplemented literals z_k for each $k = 1, 2, \ldots, m$ for which no literal z'_k appears.

Return to Step 1.

After identifying essential MOPIs, inessential MOPIs are determined. Procedure 7.4 is a follow-up procedure to Procedure 7.3 used to identify inessential MOPIs for each interval $[g_j, h_j]$. In Procedure 7.4, if a MOPI is identified as inessential for $[g_j, h_j]$, then the tag z'_j is inserted in the Z-part of the term associated with the MOPI in M_{all}. The association of the literal z'_j with a MOPI denotes that the MOPI should not be used to develop the formula F_j to represent a function f_j in the interval $[g_j, h_j]$. Similar to Procedure 7.2, if any terms exist in M_{all} at the conclusion of Procedure 7.4 which contain a Z-part of the form $z'_1 z'_2 \cdots z'_m$, i.e., all Z-variables appear and each is in complemented form, then such terms are deleted from M_{all} since they are either useless or inessential for all intervals in $[\underline{g}, \underline{h}]$.

A second action is taken in Procedure 7.4. After determining the inessential MOPIs for an interval $[g_j, h_j]$, a revised lower-bound function $\hat{g}_j(X)$ is derived to use in base formation and development of inclusion formulas.

Procedure 7.4 (Inessential MOPIs and Formation of $\hat{\underline{g}}$): Given the set $\{\tilde{g}_1, \tilde{g}_2, \ldots, \tilde{g}_m\}$ of functions, the set M_{all} which represents the set of all multiple-output prime implicants, and the set $Z = \{z_1, z_2, \ldots, z_m\}$ of Z-variables, the inessential MOPIs of M_{all} are identified in the following manner:

Step 0. Initialise the set $\widehat{Z}$ by copying into it the contents of Z.

Step 1. Remove the first literal z_j from $\widehat{Z}$.

Step 2. Form M^j which is the set of all terms in M_{all} which contain neither z'_j nor z_j.

Step 3. Form M^j_{ess} which is the set of all terms in M_{all} which contain the literal z_j.

Step 4. Remove the Z-parts from terms in M^j and M^j_{ess} to form $M^j(X)$ and $M^j_{ess}(X)$, respectively.

Step 5. Using Procedure 5.3 (Covered Terms), $M^j_{ess}(X)$, and $M^j(X)$, determine the set of inessential MOPIs associated with function $f_j(X)$. For each new inessential MOPI, insert the literal z'_j in the Z-part of the associated term in M_{all}.

Step 6. Form the function $\hat{g}_j(X) = \tilde{g}_j(X) - m^j_{ess}(X)$.

Step 7.

- If $\widehat{Z}$ is empty, then we have completed the identification of inessential MOPIs and formation of $\underline{\hat{g}}$. Continue to Step 8.
- Otherwise, return to Step 2.

Step 8.

1. Any terms in M_{all} which contain the Z-part of the form $z_1' z_2' \cdots z_m'$, i.e., all Z-variables appear and each is in complemented form, are inessential for all intervals in $[\underline{g}, \underline{h}]$. Delete such terms from M_{all}.
2. Return M_{all} and the set $\{\hat{g}_1, \hat{g}_2, \ldots, \hat{g}_m\}$.

At the completion of Procedure 7.4, the set of MOPIs is partitioned into essential, useless and/or inessential, and conditionally-eliminable MOPIs with respect to each interval $[g_j, h_j]$. Additionally, a revised set $\underline{\hat{g}}(X)$ of lower-bound functions is formed. Example 7.4 demonstrates the partitioning of MOPIs using the 1-normal form specification of Example 7.2.

Example 7.4 Using the 1-normal form specification given in Example 7.2, we develop the functions $\underline{g}(X)$:

$$\begin{aligned} g_1 &= a'b'c'd' + abd + ab'cd' + a'bcd' \\ g_2 &= a'b'c'd' + abd + a'bcd'. \end{aligned} \tag{7.23}$$

The formula M_{all} developed in Example 7.3 is

$$\begin{aligned} &a'c'd'z_2' + a'cd'z_1' + a'bcd' + a'bd'z_2' + a'b'c' + a'b'd'z_1' \\ &+ acz_2' + abd + ab'cd' + b'cd'z_1' + bcd'z_2'. \end{aligned} \tag{7.24}$$

Using Procedure 7.2, the functions $\underline{g}$ and M_{all} are used to determine the useless MOPIs. The MOPIs $ab'cd'$ and $b'cd'$ which are associated with $[g_2, h_2]$ are found to be useless. When the literal z_2' is added to the term $b'cd'z_1'$ in (7.24), the term then contains the literals $z_1'z_2'$. Hence, $b'cd'z_1'z_2'$ is deleted from M_{all}. Thus, M_{all} as returned by Procedure 7.2 is the formula

$$a'c'd'z_2' + a'cd'z_1' + a'bcd' + a'bd'z_2' + a'b'c' \qquad (7.25)$$
$$+ a'b'd'z_1' + acz_2' + abdz_1z_2 + ab'cd'z_2' + bcd'z_2'.$$

Using Procedure 7.3, the functions $\underline{g}(X)$ and M_{all} are used to determine the essential MOPIs. The only essential MOPI found among the set of MOPIs is abd. Since abd may be use to form both F_1 and F_2, it is essential for both $[g_1, h_1]$ and $[g_2, h_2]$. At the conclusion of Procedure 7.3, M_{all} is define by the formula

$$a'c'd'z_2' + a'cd'z_1' + a'bcd' + a'bd'z_2' + a'b'c' + a'b'd'z_1' \qquad (7.26)$$
$$+ acz_2' + abdz_1z_2 + ab'cd' + b'cd'z_1' + bcd'z_2'.$$

The functions $\underline{\tilde{g}}(X)$ are also formed in Procedure 7.3. In this example, we develop the functions

$$\begin{aligned} \tilde{g}_1 &= a'b'c'd' + ab'cd' + a'bcd' \qquad (7.27) \\ \tilde{g}_2 &= a'b'c'd' + a'bcd'. \end{aligned}$$

No MOPIs are identified as inessential using Procedure 7.4. Additionally, the functions $\underline{\hat{g}}$ are equal to $\underline{\tilde{g}}$. Hence, M_{all} after the partitioning of the MOPIs is the formula (7.26); $\underline{g}$ is defined by (7.27).

After the completion of Procedure 7.4, the set of MOPIs is partitioned into essential, inessential and/or useless, and conditionally-eliminable categories for each interval $[g_j, h_j]$. At this point, we can form a base for $[\underline{g}, \underline{h}]$.

Formation of Bases

After partitioning the MOPIs for each interval $[g_j, h_j]$ in $[\underline{g}, \underline{h}]$, the next step in our methodology for developing a minimal vector $\underline{F}$ of formulas to represent $\underline{f}(X)$ is to form a base for $[\underline{g}, \underline{h}]$. We present two bases for the intervals in $[\underline{g}, \underline{h}]$. The first base is the set of MOPIs which are conditionally-eliminable with respect to at least one interval in $[\underline{g}, \underline{h}]$. The second base is formed in a fashion similar to Base #3 for the single-output case.

Base #1 - All Conditionally-Eliminable MOPIs. The first base we propose is the set of MOPIs which are conditionally-eliminable with respect to at least one interval in $[\underline{g}, \underline{h}]$. This base is similar to Base #1 for single-output functions. MOPIs which form this base are easily identified using M_{all}. After identifying essential, inessential and/or useless MOPIs, terms in M_{all} corresponding to MOPIs which are conditionally eliminable contain a Z-part in which at least one Z-variable z_j, $j = 1, 2, \ldots, m$, does not appear in either complemented or uncomplemented form.

As in Base #1 of the single-output case, this base is used when an implementation of an algorithm which utilises the base is hosted on a computer which has sufficient memory capacity given the specification for which a minimal $\underline{F}$ is being developed. This base has the most terms; hence, the number of inclusion formulas representing coverage of each term of the base by subsets of the MOPIs will be larger than when employing the second base we will discuss. Thus, a large memory capacity is required to form inclusion formulas and then apply reduction rules. It is the most desirable base, however, because the formation of a minimal $\underline{F}$ which requires the use of search is less likely to be needed using this base.

Base #2 - CE MOPIs Covering $\hat{\underline{g}}$. The second base is a set of conditionally-eliminable MOPIs which covers terms in the formulas $\widehat{\underline{G}}$ which represent the functions $\hat{\underline{g}}$. This base corresponds to Base #3 in the single-output case and is similarly formed. However, a given MOPI p can only

cover terms in a formula G_j such that p is conditionally-eliminable for $[g_j, h_j]$. Procedure 7.5 returns this base as a result.

Procedure 7.5 (Base #2 - CE MOPIs Covering $\hat{\underline{g}}$): Given M_{ce} and $\hat{\underline{g}}$, the set M_{base} which consists of MOPIs which are sufficient to cover the terms in the formulas representing $\hat{\underline{g}}$ is developed in the following manner:

Step 0.

1. Initialise an accumulator $BASE$ to the empty set $\emptyset$.
2. Initialise an accumulator $\widehat{\underline{G}}_{covered}$ to the empty set $\emptyset$. $\widehat{\underline{G}}_{covered}$ will serve as an association list in which each element is a list containing a term in $\widehat{\underline{G}}$ along with the prime implicants which cover the term. $\widehat{\underline{G}}_{covered}$ is completely formed in Steps 1 through 3.
3. Initialise an accumulator $M_{covers} = \emptyset$. M_{covers} will serve as an association list in which each element is list containing a MOPI which covers at least one term in $\widehat{\underline{G}}$ along with the terms that it covers. M_{covers} is completely formed in Steps 4 through 6.

Step 1.

1. Remove the first formula G_j from $\widehat{\underline{G}}$.
2. Form the set M^j of terms of M_{ce} which are conditionally-eliminable with respect to $[g_j, h_j]$. These terms are members of M_{ce} in which neither z_j nor z_j' appears.

Step 2. For each term t in G_j:

1. Form a list T_{assoc} by placing t into it.
2. Determine the members of M^j which contain a MOPI p in the X-part such that $t \leq p$, i.e., the MOPI covers p.
3. For MOPIs p which cover t, place the term associated with p from M_j in T_{assoc}.

Place T_{assoc} in $\widehat{\underline{G}}_{covered}$.

Step 3.

- If $\widehat{\underline{G}}$ is empty, then we have determined for each term in $\widehat{\underline{G}}$ the set of MOPIs that covers it. Continue to Step 4.
- Otherwise, return to Step 1.

Step 4. Remove the first element of $\widehat{\underline{G}}_{covered}$ and denote it T_{assoc}.

Step 5. Let us denote t the term in T_{assoc} from $\widehat{\underline{G}}$. For each term $t(X, Z)$ in T_{assoc}, take one of the following actions:

- If $t(X, Z)$ does not have a corresponding element M_{assoc} in the association list M_{covers}, then create one by forming a list containing $t(X, Z)$ and the term t from the list T_{assoc}.
- Otherwise, append the term t to the list M_{assoc} corresponding to $t(X, Z)$ in M_{covers}.

Step 6.

- If $\underline{\widehat{G}}_{covered}$ is empty, then M_{covers} has been formed. Continue to Step 7.
- Otherwise, return to Step 4.

Step 7. Sort M_{covers} such that the MOPI which covers the most terms is first, the one which covers the second most terms is second, and so on.

Step 8. Remove the first element M_{assoc} from M_{covers} and add the term $t(X, Z)$ in M_{assoc} to $BASE$.

Step 9. Determine the terms t that the MOPI p that comprises the X-part of $t(X, Z)$ in M_{assoc} covers (stored in M_{assoc}). For each term t in M_{assoc}:

1. Remove the corresponding list T_{assoc} from $\underline{\widehat{G}}_{covered}$.
2. For each term $t(X, Z)$ in T_{assoc}, remove the terms t from their corresponding lists M_{assoc} in M_{covers}.

Step 10. Remove lists M_{assoc} from M_{covers} which no longer contain any terms t.

Step 11.

- If M_{covers} is empty, then we have formed the base. Return $BASE$.
- Otherwise, re-sort M_{covers} such that the MOPI which covers the most remaining terms is first, the one which covers the second most terms is second, and so on. Perform this sort in a manner such that a MOPI p which covers fewer terms than previously precedes the MOPIs which previously covered the same number of terms that p now covers, e.g., if p used to cover three terms, but now covers only one, then place it before MOPIs that used to cover one term. Return to Step 8.

If we are not able to use Base #1 due to memory constraints, the base formed by Procedure 7.5 may provide a useful alternative. It is relatively simple to form. Additionally, the terms of the base are very likely to be included in the final vector $\underline{F}$ of formulas. This information is very useful in the search process, if search is required.

Formation of a Multiple-Output Inclusion Formulas

After developing a base for $[\underline{g}, \underline{h}]$, the next step in the process of developing a minimal vector $\underline{F}$ of formulas is to develop a set IF of inclusion formulas which denote coverage of terms of the base. Forming IF for multiple-output functions is similar to the process used for single-output functions. When developing a minimal formula F to represent a single-output function, a single inclusion formula is formed for each prime implicant in the base. After the formation of an inclusion

formula for each term in the base, reduction rules are applied to identify prime implicants to place in F as well as PIs to discard from consideration. In the process, the set IF of inclusion formulas is reduced with respect to the number of contained terms and literals. In the case of multiple-output functions, however, the formation of an inclusion formula for a term in the base is more complicated than for single-output functions. We would like to form a single formula IF_l for each MOPI p_l of the base in order to facilitate the use of the same reduction rules as used in the single-output case.[2] To accomplish this task, we must first develop a set of inclusion formulas—one for each interval $[g_j, h_j]$ that the term p_l may be associated with—and then form the product of the resulting formulas. Only in this manner can we determine the subsets of MOPIs that can be used in place of p_l to constitute the different formulas in $\underline{F}$. This approach was originally used by Cutler (Cutle 80).

Let IF_l^j denote the inclusion formula for the term p_l of the base with respect to $[g_j, h_j]$. Then, the single inclusion formula IF_l for p_l is formed by the product of the formulas IF_l^j formed for each $[g_j, h_j]$ in which p_l may be used to form F_j, i.e.,

$$IF_l = \prod_{\{j | p_l \text{ can form } F_j\}} IF_l^j. \tag{7.28}$$

If p_l can only be used to form a single formula F_j, then the inclusion formula IF_l is equal to IF_l^j. We call an inclusion formula formed as shown in (7.28) a ***multiple-output inclusion formula.*** Only those MOPIs which are useful and conditionally eliminable with respect to the interval $[g_j, h_j]$ are used in the process of forming an inclusion formula IF_l^j for p_l. Additionally, all MOPIs in M_{all} are associated with distinct labels P_k prior to forming any inclusion formulas.

We now state Procedure 7.6 (Formation of a Multiple-Output Inclusion Formula) which uses Procedure 6.3 (Formation of an Inclusion Formula) to form IF_l^j. After each formula IF_l^j is developed using Procedure 6.3, the formula IF_l is generated by:

[2] Since we use the subscripts i and j to denote inputs and outputs, respectively, in the multiple-output case, we will use throughout this chapter the subscript l to denote a terms of the base and the subscript k to denote MOPIs used to cover terms of the base.

1. forming $\prod IF_l^j$; and
2. generating $IF_l = ABS(\prod IF_l^j)$.

Procedure 7.6 (Formation of a Multiple-Output Inclusion Formula): Given a MOPI p_l, the set M_{all}, a set $LABS = \{P_1', P_2', \ldots, P_k'\}$ of labels associated with each member of M_{all}, and the set $\underline{\hat{g}}$ of lower-bound functions, an inclusion formula IF_l denoting the coverage of p_l by conditionally-eliminable MOPIs of each of the intervals $[g_j, h_j]$ for which p_l may be use to form F_j is formed as follows:

Step 0.

1. Initialise a set $\hat{Z}$ which consists of the variables z_j such that p_l is useful and conditionally-eliminable with respect to $[g_j, h_j]$ (determinable by the absence of variables z_j in the Z-part of the term associated with p_l in M_{all}).
2. Initialise an accumulator IF_{acc} to the empty set $\emptyset$.
3. Initialise IF_l to the term 1.

Step 1. Remove the first variable z_j from $\hat{Z}$ and take the following actions:

1. Develop a set M^j which represents all of the MOPIs which are conditionally eliminable for interval $[g_j, h_j]$. This set consists of all terms in M_{all} which contain neither z_j nor z_j'.
2. Develop a set $LABS^j$ which consists of all of the labels in $LABS$ associated with the terms in M^j.
3. Remove the Z-part from terms in M^j to form $M^j(X)$.
4. Using Procedure 6.3 (Formation of an Inclusion Formula), the MOPI p_l, the set $M^j(X)$, the set $LABS^j$, and the lower-bound function $\hat{g}_j$, develop the inclusion formula IF_l^j.
5. Add IF_l^j to IF_{acc}.

Step 2.

- If $\hat{Z}$ is empty, then each formula IF_l^j has been formed. Continue to Step 3.
- Otherwise, return to Step 1.

Step 3.

1. Remove the first formula from IF_{acc} and multiply it by IF_l.
2. Form $ABS(IF_l)$.

Step 4.

- If $IF_{acc} = \emptyset$, then IF_l has been formed. Return IF_l.
- Otherwise, return to Step 3.

Reduction Rules for Multiple-Output Functions

After an inclusion formula IF_l is formed for each term of the base, we may reduce the set IF of inclusion formulas using reduction rules. Because of the manner in which the inclusion formulas are formed, the same reduction rules as used in the single-output case may be used in developing a multiple-output design.

When using Base #1, all conditionally-eliminable MOPIs, Procedure 6.11 (Reduction Rules - Set #1), is used to reduce IF. The set of rules is used because Base #1 in this instance corresponds to Base #1 of the single-output case. Similarly, Base #2, CE MOPIs Covering $\underline{\hat{g}}$, corresponds to Base #3 of the single-output case. Hence, Procedure 6.14 (Reduction Rules - Set #2) is used to reduce the set IF of inclusion formulas when Base #2 is used.

Reducing the set IF of inclusion formulas is the final step of the process of forming a minimal vector $\underline{F}$ of formulas prior to having to use a search process to determine the remaining MOPIs to include in $\underline{F}$. For many functions a minimal $\underline{F}$ is formed after the use of reduction rules, particularly if Base #1 is used. The search process is discussed in Chapter 9. We now state two algorithms which integrate the steps introduced up to the point at which search is required to develop a minimal $\underline{F}$ to represent functions $\underline{f}$ in a set $[\underline{g}, \underline{h}]$ of intervals.

Minimisation Algorithms for Multiple-Output Functions

In this chapter we have presented the first six steps of a methodology for forming a minimal vector $\underline{F}$ of SOP formulas to represent functions $\underline{f}$ in the intervals $[\underline{g}, \underline{h}]$. We summarize the steps in the process:

1. derive a 1-normal form specification $\phi(X, Z) = 1$, if not already formed;
2. form a general solution of $\phi(X, Z) = 1$ for Z to develop a set of intervals such as (7.2);
3. form the set of all *multiple-output prime implicants* of the upper-bound functions $\underline{h}$;
4. develop a base for $[\underline{g}, \underline{h}]$;

5. develop inclusion formulas representing coverage of the terms of the base by the multiple-output prime implicants;
6. reduce the inclusion formulas using reduction rules—identifying prime implicants to include in formulas in $\underline{F}$ as well as to discard from consideration; and
7. use search to determine the remaining prime implicants to include in formulas in $\underline{F}$.

If a minimal $\underline{F}$ is not formed after the first six steps, then a search process must be used to determine the remaining MOPIs which compose the formulas F_j in $\underline{F}$. We now present two algorithms for performing these steps. We will compare the algorithms in an ensuing section.

Algorithm Using Base #1. The first algorithm for forming a minimal $\underline{F}$ uses the set of all useful, conditionally-eliminable MOPIs as a base. A synopsis of this algorithm is:

1. derive a 1-normal form specification $\phi(X, Z) = 1$, if not already formed;
2. form a general solution of $\phi(X, Z) = 1$ for Z to develop a set of intervals such as (7.2);
3. form the set of all *multiple-output prime implicants* of the upper-bound functions $\underline{h}$;
4. develop a base for $[\underline{g}, \underline{h}]$ consisting of all useful, conditionally-eliminable MOPIs;
5. use Procedure 7.6 develop multiple-output inclusion formulas representing coverage of the terms of the base by the MOPIs;
6. use Reduction Rule Set #1 to reduce the inclusion formulas—identifying MOPIs to include in formulas in $\underline{F}$ as well as to discard from consideration; and
7. use a search process to determine the remaining prime implicants to include in formulas in $\underline{F}$.

Algorithm 7.1 implements the first six steps of the aforementioned process. The search process is introduced in Chapter 9.

Algorithm 7.1 (Minimization Algorithm #1): Given a 1-normal form specification $\phi(X, Z) = 1$, a minimal vector $\underline{F}$ of formulas which represent functions $\underline{f}(X)$ belonging to the intervals $[\underline{g}(X), \underline{h}(X)]$ developed from $\phi(X, Z) = 1$ is generated in the following manner:

Step 0.

1. Initialise a partial sum PS to the empty set $\emptyset$.
2. Initialise a variable $P_{discard}$ to the empty set $\emptyset$.

Step 1. For $j = 1, 2, \ldots, m$:

1. Form the set Z_j which is the set of variables in Z less the variable z_j.
2. Form $\tilde{\phi}_j(X, z_j) = EDIS(\phi(X, Z), Z_j)$.
3. Form $g_j(X) = \tilde{\phi}_j'(X, 0) \cdot \tilde{\phi}_j(X, 1)$.
4. Develop a simplified formula to represent $g_j(X)$ using Procedure 2.15 (Simplification). Call the simplified formula G_j.

Step 2. Using Procedure 7.1 (Formation of $\Phi_H(X, Z)$), develop a formula $\Phi_H(X, Z)$ which will be used to form the set of all multiple-output prime implicants.

Step 3.

1. Develop $BCF(\Phi_H(X, Z))$ using Procedure 2.20 (Blake canonical form).
2. Delete the term $z_1' z_2' \cdots z_m'$ in $BCF(\Phi_H(X, Z))$.
3. The formula which results after substeps 1 and 2 is M_{all}.

Step 4. Using Procedure 7.2 (Useless MOPIs), the set $\underline{g}$ of functions, and M_{all}, determine the useless MOPIs with respect to each interval $[g_j, h_j]$. The set M_{all} is revised by Procedure 7.2 to denote the useless MOPIs.

Step 5. Using Procedure 7.3 (Essential MOPIs), the set $\{g_1, g_2, \ldots, g_m\}$, and M_{all}, determine the set of essential MOPIs with respect t each interval $[g_j, h_j]$.

1. The set M_{all} is revised by Procedure 7.3 to denote the essential MOPIs.
2. Replace the set $\{g_1, g_2, \ldots, g_m\}$ of functions with the set $\underline{\tilde{g}} = \{\tilde{g}_1, \tilde{g}_2, \ldots, \tilde{g}_m\}$ returned by Procedure 7.3.

Step 6. Using Procedure 7.4 (Inessential MOPIs and Formation of $\hat{g}$), the set $\underline{\tilde{g}}$ of functions, and M_{all}, determine the set of inessential MOPIs with respect to each interval $[g_j, h_j]$.

1. The set M_{all} is revised by Procedure 7.4 to denote the inessential MOPIs.
2. Replace the set $\underline{\tilde{g}}$ of functions with the set $\underline{\hat{g}} = \{\hat{g}_1, \hat{g}_2, \ldots, \hat{g}_m\}$ returned by Procedure 7.4.

Step 7. Develop a set M_{base} which consists of terms in M_{all} which have a Z-part in which at least one Z-variable z_j does not appear in either complemented or uncomplemented form. M_{base} corresponds to MOPIs which are conditionally eliminable with respect to at least one interval $[g_j, h_j]$.

Step 8. Form a set $LABS = \{P_1', \ldots, P_k'\}$ of labels which will be used to denote the MOPIs in M_{base}.

Step 9. Initialise $IF = \emptyset$. Then, for each term in M_{base}:

1. Remove the Z-part from the term to form p_l.
2. Using Procedure 7.6 (Formation of a Multiple-Output Inclusion Formula), the MOPI p_l, the set M_{base}, the set $LABS$ associated with terms in M_{base}, and the set $\underline{\hat{g}}$ of functions, develop an inclusion formula IF_l denoting the coverage of p_l by conditionally-eliminable MOPIs of each of the intervals $[g_j, h_j]$ for which p_l may be use to form F_j.
3. Add IF_l to IF.

The set IF contains the inclusion formulas IF_l developed for each term in M_{base}.

Step 10. Using Procedure 6.4 (Assignment of Cost to Terms), M_{base}, $LABS$, and $CRITERION$ = fewest gates, develop an association list affiliating a term with a cost. Each cost is paired with the label in $LABS$ which denotes a corresponding PI in M_{base}. Call the resulting list $LAB/COSTS$.

Step 11. Using Procedure 6.11 (Reduction Rules - Set #1), the set IF of inclusion formulas, and the cost list $LAB/COSTS$, apply rule reduction to the set IF. Procedure 6.11 returns a revised set IF_{rev} of inclusion formulas, a set PS_{new} of variables identified for use in $\underline{F}$, and a set $\widetilde{P}_{discard}$ of variables to discard.

1. Replace IF with IF_{rev}.
2. Replace PS to PS_{new}.
3. Replace $P_{discard}$ with $\widetilde{P}_{discard}$.

Step 12. For each variable in PS:

1. Determine the associated term in M_{base}.
2. For the term equal to M_{base} in M_{all}, fill the Z-part of the associated term in M_{all} with uncomplemented literals z_k for each $k = 1, 2, \ldots, m$ for which no literal z'_k appears.

Step 13. For each variable in $P_{discard}$:

1. Determine the associated term in M_{base}.
2. For the term equal to M_{base} in M_{all}, fill the Z-part of the associated term in M_{all} with complemented literals z'_k for each $k = 1, 2, \ldots, m$ for which no literal z'_k appears.
3. If the term in M_{all} then contains a Z-part of the form $z'_1 z'_2 \cdots z'_m$, i.e., all Z-variables appear and each is in complemented form, delete the term from M_{all}.

Step 14.

- If $IF = \emptyset$, then a vector $\underline{F}$ of formulas has been developed. Continue to Step 15.
- Otherwise, a search process must be used to complete $\underline{F}$. Skip to Step 16.

Step 15. For $j = 1, 2, \ldots, m$, form F_j:

1. Examine each term $t(X, Z)$ in M_{all} to determine if the literal z_j appears in the term.
   - If z_j appears in $t(X, Z)$, then place the X-part $u(X)$ in F_j.
   - Otherwise, do not place $u(X)$ in F_j.
2. After each term in M_{all} has been examined, form $ABS(F_j)$.

After each formula $ABS(F_j)$ has been formed, the development of $\underline{F}$ is complete. Return $\underline{F}$.

Step 16.

1. Return the current inclusion formulas IF, M_{base}, and M_{all}.
2. Also return $LAB/COSTS$ and $LABS$ for use in the search process.

Algorithm 7.1 is different from other algorithms found in the literature in the following ways:

- the minimization process begins with a 1-normal form specification $\phi(X, Z) = 1$;
- the partitioning of the MOPIs allows concentration of effort on the useful, conditionally-eliminable MOPIs;
- the base consists of useful, conditionally-eliminable MOPIs; and
- the formation of each inclusion formula IF_i^j is simplified through the use of the $\widehat{G}$-constraint.

Algorithm Using Base #2. We now present an algorithm for forming a minimal $\underline{F}$ which uses a set of useful, conditionally-eliminable MOPIs which cover the functions $\hat{\underline{g}}$. We summarize the algorithm as follows:

1. derive a 1-normal form specification $\phi(X, Z) = 1$, if not already formed;
2. form a general solution of $\phi(X, Z) = 1$ for Z to develop a set of intervals such as (7.2);
3. form the set of all *multiple-output prime implicants* of the upper-bound functions $\underline{h}$;
4. use Procedure 7.5 to develop a base for $[\underline{g}, \underline{h}]$ consisting of useful, conditionally-eliminable MOPIs which cover the functions $\hat{\underline{g}}$;
5. use Procedure 7.6 develop multiple-output inclusion formulas representing coverage of the terms of the base by the MOPIs;
6. use Reduction Rule Set #2 to reduce the inclusion formulas—identifying MOPIs to include in formulas in $\underline{F}$ as well as to discard from consideration; and
7. use a search process to determine the remaining MOPIs to include in formulas in $\underline{F}$.

The first six steps of the foregoing process are implemented by Algorithm 7.2. The search process is introduced in Chapter 9.

Algorithm 7.2 (Minimization Algorithm #2): Given a 1-normal form specification $\phi(X, Z) = 1$, a minimal vector $\underline{F}$ of formulas which represent functions $\underline{f}(X)$ belonging to the intervals $[\underline{g}(X), \underline{h}(X)]$ developed from $\phi(X, Z) = 1$ is generated in the following manner:

Step 0.

1. Initialize a partial sum PS to the empty set $\emptyset$.
2. Initialize a variable $P_{discard}$ to the empty set $\emptyset$.

Step 1. For $j = 1, 2, \ldots, m$:

1. Form the set Z_j which is the set of variables in Z less the variable z_j.

2. Form $\tilde{\phi}_j(X, z_j) = EDIS(\phi(X, Z), Z_j)$.
3. Form $g_j(X) = \tilde{\phi}'_j(X, 0) \cdot \tilde{\phi}_j(X, 1)$.
4. Develop a simplified formula to represent $g_j(X)$ using Procedure 2.15 (Simplification). Call the simplified formula G_j.

Step 2. Using Procedure 7.1 (Formation of $\Phi_H(X, Z)$), develop a formula $\Phi_H(X, Z)$ which will be used to form the set of all multiple-output prime implicants.

Step 3.

1. Develop $BCF(\Phi_H(X, Z))$ using Procedure 2.20 (Blake canonical form).
2. Delete the term $z'_1 z'_2 \cdots z'_m$ in $BCF(\Phi_H(X, Z))$.
3. The formula which results after substeps 1 and 2 is M_{all}.

Step 4. Using Procedure 7.2 (Useless MOPIs), the set $\underline{g}$ of functions, and M_{all}, determine the useless MOPIs with respect to each interval $[g_j, h_j]$. The set M_{all} is revised by Procedure 7.2 to denote the useless MOPIs.

Step 5. Using Procedure 7.3 (Essential MOPIs), the set $\{g_1, g_2, \ldots, g_m\}$, and M_{all}, determine the set of essential MOPIs with respect to each interval $[g_j, h_j]$.

1. The set M_{all} is revised by Procedure 7.3 to denote the essential MOPIs.
2. Replace the set $\{g_1, g_2, \ldots, g_m\}$ of functions with the set $\tilde{\underline{g}} = \{\tilde{g}_1, \tilde{g}_2, \ldots, \tilde{g}_m\}$ returned by Procedure 7.3.

Step 6. Using Procedure 7.4 (Inessential MOPIs and Formation of $\hat{\underline{g}}$), the set $\tilde{\underline{g}}$ of functions, and M_{all}, determine the set of inessential MOPIs with respect to each interval $[g_j, h_j]$.

1. The set M_{all} is revised by Procedure 7.4 to denote the inessential MOPIs.
2. Replace the set $\tilde{\underline{g}}$ of functions with the set $\hat{\underline{g}} = \{\hat{g}_1, \hat{g}_2, \ldots, \hat{g}_m\}$ returned by Procedure 7.4.

Step 7.

1. Develop a set M_{ce} which consists of terms in M_{all} which have a Z-part in which at least one Z-variable z_j does not appear in either complemented or uncomplemented form. M_{ce} corresponds to MOPIs which are conditionally eliminable with respect to at least one interval $[g_j, h_j]$.
2. Using Procedure 7.5 (Base #2 - CE MOPIs Covering $\hat{\underline{g}}$), M_{ce}, and $\hat{\underline{g}}$, develop a set M_{base} which consists of MOPIs which are sufficient to cover the terms in the formulas representing $\hat{\underline{g}}$.

Step 8. Form a set $LABS = \{P'_1, \ldots, P'_k\}$ of labels which will be used to denote the MOPIs in $MOPI_{ce}$.

Step 9. Initialise $IF = \emptyset$. Then, for each term in M_{base}:

1. Remove the Z-part from the term to form p_l.
2. Using Procedure 7.6 (Formation of a Multiple-Output Inclusion Formula), the MOPI p_l, the set M_{ce}, the set $LABS$ associated with terms in M_{ce}, and the set $\hat{\underline{g}}$ of functions, develop an inclusion formula IF_l denoting the coverage of p_l by conditionally-eliminable MOPIs of each of the intervals $[g_j, h_j]$ for which p_l may be use to form F_j.

The set IF contains the inclusion formulas IF_l developed for each term in M_{base}.

Step 10. Using Procedure 6.4 (Assignment of Cost to Terms), M_{base}, $LABS$, and $CRITERION =$ fewest gates, develop an association list affiliating a term with a cost. Each cost is paired with the label in $LABS$ which denotes a corresponding PI in M_{ce}. Call the resulting list $LAB/COSTS$.

Step 11. Using Procedure 6.14 (Reduction Rules - Set #2), the set IF of inclusion formulas, and the cost list $LAB/COSTS$, apply rule reduction to the set IF. Procedure 6.14 returns a revised set IF_{rev} of inclusion formulas, a set PS_{new} of variables identified for use in $\underline{F}$, and a set $\widetilde{P}_{discard}$ of variables to discard.

1. Replace IF with IF_{rev}.
2. Replace PS to PS_{new}.
3. Replace $P_{discard}$ with $\widetilde{P}_{discard}$.

Step 12. For each variable in PS:

1. Determine the associated term in M_{ce}.
2. For the term equal to M_{ce} in M_{all}, fill the Z-part of the associated term in M_{all} with uncomplemented literals z_k for each $k = 1, 2, \ldots, m$ for which no literal z'_k appears.

Step 13. For each variable in $P_{discard}$:

1. Determine the associated term in M_{ce}.
2. For the term equal to M_{ce} in M_{all}, fill the Z-part of the associated term in M_{all} with complemented literals z'_k for each $k = 1, 2, \ldots, m$ for which no literal z'_k appears.
3. If the term in M_{all} then contains a Z-part of the form $z'_1 z'_2 \cdots z'_m$, i.e., all Z-variables appear and each is in complemented form, delete the term from M_{all}.

Step 14.

- If $IF = \emptyset$, then a vector $\underline{F}$ of formulas has been developed. Continue to Step 15.
- Otherwise, a search process must be used to complete $\underline{F}$. Skip to Step 16.

Step 15. For $j = 1, 2, \ldots, m$, form F_j:

1. Examine each term $t(X, Z)$ in M_{all} to determine if the literal z_j appears in the term.
   - If z_j appears in $t(X, Z)$, then place the X-part $u(X)$ in F_j.
   - Otherwise, do not place $u(X)$ in F_j.
2. After each term in M_{all} has been examined, form $ABS(F_j)$.

After each formula $ABS(F_j)$ has been formed, the development of $\underline{F}$ is complete. Return $\underline{F}$.

Step 16.

1. Return the current inclusion formulas IF, M_{ce}, and M_{all}.
2. Also return $LAB/COSTS$ and $LABS$ for use in the search process.

The following aspects of Algorithm 7.2 are novel:

- the minimisation process begins with a 1-normal form specification $\phi(X, Z) = 1$;
- the partitioning of the MOPIs allows concentration of effort on the useful, conditionally-eliminable MOPIs;
- the base consists of a subset of the useful, conditionally-eliminable MOPIs derived using the lower-bound functions $\underline{\hat{g}}$;
- the formation of each inclusion formula IF_i^j is simplified through the use of the $\widehat{G}$-constraint; and
- rule reduction is facilitated through the use of a revised set of reduction rules.

Comparison of the Algorithms. The utility the foregoing algorithms is dependent on the available computational resources as well as the complexity of the functions for which we are attempting to form a minimal $\underline{F}$. When memory is available, Algorithm 7.1 is the best technique to use since the resulting inclusion formulas give us the most information about MOPIs to place in formulas F_j in $\underline{F}$ as well as to discard from consideration. Hence, the least amount of work must be performed by the ensuing search process when Base #1 is used. However, if we are constrained with respect to resources, then Algorithm 7.2 is used.

Algorithm 7.1 has the largest base and subsequently the largest set of inclusion formulas. With a larger set of formulas, we are able to identify the largest set of MOPIs for $\underline{F}$. Moreover, the inclusion formulas are reduced to the greatest possible extent. Thus, if search is required, it will not involve as much effort. However, this method uses a considerable amount of memory because of the number of MOPIs and corresponding inclusion formulas. With many inclusion formulas, the process of applying reduction rules is memory intensive.

When less memory is available, we must resort to using a smaller base. Fewer inclusion formulas are generated when a smaller base is used. Hence, the demands on memory resources are not as severe as when we use Base #1. However, the cost of developing fewer inclusion formulas is that we cannot in general reduce the set of inclusion formulas as much as when a larger base is used. As a result, effort is shifted to the search process.

Summary

In this chapter we have introduced a methodology for developing a vector $\underline{F}$ of formulas which represents functions $\underline{f}$ belonging to a set $[\underline{g}, \underline{h}]$ of intervals. The resulting formulas correspond to a multiple-output design which meets a specification $\phi(X, Z) = 1$. Algorithms presented in this chapter contain all steps required to form a minimal $\underline{F}$ up to the point at which the use of search is required. The search process is discussed in Chapter 9.

The concepts presented in this chapter have been extensions of the techniques presented for single-output functions in Chapter 6. We have highlighted the differences between the techniques used in the single-output case and the methods described in this chapter.

We have extended a number of new ideas for the multiple-output case in this chapter:

- Two algorithms, based on different multiple-output prime-implicant bases, were presented so that we have available an algorithm appropriate for available computing resources as well as the complexity and size of the function which corresponds to the circuit specification.
- We begin each algorithm with a 1-normal form specification $\phi(X, Z) = 1$ with which we form a general solution for Z, thus yielding the intervals $[\underline{g}(X), \underline{h}(X)]$. This is done to emphasize that deriving a design corresponds to solving a Boolean equation.
- A methodology for partitioning of the MOPIs was presented which concentrates effort on the useful, conditionally-eliminable MOPIs.
- We have extended the equation-based approach to the generation of inclusion formulas incorporating the use of constraints for MOPIs. This approach provides a theoretically-sound foundation for the reasoning process utilised to generate inclusion formulas, something that has been lacking in previous work. The use of constraints makes the process of generating inclusion formulas more efficient.
- A modified set of reduction rules was used for situations where only a subset of the set of conditionally-eliminable MOPIs is contained in the base. These rules facilitate a moderate reduction in the inclusion formulas prior to the use of search.

VIII. An Introduction to Search

The last step in our process of forming a minimal vector $\underline{F}$ of formulas which corresponds to a design is to perform a search process. The search process, as implemented in this research, is presented in Chapter 9. In this chapter, we present background material on the concepts of search for those who may be unfamiliar with such techniques. This chapter may be scanned by someone familiar with informed search techniques. However, information presented in this chapter will be used in our discussion of the application of search in developing minimal designs as presented in Chapter 9.

Our presentation in this chapter is similar to discussions of search techniques as found in texts on artificial intelligence. We introduce state-space and problem-reduction representations used in search processes, which are two means of abstractly portraying a problem. Once a problem is represented, a number of search strategies may be applied which use the representations for the purpose of developing a solution to a problem. We present a number of different search strategies which we may apply to our problem; specific attention is placed on those strategies used in this work. To make a search strategy more efficient, heuristics are used to reduce the number of choices that may have to be considered in the course of a search; the importance and uses of heuristics in search are discussed. Sources for information found in this chapter are *The Handbook of Artificial Intelligence, Volume I* (Barr 81), *Artificial Intelligence* (Rich 83), *Heuristics: Intelligent Search Strategies for Computer Problem Solving* (Pearl 84), and *Artificial Intelligence* (Winst 84).

Minimisation Versus Satisficing

In this work we are concerned with developing *minimal* or least-cost designs to meet circuit specifications. Hence, the task we are performing is called *minimization*, which is a process in which we develop a solution which is at least as good as any other solution. If our goal was to find a solution to the problem as quickly as possible without regard to the resulting quality of the solution the task

would be called *satisficing*. In many cases, the cost of the process to develop a minimal solution is prohibitive or attaining one is impossible; in those instances, we accept *near-minimal* solutions to the problem. When we accept near-minimal solutions within some bound of a minimal solution, we perform *semi-minimization*. Semi-minimization is divided into two categories: *near-minimization* and *approximate-minimization*. Near-minimization is the case in which the acceptance region with respect to a minimal solution is within defined bounds, i.e., the resulting solution is guaranteed to be within some bound of a minimal one. In approximate-minimization tasks, there is a high probability that the resulting solution is near-minimal.

We use the terminology in the foregoing paragraph to frame our discussion of search processes in the remainder of this chapter. The definitions are adapted from terminology used by Pearl, who presented analogous terms in a discussion of optimization (Pearl 84:14-15).

Problem Representations

Two common methods for abstractly depicting a problem in search processes are *state-space* and *problem-reduction* representations. Both methods are means of systematically representing a problem to facilitate problem-solving. In the current section, we introduce these representations as well as the use of graphs in state-space and problem-reduction representations.

State-Space Representations. When we solve a problem we would like to do so in a methodical manner. To accomplish this, we need a clearly-defined starting point as well as distinct goals or objectives. Once we have determined our starting and ending points, we then must develop an orderly way to realize the goals, progressing from the starting point. In many problems we use a technique called *forward reasoning* in which we move from an initial configuration of the problem through intermediate ones until we ultimately reach a goal position. A configuration which represents a condition of the problem at a stage in the solution process is called a *state* of the problem. Additionally, we call the initial problem configuration the *initial state* and any goal

condition a *goal state*. Rules which guide the transformation of one state of the problem to another are called *operators*. Moreover, the set of all possible states derivable from the initial state using the operators is called a *state-space*. A problem representation based on forward reasoning which uses operators to transform one state to another on the path from the initial state to a goal state is called a *state-space representation* of a problem.

After we devise the state-space representation for the problem, we apply techniques to attain a goal state from the initial configuration of the problem. This often entails determining the appropriate sequence of operator applications. Starting from the initial state, we say that we are *searching* for a goal state. A specific set of rules for moving from the initial problem state to a goal state is called a *search strategy*. In forming these rules, we may find that there are short-cuts that allow us to get to a goal state quickly, although we may forsake the opportunity to get to the best goal state. Such techniques are called *heuristics*. Search strategies and heuristics will be discussed in later sections of this chapter.

The important aspects of a state-space representation of a problem are that

- each state is a symbolic configuration of a fixed set of elements;
- operators are used to move from one state to another; and
- a search strategy is used to choose a good sequence of operator applications in order to move forward from the initial state to a goal state.

Although a state-space representation of a problem may often be appropriate, in many circumstances an alternative means of depicting a problem called a problem-reduction representation may be more suitable.

Problem-Reduction Representations. In many situations a problem may be *decomposable* into a set of smaller subproblems such that the combined solutions of the subproblems constitute a solution for the original problem. A problem representation in which problems are reduced to a set of subproblems, each of which is then solved independently, is called a *problem-reduction*

representation. Whereas operators in a state-space representation are used to transform one state to another, operators in a problem-reduction representation typically decompose a problem into a set of subproblems. Once all of the subproblems are solved, the subproblem solutions are combined to form a global problem solution. Since work proceeds backward from subproblem solutions, i.e., goal states, to the initial state to form a global solution, problem-reduction approaches are said to employ *backward reasoning.*

The concepts which distinguish a problem-reduction representation from a state-space representation are that

- a set of operators is used to transform a large problem into a set of subproblems;
- after decomposing the problem into subproblems, all subproblems are independently solved; and
- after each subproblem is solved, reasoning proceeds backward from the initial goal, i.e., the combination of subproblem solutions, to the initial problem state.

Although state-space and problem-reduction representations are different approaches for solving a problem, graphs are used in both methods as a vehicle for the representation. The use of graphs in both of these representations in discussed in the next section.

Search Graphs. The typical method for portraying state-space and problem-reduction representations is through the use of *search graphs.* A search graph is a representation of the entire *search space.* In a search graph, the initial state is called the *root node* of the graph. Successors to a given node n are called *children* of the node; n is called that *parent* of the children. Operators are used to develop successor nodes from a given node; the *arcs* of the graph depict operator applications. A sequence of arcs that is traversed to go from one node to another is called a *path.* The average number of children that a node may have in a search graph is called the *branching factor* of the graph.

A search graph may be explicitly depicted; however, a search graph is more likely to be implicitly used by a search strategy, i.e., it only exists in an abstract sense. On the other hand, specific nodes are stored explicitly when generated by a search strategy. A node contains relevant information such as an encoding of the portion of a problem that it represents as well as a means for describing all points between it and the root node, i.e., how it was generated from the root node. A node also contains information that allows us to form successor nodes for it. At a point in the search process, the set of all nodes may be partitioned into four groups:

- nodes that have been *expanded*;
- nodes that have been *explored*, but not expanded;
- nodes that have been *generated*, but not explored; and
- nodes that have not been generated. (Pearl 84:34)

A node is said to be *generated* when it is produced and stored explicitly in the course of a search process. A node is *explored* when at least one of its successor nodes has been generated. When all of the children of a node n have been generated, then n is said to be *expanded*. In search strategies two lists are typically maintained: one contains nodes that have been generated but not yet expanded and the other nodes that have been expanded; these lists are called *OPEN* and *CLOSED* lists, respectively.

In a state-space representation, each node in a search graph corresponds to a state in a state-space. An arc in the search graph corresponds to the transformation of one state to another via an operator application. A path in the search graph from the initial state to a goal state represents a solution to the problem to be solved. The nodes which are generated in the course of a search depict portions of the total state-space which have been explored. In general, only a small portion of the entire state-space is examined. From a given node n, a solution may possibly be found via a path through n and subsequently through one of its successor nodes; each successor node corresponds to an alternative path that may be taken in the course of finding a solution. Paths through n to

each of its successor nodes are said to be *competing branches* of the search; hence, arcs from n to the successor nodes are called *OR arcs*. Additionally, successor nodes reached through OR arcs are called *OR nodes*. A graph in which the children of every node n correspond to alternative solutions is called an *OR graph*. State-space representations are normally depicted using OR graphs.

A graph used to portray a problem-reduction representation is similar in some respect to graphs corresponding to state-space representations. Each node in a search graph corresponds to a state in search space. For some nodes the children may lead to alternative solutions; the arcs leading to the children are OR arcs. However, for other nodes in the graph, a subset of the node's children may correspond to a decomposition of the problem. Each of the subproblems must be solved to develop a solution through the parent node. Arcs which lead to successor nodes corresponding to subproblems are called *AND arcs*. Successor nodes corresponding to subproblems are called *AND nodes*. A graph which includes both AND and OR arcs is called an *AND-OR graph*. Problem-reduction representations are depicted using AND-OR graphs. Because of the different types of arcs in AND-OR graphs, search strategies used to develop solutions from AND-OR graphs are more complicated than those used to derive solutions from OR graphs.

In general, a search graph is used to depict a search space. However, any search graph has a corresponding *search tree*. Searching through a search tree is simpler than searching through a search graph. However, in some problems effort may be duplicated when using a search tree due to the fact that in a search tree the same node may be generated several times in different paths and hence will have to be processed repetitively. This problem may be handled by comparing a newly-generated node to previously-generated nodes so that duplicate nodes are discarded. We will represent the search space for the problem of developing a minimal digital design as a search tree; hence, we restrict our discussion of search spaces in the remainder of this chapter to search trees.

Figure 8.1 depicts an AND-OR tree. Node A is the root node of the tree. Nodes B, C, and D are children of node A. The children of node B—nodes E, F, and G—are OR nodes. Nodes H, I, and

J are AND nodes. The set of AND arcs, i.e., those leading to the AND nodes corresponding to a set of subproblems, are linked together by a line which connects the arcs. In some cases, a node may have successor nodes which are OR nodes as well as AND nodes; node D is such a node. Node K is an OR node; nodes L and M are AND nodes.

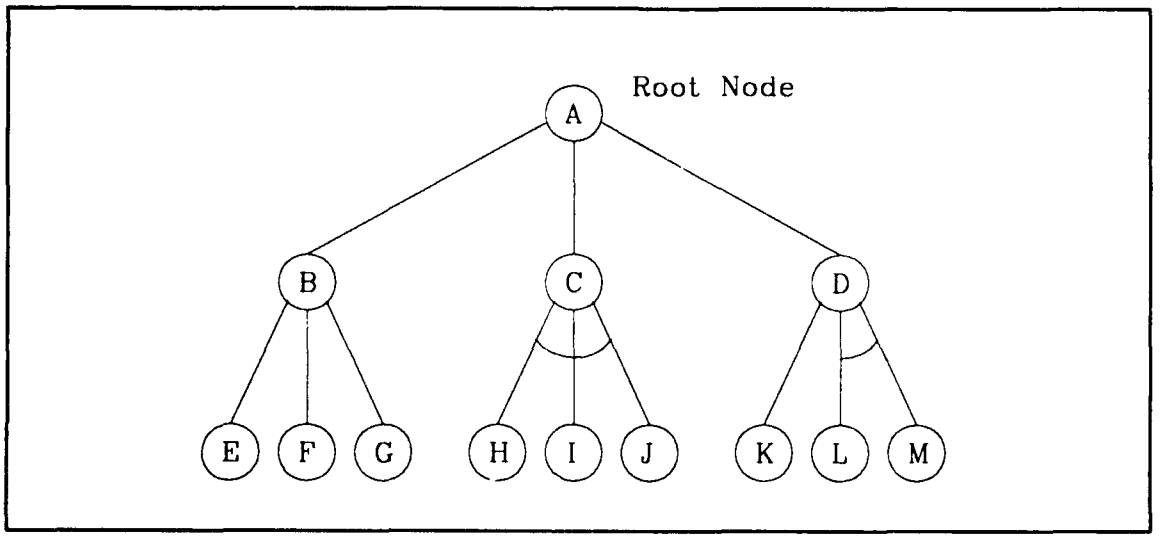


Figure 8.1. An AND-OR Tree

Search strategies are used to develop a solution from a search graph. Information about the problem domain is used by a search strategy to select a node to expand at a given point in the search process. Each search strategy uses the domain information in a specific manner to select a node for expansion. In most search strategies a least-cost node is selected for expansion. An ***evaluation function*** $f(n)$ is used to evaluate the cost of a node n. The function $f(n)$ comprises two component functions: $g(n)$ and $h(n)$. The function $g(n)$ is a measure of the exact cost of getting from the root node to n. The function $h(n)$—which is called a ***heuristic function***—is an estimate of the cost of getting from n to a goal node. In general, $f(n)$ is defined by the equation

$$f(n) = g(n) + h(n). \tag{8.1}$$

If $h^*(n)$ is the actual cost of getting from n to a goal node, then $f^*(n) = g(n) + h^*(n)$ is the actual cost of a path from the root node to a goal node which passes through n. The manner in which $f(n)$, $g(n)$, and $h(n)$ are used differs depending on the search strategy that is employed. A search strategy that employs a heuristic function is called *heuristic* or *informed* search. Search processes which do not use a heuristic function are called *uninformed* search.

The nature of the heuristic function $h(n)$ is particularly important in search strategies. However, the development of a heuristic function is only one way that heuristic information is used by search processes. In most cases we desire to form a minimal solution. In other cases, we will accept a near-minimal solution if the search process can be performed more efficiently. Heuristic information is often used to reduce the effort required in the course of a search—sometimes at the expense of forming a near-minimal solution rather than a minimal one. The use of heuristics is one of the most important issues in applying search to practical problems; hence, we address the use of heuristics in search in greater detail.

Heuristics in Search

An Overview of Heuristics. Search processes are used to develop solutions for a myriad of problems. In most cases, the tasks to which we apply search are nontrivial. In fact, many—if not most—problems in which search is used are NP-complete. These problems are characterized by what is often termed a "combinatorial explosion" inherent in attempting to find a solution. Exhaustive blind search is totally impractical for NP-complete problems because the number of states that must be examined is too immense to explore in a lifetime. To handle such problems, we often must devise mechanisms to reduce the effort that must be expended by a search process by either

1. reducing the number of choices that must be made, or
2. differentiating among possible choices.

Rules based on task-dependent information used to reduce search effort are called *heuristics*. According to Pearl, heuristics are

> criteria, methods, or principles for deciding which among several alternative courses of action promises to be most effective in order to achieve some goal, (Pearl 84:3)

i.e., a solution to a problem. Heuristics allow a search space to be trimmed to a size which is computationally manageable. Hence, heuristics facilitate the ability to solve problems that would otherwise be intractable.

Heuristics are used in two ways in search processes. First, heuristics are used to eliminate paths in a search tree. In this case, heuristics are simplifying choices or rules of thumb which are used to identify and discard from consideration nodes which most likely lead either to non-minimal solutions or to no solution at all. Such nodes are said to be "pruned" from the search tree. The second way in which heuristics are used is to generate estimates of the distance between intermediate nodes and goal nodes, i.e., via a heuristic function $h(n)$. Heuristic functions are used to assign costs to nodes to determine to order of node expansion. In either case, the use of heuristics may greatly increase the efficiency of the development of a solution. However, the cost of this efficiency is that the "best" path may be overlooked, i.e., a non-minimal solution might be developed. A non-minimal solution is often acceptable as long as we believe its cost to be within a reasonable bound of the cost of a minimal solution.

There exist a number of qualities which distinguish good heuristics from poor ones. These qualities include ease of calculation, whether a minimal solution is insured, and whether a reasonable search is performed. We would like the number of calculations involved in generating heuristic information to be as small as possible; a heuristic should be a simple way of indicating which course of action is preferable. Otherwise, we may spend more time calculating which alternative to choose in a search than performing the search itself. If a minimal solution is guaranteed to be found (if a solution exists) we often consider a heuristic to be good. However, in some cases we

would rather use a heuristic that minimizes search effort rather than the solution cost. Thus, a definitive statement that a given heuristic is better than another cannot be made apart from the problem to which they will be applied. Ideally, we would like a heuristic to expedite the pruning of a search, because we desire only to explore nodes along a minimal path from the start state to a goal state. In some situations, however, the ability to attain a best-cost solution may come at the expense of an exorbitant search effort. A heuristic that is good for one problem may be bad for another. Thus, a balance must be struck between the need to minimize solution cost and to minimise search effort. This balance must be examined on a case-by-case basis.

Heuristic Functions. The essential algorithmic difference among various search strategies is the nature of the heuristic function $h(n)$ that is used to guide the search. No heuristic function is used in uninformed search processes; alternatively, we may view uninformed search processes as having a heuristic function in which $h(n) = 0$. A given heuristic function $h_1(n)$ may have certain properties which make its use more desirable than an alternative function $h_2(n)$. The properties that distinguish one heuristic function from another are

- admissibility,
- consistency, and
- informedness.

We now discuss each of these properties.

Let $h^*(n)$ be the actual cost of getting from node n to a goal node. A heuristic function $h(n)$ is said to be *admissible* if the condition

$$h(n) \leq h^*(n) \qquad \forall n \tag{8.2}$$

is true. Thus, an admissible heuristic function $h(n)$ never overestimates the actual cost of the distance between a node n and some goal node, i.e., $h(n)$ is an *optimistic* estimate. Admissible

heuristic functions are very important in search processes. In some search strategies, the A* algorithm for example, the solution produced by the search process is guaranteed to be minimal if an admissible heuristic function is used. An algorithm is said to be *admissible* if it is guaranteed to return a least-cost solution if a solution exists.

Ideally, a heuristic function $h(n)$ exactly estimates the actual cost $h^*(n)$. This is not normally possible; however, we may be able to develop a heuristic function $h_2(n)$ which always yields a better estimate of $h^*(n)$ than does a second function $h_1(n)$. A heuristic function $h_2(n)$ is said to be more *informed* than $h_1(n)$ if both are admissible and

$$h_2(n) > h_1(n), \qquad \text{for every nongoal node } n. \tag{8.3}$$

Less effort is required in the search process if the most informed heuristic function is used. An algorithm which uses the more informed heuristic function $h_2(n)$ is said to be more *informed* than an algorithm which uses the function $h_1(n)$. In the A* algorithm, using a more informed heuristic function will generally result in fewer node expansions in the process of finding a minimal solution than if a less informed heuristic function is used, i.e., a minimal solution will be developed with minimal effort expended.

In any search strategy we would like to avoid reopening a node that was placed on the CLOSED list. A heuristic function that ensures this is called a *consistent* heuristic function. Let $k(n, n')$ be the cost of a cheapest path from node n to node n'. Then a heuristic function $h(n)$ is consistent if the condition

$$h(n) \leq k(n, n') + h(n') \qquad \forall n, n' \tag{8.4}$$

is satisfied. This condition is known as the *triangle inequality* which must hold throughout the search space. Every consistent heuristic function is also an admissible one. It has been shown that

the A* algorithm is not minimal with respect to the number of node expansions if this condition does not hold.

We will discuss the properties of heuristic functions in more detail in our discussion of the A* algorithm in the next section in which we survey the A* algorithm as well as a number of other search strategies.

Search Strategies

In this section we survey different search strategies which may be used to develop a solution from a search space. Descriptions of the search types are included in this chapter. Algorithms which implement each of the search strategies are listed in Appendix D. However, prior to introducing search strategies we introduce terminology used to differentiate them. After introducing search classifications, we present a number of uninformed and informed search techniques.

Search Strategy Classifications. Search algorithms may be categorized as blind, uninformed or informed. In blind searches the order in which nodes of the search tree are examined is determined by the search method and not by any property of the corresponding states, i.e., neither $g(n)$ nor the heuristic function $h(n)$ is used. The function $g(n)$ is used to evaluate nodes in uninformed search strategies. In informed search, properties of the corresponding states are used to guide the search in order to limit the amount of searching that must be performed, i.e., heuristic functions are used. An evaluation function $f(n) = g(n) + h(n)$ is used to estimate the relative benefit or cost of continuing a search from a given node.[1]

An algorithm is said to be *complete* if it terminates with a solution if one exists; it is *admissible* if it is guaranteed to return an optimal solution if a solution exists. An algorithm A_1 is said to *dominate* an algorithm A_2 if every node expanded by A_1 is also expanded by A_2. A_1 *strictly*

[1] The literature does not distinguish between uninformed search and blind search. However, we make a distinction between the two since we use information about a node, i.e., $g(n)$, to select a node for expansion in uninformed search. In a blind searches, such as breadth-first search, the order of node expansions is based solely on the topology of the search tree.

dominates A_2 if A_1 dominates A_2 and A_2 does not dominate A_1. Hence, an algorithm A_1 which strictly dominates A_2 is more efficient than A_2. Furthermore, an algorithm is said to be *optimal* over a class of algorithms if it dominates all members of that class. (Pearl 84:75)

In many search strategies, decisions are made such that nodes are discarded from further consideration in the course of the search process. Strategies which make such decisions are said to be *irrevocable*. If a strategy is not irrevocable, then it is said to be *revocable*. We will introduce search strategies which are both revocable and irrevocable.

Blind Search Techniques.

Breadth-First Search. *Breadth-first* search is a brute-force approach in which the nodes of the search tree are examined level-by-level. No node on a level is examined until every node on the previous level has been examined. We can view breadth-first search as using an evaluation function $f(n)$ in which $g(n)$ is the number of arcs between the root node and n and $h(n)$ is equal to zero. This type of search is managed by an OPEN list which is a queue. When a node is expanded, its children are placed at the end of the queue to await their turn to be expanded. The most important advantage of this type of search is that it will always find a shortest path from the initial node to a goal node. However, a shortest path to a solution may not be a least-cost path.

Figure 8.2 illustrates a breadth-first search. The numbers in the nodes indicate the order in which the nodes in the search tree are expanded. For complex problems, breadth-first search requires too much memory since many nodes may be added to the OPEN list before the search reaches a goal node, especially when the path to a goal node is long. Hence, breadth-first search is only feasible for relatively simple problems. Procedure 8.1 (Breadth-First Search) in Appendix D implements breadth-first search.

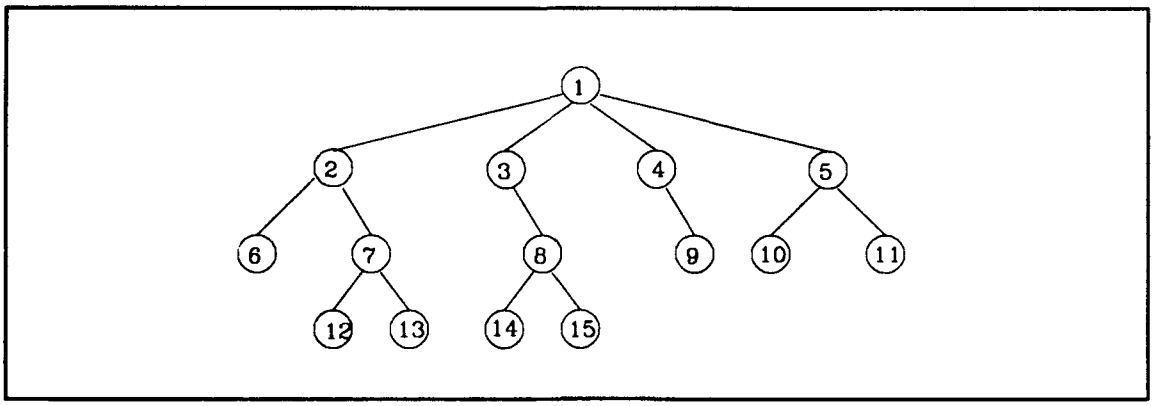


Figure 8.2. Illustration of Breadth-First Search

Depth-First Search. Another type of search that is also a brute-force approach is ***depth-first*** search. In depth-first search a search tree is explored in as deep a manner as possible before backtracking. Once a node is reached in which the search may go no deeper, backtracking occurs to its parent node and another child is explored. Hence, depth-first search minimizes backtracking as much as possible. It is a good method to use when backtracking involves complex computations. An OPEN list in depth-first search is usually implemented with a stack. When a node is expanded, its children are placed at the top of the stack. In complex situations, where the number of possible nodes is large, a *depth bound* may be imposed to force backtracking. This forces the search tree to be spread out rather than exploring a single path in the search.

Figure 8.3 illustrates a depth-first search. The numbers in the nodes indicate the order in which the nodes in the search tree are expanded. Procedure 8.2 in Appendix D (Depth-First Search) implements depth-first search. Depth-first search is good for finding a solution quickly, although it may not be minimal. However, if a number of deep paths in a tree must be traveled which initially bypass a solution which appears at a higher level in the search tree, then depth-first search would be very inefficient. Depth-first search is often used to quickly find an initial solution.

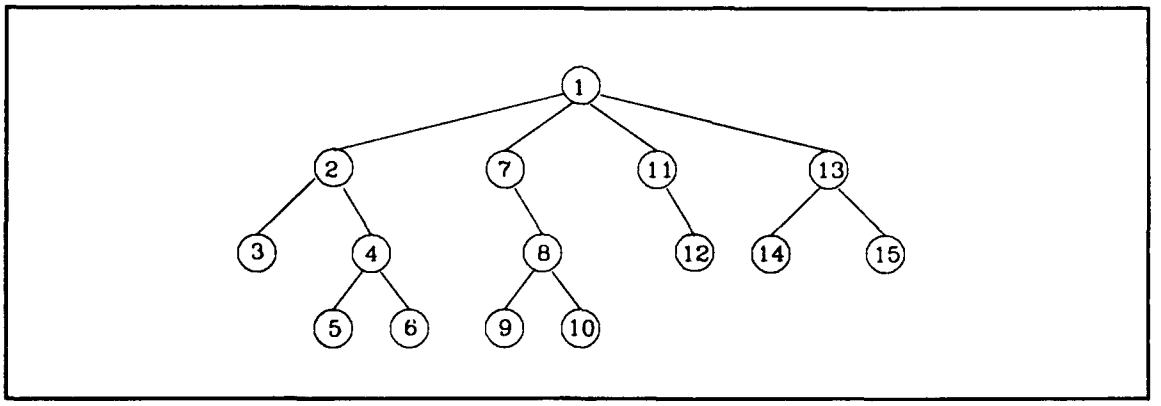


Figure 8.3. Illustration of Depth-First Search

Blind AND/OR Graph Search. Blind search strategies exist for solutions of problems represented by AND/OR graphs. AND/OR graphs are used in problem-reduction representations. Rather than finding a solution path through a single node, a solution path must be formed through each of the AND nodes since all subproblems represented by the AND nodes must be solved to form a solution through the corresponding parent node. The methods for the uninformed search of AND/OR graphs are adaptations of the methods used in search trees. For example, there are depth-first and breadth-first search algorithms for AND/OR graphs. The difference between searching the AND/OR graph and an OR graph is that instead of testing a single node for termination, a set of nodes must be tested.

Uninformed Search Techniques.

Branch-and-Bound Search. An algorithm which generalizes the breadth-first search procedure to ensure that a minimal solution is found is called *branch-and-bound* search. We again define an evaluation function $f(n)$ as being equal to a function $g(n)$. However, rather than $g(n)$ being equal to the number of arcs between the root node and n, we define it to be equal to the cost of a path between the root node and n. Given a nonnegative cost associated with each arc, the cost of a path between the root node and n is the sum of the arc-costs along the path. Note that

the measure of the cost between the start node and nodes in the search tree does not involve the use of a heuristic function $h(n)$; hence, branch-and-bound search is not a heuristic search.

A sorted list is used to form the OPEN list in branch-and-bound search. When a node is expanded, each of its successor nodes is evaluated using $g(n)$ to determine the cost of the path between the root node and a successor node. The successor nodes are inserted into the OPEN list in a manner that maintains that the ordering of the OPEN list, such that nodes appear in the OPEN list in ascending order of cost. Nodes are expanded such that the least cost path from the start state to the node to be expanded is taken, i.e., a partial path with the least cost is always extended.

Other names for branch-and-bound search are *cheapest-cost* and *uniform-cost* search. Branch-and-bound search is generally more efficient than breadth-first search. However, the use of this search may again be impractical for complex problems. Procedure 8.3 (Branch-and-Bound Search) in Appendix D implements branch-and-bound search.

Greedy Method. An algorithm which is an irrevocable variation of branch-and-bound search is *greedy* search. In greedy search a function $g(n)$ is used to determine the cost of a path between the root node and n. Initially, the root node is expanded to generate all of its children. Each child is then evaluated to determine the associated cost $g(n)$. After each child is evaluated, all but the node with the cheapest associated cost are discarded. The process then continues iteratively until either a solution is found or no more children may be generated. Hence, greedy search is not guaranteed to find a solution, and if a solution is found, it may not be minimal. Like depth-first search, greedy search may be useful for quickly finding an initial problem solution. Procedure 8.4 (Greedy Method) in Appendix D implements greedy search.

Informed Search Techniques.

Hill-Climbing. *Hill-climbing* is a variation of depth-first search which uses the local knowledge of a given node n to form a heuristic function $h(n)$ to estimate the distance between n and a goal node. The function $h(n)$ is used to select a node to expand based on what appears to be the node which is the shortest distance from a solution. In hill-climbing, the root node is expanded to generate all of its children. Each child is then evaluated to determine the node which is estimated to be closest to a solution. After each child is evaluated, all but the node which is estimated to be the closest to a solution are discarded. The process continues iteratively until either a solution is found or no more children may be generated. Hence, hill-climbing is not guaranteed to find a solution. Additionally, if a solution is found, it may not be minimal.

Similar to the greedy method, hill-climbing is an irrevocable strategy since much of the search space is eliminated from consideration in the course of the search. One problem with hill-climbing is that we may come to a local minimum—called the *foothill* problem—which either may appear as the minimum solution when in fact it may be sub-minimal. A second problem is called the *plateau* problem in which a "best" direction cannot be determined. Hence, we may be at a dead-end at which there exists no solution. In addition, a third problem is called the *ridge* problem in which we reach a state from which we cannot move in a single step. A revocable modification of the hill-climbing approach allows backtracking.

The hill-climbing search strategy is implemented by Procedure 8.5 (Hill-Climbing) in Appendix D. Similar to the greedy method, an application of hill-climbing is to quickly develop an initial solution to a problem.

Best-First Search. *Best-first* search is implemented in a fashion similar to branch-and-bound search. However, the difference between the two strategies is that the function $g(n)$ is used to evaluate each node in branch-and-bound search, whereas $h(n)$ is used to evaluate each

node in best-first search. All nodes which are generated are placed in the OPEN list. The nodes in the OPEN list are ordered so that the nodes appear in the OPEN list in ascending order of the value of $h(n)$. In a sense, best-first search is similar to hill-climbing in that the best apparent path is always taken. The difference is that when we choose to expand a node and we evaluate all of its children with $h(n)$, we do not necessarily pick the best child as the next node to expand as in hill-climbing. Rather, we then look at the entire search tree and examine all nodes that have been evaluated and pick the global best. Hence, we often will jump around the search tree choosing the next node to expand.

Best-first search will generally yield a good solution, but not necessarily a minimal one. Variants of the best-first search exist that will guarantee a minimal solution. The power of the best-first search is dependent on the heuristic function. Procedure 8.6 (Best-First Search) in Appendix D implements best-first search.

Beam Search. A modification of breadth-first search in which the search progresses level-by-level is called *beam search*. However, instead of expanding all nodes at a level in the search tree, only the "best" w nodes are expanded. The pre-defined constant w is called the *width* of the search. After children are generated at each level in the tree, an evaluation function $f(n)$ is used to determine the best w nodes. The best w nodes are kept, and the remaining nodes are discarded for further consideration. Thus, beam search is irrevocable. This process continues until a solution is reached or until no children are produced. Given that b is the branching factor of the search tree, the average number of nodes that must be generated at each level of the search tree is $w \cdot b$.

The evaluation function $f(n)$ may incorporate the use of the actual cost between the root node and a node n, i.e., $g(n)$, as well as an estimate $h(n)$ of the distance between n and a goal node. A variation of beam search is to maintain an OPEN list in which a single node is expanded at each point in the search, its children are inserted into the sorted OPEN list, and all but the first w nodes are deleted from the list prior to the next node expansion.

Beam search may be used when memory resources available to the search process are limited. Additionally, it generally produces a good solution rather quickly; beam search is often used when performing approximate minimization. However, since pruning occurs throughout the search process, a solution—minimal or non-minimal—may not be developed. Beam search is implemented by Procedure 8.7 (Beam Search) in Appendix D.

A* Search. The *A* search* algorithm is perhaps the most widely used—as well as the most extensively studied—search strategy used in the field of artificial intelligence. The A* search algorithm uses an evaluation function $f(n)$ to determine which node to expand. The evaluation function $f(n)$ in the A* algorithm comprises two components which yield an estimate of the shortest path between the start node and a goal node. The first component, $g(n)$, is a measure of the exact cost of the path between the start node and node n. The second component is an estimate of the distance between node n and a goal node; this component is the heuristic function $h(n)$. The total evaluation function $f(n)$ for the A* algorithm is defined by the equation

$$f(n) = g(n) + h(n) \qquad \forall n. \tag{8.5}$$

If the function $h(n)$ is admissible, then the A* algorithm is guaranteed to return a minimal solution.

The A* algorithm is implemented in the same fashion as the best-first algorithm with the exception that the evaluation function $f(n)$ is used rather than $h(n)$ alone. The A* algorithm is actually a general algorithm. By modifying the evaluation function it degenerates into other searches. For example, by setting $g(n)$ equal to zero the A* search becomes best-first search. If $h(n)$ is set equal to zero, then A* degenerates to a branch-and-bound search. Procedure 8.8 (A* Search) in Appendix D implements A* search.

The properties of the heuristic function $h(n)$ that may be used in the A* search have been extensively studied to determine the effect that a given function $h(n)$ may have on the A* algorithm.

The advantage of admissible heuristic functions $h(n)$ is that they guarantee that a solution path of minimal cost will be found if a solution path exists. The advantage of a consistent heuristic function $h(n)$ is that an A* search will proceed with a minimal number of node expansions. In an ideal situation we would like to find a minimal cost path. However, in practicality we often must relax our desire for a minimal cost solution and settle for some sub-minimal solution. If the heuristic function $h(n)$ is computationally expensive, the cost of performing an A* search that yields a minimal solution may be prohibitive. It may be more important to find a sub-minimal solution in a search of reasonable length. In some cases when an admissible $h(n)$ is used, the combinatorics of a problem will prohibit the search from terminating. An example in which an admissible $h(n)$ causes problems is when there are several roughly equal candidates to choose from at each stage of the search. A disproportionate amount of time will be spent on determining a minimal path.

Sometimes we would rather that a heuristic function $h(n)$ give a very accurate estimate of $h^*(n)$ for a node n in most cases, even if the actual distance to a goal is occasionally overestimated. In this case, we would have an inadmissible heuristic function and algorithm. Assuming that a heuristic function $h_1(n)$ generally gives a good estimate of $h^*(n)$—although it may at times overestimate—it may be preferable to a function $h_2(n)$ which does not overestimate but does not give as good an estimate of $h^*(n)$. Using $h_1(n)$ rather than $h_2(n)$ will generally result in fewer nodes being expanded and thus generated. Two benefits are derived from this occurrence. First, because fewer nodes are then stored on the OPEN list, the stack space used by the search process will be greatly reduced. Hence, less memory is required. Additionally, because fewer nodes are process, the speed of the search will be greatly increased. The benefit of decreasing memory usage and increasing speed comes at the expense of the guarantee that a least-cost solution will be produced.

A useful theorem regarding inadmissible heuristic functions is called the *Graceful Decay of Admissibility* (Rich 91:79). Let $h^*(n)$ be the actual cost of the shortest path between a node n and a goal node. Then the Graceful Decay of Admissibility is stated as follows:

If $h(n)$ rarely overestimates $h^*(n)$ by more than δ, then the A* algorithm will rarely find a solution whose cost is more than δ greater than the cost of the optimal solution.

Hence, if we use an $h(n)$ which overestimates $h^*(n)$ by a known bound, then the A* algorithm is useful in developing a solution which is within the bound. In this case, the A* algorithm would be useful for near-minimisation.

Another consideration regarding the heuristic function is that it may be difficult to derive a good admissible $h(n)$. If the heuristic function is poor and admissible, e.g., approaches 0, it may turn out that an A* search will degenerate into an uninformed, i.e., branch-and-bound, search. This is a situation that we would like to avoid in many problems. Hence, rather than settle for a bad admissible heuristic, it may be desirable to use an inadmissible heuristic function.

Dynamic Weighting. The use of an inadmissible heuristic function $h(n)$ which is within some bound of $h^*(n)$ has proven useful for many problems. In particular, a slight overestimate of the actual $h^*(n)$ may be useful high in a search tree in order to discriminate early among paths which appear to be equally good. The utility of this approach has been applied to develop a search strategy called *dynamic weighting*.

In this strategy, the evaluation function $f(n)$ is defined by the equation

$$f(n) = g(n) + h(n) + \epsilon \cdot [1 - d(n)/N] \cdot h(n), \tag{8.6}$$

for which

- $g(n)$ and $h(n)$ are defined as usual,
- ϵ is a pre-defined constant,
- $d(n)$ is the depth of node n, and
- N is the anticipated depth of the goal node.

Early in the search tree, the heuristic function $h(n)$ is thus weighted by a factor that is close to $(1+\epsilon)$; hence, the evaluation function $f(n)$ is close to $g(n)+(1+\epsilon)h(n)$. As the search proceeds to deeper levels in the tree, the evaluation function $f(n)$ approaches $g(n)+h(n)$. Thus, depth-first traversals of the search tree are encouraged early in the search tree; A*-type searches occur deep in the tree. If the heuristic function $h(n)$ is admissible, the dynamic-weighting algorithm is said to be ϵ-admissible, i.e., the solution found by dynamic weighting is guaranteed to be at most a factor of $(1+\epsilon)$ over a minimal solution. Thus, dynamic weighting is useful for near-minimization, i.e., when we desire a solution guaranteed to be within some bound of a minimal solution. An implementation of dynamic weighting is given by Procedure 8.9 (Dynamic Weighting) in Appendix D.

Static Weighting. Whereas the dynamic-weighting method decreases the weighting of $h(n)$ as the search proceeds to lower level in the search tree, it is useful in some problems to weight $h(n)$ by the same factor throughout the search. We call such an approach *static weighting*. Let W be a constant which may be greater than or equal to zero. (In most cases W is fixed at a value greater than one.) Then the evaluation function $f(n)$ is defined by the equation[2]

$$f(n) = g(n) + W \cdot h(n). \tag{8.7}$$

The static weighting strategy generally yields a solution which is close to a minimal solution. However, since we cannot guarantee that a resulting solution is within a defined bound of a minimal solution, static weighting is used for approximate-minimization. The static weighting method is implemented by Procedure 8.10 (Static Weighting) in Appendix D.

AO* Search. The AO* algorithm is the analog to the A* algorithm for AND/OR graphs. Because this algorithm must deal with AND/OR graphs rather than ordinary OR graphs it is more complicated than the A* algorithm. An evaluation function f is also used in this

[2] In the literature, the evaluation function is sometimes defined as $f(n) = (1-W)g(n) + Wh(n)$, in which W is a number which may range from 0 to 1. The choice is strictly arbitrary.

algorithm, but it is based only on the distance between a node n and a set of goal nodes—the cost of going from the start node to a given node is not used. However, it is more complicated because in the case of an AND branch, the cost of going from a node to a set of solutions must be estimated. Differing from the A* algorithm, as successor nodes are explored in the graph, the values of the heuristic function are propagated upward in the graph. During the upward propagation, the values of the heuristic functions for ancestor nodes are recalculated. After all calculations are complete, the best current path is selected for search. The AO* algorithm is guaranteed to derive a minimal solution to a problem.

Comparison of Search Strategies. A number of metrics allow us to compare the relative worth of different search strategies for a specific problem. The metrics that we will use are:

- the number of children generated;
- the number of nodes expanded;
- the number of nodes on the OPEN list when a solution is found; and
- the proximity of the developed solution to a minimal solution.

The first three metrics give indications of the amount of work and memory space required to develop a minimal problem solution. The last criterion measures the "goodness" of the developed solution.

In some situations, we may use one search to develop an initial solution which serves as an ***upper bound*** for a minimal solution. Hence, after a node n is evaluated, n is discarded from future consideration if the value of $f(n)$ is greater than the upper bound. A metric which gives an indication of the utility of the method used to develop the initial upper-bound as well as the heuristic function $h(n)$ used in the primary search is the number of nodes which are found to be greater than or equal to the upper bound.

We will use these metrics to compare search strategies used to develop minimal digital designs.

Summary

In this chapter, we have presented background material on the concepts of search for those who may be unfamiliar with such techniques. Topics discussed included problem representation formats, the use of heuristics in search, and search strategies.

IX. The Search Process in Minimization

In our process for forming a minimal vector $\underline{F}$ of formulas corresponding to a design, reduction rules are used to reduce the set of inclusion formulas representing coverage of the terms of the base by prime implicants.[1] In many cases, however, all of the prime implicants required to constitute formulas in $\underline{F}$ will not be identified using the reduction rules; a set of inclusion formulas remains after rule reduction which contains the information that is necessary to determine the remaining prime implicants which compose formulas in $\underline{F}$. A search process is used to determine these prime implicants from the inclusion formulas. In this chapter, we discuss our application of search for ascertaining the remaining prime implicants to constitute formulas in $\underline{F}$.

We begin the chapter by introducing a number of basic issues which must be considered when applying search to a problem. We use these issues as a framework for our discussion throughout the chapter. Generally, these issues concern the representation of information in the search process, the rules used to make decisions based on the given information, and the use of heuristics in the search process. An overview of the use of state spaces in logical design is presented in light of these issues. We introduce two state-space classifications for logical design.

Each aspect of the search process is addressed in this chapter. Specific topics include search trees, node representations, heuristic functions, and search strategies. Additionally, we present two problem-reduction techniques for decomposing the search problem; the use of these techniques greatly reduces the complexity of the resulting search process. If the A* search strategy is applied, the resulting vector $\underline{F}$ of formulas is guaranteed to be minimal. The results may only be near-minimal—but often are minimal—when other search strategies which we present are used.

Because of the manner in which inclusion formulas are formed for both single and multiple-output circuit specifications, the same search process may be used for the resulting inclusion for-

[1] In this chapter, we will use the term *prime implicant* to denote prime implicants as discussed in Chapter 6 as well as multiple-output prime implicants (MOPIs) as used in Chapter 7.

mulas. The search process returns a set of labels; each label is associated with a prime implicant. The only difference between the single-output and multiple-output cases is how we develop each formula F_j in $\underline{F}$ using the labels associated with each prime implicant.

Near the end of the chapter, we present a general search algorithm which may be applied to carry out the search process and construct $\underline{F}$. This algorithm ties together the components of the search process which are described throughout the chapter.

The Search Process in Logical Design

In her book *Artificial Intelligence* (Rich 83:56), Rich states five important issues which arise in formulating search processes for a given problem. These issues provide a framework which we will use to discuss the application of search to logical design. These issues are:

1. the direction in which to conduct the search;
2. the topology of the search process;
3. how each node will be represented;
4. selecting applicable rules; and
5. using a heuristic function to guide the search.

The fourth issue concerns the search strategy that we may use to guide a search; the use of a heuristic function $h(n)$ is the fifth issue. In the remainder of this section, we discuss state spaces in logical design with respect to these issues.

In many problems we reason forward from initial states to the goal states. In other problems, it may be possible—and advantageous—to reason backward from the goal states to the initial states. Finally, it is possible in some cases to reason simultaneously in both the forward and backward directions. Since our problem is to generate a minimal representation of a circuit specification, we do not know at the outset the resulting circuit design. Hence, we are forced to pursue a forward-reasoning strategy in which we proceed from an initial state to a yet-to-be-determined goal state.

This makes the circuit minimization problem a natural to be cast in a state-space representation. Our first issue in the formulation of a search process is concerned with the direction of the search process. In circuit minimisation we usually begin with some initial state and proceed to a goal state.

Our second and third criteria for discussing search processes—the topology of the search process and node representation—are issues that have to be examined when we formulate the state space for a problem. Two alternatives with respect to search topology are whether to represent the search problem as a search graph or as a search tree. In the general case, the search problem is represented as a graph, because a tree is a specialization of a graph. In either topology, we start with the initial state and generate intermediate states until we reach a suitable goal state. In order to apply rules (issue four) which facilitate movement from one state to another, we need to keep track of information which represents a given state and well as the utility of the state with respect to the problem. This information is used to form the node representation in a search graph. The content of the information is problem-dependent. However, there are two ways we can represent an arbitrary state at the node which incorporates the state in the search graph:

- represent the state explicitly by making the appropriate modifications to the initial state required to derive the current state, or
- represent the state implicitly by storing the changes that must be made to the initial state to derive the current state. (Rich 83:64)

From the point of view of our first three issues, we may discuss the form of state spaces that have been used in existing circuit design methods. State spaces used in logical design are typically representations of circuits or subcircuits, for which we present two different classifications:

- *circuit formation* state spaces, and
- *circuit transformation* state spaces.

Nearly all circuit design techniques fall into one of these two categories. We will describe each category in turn.

A circuit formation state space is one in which the state space represents the *formation* of a design. The root node contains no circuit; it contains information required to develop the circuit. Each arc in the circuit formation state space denotes the addition of a subcircuit to the circuit represented by the corresponding parent node. Thus, as the search process proceeds to lower levels in a search tree, the nodes represent partially-completed circuits. Leaves in the search tree correspond to completed circuits which meet the specification; however, only certain leaves represent minimal circuits. A circuit formation state space is generally used by techniques used to develop minimal circuit designs.

A circuit transformation state space represents the *transformation* of a design. The root node contains a completed circuit which meets the specification, although it is not minimal. Each arc in the state space denotes the modification of the circuit represented by the corresponding parent node. Thus, each node in the search space represents a complete design. Nodes lower in the tree contain circuits which have been modified in a manner which reduces their associated cost. Leaves in the search tree contain "solutions", i.e., circuits which cannot be modified in a manner which makes them simpler using the available transformation rules. A circuit transformation state space is generally used to develop near-minimal circuits. It is the approach used by local transformation systems for minimising circuit designs such as (Darri 81), (deGeu 85), and (Enomo 85).

The circuit formation state space is the approach we will use for applying search to the problem of developing a minimal vector $\underline{F}$ of formulas which correspond to a design. In both circuit formation and circuit transformation state spaces, two common heuristics are used to guide and limit the search effort:

- choosing circuits or subcircuits based on some measure of cost, and
- limiting the search to some "reasonable" set of circuits or subcircuits.

We use both of these heuristics in the application of search to the problem of developing a minimal $\underline{F}$.

Knowledge Representation

Given the information available at the outset of the search process, forward reasoning is used to reach a goal state from the start state. In this section we describe how information is represented while building a search tree during the search process. Two different search tree topologies are introduced. Additionally, the data stored in each node of the search tree is described. At the conclusion of this section, we will have addressed the first three of the five issues for formulating the search process for our problem.

Information at Outset of the Search Process. A search process is required if the application of rule reduction does not yield the complete set of prime implicants which constitute formulas in $\underline{F}$. In this event, there remains a set IF of inclusion formulas at the completion of rule reduction. The set IF contains the information which facilitates the selection of a prime implicants to form a minimal $\underline{F}$. Hence, IF is used by the search process to select the remaining prime implicants.

In addition to the set IF of inclusion formulas, the association list $LAB/COSTS$ must be used by the search process. $LAB/COSTS$ is a list consisting of a set of pairs in which a label P_i associated with a prime implicant p_i forms a pair with the cost c_i associated with p_i. Cost information is required by the search process to ensure that a least-cost set of prime implicants is selected for containment in formulas in $\underline{F}$.

In summary, the following information is available for use by the search process:

1. the set IF of inclusion formulas, and
2. the association list $LAB/COSTS$.

The search process yields a minimal set of labels associated with prime implicants to be placed in $\underline{F}$. A follow-up step to the search process uses these labels to construct each F_j in $\underline{F}$. The actual prime implicants are not used in any way during the search process.

Topologies. Using the set IF of inclusion formulas, we form two topologies to represent the state space. Both topologies depict the search space in the form of a search tree. The rationale for using a search tree rather than a search graph for this problem is first discussed. Each topology is then described.

Rationale for Using a Search Tree. A search problem is generally depicted by a search graph rather than a search tree. A search graph is normally used to reduce computational effort and memory usage caused by the occurrence of nodes with duplicate states which may occur in a search tree. However, if a newly-generated node n is compared to existing nodes in the OPEN and CLOSED lists to prevent duplication, this problem is overcome. If n is cheaper than a duplicate node n' in the OPEN list, then it replaces n' in the OPEN list. Otherwise, n is discarded. If n is a duplicate of a node n' in the CLOSED list and is cheaper than n', then n' is removed from the CLOSED list and n is placed in the OPEN list. Otherwise, n is discarded.

The CLOSED list contains only nodes which have been expanded. However, for some problems once a node is expanded it is rarely regenerated. (Using the A* algorithm, a node is never regenerated once expanded if a consistent heuristic function $h(n)$ is used.) We have observed based on experimentation with the search process in this research that a node is rarely regenerated once placed in the CLOSED list. Hence, a CLOSED list is not maintained. Once a node is expanded, it is simply discarded from consideration. In the rare case that it is regenerated, we simply treat it as if it never existed. This reduces computations as well as memory usage. First, newly-generated nodes do not have to be compared to nodes on a CLOSED list. Second, the memory is not used to maintain a CLOSED list. Hence, since we do not have to maintain a CLOSED list, a search tree topology is facilitated by simply comparing newly-generated nodes to nodes in the OPEN list.

Topology #1. In the first topology, the search space is depicted as a binary tree in which each node in the tree has exactly two children. For each node n, a prime implicant p_i is selected[2] to use in the formation of node n's children. The arcs leading to n's children correspond to

- the selection of p_i for containment in $\underline{F}$, and
- the removal of p_i from further consideration, respectively.

The information contained in n is modified in view of these decisions to form n's children. Figure 9.1 depicts Topology #1. This topology is similar to that used in a technique called the ***branch-and-bound method*** for selecting prime implicants from a cyclic prime implicant table (Murog 79:175). In the branch-and-bound method, a prime implicant p_i is first assumed to be contained in a minimal formula; the prime implicant table is then reduced accordingly using domination rules. If the reduced table is cyclic, the process is performed recursively. Then, after a solution is formed based on the selection of p_i, p_i is assumed not be contained in a minimal formula. The PI table is reduced, and the process is performed recursively until a solution is developed. After both solutions are developed, the least-cost solution is kept and the other is discarded.

Forming the search tree in this manner, the branching factor for every node is two. If k is the number of distinct labels in the set IF of inclusion formulas, then there are k associated prime implicants. Thus, the maximum depth of the search tree is k. Since the branching factor is two, the maximum number of goal nodes is 2^k. Hence, there are 2^k possible paths between the root node and possible goal nodes. Additionally, the maximum number of states in the search space is $2^{k+1} - 1$. Our goal in the search process is to quickly find a minimal path to a goal node while generating the fewest states possible of the states which exist in the search space.

[2]How p_i is selected will be discussed later.

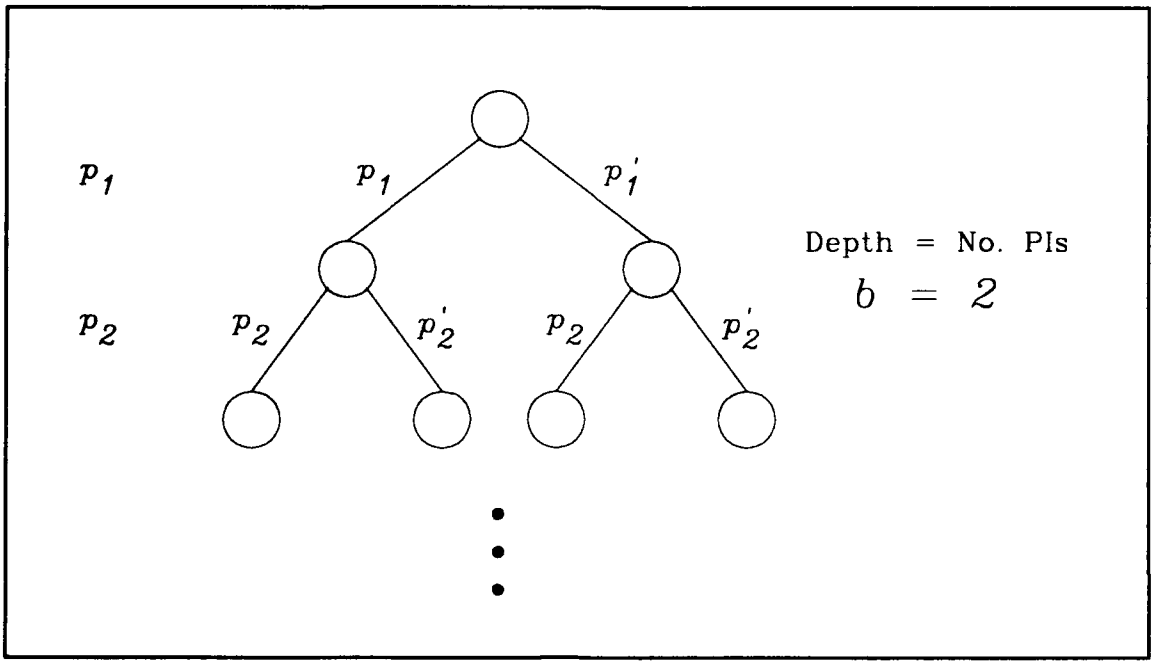


Figure 9.1. Topology #1 - Binary Search Tree

Topology #2. An inclusion formula IF_j in IF is selected for use in forming the children of a node in our second topology. Each term in a formula IF_j denotes combinations of prime implicants which cover a term t_j in the base. The set of terms in IF_j denotes the different possible ways that prime implicants may be used to cover t_j. Using the inclusion formula IF_j, we form children for a node n such that each arc leading to a child corresponds to the selection of prime implicants denoted by a term in IF_j. Suppose, for example, that the inclusion formula $P_1' + P_2'P_3' + P_2'P_4'$ is selected for use in forming node n's children. Then n would have three children; the first child is formed based on the selection of the prime implicant associated with P_1 for containment in $\underline{F}$, the second child is formed based on the selection of the prime implicants associated with P_2 and P_3 for containment in $\underline{F}$, and so on. Figure 9.2 depicts a search tree based on Topology #2.

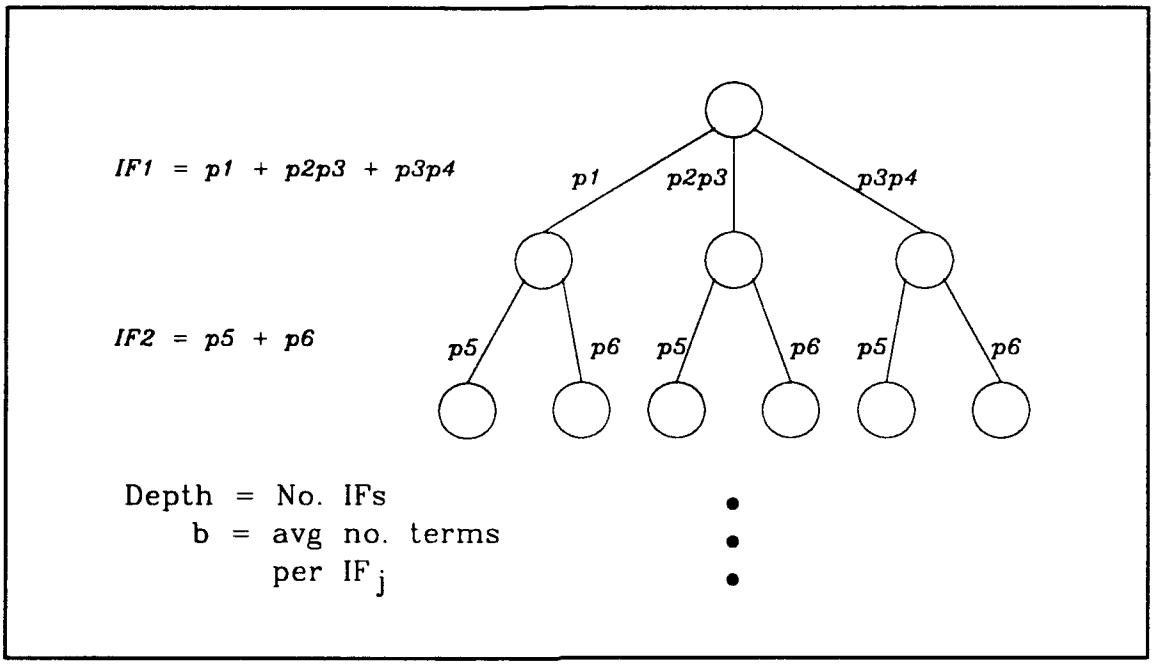


Figure 9.2. Topology #2 - Search Tree

Let m be the number of inclusion formulas in IF. The maximum depth of the search tree is m. Moreover, let b be the average number of terms in inclusion formulas in IF; then b is the branching factor for the search tree. An approximation of the number of goal nodes is b^m. The number of states in the search space is approximated by

$$\frac{b^{m+1}-1}{b-1}. \tag{9.1}$$

For example, suppose the maximum depth and branching factor are $m = 3$ and $b = 3$, respectively. Then, counting the root node as the first level, the number of nodes at each level of the tree is 1, 3, 9, and 27. The number of goal nodes is the number of nodes at the lowest level, i.e., $3^3 = 27$. Moreover, the number of states is calculated to be $(3^{3+1}-1)/(3-1) = 40$, which is in fact the sum of the nodes at each level of the tree.

This topology is useful if the average number of terms in the inclusion formulas is relatively low. If the average number of terms is high, then the branching factor of the search tree will be high. Hence, the search space will be large. Since the figures used to calculate the number of states in the state space are easily obtained from the inclusion formulas, we can calculate the projected number of states for each topology prior to the search process. Based on a heuristic that the smaller projected search space will engender a reduction of effort in the search process, we then may use the topology for which the projected search space is smaller.

Node Representation. Now that we have presented the forms of the search tree, we discuss the nodes of the search tree. We describe the information that is stored in each node of the tree as well as the processes of node generation and expansion. Two ways of representing state information in a node are introduced.

Information Contained in a Node. The set IF of inclusion formulas is the information given at the outset of the search process. The root node may be viewed as being associated with the set IF of inclusion formulas, i.e., IF is the initial state in the state space. Each node n in the search tree either contains a modified set $\widehat{IF}$ of inclusion formulas based on choices made to get to n from the root node or contains information which facilitates the generation of $\widehat{IF}$ from IF. Thus, the modified set $\widehat{IF}$ of inclusion formula is the state associated with a node n. A goal node contains a state such that the set of inclusion formulas is the empty set, i.e., no further decisions have to be made.

In addition to containing a state, a node n contains information about the path taken from the root node to get to n. The path must be stored in a node in our problem, since a CLOSED list is not maintained in the process. In our topologies, arcs leading to a node either denote the selection of prime implicants for containment in $\underline{F}$, or—in the case of Topology #1—denote the removal of a prime implicant from further consideration. Hence, n contains a representation of the arcs between the root node and n.

Finally, when a node n is generated, the utility of n in the search space must be evaluated so that n is handled as appropriate for the search strategy being used. An evaluation function $f(n)$ is used to evaluate the utility of n. Depending on the search strategy, $f(n)$ may comprise one or both of the functions $g(n)$ and $h(n)$. The function $g(n)$ evaluates the cost of the path between the root node and n. The heuristic function $h(n)$ uses $\widehat{IF}$ to evaluate n's proximity to a goal node. The values of $f(n)$ and $g(n)$ are stored within n.

In summary, a given node n in the search tree contains the following information:

1. a set $\widehat{IF}$ of inclusion formulas,
2. a representation of the path between the root node and n,
3. the value of $f(n)$, and
4. the value of $g(n)$.

We now discuss how this information is used to generate children for n.

Generation of a Node. For a node n, arcs leading to a child n' of n denote either the selection of prime implicants for containment in $\underline{F}$ or the removal of a prime implicant from further consideration. Let $a(n, n')$ denote the arc between n and n'. To form a node n', information contained in n is modified in view of the meaning of $a(n, n')$, i.e., whether $a(n, n')$ represents the selection or deletion of prime implicants. We first discuss the generation of a node n' when $a(n, n')$ denotes the selection of prime implicants for containment in $\underline{F}$.

To create a new state associated with node n', prime implicants denoted by $a(n, n')$ are used to modify the set IF of inclusion formulas associated with n, thus forming a revised set $\widehat{IF}$ of inclusion formulas. Once prime implicants are selected for containment in $\underline{F}$, they are treated as secondary essential prime implicants. In the reduction rules presented in Chapter 6, when a prime implicant p_i is secondary essential, the literal P_i' associated with p_i is removed from all terms in the set IF of inclusion formulas. Procedure 6.6 (Removal of Literals) may be used to carry out

this process. After all literals associated with prime implicants denoted by $a(n, n')$ are removed from terms of the inclusion formulas in n, reduction rules are applied to develop the set of inclusion formulas associated with n'.

When applying reduction rules, the same set of rules as applied prior to the search process must be applied throughout the search process. For example, if the original base used to form the set of inclusion formulas was the set of all conditionally-eliminable prime implicants, then Reduction Rule Set #1 (Procedure 6.11) is used to reduce the resulting inclusion formulas. Reduction Rule Set #1 is then used throughout the search process. Likewise, if Reduction Rule Set #2 (Procedure 6.14) was used prior to the search process, then Reduction Rule Set #2 is used throughout the search process.

In addition to developing a set $\widehat{IF}$ of inclusion formulas associated with n', the path between the root node and n' is constructed. The path denotes prime implicants selected as well as removed from consideration between the root node and n'. Since node n contains a path between the root node and n, we simply append the prime implicants denoted by $a(n, n')$ to n's path. The new path is stored in n'.

After forming the state associated with n', i.e., the set of inclusion formulas associated with n', and the path between the root node and n', an evaluation function $f(n)$ is used to determine the utility of n'. The composition of $f(n)$ is dependent on the search strategy; the functions $g(n)$ and $h(n)$ may be used to form $f(n)$. If the function $g(n)$ is used to form $f(n)$, then the value of $g(n)$ is stored in each node n. The function $g(n)$ is the cost of the prime implicants denoted by the path between the root node and n. To form the value of $g(n')$, the cost of the prime implicants denoted by $a(n, n')$ is added to value of $g(n)$. (The value of $g(n)$ is zero for the root node.)

For search strategies which require the use of a heuristic function $h(n)$, the set $\widehat{IF}$ of inclusion formulas associated with n' is evaluated using the heuristic function. The heuristic functions used in this work are discussed in a later section of this chapter. After applying the heuristic function

to form the value of $h(n')$, the evaluation function $f(n')$ is formed based on $g(n')$ and $h(n')$. The value of $f(n')$ is stored in each node n' for use by the search strategy.

Procedure 9.1 (Generation of a Node) is used to form a node n' in which $a(n, n')$ denotes the selection of prime implicants for containment in formulas in $\underline{F}$.

Procedure 9.1 (Generation of a Node): Given a node n, an arc $a(n, n')$, an association list $LAB/COSTS$, a heuristic function $h(n)$, and an evaluation function $f(n)$, a node n' is constructed in the following manner:

Step 1. Let IF be the set of inclusion formulas associated with node n.

1. Determine the set P of labels P_i associated with prime implicants denoted by the arc $a(n, n')$.
2. For each label P_i in P, use Procedure 6.6 (Removal of Literals) to remove each literal P_i' from terms in IF.
3. Let $\widehat{IF}_{temp}$ be the resulting set of inclusion formulas.

Step 2.

1. Use either Reduction Rule Set #1 (Procedure 6.11) or Reduction Rule Set #2 (Procedure 6.14) to reduce the set $\widehat{IF}_{temp}$ of inclusion formulas. (The choice is dependent on the set of rules used prior to the search process.)
2. The resulting set $\widehat{IF}$ of inclusion formulas forms the state associated with n'.

Step 3.

1. Determine the path stored in node n.
2. Form a new path by appending the labels associated with prime implicants denoted by $a(n, n')$ to n's path.
3. Store the new path in n'.

Step 4. For search strategies in which $g(n)$ is used to form $f(n)$:

1. Determine the value $g(n)$ stored in node n.
2. Using the labels associated with prime implicants denoted by $a(n, n')$ and the association list $LAB/COSTS$, calculate the sum of the costs of the prime implicants denoted by $a(n, n')$.
3. Add the value determined in step 2 to $g(n)$; it is the value $g(n')$.
4. Store the value of $g(n')$ in n'.

Step 5. For search strategies in which $h(n)$ is used to form $f(n)$, use $\widehat{IF}$ and $LAB/COSTS$ to determine the value of $h(n')$.

Step 6.

1. Develop the value of $f(n)$ as required for the given the search strategy.
2. Store the value of $f(n')$ in n'.

Using Topology #1, an arc $a(n, n')$ may denote a prime implicant p_i to be removed from further consideration. In this case, generation of a node n' is handled differently than when an arc denotes the selection of prime implicants. Based on the removal of a prime implicant from consideration, the inclusion formulas IF associated with node n are modified to form a revised set $\widehat{IF}$ of inclusion formulas associated with n'. When a prime implicant is removed from consideration, it is treated as an inessential prime implicant. In the reduction rules presented in Chapter 6, when a prime implicant p_i is inessential, terms containing the literal P_i' associated with p_i are removed from each formula in the set IF. Procedure 6.5 (Deletion of Terms) may be used to perform this process. After all terms containing P_i' are removed from formulas in IF, reduction rules are applied to develop the set $\widehat{IF}$ associated with n'.

As in the previous case, the same reduction rules as used prior to the search process must be used during the search process. However, in this instance, the removal of a prime implicant p_i from consideration may cause a second prime implicant to become secondary essential. The reduction rules identify and return such prime implicants.

After developing a set of inclusion formulas associated with n', the path between the root node and n' is constructed. The path denotes prime implicants selected as well as removed from consideration between the root node and n'. In this case, we must have a mechanism for depicting that a prime implicant p_i is to be removed from consideration. Using this mechanism, we form the path between the root node and n' by appending the notation to n's path that p_i is to be removed from consideration. Additionally, if any prime implicants become secondary essential due to the deletion of p_i, then these prime implicants are also added to the path. The new path is then stored in n'.

After forming the set of inclusion formulas associated with n', an evaluation function $f(n)$ is used to determine the utility of node n'. The form of $f(n)$ is dependent on the search strategy. If the function $g(n)$ is to be used to derive $f(n)$, then we must determine the value of $g(n')$. Since $a(n, n')$ denotes the removal of a prime implicant from consideration, in most cases the value of $g(n')$ is equal to the value of $g(n)$, i.e., $a(n, n')$ does not depict the addition of a prime implicant to the path. However, if prime implicants become secondary essential due to the deletion of a prime implicant denoted by $a(n, n')$, then the cost of these prime implicants must be added to $g(n)$ to form the value of $g(n')$.

For search strategies which require the use of a heuristic function $h(n)$, the set $\widehat{IF}$ of inclusion formulas associated with n' is evaluated using the heuristic function. After applying the heuristic function to form the value of $h(n')$, the evaluation function $f(n')$ is formed based on $g(n')$ and $h(n')$. The value of $f(n')$ is stored in each node n' for use by the search strategy.

Procedure 9.2 (Generation of a Node) is used to form a node n' in which $a(n, n')$ denotes the removal of a prime implicant from consideration for containment in formulas in $\underline{F}$.

Procedure 9.2 (Generation of a Node): Given a node n, an arc $a(n, n')$, an association list *LAB/COSTS*, a heuristic function $h(n)$, and an evaluation function $f(n)$, a node n' is constructed in the following manner:

Step 1. Let IF be the set of inclusion formulas associated with node n.

1. Determine the label P_i associated with the prime implicant p_i denoted by the arc $a(n, n')$.
2. Use Procedure 6.5 (Deletion of Terms) to delete terms containing the literal P_i' from formulas in IF.
3. Let $\widehat{IF}_{temp}$ be the resulting set of inclusion formulas.

Step 2.

1. Use either Reduction Rule Set #1 (Procedure 6.11) or Reduction Rule Set #2 (Procedure 6.14) to reduce the set $\widehat{IF}_{temp}$ of inclusion formulas. (The choice is dependent on the set of rules used prior to the search process.)
2. The resulting set $\widehat{IF}$ of inclusion formulas is the state associated with n'.
3. Let P be the set of prime implicants identified as secondary essential prime implicants during rule reduction.

Step 3.

1. Determine the path stored in node n.
2. Form a new path by appending to n's path the denotation that the p_i, as denoted by $a(n, n')$, is to be removed from consideration.
3. If $P \neq \emptyset$, then one or more prime implicants became secondary essential due to the deletion of p_i. Add these prime implicants to the path.
4. Store the new path in n'.

Step 4. For search strategies in which $g(n)$ is used to form $f(n)$, determine the value $g(n)$ stored in node n.

- If $P = \emptyset$, then no prime implicants are secondary essential due to the deletion of p_i. Set $g(n')$ equal to $g(n)$.
- If $P \neq \emptyset$, then one or more prime implicants are secondary essential due to the deletion p_i.
  1. Using the labels associated with prime implicants in P and the association list $LAB/COSTS$, calculate the sum of the costs of the prime implicants in P.
  2. Add the resulting sum to $g(n)$; it is the value of $g(n')$.

Store the value of $g(n')$ in n'.

Step 5. For search strategies in which $h(n)$ is used to form $f(n)$, use $\widehat{IF}$ and $LAB/COSTS$ to determine the value of $h(n')$.

Step 6.

1. Develop the value of $f(n)$ as required for the given the search strategy.
2. Store the value of $f(n')$ in n'.

Explicit Versus Implicit Representation. In our problem, each state in the state space is a modified set $\widehat{IF}$ of inclusion formulas such that $\widehat{IF}$ reflects the decisions that have been made between the root node and the node associated with $\widehat{IF}$. An explicit node representation is formed by making modifications to the initial state required to derive the current state and storing the new state in the node associated with the state. Hence, in an explicit node representation, the node n' associated with the modified set $\widehat{IF}$ of inclusion formulas actually contains $\widehat{IF}$. On the other hand, in an implicit node representation, only the changes that must be made to the initial state to derive the current state are stored in the node associated with the state. In our case, the path stored in a node contains this information.

Whether an explicit or implicit node representation is used, the following information is stored in a node n:

1. a representation of the path between the root node and n,
2. the value of $f(n)$, and
3. the value of $g(n)$.

Additionally, in a explicit node representation, the state denoted by the revised set $\widehat{IF}$ of inclusion formulas is stored in n. We present the advantages and disadvantages of each approach.

In an explicit node representation, all of the information associated with a node n in the search space is contained in the node. Most importantly, the state denoted by a set $\widehat{IF}$ of inclusion formulas is contained in the node. This is advantageous when generating a node's children, because then we only need to modify $\widehat{IF}$ to develop the states associated with n's children. The disadvantage is that if the set of inclusion formulas is large, then the memory used to store each node of the state space may be extensive. Hence, an explicit node representation is computationally efficient at the cost of memory usage.

State information is not contained in an implicit node representation. However, when a node n is first generated, the inclusion formulas $\widehat{IF}$ associated with n must be developed in order to derive the value of $h(n)$ for n. The set $\widehat{IF}$ may then be discarded. When n is expanded, the set $\widehat{IF}$ must be regenerated using path information as well as the set IF of inclusion formulas associated with the root node. $\widehat{IF}$ is used to generate the state associated with each child n' of n in order to evaluate $h(n')$. The advantage of the implicit node representation is that memory usage is greatly reduced due to the fact that only the original set IF of inclusion formulas is stored in memory. The disadvantage of this approach is that the set $\widehat{IF}$ of inclusion formulas associated with a node n has to be derived once when n is generated and again when n is expanded.

The use of an explicit versus implicit node representation is dependent on whether memory is at a premium on the computer system on which we host an implementation of the search strategy.

If an implicit node representation is used, then the set $\widehat{IF}$ of inclusion formulas associated with a node n' is discarded after Step 5 in Procedures 9.1 and 9.2. $\widehat{IF}$ is then regenerated when n' is expanded.

Expansion of a Node. Given the procedures used to generate a node, we may now describe how a node n is expanded. The expansion process differs depending on the topology used in the search process. Specifically, the first step in the node expansion process—the formation of arcs $a(n, n')$ between n and each child n' of n—changes based on the topology. We first describe the methodology for expanding a node when Topology #1 is used.

If an explicit node representation scheme is being used, then the set $\widehat{IF}$ of inclusion formulas associated with n is stored in n. On the other hand, if an implicit node representation scheme is being used, then the set $\widehat{IF}$ of inclusion formulas associated with n must be derived. The path information stored in n and the set IF of inclusion formulas associated with the root node is used to form $\widehat{IF}$. Based on the path information, IF is modified to recreate each state associated with the nodes on the path between the root node and n, until $\widehat{IF}$ is eventually formed.

In Topology #1, each node n has two children. Arcs $a(n, n')$ leading to the two children denote the selection of a prime implicant p_i and the deletion of p_i, respectively. (We defer to a later section the discussion of how such a prime implicant p_i is determined.) Given p_i, we first form the child corresponding to the selection of p_i using Procedure 9.1 (Generation of a Node). Subsequently, the second child of n corresponding to the removal of p_i from consideration is formed using Procedure 9.2 (Generation of a Node). After the children are generated, the nodes are placed on the OPEN list as determined by the search strategy.

Procedure 9.3 (Expansion of a Node - Topology #1): Given a node n and a prime implicant p_i, node n is expanded in the following manner:

Step 1.

- If an explicit node representation scheme is being used, then the set $\widehat{IF}$ of inclusion formulas associated with n is stored in n.
- Otherwise, an implicit node representation scheme is being used. Using the path stored in n and the set IF of inclusion formulas associated with the root node, form $\widehat{IF}$.

Step 2. Form a child corresponding to the selection of p_i using Procedure 9.1 (Generation of a Node).

Step 3. Form a child corresponding to the removal of p_i using Procedure 9.2 (Generation of a Node).

Step 4. Return the children for placement on the OPEN list.

In the second topology, an inclusion formula IF_j is used to generate the children for a node n. (We defer to a later section the discussion of how such a formula IF_j is determined.) The number of children generated is dependent on the number of terms in IF_j, since each term in IF_j corresponds to the selection of one or more prime implicants for containment in $\underline{F}$. Hence, arcs $a(n, n')$ leading to the children denote the selection of prime implicants given by a term in IF_j. Thus, for each term in IF_j, a child n' corresponding to the selection of prime implicants is formed using Procedure 9.1 (Generation of a Node). After the children are generated, the nodes are placed on the OPEN list as determined by the search strategy.

Procedure 9.4 (Expansion of a Node - Topology #2): Given a node n and an inclusion formula IF_j, node n is expanded in the following manner:

Step 1.

- If an explicit node representation scheme is being used, then the set $\widehat{IF}$ of inclusion formulas associated with n is stored in n.
- If an implicit node representation scheme is being used, then the set $\widehat{IF}$ of inclusion formulas associated with n must be derived. Using the path stored in n and the set IF of inclusion formulas associated with the root node, form $\widehat{IF}$.

Step 2. For each term in IF_j, form a child corresponding to the selection of prime implicants denoted by the term using Procedure 9.1 (Generation of a Node).

Step 3. Return the children for placement on the OPEN list.

We have described how our problem may be represented using a search tree methodology as well as how nodes are generated and expanded in the course of a search process. We have yet to discuss heuristics used in the search process as well as the search strategies used for manipulating and making decisions about the nodes in the search tree; these aspects will be introduced in later sections.

The point of view taken up to this juncture has been that the search problem is represented as a state-space and handled accordingly. However, in the next section we introduce a method for decomposing the problem into a set of state-space searches. Hence, we handle the search process globally using a problem-reduction format.

Search Space Partitioning

In many cases there exists a set IF of inclusion formulas at the completion of rule reduction, signifying that the application of rule reduction does not yield the complete set of prime implicants which constitute formulas in $\underline{F}$. Thus, the use of a search process is required to determine the remaining prime implicants to form a minimal $\underline{F}$. As described in foregoing sections, the set IF contains the information used by the search process to select the remaining prime implicants. In most cases, IF is of a form such that it can be partitioned; a search process is then performed independently for each block of the partition. The results of each search process are combined to form the remaining prime implicants to be contained in $\underline{F}$.

Let $P_1' + P_2'P_3'$ and $P_4' + P_5'P_6'$ be the inclusion formulas contained in IF after rule reduction. Clearly, any choice we make regarding the selection of prime implicant denoted by the inclusion formula $P_1' + P_2'P_3'$ has no bearing on the choices made regarding prime implicants associated with the inclusion formula $P_4' + P_5'P_6'$, and vice versa. Hence, the decisions we make regarding the two formulas may be performed independently. When the choice of a prime implicant p_i has a bearing on the selection and/or non-selection of another prime implicant p_j, we say that p_i and p_j are *related* to each other. Otherwise, two prime implicants are *not related* to each other. Thus, given the inclusion formulas $P_1' + P_2'P_3'$ and $P_3' + P_7'P_8'$, the prime implicants denoted by the labels in the set $P = \{P_1, P_2, P_3, P_7, P_8\}$ are related to each other, since the literal P_3' appears in both formulas. Given a third formula $P_4' + P_5'P_6'$, the prime implicants denoted by P_4, P_5, and P_6 are not related to any prime implicants denoted by labels in P.

We thus partition a set IF of inclusion formulas such that:

1. all prime implicants denoted by inclusion formulas in a block of the partition are related to each other, and
2. no prime implicant in a block of the partition is related to any prime implicant denoted by inclusion formulas in another block.

We call a partition which may be formed after the application of reduction rules an *intrinsic* partition. The derivation and use of an intrinsic partition facilitates a natural decomposition of the search process into a set of independent tasks. This decomposition greatly reduces the computational complexity of the search process. The concept of intrinsic partitions is related to the partitioning which may be performed in the course of solving a prime implicant table, although we apply the concept to the partitioning of inclusion formulas. The fact that a prime implicant table may be partitioned has been known for over thirty years. A recent application of the partitioning of a prime implicant table is in the ESPRESSO-EXACT algorithm in which a reduced form of a prime implicant table is partitioned (Rudel 89).

Using an intrinsic partition, our global search process may be portrayed in the form of a problem-reduction representation. The first step of the search process is to decompose the initial set IF of inclusion formulas via an intrinsic partition. An intrinsic partition, as incorporated into our search process, is depicted in Figure 9.3. Each of the second-level nodes in Figure 9.3 corresponds to a root node for a state-space search as presented earlier. Once a state-space search is accomplished for each of the second-level nodes, the results are combined to form the global solution of the search process. Procedure 9.5 (Intrinsic Partition) accepts a set IF of inclusion formulas and returns an intrinsic partition of the inclusion formulas. Example 9.1 demonstrates an intrinsic partition of a set of inclusion formulas.

Procedure 9.5 (Intrinsic Partition): Given a set IF of inclusion formulas, an intrinsic partition of IF is formed as follows:

Step 0.

- Initialize an accumulator $IF_{part} = \emptyset$.
- Initialize an accumulator $IF_{block} = \emptyset$.

Step 1.

1. Remove an inclusion formula IF_j from IF and determine the set P of literals contained in terms in IF_j.
2. Place IF_j in IF_{block}.

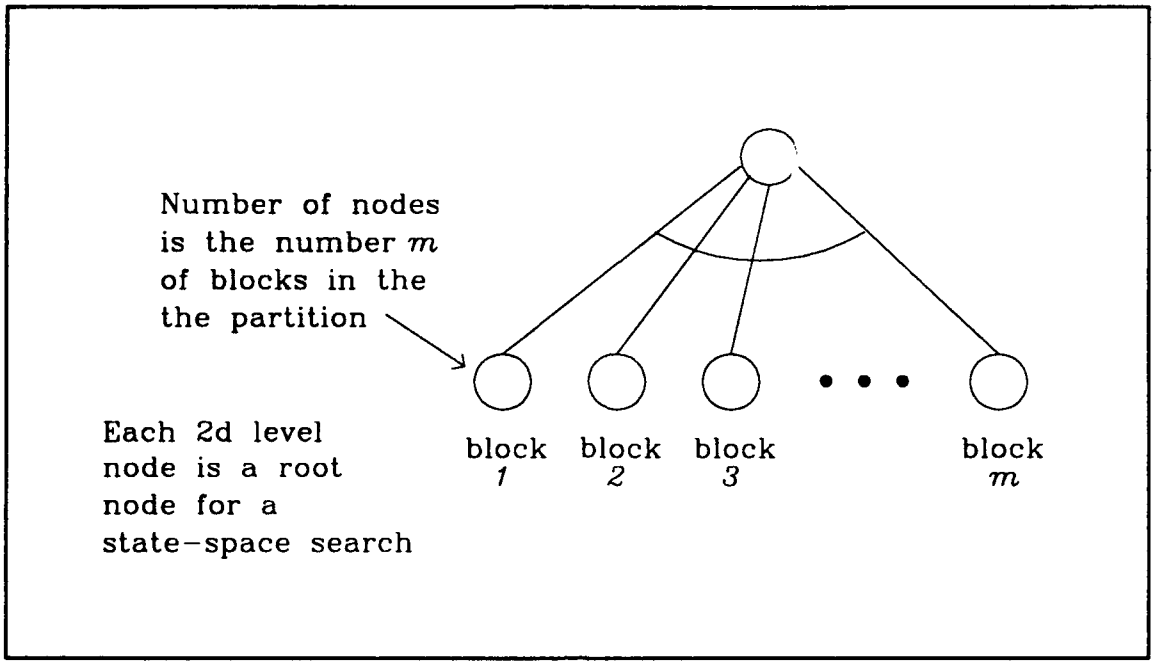


Figure 9.3. Problem-Reduction Using an Intrinsic Partition

Step 2. Of the remaining inclusion formulas in IF, determine the set $IF_{related}$ which contains literals appearing in P.

- If $IF_{related} = \emptyset$, then a block of the partition comprised of a subset of the original set IF of inclusion formulas has been formed.
  1. Place IF_{block} in IF_{part}.
  2. Reset the accumulator $IF_{block} = \emptyset$.
- Otherwise, we are not finished with forming the current block. Continue to Step 3.

Step 3.

1. Remove from IF formulas appearing in $IF_{related}$.
2. Place formulas in $IF_{related}$ in IF_{block}.
3. Add the literals appearing in formulas in $IF_{related}$ to P.
4. Return to Step 2.

Example 9.1: In the course of employing Algorithm 6.2 to develop a minimal formula F to represent the function B6, after the application of Reduction Rule Set #2 (Procedure 6.14) there remain 13 inclusion formulas. A search process must be applied to select a set of prime implicants to complete the formation of F using these inclusion formulas.

Applying Procedure 9.5 (Intrinsic Partition), we form a partition of the 13 formulas consisting of five blocks:

Block I:

$$P'_{28} + P'_{122}P'_{25}$$

Block II:

$$P'_{20} + P'_{121}P'_{9}$$

Block III:

$$P'_{126} + P'_{111}P'_{33} + P'_{33}P'_{91}$$
$$P'_{105} + P'_{88}P'_{97} + P'_{112}P'_{88}$$
$$P'_{91} + P'_{111}P'_{88} + P'_{126}P'_{88} + P'_{111}P'_{89} + P'_{126}P'_{89}$$

Block IV:

$$P'_{108} + P'_{60}P'_{86} + P'_{62}P'_{86}$$
$$P'_{63} + P'_{56}P'_{62} + P'_{59}P'_{62}$$
$$P'_{59} + P'_{56}P'_{57} + P'_{57}P'_{63}$$
$$P'_{117} + P'_{56}P'_{7} + P'_{50}P'_{56}$$

Block V:

$$P'_{127} + P'_{113}P'_{3} + P'_{113}P'_{6}$$
$$P'_{114} + P'_{106}P'_{113} + P'_{106}P'_{110}$$
$$P'_{109} + P'_{102}P'_{79}P'_{90} + P'_{102}P'_{110}P'_{79}$$
$$P'_{3} + P'_{2}P'_{6} + P'_{127}P'_{2}$$

Five independent search processes are performed using the inclusion formulas contained in each block. For many blocks, the identification of a prime implicant to be placed in F is trivial. For example, assuming that the prime implicants associated with each literal P'_i are of equal cost, the prime implicant associated with P_{28} is selected from Block I, and the prime implicant associated

with P_{20} is selected from Block II. A search process easily determines the prime implicants to place in F for the remaining blocks.

Once each block is formed in the partitioning process, a state-space search is performed in which the inclusion formulas contained in a block form the initial state associated with the root node of the search. In our discussion of the search tree topology and node representation, we deferred discussion of heuristic functions $h(n)$ and the selection of either prime implicants p_i or inclusion formulas IF_j corresponding to levels in the search tree for use during node expansion. These issues are addressed in the next section.

The Use of Heuristics

The use of heuristics—heuristic functions in particular—is the fifth issue for formulating a search process for solving a problem. In this section we present two heuristic functions used to evaluate the utility of a node. Additionally, we elaborate on heuristics used to guide the selection of a prime implicant p_i or an inclusion formula IF_j for use during node expansion. In each case, the choice is based on information developed in the course of forming the value of $h(n)$ for a set of inclusion formulas associated with a node n.

Heuristic Functions. In developing a heuristic function we are concerned with the inherent properties of the function. For example, the admissibility of a heuristic function is an important consideration. However, an admissible function $h_1(n)$ may not be as good as an inadmissible function $h_2(n)$ if

1. the value of $h_2(n)$ for a node n is generally closer to the actual distance $h^*(n)$ between n and a goal node than is the value of $h_1(n)$, and
2. $h_2(n)$ usually does not overestimate the value of $h^*(n)$.

In this section, we present two heuristic functions, $h_1(n)$ and $h_2(n)$, for estimating the distance between a node n and a goal node. The first $h_1(n)$ is an admissible function used to guarantee a minimal solution when the A* search strategy is used. The second function $h_2(n)$ is an inadmissible function which generally gives a better estimate of the actual value $h^*(n)$ than does $h_1(n)$.

Heuristic Function #1. The first heuristic function $h_1(n)$ we present is an admissible function, i.e., $h_1(n)$ always underestimates the actual cost of the cheapest path between n and a goal node. In our problem, $h_1(n)$ is used to estimate the value of a least-cost set of prime implicants which may be selected to form a cover for terms of the base associated with a set IF of inclusion formulas.

Let c_i be the cost associated with a prime implicant p_i. Given a set IF, let n_i be the number of inclusion formulas in IF in which the literal P_i' which denotes p_i appears. Then, we define the ***utility*** u_i of a prime implicant by the equation

$$u_i = c_i/n_i. \tag{9.2}$$

The utility u_i is the prorated cost of the prime implicant p_i based on a supposition that p_i may be used alone to form a cover of the terms associated with the inclusion formulas in which P_i' appears.

Once the utility of each prime implicant is determined, we derive a value v_j for each inclusion formula IF_j. First, a value v_k^t is formed for each term t_k in IF_j, in which v_k^t is defined to be the sum of the utilities of the prime implicants denoted by literals in t_k, i.e.,

$$v_k^t = \sum_{\{i|t_k \leq P_i'\}} u_i. \tag{9.3}$$

Then, IF_j is assigned the value associated with term t with the lowest value of all of the terms in IF_j. Thus, if there are l terms in IF_j, the value v_j is defined by the equation

$$v_j = \min(v_1^t, v_2^t, \ldots, v_l^t). \tag{9.4}$$

For example, suppose we are given an inclusion formula $IF_1 = P_1' + P_2'P_3'$ and the utilities $u_1 = 0.5$, $u_2 = 1$, and $u_3 = 0.5$. Then the value associated with term P_1' is 0.5; the value associated with the term $P_2'P_3'$ is $1 + 0.5 = 1.5$. It follows that the value v_1 of IF_1 is 0.5.

After v_j is formed for each IF_j in IF, the set of values for formulas in IF is summed to form a total value V associated with IF. The resulting value is an optimistic estimate of the cost of a cover for the terms associated with inclusion formulas in IF. In many cases, V may be represented by a fraction. Since a total cost is a whole number, we may safely round up V to the next highest whole number. The result is defined to be the value of $h_1(n)$ for the node with the associated state IF. Hence, if IF contains m inclusion formulas

$$h_1(n) = \left\lceil \sum_{j=1}^{m} v_j \right\rceil. \tag{9.5}$$

Examples 9.2 and 9.3 demonstrate the application of $h_1(n)$. Procedure 9.6 (Heuristic Function #1) implements $h_1(n)$.

Example 9.2: Suppose we are given the set $IF = \{IF_1, IF_2, IF_3\}$ of inclusion formulas for which we must evaluate $h_1(n)$:

$$\begin{aligned} IF_1 &= P_1' + P_2'P_3' \\ IF_2 &= P_2' + P_4'P_5' \\ IF_3 &= P_3' + P_4'P_6'. \end{aligned} \tag{9.6}$$

We first derive the utilities u_i for prime implicants p_i appearing in IF. The calculation of u_i is demonstrated using Table 9.1; we call the matrix appearing in Table 9.1 a ***utility matrix.*** A 1 appears in a cell if the label P_i' which denotes prime implicant p_i appears in inclusion formula IF_j.

	IF_1	IF_2	IF_3	c_i	n_i	$u_i = c_i/n_i$
p_1	1			1	1	1
p_2	1	1		1	2	0.5
p_3	1		1	1	2	0.5
p_4		1	1	1	2	0.5
p_5		1		1	1	1
p_6			1	1	1	1

Table 9.1. Utility Matrix for Example 9.2

After the development of the utilities u_i for each prime implicant p_i, the value v_j is determined for each inclusion formula IF_j. We first demonstrate the formation of v_1 for IF_1. The value v_k^t associated with each term is the sum of the utilities of the prime implicants denoted by literals in t_k. The values v_k^t associated with terms in IF_1 are depicted as follows:

$$IF_1 = \overbrace{P_1'}^{1} + \overbrace{P_2'P_3'}^{0.5+0.5}. \tag{9.7}$$

The value v_1 is the minimum of the values associated with terms in IF_1; hence $v_1 = 1$. Similarly, we demonstrate the values associated with terms in IF_2:

$$IF_2 = \overbrace{P_2'}^{0.5} + \overbrace{P_4'P_5'}^{0.5+1}. \tag{9.8}$$

Thus, the value $v_2 = 0.5$. Additionally, we can determine that $v_3 = 0.5$. Then, since $h_1(n) = \left\lceil \sum_{j=1}^{m} v_j \right\rceil$, we calculate $h_1(n) = \lceil 1 + 0.5 + 0.5 \rceil = 2$. In this instance, $h_1(n) = h^*(n)$, since a least-cost set of prime implicants which forms a cover for terms of the base associated with the inclusion formulas (9.6) is $\{p_2, p_3\}$.

Example 9.3: Suppose we are given the set $IF = \{IF_1, IF_2, IF_3\}$ of inclusion formulas for which we must develop a value $h_1(n)$:

$$\begin{aligned} IF_1 &= P_1' + P_3'P_5' \\ IF_2 &= P_2' + P_4'P_6' \\ IF_3 &= P_3' + P_1'P_4'. \end{aligned} \tag{9.9}$$

We first determine the utilities u_i for prime implicants p_i appearing in IF. The calculation of u_i is demonstrated using Table 9.2.

	IF_1	IF_2	IF_3	c_i	n_i	$u_i = c_i/n_i$
p_1	1		1	1	2	0.5
p_2		1		1	1	1
p_3	1		1	1	2	0.5
p_4		1	1	1	2	0.5
p_5	1			1	1	1
p_6		1		1	1	1

Table 9.2. Utility Matrix for Example 9.3

After the development of the utilities u_i for each prime implicant p_i, the value v_j is determined for each inclusion formula IF_j. We first form v_1 for IF_1. The value v_k^t associated with each term is the sum of the utilities of the prime implicants denoted by literals in t_k. The values v_k^t associated with terms in IF_1 are depicted as follows:

$$IF_1 = \overbrace{P_1'}^{0.5} + \overbrace{P_3'P_5'}^{0.5+1}. \tag{9.10}$$

Hence, $v_1 = 0.5$. Similarly, we demonstrate the values associated with terms in IF_2:

$$IF_2 = \overbrace{P_2'}^{1} + \overbrace{P_4'P_6'}^{0.5+1}. \tag{9.11}$$

Thus, the value $v_2 = 1$. Furthermore, we can derive $v_3 = 0.5$. Since $h_1(n) = \left\lceil \sum_{j=1}^{m} v_j \right\rceil$, we then calculate $h_1(n) = \lceil 0.5 + 1 + 0.5 \rceil = 2$. In this instance, $h_1(n)$ underestimates $h^*(n)$ since at least three prime implicants are required to form a cover, i.e, $h^*(n) = 3$.

Procedure 9.6 (Heuristic Function #1): Given a set IF of inclusion formulas and the association list $LAB/COSTS$, we derive the value $h_1(n)$ associated with the state IF in the following manner:

Step 1. Determine the set P of distinct labels which appear in IF.

Step 2. For each prime implicant p_i in which P_i appears in P:

1. Determine the number n_i of inclusion formulas in which the literal P_i' appears.
2. Get the cost c_i associated with p_i from $LAB/COSTS$.
3. Calculate the utility $u_i = c_i/n_i$.

Step 3. For each inclusion formula $IF_j \in IF$, determine v_j:

1. For each term t_k in IF_j, derive v_k^t by summing the utilities u_i of the prime implicants p_i denoted by literals P_i' in t_k, i.e.,

$$v_k^t = \sum_{\{i | t_k \leq P_i'\}} u_i. \tag{9.12}$$

2. Form v_j by selecting the lowest v_k^t of the l values developed in substep 1, i.e.,

$$v_j = \min(v_1^t, v_2^t, \ldots, v_l^t). \tag{9.13}$$

Step 4. Form $h_1(n)$ by adding the m values v_j and rounding up:

$$h_1(n) = \left\lceil \sum_{j=1}^{m} v_j \right\rceil. \tag{9.14}$$

An important consideration is that the heuristic function $h_1(n)$ implemented by Procedure 9.6 is admissible. An admissible heuristic function is required to guarantee the admissibility of the A* algorithm. If A* is admissible, then it returns a minimal solution. Theorem 9.1 presents the admissibility of $h_1(n)$.

Theorem 9.1 (Admissibility of $h_1(n)$): *The heuristic function $h_1(n)$ implemented by Procedure 9.6 is admissible.*

Proof. The utility u_i associated with each prime implicant p_i is the prorated cost of the prime implicant p_i based on a supposition that p_i may be used alone to form a cover of the terms associated

with the inclusion formulas in which P_i' appears. Thus, u_i is an optimistic proration of the cost associated with each p_i, i.e., we are forcing each prime implicant to appear to be more valuable than it actually is.

The value v_k^t associated with a term t_k of a formula IF_j is formed using utilities u_i associated with the prime implicants p_i denoted by literals P_i' appearing in t_k. Each v_k^t is the prorated cost of using the prime implicants denoted by literals in t_k to cover the term of the base associated with IF_j. Since we chose the value associated with the smallest v_k^t to form v_j, we are necessarily optimistically estimating the prorated cost of covering the term of the base associated with IF_j.

Since each value v_j is optimistic, it follows that the sum V of the values v_j associated with each $IF_j \in IF$ is an optimistic estimate of the actual cost of a minimal cover for the terms of the base associated with formulas in IF. Rounding V up to the next highest whole number to form $h_1(n)$ does not affect the admissibility of $h_1(n)$, since the actual cost is a whole number. This completes the proof. □

Heuristic Function #2. Function $h_1(n)$ is useful if we would like to guarantee the development of a minimal vector $\underline{F}$ of formulas which correspond to a design. However, it does not always yield a good estimate of $h^*(n)$, e.g., Example 9.3. Because $h_1(n)$ does not always yield a good estimate of $h^*(n)$, increased effort results during the search process. In this section, we introduce a second heuristic function $h_2(n)$ which generally is a very good estimator of $h^*(n)$, although in some cases it may overestimate $h^*(n)$.

The difference between $h_1(n)$ and $h_2(n)$ is the manner in which we form the utility u_i associated with each prime implicant p_i. To differentiate between the utilities used in the two heuristic function, let $\hat{u}_i$ be the utility of a prime implicant p_i for $h_2(n)$. Suppose we are given a set $IF = \{IF_1, IF_2\}$ of inclusion formulas in which

$$\begin{aligned} IF_1 &= P_1' + P_2'P_3' \\ IF_2 &= P_2' + P_1'P_3'. \end{aligned} \tag{9.15}$$

The utility matrix for IF is depicted by Table 9.3. Rather than placing a 1 in columns associated with an inclusion formula IF_j in which appears the literal P_i' which denotes p_i, we instead place the reciprocal of the shortest term in IF_j in which P_i' appears. Thus, since P_2' appears in a two-literal term in IF_1, it only does one-half the work in forming a cover $p_2 + p_3$ for the term of the base associated with IF_j. Then $\hat{n}_i$ is the sum of the number that appear in the columns associated with the inclusion formulas for row i. The utility $\hat{u}_i$ is defined by the equation $\hat{u}_i = c_i/\hat{n}_i$.

	IF_1	IF_2	c_i	$\hat{n}_i$	$\hat{u}_i = c_i/\hat{n}_i$
p_1	1	0.5	1	1.5	0.667
p_2	0.5	1	1	1.5	0.667
p_3	0.5	0.5	1	1	1

Table 9.3. Utility Matrix for Heuristic Function #2

We now present this methodology on a formal basis. Let us define an operator length(t) to be the number of literals in term t. Let t_k^j be the k-th term of inclusion formula IF_j. Then, n_i^j—the number in row i, column j of the utility matrix—is defined by the equation

$$n_i^j = \frac{1}{\min_{\{k|t_k^j \leq P_i'\}}(\text{ length}(t_k^j))}. \tag{9.16}$$

The metric $\hat{n}_i$ is defined by the equation

$$\hat{n}_i = \sum_{\{j|P_i' \text{ appears in } IF_j\}} n_i^j \tag{9.17}$$

The utility $\hat{u}_i$ for our second heuristic function is then defined by the equation

$$\hat{u}_i = c_i/\hat{n}_i. \tag{9.18}$$

Once we formulate the utility $\hat{u}_i$ for each prime implicant p_i, the remainder of the process of forming $h_2(n)$ is the same as when forming $h_1(n)$ except that $\hat{u}_i$ is used in place of u_i.

Example 9.4: Suppose we are given the set $IF = \{IF_1, IF_2, IF_3\}$ of inclusion formulas from Example 9.3 for which we would like to evaluate $h_2(n)$:

$$\begin{aligned} IF_1 &= P_1' + P_3'P_5' \\ IF_2 &= P_2' + P_4'P_6' \\ IF_3 &= P_3' + P_1'P_4'. \end{aligned} \tag{9.19}$$

We first determine the utilities $\hat{u}_i$ for prime implicants p_i appearing in IF. The calculation of $\hat{u}_i$ is depicted in Table 9.4.

	IF_1	IF_2	IF_3	c_i	$\hat{n}_i$	$\hat{u}_i = c_i/\hat{n}_i$
p_1	1		0.5	1	1.5	0.667
p_2		1		1	1	1
p_3	0.5		1	1	1.5	0.667
p_4		0.5	0.5	1	1	1
p_5	0.5			1	0.5	2
p_6		0.5		1	0.5	2

Table 9.4. Utility Matrix for Example 9.4

After the development of the utilities $\hat{u}_i$ for each prime implicant p_i, the value v_j is determined for each inclusion formula IF_j. We first form v_1 for IF_1. The value v_k^t associated with each term is the sum of the utilities of the prime implicants denoted by literals in t_k. The values v_k^t associated with terms in IF_1 are depicted as follows:

$$IF_1 = \overbrace{P_1'}^{0.667} + \overbrace{P_3'P_5'}^{0.667+2}. \tag{9.20}$$

The value v_1 is the minimum of the values associated with terms in IF_1; hence, $v_1 = 0.667$. Similarly, we demonstrate the values associated with terms in IF_2:

$$IF_2 = \overbrace{P_2'}^{1} + \overbrace{P_4'P_6'}^{1+2}. \tag{9.21}$$

In the case, the value $v_2 = 1$. We also can determine that $v_3 = 0.667$. Then, since $h_2(n) = \left\lceil \sum_{j=1}^{m} v_j \right\rceil$, we calculate $h_2(n) = \lceil 0.667 + 1 + 0.667 \rceil = \lceil 2.333 \rceil = 3$. The function $h_2(n)$ accurately estimates $h^*(n)$ since three prime implicants are required to form a cover, i.e, $h^*(n) = 3$.

A characteristic of function $h_2(n)$ is that it sometimes overestimates $h^*(n)$. Example 9.5 is a contrived example which illustrates this characteristic as observed when using $h_2(n)$ for more complex sets of inclusion formulas.

Example 9.5: Suppose we are given a set $IF = \{IF_1, IF_2, IF_3, IF_4\}$ of inclusion formulas for which we would like to evaluate $h_2(n)$:

$$\begin{aligned} IF_1 &= P_1' + P_2'P_3' \\ IF_2 &= P_2' + P_3'P_4' \\ IF_3 &= P_3' + P_1'P_2' \\ IF_4 &= P_4' + P_1'P_2'. \end{aligned} \tag{9.22}$$

We first form the utilities $\hat{u}_i$ for prime implicants p_i appearing in IF. The calculation of $\hat{u}_i$ is demonstrated using Table 9.5.

	IF_1	IF_2	IF_3	IF_4	c_i	$\hat{n}_i$	$\hat{u}_i = c_i/\hat{n}_i$
p_1	1		0.5	0.5	1	2	0.5
p_2	0.5	1	0.5	0.5	1	2.5	0.4
p_3	0.5	0.5	1		1	2	0.5
p_4		0.5		1	1	1.5	0.667

Table 9.5. Utility Matrix for Example 9.5

After the development of the utilities $\hat{u}_i$ for each prime implicant p_i, the value v_j is determined for each inclusion formula IF_j. We first form v_1 for IF_1. The values v_k^1 associated with terms in IF_1 are depicted as follows:

$$IF_1 = \overbrace{P_1'}^{0.5} + \overbrace{P_2'P_3'}^{0.4+0.5} . \tag{9.23}$$

The value v_1 is the minimum of the values associated with terms in IF_1; hence, $v_1 = 0.5$. The values associated with terms in IF_2 are demonstrated:

$$IF_2 = \overbrace{P_2'}^{0.4} + \overbrace{P_3'P_4'}^{0.5+0.667} . \tag{9.24}$$

In the case, the value $v_2 = 0.4$. The values associated with IF_3 and IF_4 are $v_3 = 0.5$ and $v_4 = 0.667$, respectively. Then, $h_2(n) = \lceil 0.5 + 0.4 + 0.5 + 0.667 \rceil = \lceil 2.0667 \rceil = 3$. The function $h_2(n)$ overestimates $h^*(n)$ since the only prime implicants p_1 and p_2 are required to form a cover, i.e, $h^*(n) = 2$.

In practical applications, heuristic function $h_2(n)$ rarely overestimates $h^*(n)$. Moreover, the instances when $h_2(n)$ overestimates do not seem to affect the quality of the resulting solution when A* search is used, i.e, the use of $h_2(n)$ almost always leads to the development of a minimal solution. We believe the function $h_2(n)$ is an example for which the Graceful Decay of Admissibility Theorem is applicable. Procedure 9.7 (Heuristic Function #2) implements the heuristic function $h_2(n)$.

Procedure 9.7 (Heuristic Function #2): Given a set IF of inclusion formulas and the association list $LAB/COSTS$, we derive the value $h_2(n)$ associated with the state IF in the following manner:

Step 1. Determine the set P of distinct labels which appear in IF.

Step 2. For each prime implicant p_i in which P_i appears in P:

1. Determine the value n_i^j for prime implicant p_i with respect to inclusion formula IF_j:

$$n_i^j = \frac{1}{\min_{\{k|t_k^j \leq P_i'\}}(\text{ length}(t_k^j))}. \tag{9.25}$$

2. Determine the value $\hat{n}_i$ associated with prime implicant p_i:

$$\hat{n}_i = \sum_{\{j|P_i' \text{ appears in } IF_j\}} n_i^j. \tag{9.26}$$

3. Get the cost c_i associated with p_i from $LAB/COSTS$.
4. Calculate the utility $\hat{u}_i = c_i/\hat{n}_i$.

Step 3. For each inclusion formula $IF_j \in IF$, determine v_j:

1. For each term t_k in IF_j, derive v_k^t by summing the utilities $\hat{u}_i$ of the prime implicants p_i denoted by literals P_i' in t_k, i.e.,

$$v_k^t = \sum_{\{i|t_k \leq P_i'\}} \hat{u}_i. \tag{9.27}$$

2. Form v_j by selecting the lowest v_k^t of the l values developed in step 1, i.e.,

$$v_j = \min(v_1^t, v_2^t, \ldots, v_l^t). \tag{9.28}$$

Step 4. Form $h_2(n)$ by adding the m values v_j and rounding up:

$$h_2(n) = \left\lceil \sum_{j=1}^{m} v_j \right\rceil. \tag{9.29}$$

Information developed in the course of forming the values of $h_1(n)$ and $h_2(n)$ is used to guide the selection of a prime implicant p_i or an inclusion formula IF_j during node expansion. We discuss this issue in the next section.

Heuristics in Formation of the Search Tree. In search space Topology #1, when a node is expanded its children are formed based on the selection of a prime implicant p_i for containment in $\underline{F}$ and the removal of p_i from further consideration. Using Topology #2, an inclusion formula

IF_j is used to form a node's children such that each child corresponds to the selection of prime implicants denoted by a term in IF_j. We discuss the selection of p_i and IF_j in this section. In each case, the choice is based on information developed in the course of forming the value of $h(n)$ for a set IF of inclusion formulas associated with a node n.

Choice of a PI for Topology #1. When forming the value of $h(n)$ associated with node n, a utility u_i is calculated for each prime implicant p_i denoted by literals P_i' which may appear in the set of inclusion formulas associated with n, i.e., the state associated with n. A heuristic which seems to work well is to select a prime implicant p_i to use in the expansion of n which has the lowest utility u_i. Since the utilities for each p_i are determined when n is generated, the p_i with the lowest utility may be stored in n for use when n is expanded. At this point, we do not have a method for selecting a single p_i when several have a minimal utility, i.e., we do not have a tiebreaker.

We have observed that the order of selection of prime implicants p_i in the course of node expansion can significantly affect the efficiency, i.e., the number of nodes expanded, of the search. For some examples, using an arbitrary p_i works better than our heuristic selection, although our heuristic selection of p_i works better than an arbitrary selection of p_i over a range of functions. More study is required for developing better heuristics to guide the selection of p_i.

Choice of an Inclusion Formula for Topology #2. When forming the value of $h(n)$ associated with node n, a value v_j is calculated for each inclusion formula IF_j in the set IF of inclusion formulas associated with n. A heuristic for selecting a formula IF_j for use during the expansion of n is to select the formula in IF which has the lowest value v_j. Similar to the prime implicant p_i for Topology #1, we determine the values v_j for each IF_j when n is generated. We store in n an indication of the inclusion formula IF_j to be used when n is expanded.

Search Strategies

The use of search strategies to control the search process is our fourth—and final—issue which must be addressed in formulating a search process for a given problem. The primary difference among search strategies lies the manner in which nodes are selected for expansion at a given point in the search process. The order of node expansion is dependent on the value of $f(n)$ associated with a node n as well as how this value is used to place n on the OPEN list. These factors differ from one search strategy to another, e.g., $f(n)$ may be comprised of $g(n)$ and $h(n)$ in one search strategy and only of $h(n)$ in another. Additionally, a search strategy may implement a heuristic which selects only certain nodes for inclusion in the OPEN list, while the rest are discarded, e.g., the strategy employed beam search.

In this section we discuss several search strategies which we apply to the problem of developing a minimal vector $\underline{F}$ of formulas corresponding to a minimal digital design. We present different search strategies depending on whether the goal in developing $\underline{F}$ is minimization, near minimization, or approximate minimization. For minimization, the search process may required more effort than for approximate minimization. In the discussion of each strategy, we will address the significant aspects relevant to that strategy.

General procedures implementing each search strategy discussed in this section were described in Chapter 8 and listed in Appendix D. A general search methodology which puts together all of the components of the search process is presented in a later section.

Total Minimization - A*. For instances in which we would like to guarantee that a minimal vector $\underline{F}$ of formulas which corresponds to a design is developed, we may apply the A* algorithm using heuristic function $h_1(n)$. The A* algorithm is guaranteed to be admissible, i.e., yield a minimal solution, if the heuristic function used by the algorithm is admissible. However, in practice the function $h_2(n)$ has proven to be just as useful as $h_1(n)$.

A liability of the A* algorithm is that if there exist many solutions which are minimal or near-minimal, an inordinate amount of effort may be expended prior to developing a minimal solution. Such circumstances often occur for highly-complex circuit specifications. To minimize the work performed by the A* algorithm, we first develop a representation $\widehat{PS}$ of an initial solution; $\widehat{PS}$ is a set of labels which denote prime implicants which may be used to form the vector $\underline{F}$ of formulas. The cost UB of the prime implicants denoted by $\widehat{PS}$ serves as an upper bound on the value of $f(n)$. Hence, if the value of $f(n)$ for a node n is greater than or equal to UB, then node n is discarded from consideration since it leads to solutions only as good as—or worse than—the initial solution $\widehat{PS}$. If no solution is found which is better than $\widehat{PS}$, then $\widehat{PS}$ serves as the solution returned by the search process.

In some circumstances, the initial solution $\widehat{PS}$ and the value of the heuristic function $h_1(n)$ for the root node will indicate that the A* algorithm does not have to be applied at all. The initial solution forms an upper bound on the cost of the solution. Since $h_1(n)$ is admissible, the value of $h_1(n)$ for the root node is an optimistic estimate of the cost of the solution, i.e., it is a lower bound. If the upper and lower bounds are equal, then $\widehat{PS}$ is a minimal solution and A* search is not required.

We present two alternative strategies for developing an initial solution. The first strategy is a beam search, which we also use as a primary search strategy. However, when developing an initial solution, we will tend to use a smaller width w than would be used in a primary search process. The second strategy is a heuristic approach which uses information produced in developing the value of $h_1(n)$ for a node n. We perform an iterative process in which an estimated least-cost set of prime implicants is always chosen in each iteration for containment in the initial solution.

In the process of evaluating $h_1(n)$ for a set IF of inclusion formulas, a value v_j is developed for each inclusion formula $IF_j \in IF$. Once each v_j is developed, we then take the following actions:

1. Select the inclusion formula IF_j which has the lowest value v_j.
2. Select the term t_k in IF_j which has the lowest value v_k^t of all of the terms in IF_j.
3. Place prime implicants denoted by literals contained in t_k in the initial solution $\widehat{PS}$.
4. Using Procedure 9.1 (Generation of a Node), form a child node n' for current node n based on the choice of prime implicants denoted by the k-th term of IF_j.
5.
   - If the state of the child node is the empty set, i.e., $IF = \emptyset$, then $\widehat{PS}$ is an initial solution. Return the initial solution.
   - Otherwise, an initial solution has not been found. Form values v_j for the set of inclusion formulas in n', and return to Step 1.

During each iteration, we thus select prime implicants associated with the lowest evaluated term in the lowest evaluated inclusion formula. This is analogous to a hill-climbing strategy, since we are using heuristic information to make decisions on prime implicants to include in the initial solution. Procedure 9.8 (Determination of Lower Bound and Initial Solution) implements this strategy.

Procedure 9.8 (Determination of Lower Bound and Initial Solution): Given a set IF of inclusion formulas and the association list $LAB/COSTS$, we form a lower bound LB and an initial solution $\widehat{PS}$ in the following manner:

Step 1. Determine the set P of distinct labels which appear in IF.

Step 2. For each prime implicant p_i in which P_i appears in P:

1. Determine the number n_i of inclusion formulas in which the literal P_i' appears.
2. Get the cost c_i associated with p_i from $LAB/COSTS$.
3. Calculate the utility $u_i = c_i/n_i$.

Step 3. For each inclusion formula $IF_j \in IF$, determine v_j:

1. For each term t_k in IF_j, derive v_k^t by summing the utilities u_i of the prime implicants p_i denoted by literals P_i' in t_k, i.e.,

$$v_k^t = \sum_{\{i|t_k \leq P_i'\}} u_i. \tag{9.30}$$

2. Form v_j by selecting the lowest v_k^t of the l values developed in substep 1, i.e.,

$$v_j = \min(v_1^t, v_2^t, \ldots, v_l^t). \tag{9.31}$$

Step 4.

- If this is the initial iteration of the procedure, i.e., the first time this step is executed, then form LB by adding the m values v_j and rounding up:

$$LB = \left\lceil \sum_{j=1}^{m} v_j \right\rceil . \tag{9.32}$$

- Otherwise, do nothing.

Step 5.

1. Select the inclusion formula IF_j which has the lowest value v_j.
2. Select the term t_k in IF_j which has the lowest value v_k^t of all of the terms in IF_j.
3. Literals contained in t_k denote prime implicants which we will place in the initial solution.

Step 6. Using Procedure 9.1 (Generation of a Node), form a child node n' for current node n based on the choice of prime implicants denoted by the k-th term of IF_j.

Step 7.

- If the set IF of inclusion formulas for the new node n' is the empty set, i.e., $IF = \emptyset$, then an initial solution $\widehat{PS}$ has been formed. Return the initial solution $\widehat{PS}$ and LB.
- Otherwise, return to Step 1.

The **A*** algorithm is suitable in many cases. However, we are sometimes willing to sacrifice the minimality of the solution in return for quickly developing a near-minimal solution. In such circumstances, we may apply a dynamic-weighting strategy to the problem.

Near Minimization - Dynamic Weighting. For circumstances in which we must develop a solution $\underline{F}$ quickly at the price of minimality, but would like to guarantee that $\underline{F}$ is within a fixed-bound of the minimal solution, the dynamic-weighting search strategy is applied to construct a solution. The evaluation function $f(n)$ used by the dynamic-weighting search strategy is defined by the equation

$$f(n) = g(n) + h(n) + \epsilon \cdot [1 - d(n)/N] \cdot h(n), \tag{9.33}$$

for which ϵ is a pre-defined constant, $d(n)$ is the depth in the search tree of a node, and N is the anticipated depth of the search. If the admissible heuristic function $h_1(n)$ is used for $h(n)$ by the

dynamic weighting algorithm, then the resulting solution is guaranteed to be within a factor of $(1+\epsilon)$ of the minimal solution. The dynamic-weighting strategy is then said to be ϵ-admissible.

The anticipated depth N of the goal node used to regulate the overestimate of the heuristic portion of $f(n)$ differs depending on the topology used in the search process. For Topology #1, the maximum depth of the tree is the number k of prime implicants. However, a solution is generally found much higher in the search tree, since only a fraction of the prime implicants are selected for inclusion in the solution. Hence, we define N to be equal to half the number of prime implicants, i.e., $N = k/2$. In instances in which Topology #2 is used, the maximum depth of the search tree is the number m of inclusion formulas. In this case, the number of inclusion formulas is a good approximation of the actual depth of the search tree; hence, we define N to be equal to the number m of inclusion formulas.

We do not form an upper bound when using the dynamic-weighting strategy, because the resulting value of $f(n)$ is an intentional overestimate of the true value for a node.

Approximate Minimization. In the dynamic-weighting strategy, a near-minimal solution, i.e., one guaranteed to be within a fixed bound of the minimal solution, is developed if an admissible heuristic function is used. In some cases, we may want to relax the requirement that the resulting solution is within a fixed bound of the minimal solution if:

1. we believe the resulting solution to be very close to the minimal solution, and
2. effort expended during the search process is sufficiently reduced.

If we believe that the resulting solution is close to the minimal solution, but we cannot guarantee the proximity to the minimal solution, then we are performing approximate minimization. In this section, we present two techniques which may be applied to develop an approximate-minimal solution for our problem.

Beam Search. Beam search is an irrevocable strategy used to quickly generate a good solution. For our problem, the evaluation function $f(n)$ used to evaluate each node is $f(n) = g(n) + h_2(n)$. We use the heuristic function $h_2(n)$ because we would like to get the best estimate possible of the value of a set of nodes at each level in the search tree. Also, the resulting solution is not guaranteed to be minimal in any case, so there is no imperative for using an admissible function. In either topology presented, a width w of 3 or 4 seems to be suitable for our purposes.

We would like to discriminate among nodes at each level of the search, especially at high levels in the search tree. Some nodes which appear good in the early levels may in fact lead to non-minimal solutions. When beam search is used as our primary search strategy, we use Procedure 9.8 to develop an initial solution. The cost UB of the initial solution is used as an upper bound; a node n is discarded if $f(n) \geq UB$. Only nodes for which $f(n) < UB$ are kept at each level of the search tree, even if the number kept is less than the width w. If all generated nodes are discarded, then we quit the search and announce that the upper-bound solution $\widehat{PS}$ is the result of the search process. If there exists a tie, so that we will have to both keep and discard nodes with the same value of $f(n)$, then we keep the nodes with the highest value of $g(n)$, i.e., the nodes likely to be closer to a solution.

A liability of beam search is that since we expand w nodes at each level of the search tree, we may sometimes expand more nodes in the course of a search than when using A* or dynamic-weighting strategies. On the other hand, since all but w nodes are discarded at each level of the search process, less memory is required in beam search than in the other strategies. In the other strategies, a large number of nodes may have to be stored on the OPEN list for possible future expansion.

Static Weighting. An alternative to beam search for approximate minimization is the static-weighting strategy. Using static weighting, the evaluation function for each node is defined as $f(n) = g(n) + W \cdot h_2(n)$, for which W is a pre-defined weight. Hence, more or less weight may

be given to the heuristic function $h_2(n)$ relative to $g(n)$. Generally, when we use static weighting we desire to weight $h_2(n)$ more than $g(n)$. For our purposes, we use $W = 2$ to put twice as much weight on $h_2(n)$ relative to $g(n)$. As in beam search, since the result produced by the search is only approximate-minimal, the function $h_2(n)$ is used rather than $h_1(n)$. Additionally, because the resulting value of $f(n)$ is an intentional overestimate of the true value for a node, an upper bound is not developed prior to applying the static weighting search strategy.

Comparison of Applied Strategies. We have applied the foregoing search strategies to several sets of single-output circuit design problems. Computational results of the search strategy applications are given and discussed in Appendix C. Further work must be done to ascertain the utility of each strategy for classes of design problems. Moreover, experimentation must be conducted to determine "good" pre-defined constants to use with the dynamic-weighting, static-weighting, and beam search strategies.

Implementation of the Search Process

Up to this point in the chapter, we have discussed the different components of the search process. In this section, we present a general search algorithm which ties together the aspects of search presented in foregoing sections. We also present procedures in which the result of the search process is used to construct the vector $\underline{F}$ of formulas which correspond to a design.

General Search Algorithm. Procedures which implement the search strategies used in this work are described in Chapter 8 and listed in Appendix D. In each of these procedures, the manner in which a node is expanded and evaluated using $f(n)$, $g(n)$, and $h(n)$ was not addressed, i.e., the procedures were presented in a general form. In this section we present a general search algorithm which integrates these components into a framework used by most search routines. In a given step, if the actions to be taken differ among search strategies, the different actions are outlined. Algorithm 9.1 implements our general search algorithm. We assume that pre-defined

constants required in a particular search strategy are set *a priori*. Additionally, we presume that the choices for search topology, the heuristic function, and the search strategy are pre-determined.

Algorithm 9.1 (General Search Algorithm): Given a set IF of inclusion formulas and a list $LAB/COSTS$ which associates labels denoting prime implicants and their associated costs, a search process is formed to select a set PS of labels denoting prime implicants which form a vector $\underline{F}$ of formulas in the following manner:

Step 0. Initialise an accumulator $PS = \emptyset$.

Step 1. Using Procedure 9.5 (Intrinsic Partition), form a partition of IF. A search is performed independently for each block IF_{block} of IF.

Step 2.

- If a search has been performed for each block IF_{block} of IF, then PS is the combined result of the search processes. Return PS.
- Otherwise, continue to Step 3 to perform a search for another block.

Step 3.

- For the A* search or beam search strategies, use Procedure 9.8 (Determination of Lower Bound and Initial Solution) to form an initial solution $\widehat{PS}$ and a lower bound LB.
  1. Determine the cost UB of prime implicants denoted by the labels in the initial solution $\widehat{PS}$.
  2. If $LB = UB$, then the initial solution $\widehat{PS}$ is a minimal solution. Add the elements of $\widehat{PS}$ to PS and return to Step 2.
- For dynamic-weighting and static-weighting strategies, do nothing.

Step 4. Form a root node n using the initial set IF_{block} of inclusion formulas, and initialize the OPEN list by placing n onto the list.

Step 5 (All Except Beam Search).

- If the OPEN list is empty, then the initial solution $\widehat{PS}$ found in the upper bound calculation in Step 3 is used to form a solution. Add the elements of $\widehat{PS}$ to PS and return to Step 2.
- Otherwise, select the first node n on the OPEN list and determine if n is the goal node.
  - If n is not the goal node, then continue to Step 6.
  - Otherwise, n is the goal node. Use the path in n to form a solution $\widehat{PS}$. Add the elements of $\widehat{PS}$ to PS and return to Step 2.

Step 5 (Beam Search Only).

- If the OPEN list is empty, then the initial solution $\widehat{PS}$ found in the upper bound calculation in Step 3 is used to form a solution. Add the elements of $\widehat{PS}$ to PS and return to Step 2.

- Otherwise, determine if one of the nodes on the OPEN list is a goal node.
  - If no node on the OPEN list is a goal node, then continue to Step 6.
  - Otherwise, one or more nodes on the goal list is a goal node. Choosing the node n for which the cost is cheapest, use the path in n to form a solution $\widehat{PS}$. Add the elements of $\widehat{PS}$ to PS and return to Step 2.

Step 6 (All Except Beam Search). Expand node n from Step 5 based on the chosen search tree topology using either Procedure 9.3 (Expansion of a Node - Topology #1) or Procedure 9.4 (Expansion of a Node - Topology #2). The following aspects of the node expansion process differ depending on the search strategy and topology:

- Using Procedure 9.3, both Procedures 9.1 and 9.2 are used to create a child for node n. Using Procedure 9.4, Procedure 9.1 is used to create two or more children for n.
- The heuristic functions $h_1(n)$ and $h_2(n)$—implemented by Procedure 9.6 and Procedure 9.7, respectively—which are used depend on the search strategy as well as the minimization goal.
- The evaluation function $f(n)$ used to evaluate each node generated is also dependent on the search strategy.

Step 6 (Beam Search Only). Expand all nodes on the OPEN list based on the chosen search tree topology using either Procedure 9.3 (Expansion of a Node - Topology #1) or Procedure 9.4 (Expansion of a Node - Topology #2).

1. Evaluate each node generated using the evaluation function $f(n) = g(n) + h_2(n)$.
2. After all nodes are generated and evaluated, discard all but the w best nodes. (If there exists a tie such that we may have to both keep and discard nodes with the same value $f(n)$, then keep the nodes with the higher value for $g(n)$.)

Step 7. In this step, the children generated in Step 6 are inserted into the OPEN list such that nodes in the resulting list appear in ascending order of $f(n)$. For nodes with equal values of $f(n)$, nodes with higher values of $g(n)$ are placed before nodes with lower values of $g(n)$. For each node n, perform one of the following actions:

- If n contains the same state as a node n' appearing in the OPEN list and $f(n) < f(n')$, then remove n' from the OPEN list and place n in the list in the appropriate position.
- If n contains the same state as a node n' appearing in the OPEN list and $f(n) \geq f(n')$, then discard n.
- Otherwise, place n in the list in the appropriate position.

Return to Step 5.

Algorithm 9.1 (General Search Algorithm) returns a set PS of labels which denote prime implicants which appear in a vector $\underline{F}$ of formulas. In the next section we describe the process whereby we use PS to construct $\underline{F}$ and thus form a design.

Construction of $\underline{F}$. The process of forming a design differs between forming a formula F corresponding to a design for a single-output circuit and constructing a vector $\underline{F}$ of formulas corresponding to a multiple-output circuit. The available information for constructing the formulas is slightly different in the single-output case than for multiple-output designs. We describe each case in turn.

Formation of a Single Formula F. The partial sum PS_{search} developed in the course of a search process is combined with the partial results from Algorithms 6.1, 6.2, or 6.3 to develop a formula F which represents a function $f(X)$ belonging to an interval $[g(X), h(X)]$. The following information is available for use in constructing F after the search process:

1. a partial sum PS_{search} developed in the search process;
2. a partial sum PS_{rules} identified during the application of reduction rules consisting of labels which denote useful, conditionally-eliminable prime implicants to be contained in F;
3. the set H_{useful} of all useful, conditionally-eliminable prime implicants;
4. the set H_{ess} of all essential prime implicants; and
5. the set $LABS$ of labels corresponding to elements of H_{useful}.

The partial sum PS_{rules} is combined with the partial sum PS_{search} returned by Algorithm 9.1 to form a set PS. After PS_{rules} and PS_{search} are combined, the sets $LABS$ and H_{useful} are used to replace the labels in PS to form a set H_{ps} of prime implicants. H_{ps} is combined with H_{ess} to construct the formula F which corresponds to a design. Procedure 9.9 implements this process.

Procedure 9.9 (Formation of F): Given the sets PS_{search} and PS_{rules} of labels, the sets H_{useful} and H_{ess} of prime implicants, and the set $LABS$ of labels, the formula F is formed in the following manner:

Step 1. Combine the sets PS_{search} and PS_{rules} of labels to form a set PS.

Step 2. Replace labels in PS with their associated prime implicants in H_{useful} to form a set H_{ps}. (Labels in the set $LABS$ of labels correspond one-to-one with elements of H_{useful}. For each label in PS, find the identical label in $LABS$ and then place corresponding element of H_{useful} in H_{ps}.)

Step 3. Combine the sets H_{ps} and H_{ess} to form F. Return F.

Formation of a Vector $\underline{F}$. Formation of a vector $\underline{F}$ of formulas is different for multiple-output than for single-output designs due to the fact that we are creating a set of formulas rather than a single formula. Additionally, the available information for constructing $\underline{F}$ is different for the multiple-output case. The partial sum PS_{search} developed in the search process is combined with the results from Algorithms 7.1 or 7.2 to develop a vector $\underline{F}$ of formulas which represent functions $\underline{f}(X)$ belonging to the intervals $[\underline{g}(X), \underline{h}(X)]$. The following information is available for use in constructing $\underline{F}$ after the search process:

1. a partial sum PS_{search} developed in the search process;
2. the set M_{ce}, which represents all conditionally-eliminable multiple-output prime implicants;
3. the set M_{all}, which represents all MOPIs which are either essential or were identified for containment in $\underline{F}$ during rule reduction; and
4. the set $LABS$ of labels in one-to-one correspondence with elements of M_{ce}.

For each label in PS_{search}, we find the same label appearing in $LABS$, which then allows us to determine the associated term $t(X, Z)$ in M_{ce}. The X-part of the corresponding term in M_{ce} is the MOPI denoted by the label in PS_{search}. After finding $t(X, Z)$ in M_{all}, the Z-part is filled with the literal z_j for each j in which neither z_j nor z_j' appears. After this process is performed for all labels in PS_{search}, we then form each formula F_j of $\underline{F}$. For each term $t(X, Z)$ in M_{all}, if the literal z_j appears in $t(X, Z)$, then the associated X-part $u(X)$ is contained in F_j. After each term in M_{all} has been examined and a formula F_j is constructed, $ABS(F_j)$ is formed. After each formula $ABS(F_j)$ is derived, the development of $\underline{F}$ is complete.

Procedure 9.10 (Formation of $\underline{F}$): Given the set PS_{search} of labels, the sets M_{ce} and M_{all} which represent MOPIs, and the set $LABS$ of labels, the vector $\underline{F}$ of formulas is formed in the following manner:

Step 1. For each label in PS_{search}:

1. Find the same label appearing in $LABS$.
2. Elements of $LABS$ are in one-to-one correspondence with terms in M_{ce}. For the label of substep 1, determine the corresponding term $t(X, Z)$ in M_{ce}.

3. Locate the term $t(X, Z)$ in M_{all}.
4. Fill the Z-part of $t(X, Z)$ in M_{all} with the literal z_j for each j in which neither z_j nor z_j' appears.

Step 2. For $j = 1, 2, \ldots, m$, form F_j:

1. Examine each term $t(X, Z)$ in M_{all} to determine if the literal z_j appears in the term.
   - If z_j appears in $t(X, Z)$, then place the X-part $u(X)$ in F_j.
   - Otherwise, do not place $u(X)$ in F_j.
2. After each term in M_{all} has been examined, form $ABS(F_j)$.

Step 3. After each formula $ABS(F_j)$ has been formed, the development of $\underline{F}$ is complete. Return $\underline{F}$.

A Decomposition Strategy

In this section we outline a problem-reduction strategy for the search process. In our discussion of intrinsic partitions, we noted that a set IF of inclusion formulas may be partitioned prior to the search process. A state-space search is then performed for each of the component blocks of the partition of IF. However, during the course of the search process, we may be able to partition as well the set $\widehat{IF}$ of inclusion formulas associated with a given node. Hence, during the middle of the search process, we may again decompose the problem into a set of searches rather than perform a large search process. We call partitions which may be formed due to choices made during the course of a search process, i.e., the selection or deletion of prime implicants, ***induced partitions***. Moreover, we show that an intelligent selection or deletion of certain prime implicants may facilitate the formation of an induced partition. We discuss how to use a set of inclusion formulas and either cut-set or graph-partitioning techniques to determine prime implicants which are useful in facilitating the formation of an induced partition.

Suppose we are performing a search for the set of inclusion formulas given by Block V of the intrinsic partition in Example 9.1, i.e.,

$$\begin{aligned} IF_1 &= P'_{127} + P'_{113}P'_3 + P'_{113}P'_6 \\ IF_2 &= P'_{114} + P'_{106}P'_{113} + P'_{106}P'_{110} \\ IF_3 &= P'_{109} + P'_{102}P'_{79}P'_{90} + P'_{102}P'_{110}P'_{79} \\ IF_4 &= P'_3 + P'_2P'_6 + P'_{127}P'_2. \end{aligned} \tag{9.34}$$

Upon examination of these inclusion formulas we determine that the prime implicant in common between IF_1 and IF_2 is the PI denoted by P_{113}. Similarly, the prime implicant in common between IF_2 and IF_3 is denoted by P_{110}. Of the remaining combinations of inclusion formulas, only IF_1 and IF_4 share prime implicants; they have the prime implicants denoted by P_3, P_6, and P_{127} in common. Using the information regarding common prime implicants, we may represent the inclusion formulas and the prime implicants common among the formulas by a graph in which each vertex denotes an inclusion formula and edges in the graph depict prime implicants common among the set of inclusion formulas. A graph which denotes the shared prime implicants for the set of inclusion formulas in (9.34) is given by Figure 9.4. Suppose that the prime implicant denoted by P_{113} is used to generate children for the root node n for which the associated state is the set of inclusion formulas in (9.34). Due to the selection of P_{113}, an induced partition may then be formed for the inclusion formulas associated with each of n's children; this is demonstrated in Example 9.6.

Example 9.6: Suppose we are given the set $IF = \{IF_1, IF_2, IF_3, IF_4\}$ of inclusion formulas defined by

$$\begin{aligned} IF_1 &= P'_{127} + P'_{113}P'_3 + P'_{113}P'_6 \\ IF_2 &= P'_{114} + P'_{106}P'_{113} + P'_{106}P'_{110} \\ IF_3 &= P'_{109} + P'_{102}P'_{79}P'_{90} + P'_{102}P'_{110}P'_{79} \\ IF_4 &= P'_3 + P'_2P'_6 + P'_{127}P'_2. \end{aligned} \tag{9.35}$$

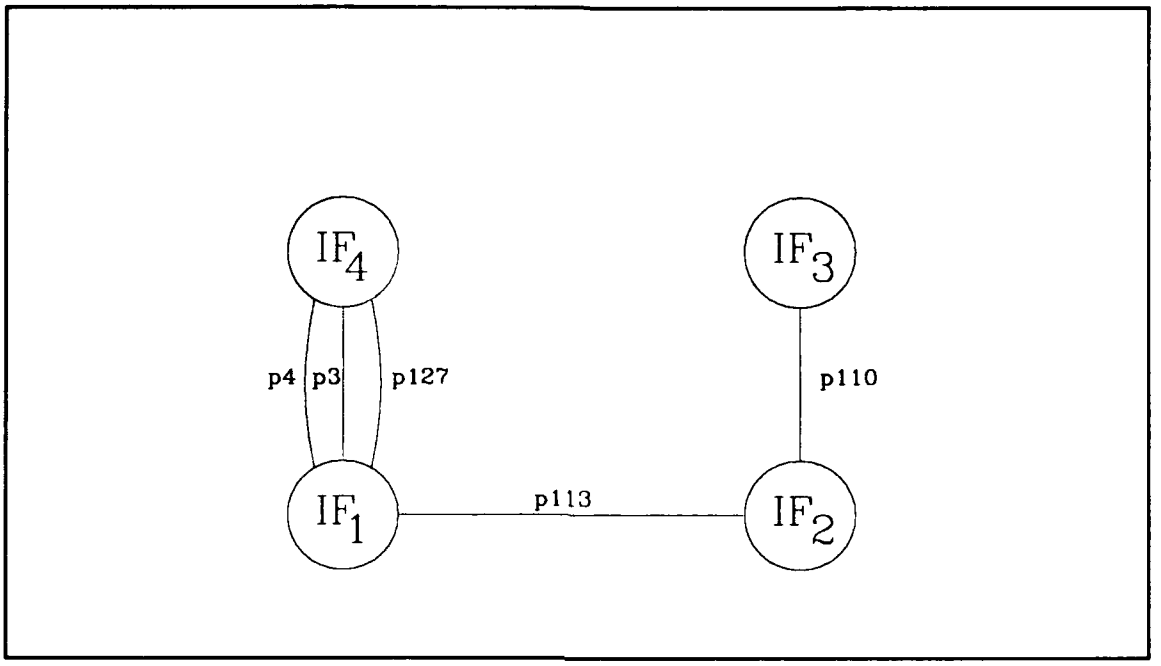


Figure 9.4. Graph Representation of Inclusion Formulas and Common PIs

The inclusion formulas in IF represent the coverage of the PIs denoted by $P_{127}, P_{114}, P_{109}$, and P_3, respectively, by conditionally-eliminable prime implicants. Using Topology #1, if the prime implicant associated with P_{113} is used to generate children for the node with the associated state IF, two children are generated—one based on the selection of the PI denoted by P_{113} and one based on the removal of the PI from consideration. We describe in turn the formation of each of the children.

We first discuss the formation of the child based on the selection of the prime implicant denoted by P_{113}. The first step taken is to modify the inclusion formulas of (9.35) based on the selection of this prime implicant to form a new set of formulas. This is performed by removing the literal P'_{113} from terms in the formulas in IF. Removing the literal P'_{113} from terms in (9.35) and deleting absorbed terms, the set

$$
\begin{aligned}
IF_1 &= P'_{127} + P'_3 + P'_6 \\
IF_2 &= P'_{114} + P'_{106} \\
IF_3 &= P'_{109} + P'_{102}P'_{79}P'_{90} + P'_{102}P'_{110}P'_{79} \\
IF_4 &= P'_3 + P'_2P'_6 + P'_{127}P'_2
\end{aligned}
\tag{9.36}
$$

of inclusion formulas is developed. Examination of the formulas in (9.36) reveals that only inclusion formulas IF_1 and IF_4 have common prime implicants. Hence, the selection of the prime implicant denoted by P_{113} results in the formation of the induced partition $\{\{IF_1, IF_4\}, \{IF_2\}, \{IF_3\}\}$ of the inclusion formulas.

In this instance, however, Reduction Rule Set #2 may be applied to further reduce the set of inclusion formulas. Since IF_1 denotes the coverage of the prime implicant denoted with P_{127}, the PI denoted by P_3 dominates the prime implicant denoted by P_{127}. Because IF_4 represents the coverage of the PI denoted by P_3, formula IF_1 may be deleted and terms which contain literal P'_{127} may be removed from remaining inclusion formulas. Additionally, since IF_2 represents the coverage of the prime implicant denoted by P_{114}, the prime implicant denoted by P_{106} dominates the prime implicant denoted by P_{114}. Then, because the PI denoted by P_{106} does not have an associated inclusion formula, the term P'_{114} is simply deleted from IF_2. The prime implicant denoted by P_{106} then becomes secondary essential; thus, it must be selected and IF_2 is deleted from the remaining set of inclusion formulas. It follows that after rule reduction we derive the set $\{P_{113}, P_{106}\}$ which denotes PIs which must appear in the resulting minimal vector $\underline{F}$ of formulas as well as the revised set

$$
\begin{aligned}
IF_3 &= P'_{109} + P'_{102}P'_{79}P'_{90} + P'_{102}P'_{110}P'_{79} \\
IF_4 &= P'_3 + P'_2P'_6
\end{aligned}
\tag{9.37}
$$

of inclusion formulas. It is then obvious that the prime implicants denoted by P_{109} and P_3 must be selected for containment in F.

If we remove the prime implicant denoted by P_{113} from consideration, then the inclusion formulas of (9.35) are modified based on the deletion of this prime implicant to form a new set of formulas. In this instance, all terms containing the literal P'_{113} are removed from formulas in IF. Removing such terms results in the inclusion formulas

$$\begin{aligned} IF_1 &= P'_{127} \\ IF_2 &= P'_{114} + P'_{106}P'_{110} \\ IF_3 &= P'_{109} + P'_{102}P'_{79}P'_{90} + P'_{102}P'_{110}P'_{79} \\ IF_4 &= P'_3 + P'_2P'_6 + P'_{127}P'_2. \end{aligned} \tag{9.38}$$

Of the formulas in (9.38), the prime implicant denoted by P_{127} is common between IF_1 and IF_4. Additionally, the prime implicant denoted by P_{110} is between IF_2 and IF_3. Hence, the deletion of the prime implicant denoted by P_{113} results in the formation of the induced partition $\{\{IF_1, IF_4\}, \{IF_2, IF_3\}\}$ of the inclusion formulas. However, applying Reduction Rule Set #2 results in the selection of the prime implicants denoted by P_{127} and P_2 in F and the deletion of formulas IF_1 and IF_4 from the revised set of inclusion formulas. We thus derive the set

$$\begin{aligned} IF_2 &= P'_{114} + P'_{106}P'_{110} \\ IF_3 &= P'_{109} + P'_{102}P'_{79}P'_{90} + P'_{102}P'_{110}P'_{79} \end{aligned} \tag{9.39}$$

of inclusion formulas. It is then apparent that the prime implicants denoted by P_{109} and P_{114} must be selected for containment in F.

The use of the prime implicant denoted by P_{113} to form children for the node associated with the set (9.34) of inclusion formulas thus facilitates the formation of an induced partition for the inclusion formulas associated with each of the children. This is clear upon examination of Figure 9.4 since the edge between the vertices of the graph denoting IF_1 and IF_2 is associated with P_{113}. The use of P_{113} also leads to further reduction of the resulting set of inclusion formulas and identification of prime implicants which constitute formulas in $\underline{F}$. Additionally, the selection of the

prime implicant denoted by P_{113} leads to the formation of an induced partition of three blocks, rather than simply a two-block partition.

Example 9.6 illustrates the selection of a prime implicant which results in the formation of an induced partition. Inclusion formulas which form each block of the induced partition may then be handled independently. The use of induced partitions thus facilitates a decomposition of the problem. Applying this technique on a global scale throughout the search process, the search process employs a problem-reduction approach rather than a state-space representation. An AND/OR graph is then used to represent the search process rather than an OR graph. Figure 9.5 depicts a search process which uses both intrinsic as well as induced partitions. For a node n which is the parent to a set of AND nodes, the results of a search for each of its children are combined to develop n's corresponding solution.

To develop a search process based on the use of induced partitions, we must deal with two issues:

1. the formation of a search strategy for solving a problem represented with an AND/OR tree, and
2. the development of a technique for identifying prime implicants (Topology #1) or inclusion formulas (Topology #2) to use in the expansion of a node which facilitates the formation of an induced partition.

Unfortunately, the author has not had the opportunity to explore these issues in depth. However, we will offer several ideas which should provide the basis for further research.

A number of search strategies for developing least-cost solutions for problems represented using an AND/OR graph, e.g., AO*, are found in *The Handbook of Artificial Intelligence, Volume I* (Barr 81) and *Artificial Intelligence* (Rich 83). The strategies given in these texts must be adapted for use in an AND/OR tree rather than the general case of an AND/OR graph. We believe that this will actually simplify the resulting search strategy. For example, the AO* strategy does not use

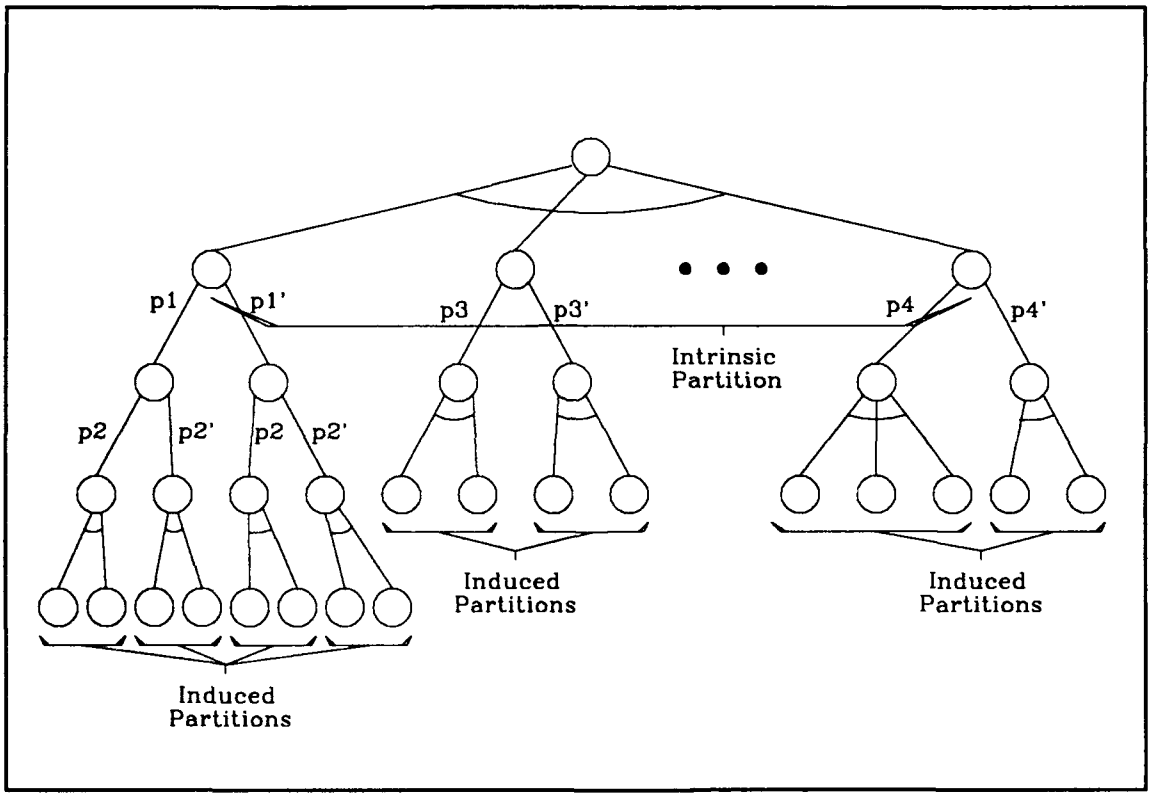


Figure 9.5. Problem Representation Using an AND/OR Graph

or calculate the value of $g(n)$ for a given node n since there may be many paths to n (Rich 91:86); however, for our problem the value of $g(n)$ may be useful since there will only be one path to n. Nodes which are considered "unsolvable" are handled by the AO* algorithm; however, all nodes in our representation are solvable, i.e., they are on the path to a solution. Other aspects of the search process will have to be examined to develop a search strategy suitable for our problem.

We give more insight on the development of a technique for identifying prime implicants (Topology #1) or inclusion formulas (Topology #2) to use in the expansion of a node to facilitate the formation of an induced partition. Using the graph-based representation of a set of inclusion formulas and common prime implicants, we can apply graph algorithms to select prime implicants or inclusion formulas for use during node expansion. To proceed with this discussion, however, we must define a number of terms used in graph theory. We use the terminology in the *Introduction to Graph Theory* (Wilso 79).

As we earlier alluded, the points in a graph G are called *vertices* and the lines which connect the points—if undirected—are called *edges*. A *subgraph* G' of G contains a subset of the vertices and edges in G. A graph is called *complete* if there exists an edge between every two distinct vertices. A *path* is a sequence of distinct edges in G in which each vertex on the path is distinct (except possibly the first and last vertices) that may be followed to get from one vertex to another. A path is *closed* if the first and last vertices are identical; a closed path containing at least one edge is called a *circuit*. G is said to be *connected* if there exists a path between any two vertices in G. Otherwise, G is said to be *disconnected*. Each connected subgraph G' of G is called a *component* of G. A connected graph has only one component; a disconnected graph has more than one component. A *disconnecting set* of a connected graph G is a set of edges whose removal disconnects G. A *cutset* is a disconnecting set in which no proper subset is a disconnecting set. A cutset which contains a single edge is called a *bridge* or *isthmus*. The *edge-connectivity* of a graph is the size of the smallest cutset; a graph G is said to be *k-edge-connected* if the edge-connectivity

of G is greater than or equal to k. A *separating set* of a connected graph G is a set of vertices whose deletion disconnects G. A *cut-vertex* or *articulation point* is a separating set which contains only one vertex. The *vertex-connectivity* of a graph G is the size of the smallest separating set; G is said to be *k-connected* if the vertex-connectivity is greater than or equal to k. Using this terminology, we outline several approaches for determining prime implicants and inclusion formulas to select for node expansion which facilitate the formation of an induced partition.

In the first step of any procedure for making a selection, we form a graph G in which vertices depict inclusion formulas associated with a node n in the search tree and edges which link vertices represent common prime implicants between associated inclusion formulas. Moreover, if a prime implicant is common among three or more inclusion formulas, then a complete subgraph G' is formed which contains the vertices depicting the inclusion formulas and edges between the vertices representing the prime implicant common among them. Additionally, if more than one prime implicant is common between two inclusion formulas, then more than one edge will appear between the respective vertices. It is important that multiple edges appear between two vertices v_i and v_j in such an instance, because we may have to deal with more than one prime implicant to break a direct link between v_i and v_j.

A simple approach for decomposing the search problem is to use bridges (Topology #1) and cut-vertices (Topology #2). At a given point in the search process, we can use an algorithm which determines all of the bridges and cut-vertices in G. Using Topology #1, we select a bridge in which the edge denotes the prime implicant to use in the formation of children for a given node. In each of the children, the associated set of inclusion formulas can be represented by a disconnected graph of two or more components, since the edge associated with the selected prime implicant is removed from G. A search process is then performed independently for each of the components. If more than one bridge exists in G, then the "best" bridge to use in the selection of a prime implicant is the one for which the removal of the corresponding edge breaks up G into components in which the

disparity in contained vertices is minimized. For example, if two bridges exist in a graph consisting of eight vertices, and the removal of the edge associated with the first bridge breaks up the graph into components of two and six vertices, and the removal of the edge associated with the second bridge breaks up the graph into components of three and five vertices, then the prime implicant denoted by the edge in the second bridge would be used to force an induced partition. Similarly, using Topology #2, a cut- vertex denotes an inclusion formula to use in the formation of a node's children such that an induced partition of the inclusion formulas associated with each of the children may be formed.

An algorithm and FORTRAN implementation which determines all bridges and cut-vertices of a graph is found in *Algorithms on Graphs* (Lau 89). The original work on which this implementation is based is (Paton 69) and (Paton 71). Additionally, the complexity of this algorithm is $\mathcal{O}(n^2)$. In many instances, it will likely be beneficial to apply an algorithm such as this one to determine the set of bridges and/or cut-vertices. This may be followed by a procedure which selects the one—bridge or cut-vertex (depending on the topology)—which causes the best decomposition of the graph and hence the problem. However, in some instances the use of bridges may only be marginally useful. Figure 9.6 illustrates the fifteenth block of a intrinsic partition formed for interval IC14 in the course of applying Algorithm 6.2. Using a cut-vertex approach, the vertex which denotes IF_{12} would be a good choice for developing children for the root node, because it would facilitate a partition of the graph into components of nearly equal number of vertices. Using bridges to select a prime implicant to form children for the root node may only be marginally useful since bridges are located only near the extremities of the graph. However, an approach which may be attempted is to use these bridges to "whittle away" the graph until no more bridges exist; a small decomposition of the problem may be better than none at all.

If no bridges exist in a graph representing a set of inclusion formulas associated with a node in the search tree, a heuristic which may work is to first select a prime implicant p_i which is most

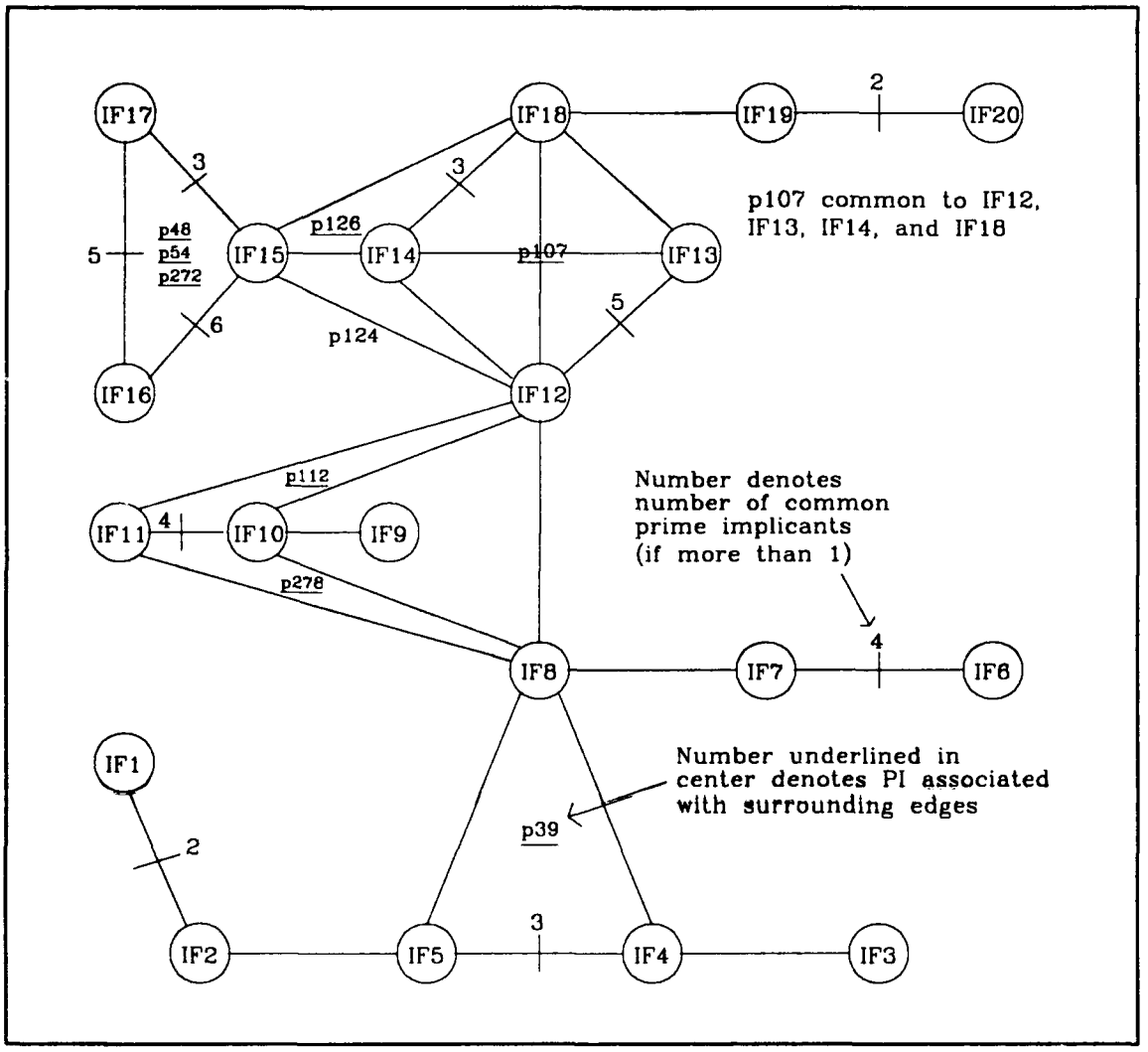


Figure 9.6. Graph Representing Block XV of IC14

common among inclusion formulas. After such a p_i is used to form children for a node, the inclusion formulas associated with each of the child nodes may be represented by a graph which contains bridges. The reason why this occurs is that p_i causes circuits in the graph; the subgraph formed considering only the vertices associated with inclusion formulas in which P_i appears is complete. Removal of the edges associated with p_i decreases the number of possible circuits in the graph, hence causing the occurrence of bridges. In Figure 9.6, the prime implicant p_{107} is associated with edges connecting four vertices of the graph. If we were to remove these edges, then the edge depicting prime implicant p_{124} is contained in a bridge in the resulting graph. Selection of p_{124} to form the next set of children facilitates an induced partition in which the components of the associated graphs contain seven and thirteen vertices.

An alternative to using an algorithm to detect bridges and cut-vertices is to use a general graph-partitioning technique. Such methods either identify cutsets to disconnect a graph into two components or identify sets of edges for partitioning the graph into an arbitrary number of components. Although an optimal partition a graph into k components, i.e., the identification of a minimal number of edges, is an NP-complete problem (Garey 79), heuristic algorithms exist to solve the graph-partitioning problem which are very efficient. Two such algorithms are the Kernighan-Lin (Kerni 70) and the Fiduccia-Mattheyses (Fiduc 82) algorithms. Each of these algorithms uses an iterative approach to determine a good graph partition. If the average number of edges incident to a vertex is small, then the computational cost for each iteration of the Kernighan-Lin algorithm for a graph of n vertices is $\mathcal{O}(n\log(n))$. For the arbitrary case, the computational cost per iteration is $\mathcal{O}(n^2)$. The computational cost for each iteration of the Fiduccia-Mattheyses algorithm is $\mathcal{O}(n\log(n))$. The underlying idea in both algorithms is to divide a set of an even number of vertices into two components with an equal number of vertices such that the cutset used to form the partition is minimal with respect to the contained edges. Each method may be generalized to form partitions of an unequal number of vertices; moreover, each may be used to develop a partition of more than two components. For our purposes, the partition of a graph into two components of

unequal number would suffice. In our problem, a cutset containing a smaller number of edges is more likely if the components are allowed to be unequal.

No matter which methodology is followed, once we determine prime implicants which facilitate an induced partition of the graph, the decomposition of the problem should make the total search process far more efficient than if a decomposition strategy is not employed. For our problems, once the number of inclusion formulas contained in a block of a partition is small, e.g., six to eight, further decomposition may not be necessary. This may be highly dependent on the connectivity of the graph. This issue requires further study.

There are several disadvantages to using induced partitions in the course of a search process. Much work may be performed to identify prime implicants and form a decomposition when in some cases a solution may be quickly developed without a problem-reduction step. Moreover, a search process which employs this technique will more complicated than when not decomposing the problem. However, for most problems the efficiency gain which results due to the decomposition of the problem will likely be significant.

In the foregoing discussion, we have provided a number of suggestions on which to base further research into this problem. More work must be done to determine the best approaches. We believe a combination of the techniques described above may prove useful in improving the overall search process.

Summary

In this chapter we presented a search process for identifying prime implicants to be used in the construction of a vector $\underline{F}$ of formulas corresponding to a design. At the beginning of the search process, we are given a set IF of inclusion formulas produced by algorithms described in earlier chapters. The search process uses IF to form a set PS of labels which denotes a least-cost set of

prime implicants to be contained in $\underline{F}$. Subsequently, the labels are replaced by their associated prime implicants to form $\underline{F}$.

We began the chapter by presenting a set of issues which must be considered in formulating a search process. Throughout the chapter, we addressed each of these issues as we described the different components of the search process. After introducing each component, a general search algorithm was presented which integrates the components. Two decomposition strategies were discussed for use in simplifying the search process.

Throughout the chapter, we described different methods for representing as well as manipulating information. The availability of alternative approaches to solving the problem facilitates our ability to make trade-offs during the design process. Two different trade-offs were discussed in the chapter:

- computational effort versus memory usage, and
- speed of the search process versus minimality of the solution.

Choices regarding these issues are made in order to satisfy our design objectives as well as time requirements. Ultimately, the circuit designer must make decisions regarding these issues. Computational resource limitations may also affect our design decisions as well.

A number of new ideas were introduced in this chapter:

- A general search algorithm which employs heuristic search was described which may be used to develop a minimal vector $\underline{F}$ of formulas which corresponds to a design. Different search topologies and node representations were presented for use in the search algorithm.
- We introduced a spectrum of search strategies which facilitate trade-offs between speed of the search process and minimality of the solution. Thus, different search strategies facilitate varying levels of effort. Search strategies were described for producing minimal, near-minimal, and approximate-minimal solutions.
- Two heuristic functions $h_1(n)$ and $h_2(n)$ were presented. The first, $h_1(n)$, is an admissible function which guarantees that a minimal solution will be found when A* search is used. Function $h_2(n)$ produces better estimates than $h_1(n)$ most of the time, although it overestimates in some cases.

- Two different state-space classifications—circuit formation and circuit transformation—for representing the digital design were introduced. All digital design approaches fall into one of these two categories.
- Two different decomposition strategies were described. The first approach, the formation of an intrinsic partition, is similar in concept to the partitioning of a prime implicant table. A graph-based partitioning scheme is used to form an induced partition in the second method. The use of either technique significantly reduces effort required during the search process.

X. Alternative Minimization Techniques

In this chapter, we present techniques for producing designs for circuit specifications which are different from the type of designs which can be generated using conventional minimization algorithms. In conventional approaches, a specification is stated and a design is generated such that the circuit outputs are functions of the circuit inputs. We propose methods which facilitate the use of signals other than the circuit inputs to produce circuit outputs.

In the late 1950s Ledley (Ledle 60) proposed a set of digital design problems for which he developed solutions based on solving Boolean equations. Ledley devised ad hoc methods for solving the proposed problems. However, his methods would not be useful for highly complex problems. In the first part of this chapter, we present techniques for dealing with Ledley's problems that result in minimal designs. An application of one of Ledley's problems is that signals from existing circuits may be used in constructing new designs which are cheaper than what could be developed without knowledge of the existing signals.

In the second part of this chapter, we present methods for developing approximate-minimal recurrent circuits. The outputs of recurrent circuits are formed using other outputs as well as circuit inputs. The cost of the resulting designs are generally lower than those of conventional designs. We introduce algorithms which are analogous to those presented in Chapter 7, with modifications to facilitate the handling of output nodes in a similar fashion as input nodes.

Ledley's Problems

An Overview of Ledley's Problems. Ledley proposed three design problems which he called *elementary problems of circuit design* (Ledle 60). Each of the problems involves a circuit of the form depicted in Figure 10.1. We denote input nodes by the vector $X = (x_1, x_2, \ldots, x_n)$ and output nodes by the vector $Z = (z_1, z_2, \ldots, z_m)$. Additionally, intermediate nodes—those which are internal to the circuit—are denoted by the vector $Y = (y_1, y_2, \ldots, y_l)$. In the first problem,

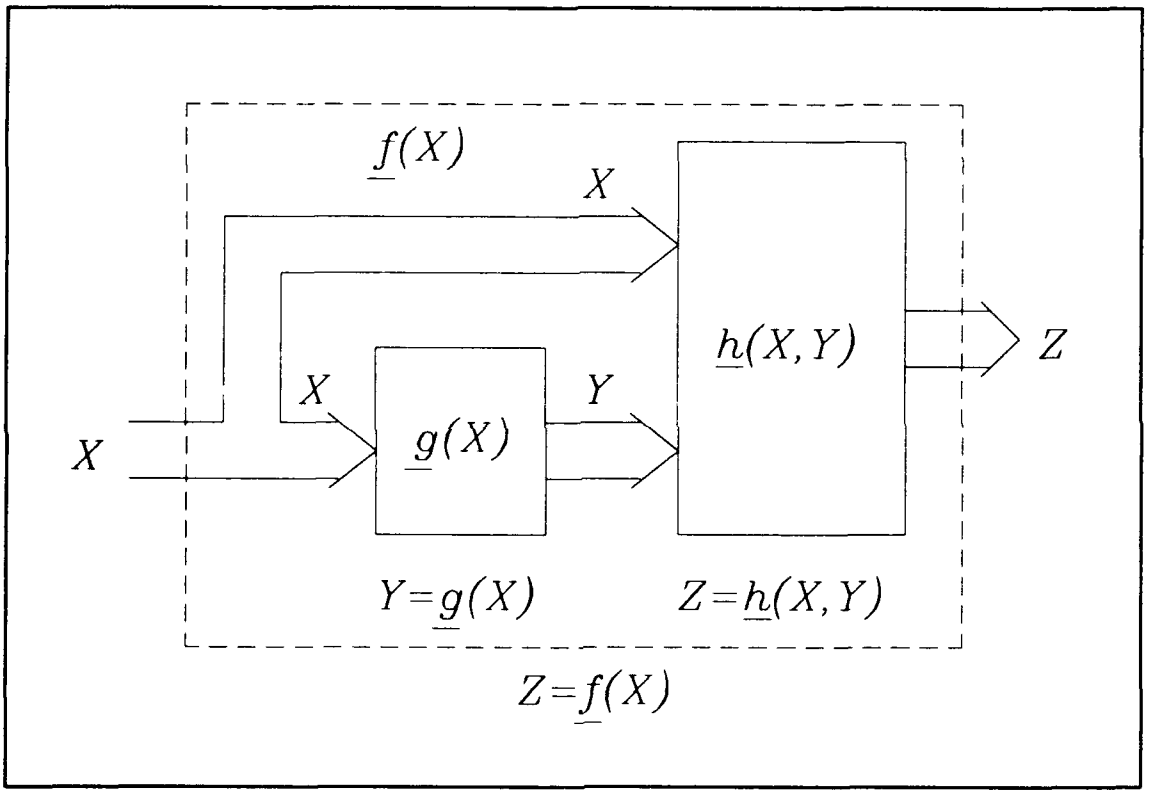


Figure 10.1. Ledley's Circuit

called a *Type 1* problem by Ledley, we know the components which compose the circuit and we are required to analyse the circuit to determine the function that the circuit implements. In the last two problems—called *Type 2* and *Type 3* problems—we are given the global circuit specification as well as functions implemented by subcircuits; the problem is to design the remaining subcircuits such that the global circuit specification is met. Using the notation given in Figure 10.1, we give a formal description of each of Ledley's problems.

Type 1 Problem: Given the vectors $\underline{g}(X)$ and $\underline{h}(X,Y)$ of functions, develop a minimal vector $\underline{F}$ of formulas to represent the vector $\underline{f}(X)$ of functions.

Type 2 Problem: Given the vector $\underline{g}(X)$ of functions and the set $[\underline{f}^l(X), \underline{f}^u(X)]$ of intervals, develop a minimal vector $\underline{H}$ of formulas to represent the vector $\underline{h}(X,Y)$ of functions.

Type 3 Problem: Given the vector $\underline{h}(X,Y)$ of functions and the set $[\underline{f}^l(X), \underline{f}^u(X)]$ of intervals, develop a minimal vector $\underline{G}$ of formulas to represent the vector $\underline{g}(X)$ of functions.

Our statement of a Type 1 problem differs from Ledley's in that his concern was for the formation of the functions $\underline{f}(X)$ rather than the formation of a minimal vector $\underline{F}$ of formulas.

Ledley developed matrix-based methods for solving the proposed problems. However, his methods did not guarantee a minimal solution—or even a good solution. In addition, his techniques for solving the problems are not very useful for highly complex specifications. For example, in a Type 2 problem in which we are given an equation $z = f(X)$, in which f is defined by the equation

$$f(X) = x_3x_4 + x_1x_4'x_5 + x_2x_3x_5' + x_2x_4, \tag{10.1}$$

and the system $Y = \underline{g}(X)$, in which $\underline{g}(X)$ is defined by the system

$$\begin{aligned} g_1(X) &= x_3x_4' + x_3'x_5 \\ g_2(X) &= x_3'x_4 + x_3x_5' \\ g_3(X) &= x_3x_4 + x_4'x_5, \end{aligned} \tag{10.2}$$

the solution given by Ledley, using his methods, is

$$h(X,Y) = y_1'y_3 + x_1y_1y_2' + x_2y_2y_3'. \tag{10.3}$$

However, a least-cost solution with respect to any typical cost criterion, e.g., fewest terms or fewest literals, is

$$h(X,Y) = x_3x_4 + x_1y_3 + x_2y_2. \tag{10.4}$$

Methods in this section return least-cost solutions for Ledley's problems.

We present a methodology to handle each of Ledley's problems. Our methodology consists of three steps:

1. reduce the information that is given about the problem to a 1-normal form $\psi(X,Y,Z) = 1$;
2. using $\psi(X,Y,Z)$, derive a 1-normal form required for the problem; and
3. use the algorithms developed in Chapters 6 and 7 to develop a minimal design which meets the specification given by the 1-normal form developed in step 2.

For a Type 1 problem, Step 3 may be modified to return a general solution of the 1-normal form developed in Step 2. Returning a general solution completes the task as originally defined by Ledley, which is an *analysis problem*—a problem in which information is determined about the circuit. We modify the problem to change it to a design problem. We apply the aforementioned methodology in turn to each of Ledley's problems.

Type 1 Problems. In a Type 1 problem, the vectors $\underline{g}(X)$ and $\underline{h}(X,Y)$ of functions are given, and we must determine the vector $\underline{f}(X)$ of functions. In this problem, we know the functions of the subcircuits which compose the circuit and are required to analyze the circuit to determine the vector $\underline{f}(X)$ of functions implemented by the global circuit. In addition to analyzing the circuit, we can use the framework of the problem to develop a new circuit which meets the same specification as does the implemented circuit. If the new design is significantly cheaper than the existing circuit, then we may find it desirable to replace the existing circuit with the new design.

Given the vectors $\underline{g}(X)$ and $\underline{h}(X,Y)$ of functions, we use reduction to form an equivalent 1-normal form $\psi(X,Y,Z) = 1$. In the first step of the process, an equivalent 1-normal form $\xi(X,Y) = 1$ is derived for the system $Y = \underline{g}(X)$. The system

$$\begin{aligned} y_1 &= g_1(X) \\ y_2 &= g_2(X) \\ &\vdots \\ y_l &= g_l(X) \end{aligned} \tag{10.5}$$

is equivalent to the equation

$$\prod_{k=1}^{l}(y_k \odot g_k(X)) = 1. \tag{10.6}$$

We define the left-hand side of (10.6) to be the function $\xi(X,Y)$. In the second step, we form an equivalent 1-normal form $\lambda(X,Y,Z) = 1$ for $Z = \underline{h}(X,Y)$. Given the system

$$\begin{aligned} z_1 &= h_1(X,Y) \\ z_2 &= h_2(X,Y) \\ &\vdots \\ z_m &= h_m(X,Y), \end{aligned} \tag{10.7}$$

we form the equation

$$\prod_{j=1}^{m}(z_j \odot h_j(X,Y)) = 1. \tag{10.8}$$

We define the function $\lambda(X,Y,Z)$ to be equal to left-hand side of (10.8). The equations $\xi(X,Y) = 1$ and $\lambda(X,Y,Z) = 1$ can be combined to formed the equation $\psi(X,Y,Z) = 1$. Subsequently, the function $\psi(X,Y,Z)$ is defined by the equation

$$\psi(X,Y,Z) = \xi(X,Y) \cdot \lambda(X,Y,Z). \tag{10.9}$$

Formation of the equation $\psi(X,Y,Z) = 1$ is the first step in our methodology for handling Ledley's problems.

The second step in our methodology for Ledley's problems in the development of 1-normal form required for the type problem. In a Type 1 problem, we must derive the 1-normal form $\phi(X,Z) = 1$ which specifies the complete circuit depicted in Figure 10.1. Elimination of the Y-variables from $\psi(X,Y,Z) = 1$ yields an equation $\phi(X,Z) = 1$, in which $\phi(X,Z)$ is defined by the equation

$$\phi(X,Z) = EDIS(\psi(X,Y,Z),Y). \tag{10.10}$$

The specification $\phi(X,Z) = 1$ is guaranteed to be tabular since it is developed from an implemented circuit.

The development of a minimal vector $\underline{F}$ of formulas is the third step of our methodology for Ledley's problems. In a Type 1 problem, we develop a design to determine if it is cheaper than the existing circuit. Using the 1-normal form $\phi(X,Z) = 1$, either Algorithm 7.1 or Algorithm 7.2 may be used to develop a new design. (For a 1-normal form $\phi(X,z) = 1$, Algorithms 6.1, 6.2, or 6.3 are

used to develop a design.) If the new design is significantly cheaper in comparison to the existing circuit, then we may find it desirable to replace the existing circuit with the redesigned circuit. Algorithm 10.1 (Type 1 Problem) may be used to analyse the circuit depicted in Figure 10.1 and develop a minimal equivalent circuit.

Algorithm 10.1 (Type 1 Problem): Given the vectors $\underline{g}(X)$ and $\underline{h}(X,Y)$ of functions, a minimal vector $\underline{F}$ of formulas is developed to represent the vector $\underline{f}(X)$ of functions in the following manner:

Step 1. Using Boolean reduction, derive an equivalent 1-normal form $\xi(X,Y) = 1$ for the system $Y = \underline{g}(X)$.

Step 2. Using Boolean reduction, derive an equivalent 1-normal form $\lambda(X,Y,Z) = 1$ for the system $Z = \underline{h}(X,Y)$.

Step 3. Form $\psi(X,Y,Z) = \xi(X,Y) \cdot \lambda(X,Y,Z)$.

Step 4. Form $\phi(X,Z) = EDIS(\psi(X,Y,Z),Y)$.

Step 5.

- Given the 1-normal form specification $\phi(X,z) = 1$, use one of Algorithm 6.1, Algorithm 6.2, or Algorithm 6.3 to develop a minimal formula F.
- Given the 1-normal form specification $\phi(X,Z) = 1$, use either Algorithm 7.1 or Algorithm 7.2 to develop a minimal vector $\underline{F}$ of formulas.

Rather than forming a minimal vector $\underline{F}$, our interest may lie—as did Ledley's—with analysing the circuit depicted in Figure 10.1 to derive the vector $\underline{f}(X)$ of functions. To form $\underline{f}(X)$, we first must form a general solution of $\phi(X,Z) = 1$ for Z. Let us define the function $\tilde{\phi}_j(X,z_j) = EDIS(\phi(X,Z),Z_j)$, in which Z_j is the set Z of variables associated with the output nodes less the output z_j. Viewing $\tilde{\phi}_j(X,z_j)$ as a function consisting of a single variable z_j, we form a general solution of $\phi(X,Z) = 1$ for Z:

$$\begin{array}{rcccl} \tilde{\phi}'_1(X,0) & \leq & z_1 & \leq & \tilde{\phi}_1(X,1) \\ \tilde{\phi}'_2(X,0) & \leq & z_2 & \leq & \tilde{\phi}_2(X,1) \\ \tilde{\phi}'_3(X,0) & \leq & z_3 & \leq & \tilde{\phi}_3(X,1) \\ & & \vdots & & \\ \tilde{\phi}'_m(X,0) & \leq & z_m & \leq & \tilde{\phi}_m(X,1). \end{array} \qquad (10.11)$$

A particular solution $Z = \underline{f}(X)$ is then developed using (10.11).

Since $\phi(X, Z) = 1$ specifies an implemented circuit, i.e., it was developed from a design rather than used as a vehicle for specifying a design, the upper and lower bounds of (10.11) are equal. Thus, the general solution represents a single particular solution. It follows that Z is defined by the system

$$\begin{aligned} z_1 &= \tilde{\phi}_1(X, 1) \\ z_2 &= \tilde{\phi}_2(X, 1) \\ &\vdots \\ z_m &= \tilde{\phi}_m(X, 1). \end{aligned} \tag{10.12}$$

Since $Z = \underline{f}(X)$, $\underline{f}(X)$ is defined by the system

$$\begin{aligned} f_1(X) &= \tilde{\phi}_1(X, 1) \\ f_2(X) &= \tilde{\phi}_2(X, 1) \\ &\vdots \\ f_m(X) &= \tilde{\phi}_m(X, 1). \end{aligned} \tag{10.13}$$

Algorithm 10.2 (Type 1 Problem) implements a procedure which returns a vector $\underline{f}(X)$ of functions.

Example 10.1 demonstrates the application of Algorithm 10.2.

Algorithm 10.2 (Type 1 Problem): Given the vectors $\underline{g}(X)$ and $\underline{h}(X, Y)$ of functions, the vector $\underline{f}(X)$ of functions is determined in the following manner:

Step 1. Using Boolean reduction, derive an equivalent 1-normal form $\xi(X, Y) = 1$ for the system $Y = \underline{g}(X)$.

Step 2. Using Boolean reduction, derive an equivalent 1-normal form $\lambda(X, Y, Z) = 1$ for the system $Z = \underline{h}(X, Y)$.

Step 3. Form $\psi(X, Y, Z) = \xi(X, Y) \cdot \lambda(X, Y, Z)$.

Step 4. Form $\phi(X, Z) = EDIS(\psi(X, Y, Z), Y)$.

Step 5. Letting Z_j be the set of Z-variables less the variable z_j, perform the following actions for each z_j:

1. Derive $\widetilde{\phi}_j(X, z_j) = EDIS(\phi(X, Z), Z_j)$.
2. Form $z_j = \widetilde{\phi}_j(X, 1)$.

Step 6. The system $Z = \underline{f}(X)$ was formed in Step 5, in which $f_j(X) = \widetilde{\phi}_j(X, 1)$. Return $\underline{f}(X)$.

Example 10.1: Let $Y = \underline{g}(X)$ be defined by the system

$$\begin{aligned} y_1 &= x_1' x_2' \\ y_2 &= x_2' x_3 \end{aligned} \tag{10.14}$$

of equations. Moreover, let $z = h(X, Y)$ be defined by the equation

$$z = y_1 y_2' + x_4 y_1. \tag{10.15}$$

Our goal is to determine the function $f(X, Y)$.

Step 1. Using Boolean reduction, we derive a 1-normal form $\xi(X, Y) = 1$ which is equivalent to $Y = \underline{g}(X)$. $\xi(X, Y)$ is defined by the equation

$$\xi(X, Y) = x_1 x_3' y_1' y_2' + x_2 y_1' y_2' + x_1 x_2' x_3 y_1' y_2 + x_1' x_2' x_3' y_1 y_2' + x_1' x_2' x_3 y_1 y_2. \tag{10.16}$$

Step 2. Using Boolean reduction, we derive an equivalent 1-normal form $\lambda(X, Y, z) = 1$ for the equation $z = \underline{h}(X, Y)$. $\lambda(X, Y, z)$ is defined by the equation

$$\lambda(X, Y, z) = y_1' z' + x_4' y_2 z' + y_1 y_2' z + x_4 y_1 z. \tag{10.17}$$

Step 3. Form $\psi(X,Y,z) = \xi(X,Y) \cdot \lambda(X,Y,z)$.

$$\begin{aligned}\psi(X,Y,z) &= x_1x_3'y_1'y_2'z' + x_2y_1'y_2'z' + x_1x_2'x_3y_1'y_2z' + \\ &\quad x_1'x_2'x_3'y_1y_2'z + x_1'x_2'x_3x_4'y_1y_2z' + x_1'x_2'x_3x_4y_1y_2z.\end{aligned} \tag{10.18}$$

Step 4. We form $\phi(X,z) = EDIS(\psi(X,Y,z),Y)$:

$$\phi(X,z) = x_1z' + x_2z' + x_3x_4'z' + x_1'x_2'x_3'z + x_1'x_2'x_4z. \tag{10.19}$$

Step 5. Since $z = h(X,Y)$ corresponds to a single-output circuit, we simply form $z = \phi(X,1)$. We thus develop the equation

$$z = x_1'x_2'x_4 + x_1'x_2'x_3'. \tag{10.20}$$

Step 6. The equation $z = f(X)$ was formed in Step 5, for which $f(X) = \phi(X,1)$. We return $f(X) = x_1'x_2'x_4 + x_1'x_2'x_3'$.

Type 2 Problems. Type 1 problems are of greatest utility in the analysis of existing designs. On the other hand, Type 2 problems are very useful in developing new designs. In a Type 2 problem, we are given the vector $\underline{g}(X)$ of functions and the set $[\underline{f}^l(X), \underline{f}^u(X)]$ of intervals and must develop a minimal vector $\underline{H}$ of formulas to represent the vector $\underline{h}(X,Y)$ of functions. An application of a Type 2 problem is the use of existing signals from previously-constructed subcircuits in the development of a new circuit which contains the subcircuits. Taking advantage of the existing signals, the new circuits may be more economical than otherwise possible.

In the first step of our methodology, we use the information given about the problem to form $\psi(X,Y,Z)$. We are initially given the systems $Y = \underline{g}(X)$ and the set $[\underline{f}^l(X), \underline{f}^u(X)]$ of intervals. The system $Y = \underline{g}(X)$ is used to form the equation $\xi(X,Y) = 1$, in which $\xi(X,Y)$ is defined as the

left-hand side of equation (10.6). If we are not given the global circuit specification $\phi(X, Z) = 1$ at the outset, then it is formed from the set $[\underline{f}^l(X), \underline{f}^u(X)]$ of intervals in which each z_j is bounded by the interval $[f_j^l(X), f_j^u(X)]$. Thus, Z is defined by the vector $[\underline{f}^l(X), \underline{f}^l(X)]$ of intervals. The function $\phi(X, Z)$ is formed as demonstrated by (4.91) through (4.94) in Chapter 4. The equation $\psi(X, Y, Z) = 1$ is formed by combining equation $\xi(X, Y) = 1$ and $\phi(X, Z) = 1$; it follows that $\psi(X, Y, Z)$ in a Type 2 problem is defined by the equation

$$\psi(X, Y, Z) = \xi(X, Y) \cdot \phi(X, Z). \tag{10.21}$$

In this problem, we partition the X-variables into two blocks. The first block is the subset X_1 of the X-variables which designate input nodes for the circuits corresponding to the system $Y = \underline{g}(X)$ and which are not inputs nodes for the circuit corresponding to $Z = \underline{h}(X, Y)$. In other words, the functions $\underline{h}(X, Y)$ are not dependent on any variables in X_1. The second block, X_2, comprises the X-variables less the variables in X_1. The *1-normal* form required for a Type 2 problem is $\lambda(X_2, Y, Z) = 1$, for which $\lambda(X_2, Y, Z)$ is defined by the equation

$$\lambda(X_2, Y, Z) = EDIS(\psi(X, Y, Z), X_1). \tag{10.22}$$

Formation of the equation $\lambda(X_2, Y, Z) = 1$ accomplishes the second step in our methodology for a Type 2 problem.

After forming $\lambda(X_2, Y, Z) = 1$, we use $\lambda(X_2, Y, Z) = 1$ as a specification for developing a minimal design. Hence, $\lambda(X_2, Y, Z) = 1$ is used as the circuit specification when applying Algorithms 7.1 and 7.2. An equation $\phi(X, Z) = 1$ is normally used as the specification in Algorithms 7.1 and 7.2. When using $\lambda(X_2, Y, Z) = 1$ rather than $\phi(X, Z) = 1$, the union of the sets X_2 and Y corresponds to the X-variables and the set Z forms the Z-variables in the two algorithms. (Given a specification $\lambda(X_2, Y, z) = 1$, one of Algorithms 6.1, 6.2, or 6.3 is used to develop a minimal

design.) Algorithm 10.3 (Type 2 Problem) returns a minimal vector $\underline{H}$ of formulas to represent the vector $\underline{h}(X,Y)$ of functions. Example 10.2 demonstrates an application of Algorithm 10.3.

Algorithm 10.3 (Type 2 Problem): Given the functions $\underline{g}(X)$ and the set $[\underline{f}^l(X), \underline{f}^u(X)]$ of intervals, a minimal vector $\underline{H}$ of formulas is developed to represent the vector $\underline{h}(X,Y)$ of functions in the following manner:

Step 1. Using Boolean reduction, derive an equivalent 1-normal form $\xi(X,Y) = 1$ for the system $Y = \underline{g}(X)$.

Step 2. Using Boolean reduction, derive an equivalent 1-normal form $\phi(X,Z) = 1$ for the set $\underline{f}^l(X) \leq Z \leq \underline{f}^l(X)$ of intervals.

Step 3. Form $\psi(X,Y,Z) = \xi(X,Y) \cdot \phi(X,Z)$.

Step 4. Partition the X-variables into subsets X_1 and X_2:

1. Let X_1 be the subset of the X-variables which designate input nodes for the circuits corresponding to the system $Y = \underline{g}(X)$ and which are not inputs nodes for design corresponding to $Z = \underline{h}(X,Y)$.
2. Let X_2 denote the X-variables less the variables in X_1.

Step 5. Form $\lambda(X_2,Y,Z) = EDIS(\psi(X,Y,Z), X_1)$.

Step 6.

- Given a 1-normal form specification $\lambda(X_2,Y,z) = 1$, use one among Algorithm 6.1, Algorithm 6.2, or Algorithm 6.3 to develop a minimal formula H. The union of the sets X_2 and Y corresponds to the X-variables and z is the Z-variable in the algorithms.
- Given a 1-normal form specification $\lambda(X_2,Y,Z) = 1$, use either Algorithm 7.1 or Algorithm 7.2 to develop a minimal vector $\underline{H}$ of formulas. The union of the sets X_2 and Y corresponds to the X-variables and the set Z forms the Z-variables in the two algorithms.

Example 10.2: Let $Y = \underline{g}(X)$ by defined by the system

$$\begin{aligned} y_1 &= x_1x_2' + x_1'x_2 \\ y_2 &= x_2x_3 \end{aligned} \tag{10.23}$$

of equations. Moreover, let $f^l(X) \leq z \leq f^u(X)$ be defined by the interval

$$x_2x_3 + x_1x_2'x_3'x_4' + x_1'x_2'x_3'x_4 \leq z \leq x_2x_3 + x_1'x_2' + x_1x_2 + x_1x_3'x_4'. \tag{10.24}$$

Suppose we would like to enforce the condition that the function $h(X,Y)$ in the equation $z = h(X,Y)$ is dependent only on the variables x_4, y_1 and y_2. Our goal is to determine the function $h(x_4, y_1, y_2)$.

Step 1. Using Boolean reduction, we derive an equivalent 1-normal form $\xi(X,Y) = 1$ for $Y = \underline{g}(X)$. $\xi(X,Y)$ is defined by the equation

$$\begin{aligned}\xi(X,Y) \;=\; & x_1'x_2'y_1'y_2' + x_1'x_2x_3'y_1y_2' + x_1'x_2x_3y_1y_2 + \\ & x_1x_2'y_1y_2' + x_1x_2x_3'y_1'y_2' + x_1x_2x_3y_1'y_2.\end{aligned} \tag{10.25}$$

Step 2. Using Boolean reduction, we derive an equivalent 1-normal form $\phi(X,z) = 1$ for the interval $f^l(X) \leq z \leq f^u(X)$. $\phi(X,z)$ is defined by the equation

$$\begin{aligned}\phi(X,z) \;=\; & x_1x_2'x_4z' + x_2'x_3z' + x_2'x_3'x_4'z + x_1'x_2'z + x_1'x_2'x_4' + \\ & x_1'x_2'x_3 + x_2x_3'z' + x_1x_2x_3' + x_2x_3z + x_1x_2z + \\ & x_1x_3'x_4z' + x_1x_3'x_4'z + x_1'x_3z + x_1'x_3'x_4'z'.\end{aligned} \tag{10.26}$$

Step 3. We form $\psi(X,Y,z) = \xi(X,Y) \cdot \phi(X,z)$:

$$\begin{aligned}\psi(X,Y,z) \;=\; & x_1'x_2'y_1'y_2'z + x_1'x_2'x_3y_1'y_2' + x_1'x_2'x_4'y_1'y_2' + \\ & x_1'x_2x_3'y_1y_2'z' + x_1'x_2x_3y_1y_2z + x_1x_2'x_3y_1y_2'z' + \\ & x_1x_2'x_4y_1y_2'z' + x_1x_2'x_3'x_4'y_1y_2'z + x_1x_2x_3'y_1'y_2' + x_1x_2x_3y_1'y_2z.\end{aligned} \tag{10.27}$$

Step 4. We partition the X-variables into the subsets X_1 and X_2. Since $h(x_4, y_1, y_4)$ is only dependent on x_4, X_1 is equal to $\{x_1, x_2, x_3\}$. Then, X_2 equals $\{x_4\}$.

Step 5. We form $\lambda(x_4, y_1, y_2, z) = EDIS(\psi(X,Y,z), X_1)$:

$$\lambda(x_4, y_1, y_2, z) = y_1'y_2' + y_1'z + y_2'z' + x_4'z + y_2z + x_4'y_2'. \tag{10.28}$$

Step 6. A general solution for z of $\lambda(x_4, y_1, y_2, z) = 1$ is developed and used to form a minimal formula H. We develop the interval

$$y_2 \leq z \leq y_2 + y_1' + x_4'. \tag{10.29}$$

Clearly, we may let $h = y_2$. We thus develop the equation $z = y_2$ as the solution for this Type 2 problem.

Now suppose we change the problem so that $g(X)$ and $h(X, Y)$ are dependent on the same set of X-variables. The modified problem is different from the original problem starting with Step 4.

Step 4. We partition the X-variables into the subsets X_1 and X_2. Since $h(X, Y)$ is dependent on all of the X-variables, $X_1 = \emptyset$. Then, $X_2 = \{x_1, x_2, x_3, x_4\}$.

Step 5. $\lambda(X, Y, z)$ is equal to $\psi(X, Y, z)$.

Step 6. A general solution of $\lambda(X, Y, z) = 1$ for z is developed and used to form a minimal formula H. We develop an interval in which the lower bound is equal to

$$x_1' x_2 x_3 y_1 y_2 + x_1' x_2' x_3' x_4 y_1' y_2' + x_1 x_2' x_3' x_4' y_1 y_2' + x_1 x_2 x_3 y_1' y_2 \tag{10.30}$$

The upper bound in Blake canonical form is equal to the formula

$$x_1' x_2' + x_2' x_3' x_4' + x_1 x_2 + x_2 x_3 + x_1' x_3 + y_1' + x_1 x_3' x_4' + y_2. \tag{10.31}$$

To develop a minimal formula H, a minimal subset of terms in (10.31) is formed to cover the terms in (10.30). There exist 12 irredundant formulas containing terms in (10.31) which cover terms in (10.30):

$$
\begin{array}{l}
x_2'x_3'x_4' + x_1'x_2' + x_2x_3 \\
x_1x_3'x_4' + x_1'x_2' + x_2x_3 \\
x_2'x_3'x_4' + y_1' + y_2 \\
x_2'x_3'x_4' + x_1'x_2' + x_1'x_3 + x_1x_2 \\
x_1x_3'x_4' + x_1'x_2' + x_1'x_3 + x_1x_2 \\
x_2'x_3'x_4' + y_1' + x_2x_3 \\
x_2'x_3'x_4' + x_1'x_3 + y_1' \\
x_1x_3'x_4' + y_1' + x_2x_3 \\
x_1x_3'x_4' + y_1' + y_2 \\
x_2'x_3'x_4' + x_1'x_2' + y_2 \\
x_1x_3'x_4' + x_1'x_2' + y_2 \\
x_1x_3'x_4' + x_1'x_3 + y_1'.
\end{array}
\tag{10.32}
$$

Of the irredundant formulas, 10 contain three terms and two contain four terms. Any formula containing only three terms may be chosen as formula H. Choosing the formula $H = x_2'x_3'x_4' + y_1' + y_2$, we thus develop the equation $z = x_2'x_3'x_4' + y_1' + y_2$ as a solution for the given Type 2 problem.

An application of the Type 2 problem is the use of existing signals from previously-constructed circuits to aid in the development of new circuits. Taking advantage of the existing signals, the new circuits may be more economical than otherwise possible. In essence, the availability of the existing signals yields a greater number of options which facilitate greater minimization. Hence, rather than implementing a circuit corresponding to the equation $Z = \underline{f}(X)$, we may use a circuit corresponding to the equation $Z = \underline{h}(X, Y)$. The choice depends on whether the design corresponding to $\underline{H}$ is cheaper than the design corresponding to $\underline{F}$. Figure 10.2 depicts the application of the Type 2 problem in which we use existing signals to develop a new design. Note that in this application, we treat Y-variables as both outputs and as intermediate nodes.

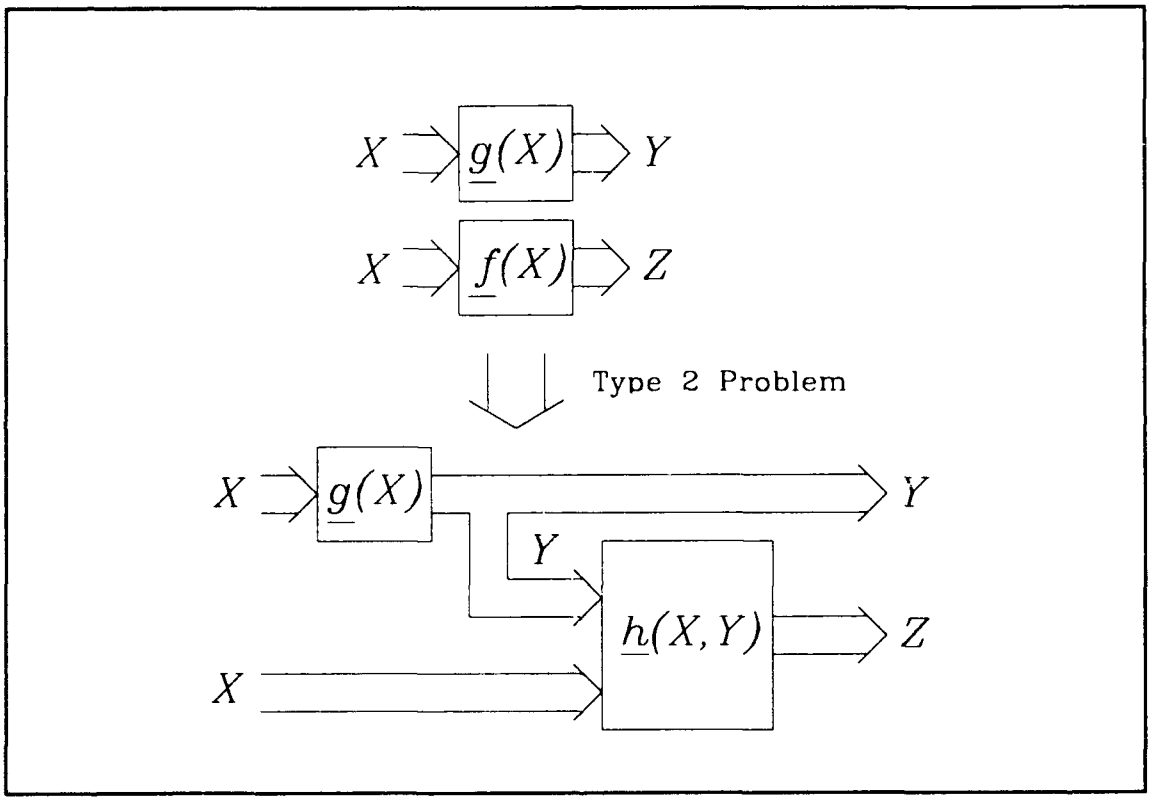


Figure 10.2. An Application of the Type 2 Problem

For this application, we assume that we are given a 1-normal form specification $\phi(X, Z) = 1$ for the portion of the circuit to be designed. Furthermore, there exists a circuit which corresponds to the equation $Y = \underline{g}(X)$. The equation $Y = \underline{g}(X)$ is reduced to an equivalent 1-normal form $\xi(X, Y) = 1$. The equations $\phi(X, Z) = 1$ and $\xi(X, Y) = 1$ are combined to form the equation $\psi(X, Y, Z) = 1$. $\psi(X, Y, Z) = 1$ is then used as the specification rather than $\phi(X, Z) = 1$ when applying the algorithms presented in earlier chapters. Example 10.3 demonstrates this application of the Type 2 problem.

Example 10.3: Suppose we are given a circuit specification $\phi(X, z) = 1$, in which $\phi(X, z)$ is represented by the formula

$$x_2'x_4'z' + x_3'x_4'z' + x_1'x_4'z' + x_1x_2x_3x_4'z + x_1x_2x_3x_4z' + x_1'x_4z + x_2'x_4z + x_3'x_4z. \tag{10.33}$$

To form a design, we develop a minimal F to represent $f(X)$ in the equation $z = f(X)$. We first form a general solution of $\phi(X, z) = 1$ for z; such a solution is formed by the interval $\phi'(X, 0) \leq z \leq \phi(X, 1)$. In this case, we develop the interval

$$x_1x_2x_3x_4' + x_1'x_4 + x_2'x_4 + x_3'x_4 \leq z \leq x_1x_2x_3x_4' + x_1'x_4 + x_2'x_4 + x_3'x_4. \tag{10.34}$$

The upper and lower bounds are equal in (10.34); furthermore, each formula is a Blake canonical form. We observe that no term may be deleted from either formula; hence, a minimal F is given by the formula $x_1x_2x_3x_4' + x_1'x_4 + x_2'x_4 + x_3'x_4$. Using the fewest-gates cost criterion, it follows that a least-cost design consists of four AND gates.

Now suppose there exists a circuit $y = g(X)$, $g(X) = x_1x_2x_3$; we may freely use node y in the development of a new circuit. We employ the circuit $y = g(X)$ to develop a minimal formula H

which represents a function $h(X, y)$. The equation $z = h(X, y)$ corresponds to a circuit which may be used in place of the circuit corresponding to $z = f(X)$. The function $\xi(X, y)$ in the equivalent 1-normal form $\xi(X, y) = 1$ for $y = g(X)$ is defined by the equation

$$\xi(X, y) = x_1'y' + x_1x_2x_3y + x_2'y' + x_3'y'. \tag{10.35}$$

Using $\phi(X, z)$ and $\xi(X, y)$, the function $\psi(X, y, z)$ is formed by the product of $\phi(X, z)$ and $\xi(X, y)$:

$$\begin{aligned} \psi(X, y, z) \quad = \quad & x_1'x_4'y'z' + x_2'x_4'y'z' + x_3'x_4'y'z' + x_1x_2x_3x_4'yz + \\ & x_2'x_4y'z + x_1'x_4y'z + x_3'x_4y'z + x_1x_2x_3x_4yz'. \end{aligned} \tag{10.36}$$

Once we form $\psi(X, y, z)$, we develop a general solution of $\psi(X, y, z) = 1$ for z. The lower bound for z is defined by the formula

$$x_1x_2x_3x_4'y + x_3'x_4y' + x_2'x_4y' + x_1'x_4y'. \tag{10.37}$$

Additionally, the Blake canonical form of the upper bound for z is defined by the formula

$$x_4'y + x_1x_2x_3x_4' + x_1'x_4 + x_3'x_4 + x_4y' + x_2'x_4 + x_1'y + x_2'y + x_1x_2x_3y' + x_3'y. \tag{10.38}$$

To develop a minimal formula H for the function $h(X, y)$, a minimal subset of terms in (10.38) is formed to cover the terms in (10.37). Upon examination of both of the formulas, we determine that the term x_4y' in (10.38) covers the last three terms in (10.37). The first term in (10.37) is covered either by $x_4'y$ or $x_1x_2x_3x_4'$ from (10.38). Hence, two alternative minimal formulas H are

$$\begin{aligned} &x_4y' + x_4'y \\ &x_4y' + x_1x_2x_3x_4'. \end{aligned} \tag{10.39}$$

Using the node y, which is the output of $y = g(X)$, facilitates a design consisting of two gates. Thus, using y yields a saving of two gates and, hence, a 50% cost savings over a design developed which did not consider the y node.

In the application of the Type 2 problem illustrated by Figure 10.2, a node y_k of the circuit $Y = \underline{g}(X)$ will generally be useful only if the X-variables in the corresponding function $g_k(X)$ are a subset of the X-variables contained in at least one function $f_j(X)$ of $\underline{f}(X)$.

Type 3 Problems. The third problem presented by Ledley is the Type 3 problem, which is handled in a similar fashion as the Type 2 problem. In a Type 3 problem, the vector $\underline{h}(X, Y)$ of functions and the set $[\underline{f}^l(X), \underline{f}^u(X)]$ of intervals are given and we must develop a minimal vector $\underline{G}$ of formulas to represent the functions $\underline{g}(X)$.

We first use the information given about the problem to form $\psi(X, Y, Z)$. In a Type 3 problem, we are initially given the system $Z = \underline{h}(X, Y)$ and the set $[\underline{f}^l(X), \underline{f}^u(X)]$ of intervals. The set $[\underline{f}^l(X), \underline{f}^u(X)]$ of intervals is used to develop $\phi(X, Z) = 1$ as described in the section on the Type 2 problem. The system $Z = \underline{h}(X, Y)$ is used to form the equation $\lambda(X, Y, Z) = 1$, in which $\lambda(X, Y, Z)$ is defined as the left-hand side of equation (10.8). The 1-normal form $\psi(X, Y, Z) = 1$ is derived by combining $\phi(X, Z) = 1$ and $\lambda(X, Y, Z) = 1$. Hence, $\psi(X, Y, Z)$ in a Type 3 problem is defined by the equation

$$\psi(X,Y,Z) = \lambda(X,Y,Z) \cdot \phi(X,Z). \tag{10.40}$$

We again partition the X-variables into two blocks. In this problem, however, the first block is the subset X_1 of the X-variables which designate input nodes for the circuits corresponding to the system $Z = \underline{h}(X,Y)$ and which are not inputs for design corresponding to $Y = \underline{g}(X)$. In other words, the functions $\underline{g}(X)$ are not dependent on any variables in X_1. The second block—X_2—is the X-variables except those variables contained in X_1. The 1-normal form required for a Type 3 problem is $\xi(X_2,Y) = 1$. The function $\xi(X_2,Y)$ is defined by the equation

$$\xi(X_2,Y) = EDIS(\psi(X,Y,Z), X_1 \cup Z). \tag{10.41}$$

Once we form the equation $\xi(X_2,Y) = 1$, the second step in our methodology is accomplished for a Type 3 problem.

The equation $\xi(X_2,Y) = 1$ is used as a specification for developing a minimal design. The system $\xi(X_2,Y) = 1$ is used as the circuit specification when applying Algorithms 7.1 and 7.2. When using $\xi(X_2,Y) = 1$ rather than $\phi(X,Z) = 1$ in the two algorithms, the set X_2 corresponds to the X-variables and the set Y correspond to the Z-variables. (For a system $\xi(X_2,y) = 1$, one among Algorithm 6.1, Algorithm 6.2, or Algorithm 6.3 is used to develop a minimal design.)

Unlike Type 1 and Type 2 problems, a Type 3 problem introduces a possible dilemma which must be considered in development of a solution. The function $EDIS(\xi(X_2,Y),Y)$ in the consistency condition $EDIS(\xi(X_2,Y),Y) = 1$ for the general solution of $\xi(X_2,Y) = 1$ for Y may not be identically equal to 1. This is due to the fact that it is not always possible to find an equation $Y = \underline{g}(X_2)$ such that $\xi(X_2,\underline{g}(X_2)) = 1$ is an identity. This situation may occur due to the fact that we cannot always make a circuit corresponding to $Y = \underline{g}(X_2)$ "fit" into the position preceding the circuit corresponding to $Z = \underline{h}(X,Y)$ in Figure 10.1.

If $EDIS(\xi(X_2, Y), Y)$ is not identically equal to 1, then the equation $\xi(X_2, Y) = 1$ is a constrained equation. A constrained solution (defined in Chapter 4) may be formed providing that the condition $EDIS(\xi(X_2, Y), Y) = 1$ is satisfied. Alternatively, we define a function $\xi_C(X_2)$:

$$\xi_C(X_2) = (EDIS(\xi(X_2, Y), Y))'. \tag{10.42}$$

We may then develop a general solution of $\xi(X_2, Y) = 1$ for Y if and only if the condition

$$\xi_C(X_2) = 0 \tag{10.43}$$

is satisfied. Such a condition is satisfied if the input combinations corresponding to minterms of $\xi_C(X_2)$ will not occur. Assuming that equation (10.43) is satisfied, input combinations corresponding to the minterms of $\xi_C(X_2)$ may be treated as don't cares and are thus useful in the minimization process. Algorithm 10.4 (Type 3 Problem) returns a minimal vector $\underline{G}$ of formulas to represent the vector $\underline{g}(X_2, Y)$ of functions.

Algorithm 10.4 (Type 3 Problem): Given the functions $\underline{h}(X, Y)$ and the set $[\underline{f}^l(X), \underline{f}^u(X)]$ of intervals, a minimal vector $\underline{G}$ of formulas is developed to represent the vector $\underline{g}(X)$ of functions in the following manner:

Step 1. Using Boolean reduction, derive an equivalent 1-normal form $\lambda(X, Y, Z) = 1$ for the system $Z = \underline{h}(X, Y)$.

Step 2. Using Boolean reduction, derive an equivalent 1-normal form $\phi(X, Z) = 1$ for the set $\underline{f}^l(X) \leq Z \leq \underline{f}^l(X)$ of intervals.

Step 3. Form $\psi(X, Y, Z) = \lambda(X, Y, Z) \cdot \phi(X, Z)$.

Step 4. Partition the X-variables into subsets X_1 and X_2:

1. Let X_1 be the subset of the X-variables which designate input nodes for the circuits corresponding to the system $Z = \underline{h}(X, Y)$ and which are not inputs nodes for design corresponding to $Y = \underline{g}(X)$.
2. Let X_2 denote the X-variables less the variables in X_1.

Step 5. Form $\xi(X_2, Y) = EDIS(\psi(X, Y, Z), X_1 \cup Z)$.

Step 6.

- Given a 1-normal form specification $\xi(X_2, y) = 1$, use one among Algorithm 6.1, Algorithm 6.2, or Algorithm 6.3 to develop a minimal formula G, in which the set X_2 corresponds to the X-variables and y corresponds to the variable z in the algorithms.
- Given a 1-normal form specification $\xi(X_2, Y) = 1$, use either Algorithm 7.1 or Algorithm 7.2 to develop a minimal vector $\underline{G}$ of formulas, in which the set X_2 corresponds to the X-variables and the set Y corresponds to the Z-variables in the two algorithms.

Step 7. Form the constraint condition $\xi_C(X_2) = 0$, for which $\xi_C(X_2) = (EDIS(\xi(X_2, Y), Y))'$. The function $\xi_C(X_2)$ is returned with the vector $\underline{G}$ of formulas developed in Step 6.

Summary of the Approach to Ledley's Problems. The foregoing discussion of the Types 1, 2, and 3 problems demonstrates the utility of the 1-normal form and the equation-solving approach for handling the design problem. Using 1-normal form specifications and solutions of Boolean equations, we are able to develop minimal designs for problems which cannot be handled using conventional minimization techniques. Additionally, whereas Ledley only presented ad hoc techniques for solving the problems that he proposed, our methodology yields minimal solutions for all three problems.

In the second part of this chapter, we once more demonstrate the utility of the 1-normal form and the equation-solving approach for developing designs for another type of circuit—recurrent circuits—which cannot be handled using conventional minimization techniques. Recurrent circuits are in many cases more economical than designs produced using conventional methods.

Recurrent Circuits

We showed in Chapter 4 that a specification corresponds to a 1-normal form $\phi(X, Z) = 1$ and that designs have a correspondence with particular solutions $Z = \underline{f}(X)$ of $\phi(X, Z) = 1$. In Chapter 7, we presented algorithms for developing a minimal vector $\underline{F}$ of formulas to represent the functions $\underline{f}(X)$ in the particular solution $Z = \underline{f}(X)$. The resulting designs are of the form depicted in Figure 10.3. In this chapter, we present techniques for constructing designs of the form shown in Figure 10.4, i.e., designs corresponding to formulas $\underline{F}(X, Z)$ which represents functions $\underline{f}(X, Z)$ in recurrent solutions $Z = \underline{f}(X, Z)$ of $\phi(X, Z) = 1$. We call such designs *recurrent* or *cascade* circuits. The method we present produces approximate-minimal recurrent circuits. The resulting circuits generally are of lower cost than those developed using conventional techniques.

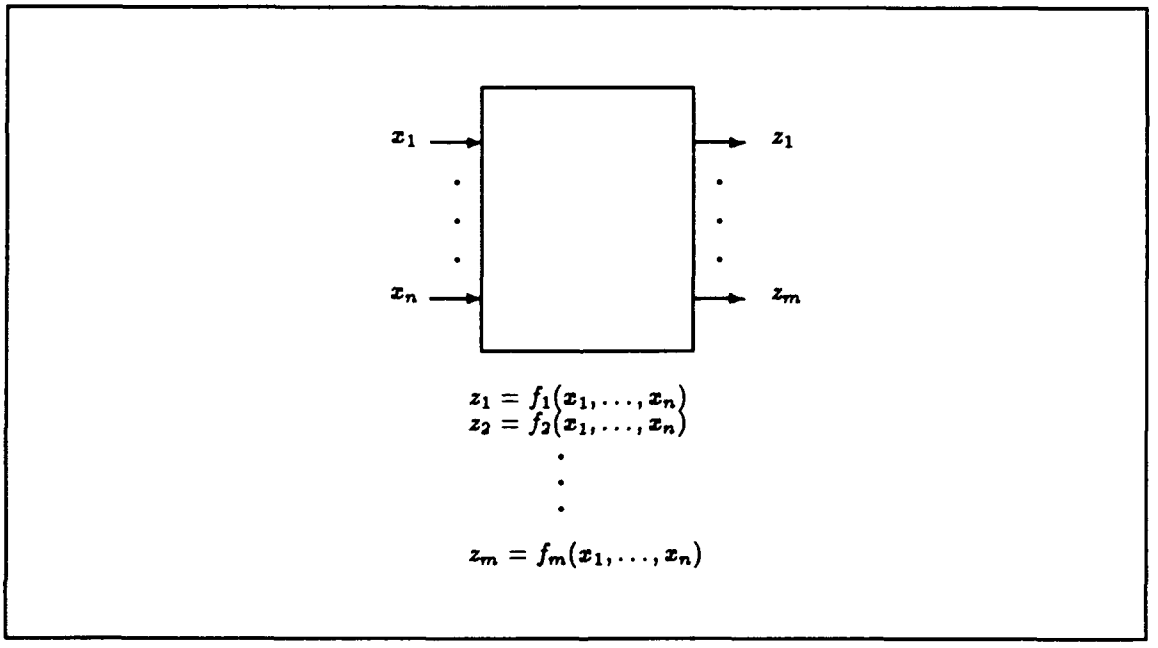


Figure 10.3. Representation of a Digital Circuit

Before presenting an algorithm for producing recurrent circuits, we first discuss the cost-advantage of recurrent designs versus non-recurrent ones.

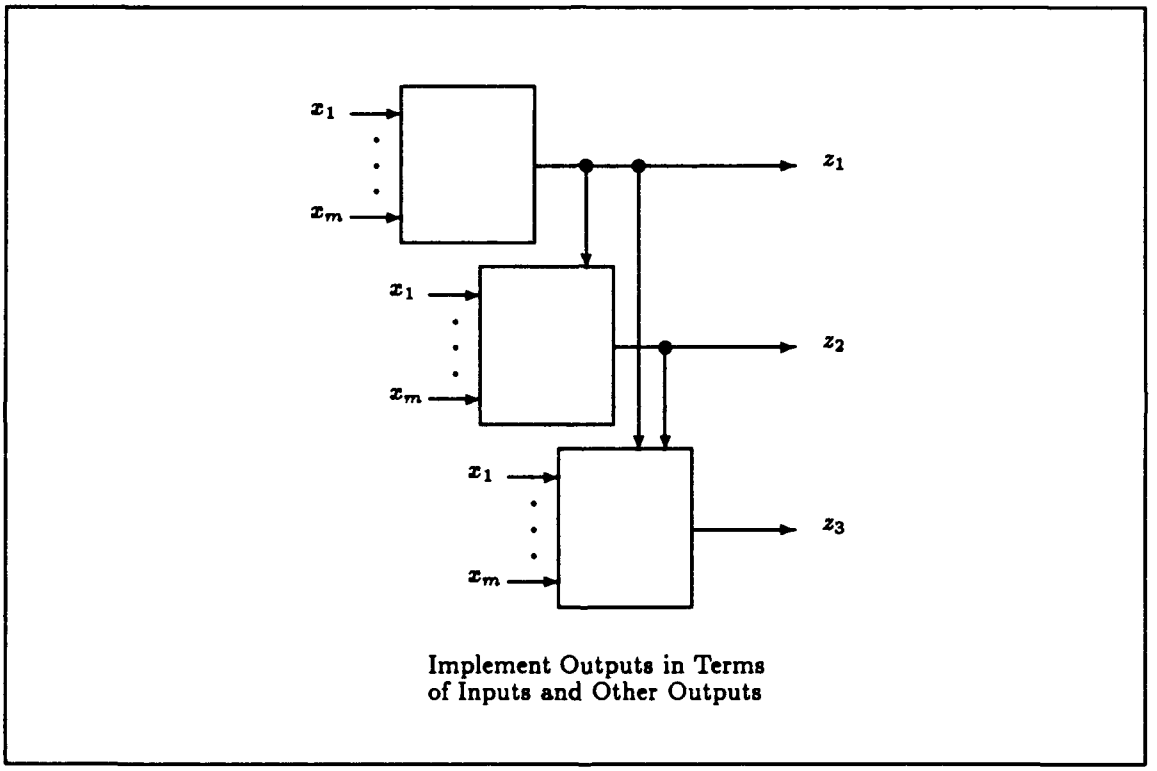


Figure 10.4. A Recurrent Design

The Advantage of Recurrent Designs. In conventional circuit minimization, a non-recurrent system such as

$$\begin{array}{rcccl}
\tilde{\phi}_1'(X,0)\cdot\tilde{\phi}_1(X,1) & \leq & z_1 & \leq & \tilde{\phi}_1'(X,0)+\tilde{\phi}_1(X,1) \\
\tilde{\phi}_2'(X,0)\cdot\tilde{\phi}_2(X,1) & \leq & z_2 & \leq & \tilde{\phi}_2'(X,0)+\tilde{\phi}_2(X,1) \\
\tilde{\phi}_3'(X,0)\cdot\tilde{\phi}_3(X,1) & \leq & z_3 & \leq & \tilde{\phi}_3'(X,0)+\tilde{\phi}_3(X,1) \\
 & & \vdots & & \\
\tilde{\phi}_m'(X,0)\cdot\tilde{\phi}_m(X,1) & \leq & z_m & \leq & \tilde{\phi}_m'(X,0)+\tilde{\phi}_m(X,1).
\end{array} \tag{10.44}$$

is used as the basis for developing a design. Using a non-recurrent system, each z_j is a function only of the input variables X. On the other hand, employing the method of successive eliminations to form a subsumptive general solution of $\phi(X,Z)=1$ for Z, we develop the recurrent system

$$\begin{array}{rcccl}
 & & 1 & \leq & \phi_0(X) \\
\phi_1'(X,0) & \leq & z_1 & \leq & \phi_1(X,1) \\
\phi_2'(X,z_1,0) & \leq & z_2 & \leq & \phi_2(X,z_1,1) \\
\phi_3'(X,z_1,z_2,0) & \leq & z_3 & \leq & \phi_3(X,z_1,z_2,1) \\
 & & \vdots & & \\
\phi_m'(X,z_1,\ldots,z_{m-1},0) & \leq & z_m & \leq & \phi_m(X,z_1,\ldots,z_{m-1},1).
\end{array} \tag{10.45}$$

In view of Theorem 4.4, we form a recurrent system based on the extended range concept:

$$\begin{array}{rcccl}
 & & 1 & \leq & \phi_0(X) \\
\phi_1'(X,0)\cdot\phi_1(X,1) & \leq & z_1 & \leq & \phi_1'(X,0)+\phi_1(X,1) \\
\phi_2'(X,z_1,0)\cdot\phi_2(X,z_1,1) & \leq & z_2 & \leq & \phi_2'(X,z_1,0)+\phi_2(X,z_1,1) \\
\phi_3'(X,z_1,z_2,0)\cdot\phi_3(X,z_1,z_2,1) & \leq & z_3 & \leq & \phi_3'(X,z_1,z_2,0)+\phi_3(X,z_1,z_2,1) \\
 & & \vdots & & \\
\phi_m'(X,z_1,\ldots,z_{m-1},0) & \leq & z_m & \leq & \phi_m'(X,z_1,\ldots,z_{m-1},0) \\
\cdot\phi_m(X,z_1,\ldots,z_{m-1},1) & & & & +\phi_m(X,z_1,\ldots,z_{m-1},1).
\end{array} \tag{10.46}$$

Finally, we derive from system (10.46) a recurrent solution

$$\begin{aligned} z_1 &= f_1(X) \\ z_2 &= f_2(X, z_1) \\ z_3 &= f_3(X, z_1, z_2) \\ &\vdots \\ z_m &= f_m(X, z_1, z_2, \ldots, z_{m-1}). \end{aligned} \tag{10.47}$$

System (10.47) is denoted by $Z = \underline{f}(X, Z)$ with the understanding that each z_j is dependent only on $z_1, \ldots, z_{j-1}$. The vector $\underline{F}(X, Z)$ which represents $\underline{f}(X, Z)$ in the recurrent solution $Z = \underline{f}(X, Z)$ corresponds to a recurrent design. Conventional minimization techniques produce only non-recurrent designs.

The advantage of using a recurrent system such as (10.46) as the basis for the design process is that a design may be developed in which we use output nodes as well as input nodes to generate a given output node. When we can use the output nodes to form inputs, we can generate designs which cost less than those developed using conventional techniques. Example 10.4 illustrates this point.

Example 10.4: Suppose we are given the 1-normal form specification $\phi(X, Z) = 1$ for which we must develop a design. Let $\phi(X, Z)$ be defined by the equation

$$\begin{aligned} \phi(X, Z) = {} & x_1'x_2'x_3'x_4'z_1'z_2' + x_1'x_2'x_3'x_4z_1'z_2'z_3 + x_1'x_2'x_3x_4'z_2'z_3 + x_1'x_2'x_3x_4z_1z_2'z_3' + \\ & x_1'x_2x_3'x_4'z_1z_2z_3' + x_1'x_2x_3'x_4z_1z_2z_3' + x_1'x_2x_3x_4'z_1'z_2'z_3 + \qquad (10.48) \\ & x_1x_2'x_3'x_4'z_3' + x_1x_2'x_3'x_4z_1z_3' + x_1x_2'x_3x_4'z_1'z_2' + x_1x_2'x_3x_4z_1'z_2'z_3 + \\ & x_1x_2x_3'x_4'z_1'z_2'z_3 + x_1x_2x_3'x_4z_1'z_3 + x_1x_2x_3x_4'z_2'z_3 + x_1x_2x_3x_4z_2z_3' + \qquad (10.49) \\ & x_1'x_2x_3x_4z_1'z_2'z_3. \end{aligned}$$

Using a conventional approach, we first develop a general solution of $\phi(X, Z) = 1$ for Z of the form (10.44):

$$\begin{array}{rcccl}
x_1x_2'x_3'x_4 + x_1'x_2'x_3x_4 + x_1'x_2x_3' & \leq & z_1 & \leq & x_1'x_2'x_3 + x_1'x_2x_3' + x_1x_2'x_3' + x_1x_2x_3 \\
x_1'x_2x_3' + x_1x_2x_3x_4 & \leq & z_2 & \leq & x_1x_2'x_3' + x_2x_3'x_4 \\
 & & & & +x_1x_2x_4 + x_1'x_2x_3' + x_1x_3'x_4 \\
x_1'x_3x_4' + x_1'x_2x_3 + x_1'x_2'x_3'x_4 & \leq & z_3 & \leq & x_1'x_2'x_4' + x_1'x_2x_3 + x_1'x_2'x_3' \\
+x_1x_2x_4' + x_1x_2x_3' + x_1x_2'x_3x_4 & & & & +x_1x_2x_3' + x_1x_2x_4' + x_1x_2'x_3 + x_3x_4'.
\end{array} \tag{10.50}$$

Using the methodology discussed in Chapter 7, a minimal $\underline{F}$ may be developed to represent functions $\underline{f}(X)$ in the particular solution $Z = \underline{f}(X)$ of $\phi(X, Z) = 1$. One such particular solution is the system:

$$\begin{array}{rcl}
z_1 & = & x_1'x_2'x_3 + x_1'x_2x_3' + x_1x_2'x_3' \\
z_2 & = & x_1x_2x_4 + x_1'x_2x_3' \\
z_3 & = & x_1'x_2x_3 + x_1'x_2'x_3' + x_1x_2x_3' + x_1x_2'x_3 + x_3x_4'.
\end{array} \tag{10.51}$$

Using the fewest-gates cost criterion, the cost of the resulting design is the number of distinct terms on the right-hand side of (10.51); hence, the cost of $\underline{F}$ is nine gates.

Now suppose we form a subsumptive general solution of $\phi(X, Z) = 1$ for Z in which we use the sequence (z_2, z_1, z_3) to construct the solution. We develop the following bounds for the Z-variables:

z_3 Lower bound:

$$x_1x_2x_3' + x_1'x_2'x_3'x_4 + x_1'x_2x_3 + x_1'x_3x_4' + x_1x_2'x_3x_4 + x_2x_3x_4'$$

z_3 Upper bound:

$$x_1'x_2'x_3' + x_1x_2x_3' + x_1'x_2x_3 + x_1x_2'x_3 + x_3x_4' + x_1'x_2'x_4' + x_1x_2x_4'$$

z_1 Lower bound:

$$x_1'x_2x_3'z_3' + x_1'x_2'x_3x_4z_3' + x_1x_2'x_3'x_4z_3'$$

z_1 Upper bound:

$$x_1'x_2'x_3 + x_1x_2'x_3' + x_2z_3' + x_1x_2x_3 + x_1'x_2x_3' + x_1x_3'z_3' + x_4z_3' + x_1'x_3z_3'$$

z_2 Lower bound:

$$x_1'x_2x_3'z_1z_3' + x_1x_2x_3x_4z_3'$$

z_2 Upper bound:

$$\begin{aligned} & x_1'x_2'x_3x_4z_1' + x_1'x_2'x_3x_4z_3 + x_1'x_3z_1'z_3' + x_1'x_3x_4'z_3' + x_1'x_2z_1 \\ & + x_1x_2'x_3' + x_1x_2'z_1 + x_1x_2x_4 + x_1x_4z_3' + x_1x_3'z_3' + x_1x_4z_1 + x_1z_1z_3' + x_1x_3'x_4 \\ & + x_3'x_4z_3' + x_2x_3'x_4 + x_3'z_1 + x_4z_1z_3 + x_2x_4z_1 + x_4'z_1z_3' + x_4z_1'z_3' + x_2z_3'. \end{aligned}$$

A suitable design F_j for each z_j is one which contains a subset of the terms of the upper bound which covers all of the terms of the lower bound. Using the bounds developed for a subsumptive general solution of $\phi(X, Z) = 1$ for Z, we can develop the following particular solution $Z = \underline{f}(X, Z)$ of $\phi(X, Z) = 1$:

$$\begin{aligned} z_1 &= x_2z_3' + x_4z_3' \\ z_2 &= x_2z_3' \\ z_3 &= x_1'x_2x_3 + x_1'x_2'x_3' + x_1x_2x_3' + x_1x_2'x_3 + x_3x_4'. \end{aligned} \tag{10.52}$$

Using the fewest-gates cost criterion, the cost of the design depicted by (10.52) is seven gates. Thus, for the given ordering of the Z-variables, two fewer gates are required when using the recurrent approach versus a conventional methodology.

As alluded to in Example 10.4, the cost of the resulting design is dependent on the ordering of the Z-variables used when forming the subsumptive general solution. One ordering of Z-variables may yield a significant cost savings, while another ordering may provide little cost savings relative to using a conventional methodology. We address this issue in the next section.

Sequence of the Z-Variables. The ordering of the Z-variables used when forming a subsumptive general solution may significantly affect the cost savings realized when using the recurrent methodology rather than a conventional approach. Ideally, we would always choose the sequence of Z-variables which leads to a least-cost recurrent design. Unfortunately, we know of no scheme which always produces a "best" ordering of variables. To guarantee the production of a minimal recurrent design, we would have to construct a design for all possible orderings—an approach which is intractable for all but trivial problems. We therefore must rely on heuristics which produce an ordering of the Z-variables which will likely bear a cost-savings when producing a design using the recurrent approach. This is the reason why designs developed using the recurrent approach are only approximate-minimal.

One consideration that may affect the choice of ordering of the output variables is the delay of each output node. The delay of an output node z_j may be estimated by measuring by the longest path with respect to the number of gates between one of the inputs x_i on which z_j depends and z_j. If an output node z_1 is used as an input for the output node z_2, there will generally be an increased delay for node z_2. To determine the delay of z_2, we then must consider the number of gates between an x_i and z_1 and then between z_1 and z_2. Hence, the disadvantage in using z_1 to create z_2 is an increased delay for z_2. This is an example of the classic time versus space dilemma: we increase circuit delay to decrease circuit cost. In some situations, however, it may be imperative that we do not increase the delay for a specific node z_j. In these cases, we specify an ordering such that z_j is produced using only input nodes x_i.

Although the determination of a good order for the Z-variables is an important consideration, we will not address further the issue of selecting an order for the Z-variables in this work. We leave the issue as an open question which requires further investigation. For the remainder of this section, we will assume that a pre-set ordering is available for use.

Methodology for Recurrent Designs. We endeavor to follow a similar approach to developing a design as described in Chapter 7; thus, we only need modify the steps for forming a design that are pertinent to the recurrent approach. We summarize the steps in our methodology for developing a vector $\underline{F}$ as presented in Chapter 7:

1. derive a 1-normal form specification $\phi(X, Z) = 1$, if not already formed;
2. form a general solution of $\phi(X, Z) = 1$ for Z;
3. form the set of all *multiple-output prime implicants* of the upper-bound functions $\underline{h}$;
4. develop a base for $[\underline{g}, \underline{h}]$;
5. develop inclusion formulas representing coverage of the terms of the base by the multiple-output prime implicants;
6. reduce the inclusion formulas using reduction rules—identifying prime implicants to include in formulas in $\underline{F}$ as well as to discard from consideration; and
7. use search to determine the remaining prime implicants to include in formulas in $\underline{F}$.

The steps which differ between the recurrent approach and the conventional methodology are Steps 2 and 3. In the conventional approach, a non-recurrent system (10.44) is formed to represent a general solution of $\phi(X, Z) = 1$ for Z; a system such as (10.46) is developed in the recurrent method. Moreover, in addition to the multiple-output prime implicants developed in Step 3, we add a set of terms which contain output variables which we call *recurrent prime implicants* (RPIs). RPIs as well as MOPIs are used to cover terms of the lower bound functions $\underline{g}$ in the recurrent method.

Formation of a Subsumptive General Solution. Given a specification $\phi(X,Z)=1$ and an ordering Z_{order} for the output variables, we develop a subsumptive general solution of $\phi(X,Z)=1$ for Z. Using the extended-range, we thus form

$$\begin{array}{rcccl}
 & & 1 & \leq & \phi_0(X) \\
\phi_1'(X,0)\cdot\phi_1(X,1) & \leq & z_1 & \leq & \phi_1'(X,0)+\phi_1(X,1) \\
\phi_2'(X,z_1,0)\cdot\phi_2(X,z_1,1) & \leq & z_2 & \leq & \phi_2'(X,z_1,0)+\phi_2(X,z_1,1) \\
\phi_3'(X,z_1,z_2,0)\cdot\phi_3(X,z_1,z_2,1) & \leq & z_3 & \leq & \phi_3'(X,z_1,z_2,0)+\phi_3(X,z_1,z_2,1) \\
 & & \vdots & & \\
\phi_m'(X,z_1,\ldots,z_{m-1},0) & \leq & z_m & \leq & \phi_m'(X,z_1,\ldots,z_{m-1},0) \\
\cdot\phi_m(X,z_1,\ldots,z_{m-1},1) & & & & +\phi_m(X,z_1,\ldots,z_{m-1},1).
\end{array} \tag{10.53}$$

For each z_j, we denote the lower-bound function by $g_j(X,Z)$ and the upper-bound function by $h_j(X,Z)$. A simplified formula $G_j(X,Z)$ is developed to represent each $g_j(X,Z)$; terms of $G_j(X,Z)$ are covered by the MOPIs and RPIs to develop $\underline{F}(X,Z)$.

Development of RPIs. Just as in conventional minimization, we form multiple-output prime implicants for use in covering terms of the formulas G_j in $\underline{G}(X,Z)$. MOPIs are used in addition to recurrent prime implicants, thus guaranteeing that the resulting design will be no worse than what would be developed using conventional techniques. A formula M_{all} is developed using the methodology described in Chapter 7 in which terms of M_{all} denote the set of all MOPIs. Terms $t(X,Z)$ in M_{all} contain an X-part $u(X)$ which is a MOPI. The Z-part $v(Z)$ denotes the formulas F_j in which the corresponding MOPI may be contained. This is convenient technique for differentiating among the functions that a MOPI may represent. In recurrent designs, however, Z-variables must also be treated in a similar fashion as X-variables. Hence, we require a method by which we may use the Z-variables in both manners; we shall address this issue shortly.

After the subsumptive general solution is developed, we form the Blake canonical form for each function $h_j(X,Z)$. Terms in each formula $BCF(h_j(X,Z))$ which contain only X-variables are

then deleted since they correspond to MOPIs. Terms t which remain in each formula may be used to cover the corresponding lower-bound function $g_j(X, Z)$, and possibly other functions $g_k(X, Z)$. The functions that each t may be used to cover is dependent on the sequence Z_{order} used when forming the subsumptive general solution, i.e., k must precede j in Z_{order}. For example, if an ordering of (z_2, z_1, z_3) is used as in Example 10.4, then a system such as

$$\begin{array}{rcccl} g_3(X) & \leq & z_3 & \leq & h_3(X) \\ g_1(X, z_3) & \leq & z_1 & \leq & h_1(X, z_3) \\ g_2(X, z_3, z_1) & \leq & z_2 & \leq & h_2(X, z_3, z_1). \end{array} \tag{10.54}$$

is formed. After the deletion of terms containing only X-variables in each H_j, terms remaining in $H_1(X, z_3)$ may not be used to cover $g_3(X)$. Additionally, terms remaining in $H_2(X, z_3, z_1)$ may not be used to cover $g_3(X)$ or $g_1(X, z_3)$. The way we denoted that terms in H_1 are not used to cover $g_3(X)$ in the conventional approach was to include a literal z_3' in terms of H_1. However, the variable z_3 may appear in both complemented and uncomplemented form in terms in $H_1(X, z_3)$ since $h_1(X, z_3)$ is a function of the z_3 variable. We must have a way for representing that H_1 may both contain the variable z_3 and may not be used to cover $g_3(X)$.

The method we use for handling this problem is to create a set Y of variables, in which each $y_j \in Y$ corresponds directly with a variable z_j in Z. The Y-variables replace the Z-variables for the purpose of being used similar to X-variables, i.e., system (10.54) is rewritten as

$$\begin{array}{rcccl} g_3(X) & \leq & z_3 & \leq & h_3(X) \\ g_1(X, y_3) & \leq & z_1 & \leq & h_1(X, y_3) \\ g_2(X, y_3, y_1) & \leq & z_2 & \leq & h_2(X, y_3, y_1). \end{array} \tag{10.55}$$

Each variable in formulas H_j is replaced with the corresponding variable y_j. After all Z-variables are replaced by Y-variables, we then develop a formula M_{rec} which will contain the set of all recurrent prime implicants. For each formula H_j, literals z_k' are concatenated to each term

in H_j for every z_k that z_j precedes in Z_{order}. For (10.55), we would concatenate z_3' to each term in $H_1(X, y_3)$; additionally, the term $z_3'z_1'$ would be concatenated to each term in $H_2(X, y_3, y_1)$. After these actions are performed for each formula H_j, the resulting terms are combined to form a single set $H(X, Y, Z)$ of terms. The formula M_{rec} is defined as equal to $ABS(H(X, Y, Z))$. M_{rec} represents the set of all recurrent prime implicants.

Terms in M_{rec} contain X, Y, and Z-parts. The X and Y parts are treated in a similar manner. The Z-parts are identical in utility to the Z-parts for M_{all}. Procedure 10.1 (Recurrent Prime Implicants) produces the formula M_{rec}.

Procedure 10.1 (Recurrent Prime Implicants): Given the vector $\underline{H}(X, Z)$ of formulas which represent the functions $\underline{h}(X, Z)$ and a sequence Z_{order} of Z-variables, the formula M_{rec} is constructed in the following manner:

Step 1. Form $BCF(h_j(X, Z))$ for each function h_j in $\underline{h}(X, Z)$.

Step 2. For each formula $BCF(h_j(X, Z))$, delete all terms which contain only X-variables.

Step 3. Form a set Y of variables which corresponds directly with the Z-variables.

Step 4. For each formula H_j resulting from Step 2, replace each appearance of a variable z_j with its corresponding Y-variable y_j.

Step 5. For each formula H_j resulting from Step 4, concatenate to each term a literal z_k' for each z_k preceded by z_j in Z_{order}.

Step 6. Combine the terms from all formulas formed in Step 5 to form a single formula $H(X, Y, Z)$.

Step 7. Form $ABS(H(X, Y, Z))$. $ABS(H(X, Y, Z))$ is the formula M_{rec}. Return M_{rec}.

The combination of M_{all} and M_{rec} represents the set of all prime implicants—MOPIs and RPIs, respectively—which may be used to cover the functions $\underline{g}(X, Z)$. An identical methodology as found in Chapter 7 may then be followed to develop $\underline{F}$. We present two algorithms for producing recurrent designs in the next section.

Algorithms for Recurrent Designs. We introduced two algorithms in Chapter 7 for producing multiple-output circuits. In this section, we present two analogous algorithms for constructing recurrent designs. The first, Algorithm 10.5, uses the set of all useful, conditionally-eliminable

MOPIs and RPIs as a base.[1] A subset of the base used in first algorithm is used as a base for the second—Algorithm 10.6.

Algorithm Using Base #1. The first algorithm for forming an $\underline{F}$ uses the set of all useful, conditionally-eliminable prime implicants as a base. A synopsis of this algorithm is:

1. derive a 1-normal form specification $\phi(X, Z) = 1$, if not already formed;
2. form a general solution of $\phi(X, Z) = 1$ for Z to develop a set of intervals such as (10.53);
3. form the set of all *multiple-output prime implicants* and *recurrent prime implicants* of the upper-bound functions $\underline{h}$;
4. develop a base for $[\underline{g}, \underline{h}]$ consisting of all useful, conditionally-eliminable PIs;
5. use Procedure 7.6 develop multiple-output inclusion formulas representing coverage of the terms of the base by the PIs;
6. use Reduction Rule Set #1 to reduce the inclusion formulas—identifying PIs to include in formulas in $\underline{F}$ as well as to discard from consideration; and
7. use a search process to determine the remaining prime implicants to include in formulas in $\underline{F}$.

Algorithm 10.5 implements the first six steps of the aforementioned process. The search process was discussed in Chapter 9.

Algorithm 10.5 (Algorithm #1 - Recurrent Designs): Given a 1-normal form specification $\phi(X, Z) = 1$ and a Z-variable sequence Z_{order}, a minimal vector $\underline{F}$ of formulas which represent functions $\underline{f}(X)$ belonging to the intervals $[\underline{g}(X), \underline{h}(X)]$ developed from $\phi(X, Z) = 1$ is generated in the following manner:

Step 0.

1. Initialize a partial sum $PS = \emptyset$.
2. Initialize a variable $P_{discard} = \emptyset$.

Step 1.

1. Using Procedure 4.2 (Subsumptive General Solution - Extended Range) and Z_{order}, develop a general solution of $\phi(X, Z) = 1$ for Z.
2. For each $j = 1, 2, \ldots, m$, develop a simplified formula G_j to represent $g_j(X, Z)$ using Procedure 2.15 (Simplification).

[1] For the remainder of this chapter, we shall refer to the combination of MOPIs and RPIs simply as prime implicants (PIs).

Step 2. Using Procedure 7.1 (Formation of $\Phi_H(X,Z)$), develop a formula $\Phi_H(X,Z)$ which will be used to form the set of all multiple-output prime implicants.

Step 3.

1. Develop $BCF(\Phi_H(X,Z))$ using Procedure 2.20 (Blake canonical form).
2. Delete the term $x_1'x_2'\cdots z_m'$ in $BCF(\Phi_H(X,Z))$.
3. The formula which results after substeps 1 and 2 is M_{all}.

Step 4.

1. Using Procedure 10.1 (Recurrent Prime Implicants), the vector $\underline{H}(X,Z)$ of formulas, and Z_{order}, derive the formula M_{rec}. M_{rec} represents the set of all recurrent prime implicants.
2. Append M_{rec} to M_{all} to form the combined set of prime implicants.
3. For all terms in each $G_j(X,Z)$, replace each variable z_j with the corresponding variable y_j.

Step 5. Using Procedure 7.2 (Useless MOPIs), the set $\underline{g}$ of functions, and M_{all}, determine the useless PIs with respect to each interval $[g_j,h_j]$. The set M_{all} is revised by Procedure 7.2 to denote the useless PIs.

Step 6. Using Procedure 7.3 (Essential MOPIs), the set $\{g_1,g_2,\ldots,g_m\}$, and M_{all}, determine the set of essential PIs with respect to each interval $[g_j,h_j]$.

1. The set M_{all} is revised by Procedure 7.3 to denote the essential PIs.
2. Replace the set $\{g_1,g_2,\ldots,g_m\}$ of functions with the set $\tilde{\underline{g}} = \{\tilde{g}_1,\tilde{g}_2,\ldots,\tilde{g}_m\}$ returned by Procedure 7.3.

Step 7. Using Procedure 7.4 (Inessential MOPIs and Formation of $\hat{\underline{g}}$), the set $\tilde{\underline{g}}$ of functions, and M_{all}, determine the set of inessential PIs with respect to each interval $[g_j,h_j]$.

1. The set M_{all} is revised by Procedure 7.4 to denote the inessential MOPIs.
2. Replace the set $\tilde{\underline{g}}$ of functions with the set $\hat{\underline{g}} = \{\hat{g}_1,\hat{g}_2,\ldots,\hat{g}_m\}$ returned by Procedure 7.4.

Step 8. Develop a set M_{base} which consists of terms in M_{all} which have a Z-part in which at least one Z-variable z_j does not appear in either complemented or uncomplemented form. M_{base} corresponds to PIs which are conditionally eliminable with respect to at least one interval $[g_j,h_j]$.

Step 9. Form a set $LABS = \{P_1',\ldots,P_k'\}$ of labels which will be used to denote the PIs in M_{base}.

Step 10. Initialise $IF = \emptyset$. Then, for each term in M_{base}:

1. Remove the Z-part from the term to form p_l.
2. Using Procedure 7.6 (Formation of a Multiple-Output Inclusion Formula), the PI p_l, the set M_{base}, the set $LABS$ associated with terms in M_{base}, and the set $\hat{\underline{g}}$ of functions, develop an inclusion formula IF_l denoting the coverage of p_l by conditionally-eliminable PIs of each of the intervals $[g_j,h_j]$ for which p_l may be use to form F_j.
3. Add IF_l to IF.

The set IF contains the inclusion formulas IF_l developed for each term in M_{base}.

Step 11. Using Procedure 6.4 (Assignment of Cost to Terms), M_{base}, $LABS$, and $CRITERION$ = fewest gates, develop an association list affiliating a term with a cost. Each cost is paired with the label in $LABS$ which denotes a corresponding PI in M_{base}. Call the resulting list $LAB/COSTS$.

Step 12. Using Procedure 6.11 (Reduction Rules - Set #1), the set IF of inclusion formulas, and the cost list $LAB/COSTS$, apply rule reduction to the set IF. Procedure 6.11 returns a revised set IF_{rev} of inclusion formulas, a set PS_{new} of variables identified for use in $\underline{F}$, and a set $\tilde{P}_{discard}$ of variables to discard.

1. Replace IF with IF_{rev}.
2. Replace PS to PS_{new}.
3. Replace $P_{discard}$ with $\tilde{P}_{discard}$.

Step 13. For each variable in PS:

1. Determine the associated term in M_{base}.
2. For the term equal to M_{base} in M_{all}, fill the Z-part of the associated term in M_{all} with uncomplemented literals z_k for each $k = 1, 2, \ldots, m$ for which no literal z'_k appears.

Step 14. For each variable in $P_{discard}$:

1. Determine the associated term in M_{base}.
2. For the term equal to M_{base} in M_{all}, fill the Z-part of the associated term in M_{all} with complemented literals z'_k for each $k = 1, 2, \ldots, m$ for which no literal z'_k appears.
3. If the term in M_{all} then contains a Z-part of the form $z'_1 z'_2 \cdots z'_m$, i.e., all Z-variables appear and each is in complemented form, delete the term from M_{all}.

Step 15.

- If $IF = \emptyset$, then a vector $\underline{F}$ of formulas has been developed. Continue to Step 16.
- Otherwise, a search process must be used to complete $\underline{F}$. Skip to Step 17.

Step 16. For $j = 1, 2, \ldots, m$, form F_j:

1. Examine each term $t(X, Z)$ in M_{all} to determine if the literal z_j appears in the term.
   - If z_j appears in $t(X, Z)$, then place the X-part $u(X)$ in F_j.
   - Otherwise, do not place $u(X)$ in F_j.
2. After each term in M_{all} has been examined, form $ABS(F_j)$.

After each formula $ABS(F_j)$ has been formed, the development of $\underline{F}$ is complete. Return $\underline{F}$.

Step 17.

1. Return the current inclusion formulas IF, M_{base}, and M_{all}.
2. Also return $LAB/COSTS$ and $LABS$ for use in the search process.

Algorithm Using Base #2. The algorithm for forming a minimal $\underline{F}$ which uses a set of useful, conditionally-eliminable PIs which cover the functions $\hat{\underline{g}}$. We summarize the algorithm as follows:

1. derive a 1-normal form specification $\phi(X, Z) = 1$, if not already formed;
2. form a general solution of $\phi(X, Z) = 1$ for Z to develop a set of intervals such as (10.53);
3. form the set of all *multiple-output prime implicants* and *recurrent prime implicants* of the upper-bound functions $\underline{h}$;
4. use Procedure 7.5 to develop a base for $[\underline{g}, \underline{h}]$ consisting of useful, conditionally-eliminable PIs which cover the functions $\hat{\underline{g}}$;
5. use Procedure 7.6 develop multiple-output inclusion formulas representing coverage of the terms of the base by the PIs;
6. use Reduction Rule Set #2 to reduce the inclusion formulas—identifying PIs to include in formulas in $\underline{F}$ as well as to discard from consideration; and
7. use a search process to determine the remaining PIs to include in formulas in $\underline{F}$.

The first six steps of the foregoing process are implemented by Algorithm 10.6. The search process was introduced in Chapter 9.

Algorithm 10.6 (Algorithm #2 - Recurrent Designs): Given a 1-normal form specification $\phi(X, Z) = 1$ and a Z-variable sequence Z_{order}, a minimal vector $\underline{F}$ of formulas which represent functions $\underline{f}(X)$ belonging to the intervals $[\underline{g}(X), \underline{h}(X)]$ developed from $\phi(X, Z) = 1$ is generated in the following manner:

Step 0.

1. Initialise a partial sum $PS = \emptyset$.
2. Initialise a variable $P_{discard} = \emptyset$.

Step 1.

1. Using Procedure 4.2 (Subsumptive General Solution - Extended Range) and Z_{order}, develop a general solution of $\phi(X, Z) = 1$ for Z.
2. For each $j = 1, 2, \ldots, m$, develop a simplified formula G_j to represent $g_j(X, Z)$ using Procedure 2.15 (Simplification).

Step 2. Using Procedure 7.1 (Formation of $\Phi_H(X, Z)$), develop a formula $\Phi_H(X, Z)$ which will be used to form the set of all multiple-output prime implicants.

Step 3.

1. Develop $BCF(\Phi_H(X,Z))$ using Procedure 2.20 (Blake canonical form).
2. Delete the term $z_1'z_2'\cdots z_m'$ in $BCF(\Phi_H(X,Z))$.
3. The formula which results after substeps 1 and 2 is M_{all}.

Step 4.

1. Using Procedure 10.1 (Recurrent Prime Implicants), the vector $\underline{H}(X,Z)$ of formulas, and Z_{order}, derive the formula M_{rec}. M_{rec} represents the set of all recurrent prime implicants.
2. Append M_{rec} to M_{all} to form the combined set of prime implicants.
3. For all terms in each $G_j(X,Z)$, replace each variable z_j with the corresponding variable y_j.

Step 5. Using Procedure 7.2 (Useless MOPIs), the set $\underline{g}$ of functions, and M_{all}, determine the useless PIs with respect to each interval $[g_j,h_j]$. The set M_{all} is revised by Procedure 7.2 to denote the useless PIs.

Step 6. Using Procedure 7.3 (Essential MOPIs), the set $\{g_1,g_2,\ldots,g_m\}$, and M_{all}, determine the set of essential PIs with respect to each interval $[g_j,h_j]$.

1. The set M_{all} is revised by Procedure 7.3 to denote the essential PIs.
2. Replace the set $\{g_1,g_2,\ldots,g_m\}$ of functions with the set $\underline{\tilde{g}} = \{\tilde{g}_1,\tilde{g}_2,\ldots,\tilde{g}_m\}$ returned by Procedure 7.3.

Step 7. Using Procedure 7.4 (Inessential MOPIs and Formation of $\underline{\hat{g}}$), the set $\underline{\tilde{g}}$ of functions, and M_{all}, determine the set of inessential PIs with respect to each interval $[g_j,h_j]$.

1. The set M_{all} is revised by Procedure 7.4 to denote the inessential PIs.
2. Replace the set $\underline{\tilde{g}}$ of functions with the set $\underline{\hat{g}} = \{\hat{g}_1,\hat{g}_2,\ldots,\hat{g}_m\}$ returned by Procedure 7.4.

Step 8.

1. Develop a set M_{ce} which consists of terms in M_{all} which have a Z-part in which at least one Z-variable z_j does not appear in either complemented or uncomplemented form. M_{ce} corresponds to PIs which are conditionally eliminable with respect to at least one interval $[g_j,h_j]$.
2. Using Procedure 7.5 (Base #2 - CE MOPIs Covering $\underline{\hat{g}}$), M_{ce}, and $\underline{\hat{g}}$, develop a set M_{base} which consists of PIs which are sufficient to cover the terms in the formulas representing $\underline{\hat{g}}$.

Step 9. Form a set $LABS = \{P_1',\ldots,P_k'\}$ of labels which will be used to denote the PIs in $MOPI_{ce}$.

Step 10. Initialise $IF = \emptyset$. Then, for each term in M_{base}:

1. Remove the Z-part from the term to form p_l.
2. Using Procedure 7.6 (Formation of a Multiple-Output Inclusion Formula), the MOPI p_l, the set M_{ce}, the set $LABS$ associated with terms in M_{ce}, and the set $\underline{\hat{g}}$ of functions, develop an inclusion formula IF_l denoting the coverage of p_l by conditionally-eliminable PIs of each of the intervals $[g_j,h_j]$ for which p_l may be use to form F_j.

The set IF contains the inclusion formulas IF_i developed for each term in M_{base}.

Step 11. Using Procedure 6.4 (Assignment of Cost to Terms), M_{base}, $LABS$, and $CRITERION$ = fewest gates, develop an association list affiliating a term with a cost. Each cost is paired with the label in $LABS$ which denotes a corresponding PI in M_{ce}. Call the resulting list $LAB/COSTS$

Step 12. Using Procedure 6.14 (Reduction Rules - Set #2), the set IF of inclusion formulas, and the cost list $LAB/COSTS$, apply rule reduction to the set IF. Procedure 6.14 returns a revised set IF_{rev} of inclusion formulas, a set PS_{new} of variables identified for use in $\underline{F}$, and a set $\tilde{P}_{discard}$ of variables to discard.

1. Replace IF with IF_{rev}.
2. Replace PS to PS_{new}.
3. Replace $P_{discard}$ with $\tilde{P}_{discard}$.

Step 13. For each variable in PS:

1. Determine the associated term in M_{ce}.
2. For the term equal to M_{ce} in M_{all}, fill the Z-part of the associated term in M_{all} with uncomplemented literals z_k for each $k = 1, 2, \ldots, m$ for which no literal z'_k appears.

Step 14. For each variable in $P_{discard}$:

1. Determine the associated term in M_{ce}.
2. For the term equal to M_{ce} in M_{all}, fill the Z-part of the associated term in M_{all} with complemented literals z'_k for each $k = 1, 2, \ldots, m$ for which no literal z'_k appears.
3. If the term in M_{all} then contains a Z-part of the form $z'_1 z'_2 \cdots z'_m$, i.e., all Z-variables appear and each is in complemented form, delete the term from M_{all}.

Step 15.

- If $IF = \emptyset$, then a vector $\underline{F}$ of formulas has been developed. Continue to Step 16.
- Otherwise, a search process must be used to complete $\underline{F}$. Skip to Step 17.

Step 16. For $j = 1, 2, \ldots, m$, form F_j:

1. Examine each term $t(X, Z)$ in M_{all} to determine if the literal z_j appears in the term.
   - If z_j appears in $t(X, Z)$, then place the X-part $u(X)$ in F_j.
   - Otherwise, do not place $u(X)$ in F_j.
2. After each term in M_{all} has been examined, form $ABS(F_j)$.

After each formula $ABS(F_j)$ has been formed, the development of $\underline{F}$ is complete. Return $\underline{F}$.

Step 17.

1. Return the current inclusion formulas IF, M_{ce}, and M_{all}.
2. Also return $LAB/COSTS$ and $LABS$ for use in the search process.

Summary of Recurrent Approach. Given a specification $\phi(X, Z) = 1$ and a sequence Z_{order} of Z-variables, Algorithms 10.5 and 10.6 produce designs which are minimal for the sequence Z_{order} of variables. However, the designs are approximate-minimal with respect to the possible designs for all possible orderings of the Z-variables. The differences between the algorithms in this chapter and those found in Chapter 7 are that

1. a subsumptive general solution of $\phi(X, Z) = 1$ is formed rather than a non-recurrent general solution, and
2. a set of recurrent prime implicants is developed and used in addition to the multiple-output prime implicants.

Since we use the RPIs in addition to the MOPIs, the resulting designs are no worse than what would be developed using the algorithms presented in Chapter 7.

The advantage of using a recurrent method rather than a conventional approach is that the resulting designs generally cost less. There are several disadvantages of the recurrent approach:

1. The delay is typically increased for outputs formed using other output nodes.
2. The addition of the RPIs to the MOPIs increases the memory requirements of the design algorithm.
3. The addition of the RPIs to the MOPIs causes many MOPIs which would be essential using a conventional approach no longer to be essential. Thus, the partitioning of the prime implicants does not initially identify as many PIs to contain in $\underline{F}$, and more PIs are then classified as conditionally-eliminable rather than essential.

The resulting designs are similar to those produced using a method developed by Brown (Brown 90) and refined by Knutson (Knuts 90). However, in their approach a subsumptive general solution of $\phi(X, Z) = 1$ is not formed. In addition, they only use the prime implicants of the upper bound formulas $H_j(X, Z)$—a subset of which is our RPIs—rather than MOPIs. Moreover, their method is based on developing sub-minimal formulas to represents functions $f_j(X, Z)$ belonging to the interval $[g_j(X, Z), h_j(X, Z)]$ rather than considering each prime implicant individually to develop a formula which is minimal with respect to the sequence of Z-variables.

Summary

In the two sections of this chapter, we presented methods in which we construct designs which cannot be developed using a conventional approach to the design problem. In our approach to solving Ledley's problems, we demonstrate the utility of the 1-normal form and the equation-solving approach in the design process. Whereas Ledley only presented ad hoc techniques for solving the problems that he proposed, our methodology yields minimal solutions for all three problems. In the second part of this chapter, we again demonstrated the utility of the 1-normal form and the equation-solving approach in a method for developing recurrent circuits. In recurrent circuits, circuit outputs may be constructed using other output nodes as well as input nodes.

The following methods are unique in this chapter:

- We introduced a methodology using the 1-normal form and Boolean equation-solving approach for solving Ledley's problems in which least-cost designs are produced.
- We presented algorithms for developing recurrent circuits based on the formation of a subsumptive general solution of a specification $\phi(X, Z) = 1$. The resulting designs are minimal with respect to a sequence Z_{order} and approximate-minimal with respect to all possible sequences of Z-variables.

XI. Conclusions and Recommendations

In this chapter, we present a summary of the work. Conclusions are stated regarding the utility of the resulting methods. In addition, we summarize the key contributions of this effort. Finally, suggestions for further research are given.

Summary

Our goal at the outset of this work was to develop new, theoretically-sound algorithms for producing minimal or near-minimal circuit designs. We endeavored to produce techniques which integrate the concepts of Boolean reasoning and informed search in the minimization process. In an effort to accomplish this goal, a seven-step methodology was developed on which to base new algorithms. We review the steps of this methodology in turn.

In the first step, a 1-normal form $\phi(X, Z) = 1$ is derived which serves as the circuit specification. A 1-normal form is useful because it is easy to develop from traditional specification formats and can specify behavior that the traditional formats cannot indicate. Formulas $\underline{F}$ which represent functions $\underline{f}(X)$ in a particular solution $Z = \underline{f}(X)$ of $\phi(X, Z) = 1$ correspond to circuit designs. The use of the 1-normal form—in conjunction with the use of Boolean equation solving—facilitates the alternative minimisation techniques discussed in Chapter 10.

A general solution of $\phi(X, Z) = 1$ for Z is developed in the second step of our methodology. If $\phi(X, Z) = 1$ is a *tabular* specification, then a general solution of $\phi(X, Z) = 1$ for Z may be represented by a system of the form

$$\begin{array}{ccccc} \alpha_1(X) & \leq & z_1 & \leq & \beta_1(X) \\ \alpha_2(X) & \leq & z_2 & \leq & \beta_2(X) \\ \alpha_3(X) & \leq & z_3 & \leq & \beta_3(X) \\ & & \vdots & & \\ \alpha_m(X) & \leq & z_m & \leq & \beta_m(X). \end{array} \tag{11.1}$$

A general solution such as (11.1) was used in the techniques presented in Chapter 7 to develop a particular solution $Z = \underline{f}(X)$. In the section on recurrent designs presented in Chapter 10, a recurrent general solution of the form

$$\begin{array}{rcccl} \alpha_1(X) & \leq & z_1 & \leq & \beta_1(X) \\ \alpha_2(X, z_1) & \leq & z_2 & \leq & \beta_2(X, z_1) \\ \alpha_3(X, z_1, z_2) & \leq & z_3 & \leq & \beta_3(X, z_1, z_2) \\ & & \vdots & & \\ \alpha_m(X, z_1, \ldots, z_{m-1}) & \leq & z_m & \leq & \beta_m(X, z_1, \ldots, z_{m-1}). \end{array} \tag{11.2}$$

was used in the development of a circuit. A recurrent solution $Z = \underline{f}(X, Z)$ for $\phi(X, Z) = 1$ is developed which corresponds to a recurrent design. System (11.2) is the most general way of depicting a general solution. It is particularly useful, because it can be used in the development of a design with a *non-tabular* specification $\phi(X, Z) = 1$.

A vector $\underline{F}$ of a *minimal design consists of prime implicants of the functions* in $\underline{\beta}$ which cover the functions in $\underline{\alpha}$. In the third step of our methodology, the set P of all prime implicants of $\underline{\beta}$ is developed. After P is formed, the set is partitioned into essential, inessential, and conditionally-eliminable categories. Inessential prime implicants may be deleted from consideration since they never appear in minimal formulas consisting of prime implicants; essential prime implicants must appear in $\underline{F}$. Hence, the minimisation effort is focused on the selection of conditionally-eliminable prime implicants (CEPIs) to constitute formulas in $\underline{F}$. Subsets of the CEPIs are used to form a base in the fourth step of our method. Prime implicants are used as a base to facilitate the use of *reduction rules in the sixth step* of our method.

The development of inclusion formulas, e.g., $P_1 + P_2P_3$, which represent the coverage of the terms of the base by CEPIs of $\underline{\beta}$ is the fifth step of our method. An equation-based approach based on the Boolean reasoning concepts of reduction and elimination is used to construct the inclusion formulas. This theoretical foundation allows the use of constraints, which are equations of the

form $C_j(X) = 0$ which constrain the values of the X-variables. Constraints are constructed using knowledge of the essential prime implicants and don't-care set of a function. The use of constraints reduces the number of computations in the elimination process. Moreover, the resulting inclusion formulas comprise fewer terms and literals, because each formula developed then only represents the portion of the corresponding term covered by the prime implicants which is not part of the don't-care set or is not covered by essential prime implicants.

After an inclusion formula is developed for each term of the base, reduction rules are applied in our sixth step to identify prime implicants which constitute formulas in $\underline{F}$ as well as those to discard from consideration. Reduction rules are based conceptually on the ideas of domination as used to reduce prime implicant tables. The reduction rules are so called because in the course of identifying prime implicants to use or delete, the inclusion formulas are reduced with respect to contained terms and literals. In many cases, the formulas in $\underline{F}$ may be completely formed after the application of reduction rules. In other cases, a search process must be used to determine the remaining set of prime implicants. The search process is the last step of our methodology.

In Chapter 9, we discussed five issues required for formulating a search process for a problem. Each of these issues was addressed in constructing a search process. Of particular importance is availability of strategies which provide the option of either developing a minimal solution—possibly at large expense—or quickly constructing near or approximate-minimal solutions. At the completion of the search process, a vector $\underline{F}$ of formulas is constructed which represents the functions $\underline{f}$ in either system (11.1) or system (11.2) and corresponds to a two-level digital design.

Conclusions

Utility of Boolean Reasoning. Boolean reasoning provided a theoretical foundation that was indispensable in the development of a new methodology for circuit minimization. This was evident in several aspects of this work. The utility of Boolean reasoning was particularly important

in the development of inclusion formulas which denote coverage of terms of the base by subsets of the prime implicants of a function. Specifically, the idea of reducing information to a single equation allowed the use of constraints to reduce the number of computations required in the process of developing inclusion formulas; it also reduced the complexity of the resulting formula with respect to the number of contained terms and literals. The application of constraints in this work was a natural outcome of viewing the design problem from a Boolean-reasoning perspective.

The reduction of information to a single equation was also important in the development of a 1-normal form specification $\phi(X, Z) = 1$. Using Boolean equation-solving techniques in conjunction with the 1-normal form facilitated the development of methods to handle design problems as well as unusual specifications in a manner not possible using conventional approaches. Two examples in this dissertation of the utility of our approach are the development of recurrent designs and the ability to formulate least-cost solutions for Ledley's elementary design problems.

Design Trade-Offs. Two issues were used to guide the development of a set of algorithms for solving the design problem. These issues concern the trade-offs that an engineer must make in the course of the design process. They are

- number of computations versus memory usage, and
- minimality of the design versus speed at which we develop a design.

The first issue pertains to the classic time versus space dilemma; the second to the complexity of the minimisation problem. For complex problems, both issues may have to be addressed by the circuit designer. An important outcome of this dissertation is a set of techniques which allow a circuit designer to make decisions regarding these two issues during the design process.

The number of computations required to solve a problem is often dependent on the memory available. A fast technique, which performs few computations, may require much memory. On the other hand, a method which requires less memory to solve the problem may perform a greater

number of computations. An example of this trade-off in this dissertation was the use of smaller bases in exchange for an increase in the amount of work that must be performed in the search process. When a small base is used—hence, fewer inclusion formulas are developed and stored—reduction rules cannot identify as many prime implicants to constitute $\underline{F}$ as when a larger base is used. A search process must then be used to determine a greater percentage of the prime implicant to constitute $\underline{F}$ than when a larger base is used. Thus, work is shifted to the search process. This savings in memory requires an increase in the number of computations. The designer may select the algorithm from our set which is suitable for the memory constraints of his hardware.

The choice between the minimality of the design versus speed at which a result is developed is facilitated by the availability of different search strategies for use in the search process. A strategy such as A*, which guarantees a least-cost result if an admissible heuristic function is used, will generally take longer to produce a result than does an approximate-minimal strategy such as static weighting. The concept in economics called *the law of diminishing returns* is especially pertinent to the minimisation problem. A strategy which guarantees only near-minimal designs, possibly one which is within one percent of a minimal solution, may take only a fraction of the time required by a technique for constructing a guaranteed least-cost design. The difference between a minimal and a near-minimal design, i.e., the last one percent of the result, is what may require a significant effort. We thus provide a spectrum of search strategies to allow a designer to determine the amount of time that he feels should be devoted to the development of a design.

Assessment

The most important contribution of this dissertation is that it is the first coherent, uniform attempt to systematically apply Boolean reasoning to the minimization problem. A methodology is developed for circuit minimization which departs from the conventional approach. In algorithms developed in this work, a circuit specification is reduced to a single equivalent Boolean equation

$\phi(X, Z) = 1$ called a 1-normal form. It is shown that developing a particular solution $Z = \underline{f}(X)$ for $\phi(X, Z) = 1$ corresponds to constructing a two-level design which meets the specification. Thus, forming a good solution for a Boolean equation corresponds directly to developing an economical digital circuit. This approach has several advantages: a number of design problems which cannot be handled using conventional methods are easily treated, atypical design specifications unusable by conventional methods are dealt with in a uniform manner, and in some cases a single algorithm, rather than a set of algorithms, suffices to solve a problem.

Of particular importance is the use of Boolean reasoning in the development of a technique for producing formulas which denote the coverage of a function by subsets of a set of functions. In the special case of all functions being terms, this formula is called an inclusion formula. An equation-based approach which incorporates the concept of constraints is presented for the generation of inclusion formulas. This approach provides a theoretically-sound foundation for the reasoning process employed to generate inclusion formulas, something that has been lacking in previous work. The use of constraints reduces the number of computations involved in the process of generating inclusion formulas; it also simplifies the resulting formula with respect to the number of contained terms and literals.

The second contribution of this dissertation is the formulation of a search process in which we could apply informed search. We introduce the use of informed search in circuit-minimization. Significant aspects of the search process developed in this work are the formulation of heuristic functions as well as strategies for the decomposition of the search process. The use of informed search has not been widely employed thus far in circuit minimization.

A set of algorithms for minimizing logic circuits which incorporates the concepts of Boolean reasoning and informed search is the third contribution of this work. The algorithms vary with respect to memory and computational requirements, which allows a trade-off between memory usage and the number of computations. For example, an algorithm which requires more memory

will generally involve fewer computations. Hence, if the memory resources of a computer used to implement the algorithms are limited, then the algorithm which requires the least amount of memory can be used to generate a design. Similarly, if a specification is highly complex, then an algorithm which is memory-conserving may be used. We normally like to use an algorithm which quickly produces a result; however, an algorithm which quickly generates a design generally is more memory-intensive than a slower method.

Recommendations

The development of a method for solving a complex problem spawns many additional problems. Such is the case with this work. Moreover, we mentioned several issues which require further attention.

We have provided a set of techniques which allow a circuit designer to make choices during the design process. However, in providing these methods, we have not developed specific guidelines which facilitate good choices. In an ideal situation, the "best" choices would be made automatically—without human intervention—based on hardware resources and function complexity. Although this may one day be possible, in the interim a set of specific guidelines should be developed to guide the selection of algorithm and search strategy. An if-then-else set of rules seems appropriate for this problem.

In the selection of search strategies to apply to this problem, we endeavored to propose one strategy from each category of minimization, i.e., minimization, near-minimization, and approximate-minimization. However, we have not performed enough experimentation and analysis to recommend settings for pre-defined constants used in the various search strategies. Examples include overestimation factor ϵ and the anticipated search depth N in dynamic weighting, the beam width w in beam search, and the weight W in static weighting. Experimentation and analysis should be performed to determine "good" values for these constants.

More study should be devoted to the use of induced partitions for problem decomposition in the search process as suggested in Chapter 9. The feasibility of the suggested alternatives should be investigated and integrated into the current search process to form an AND/OR search process. One or more of the cut-set or graph-partitioning algorithms mentioned in the text should be implemented for use in this problem.

In our presentation of the development of recurrent circuit designs, we left unaddressed the issue of selecting a good sequence of Z-variables which will generally lead a design whose cost is close to that of a least-cost design. Research should look into the existence of heuristics which may be used to guide the choice of Z-orderings.

Lastly, the methodology presented in this work may be useful in developing techniques for more elaborate digital design problems. Examples of such problems include the development of sequential circuit designs, the construction of multi-level logic circuits, and the formation of designs which meet non-tabular specifications. Of these problems, the utility of this approach may be most useful for developing designs for non-tabular specifications—a problem which has received little attention in the past.

Appendix A. Existing Methods

This appendix includes background material on previous research efforts in two-level circuit minimisation which are not addressed in the main body of the text. This appendix is not meant to be all-inclusive; rather it presents a number of notable approaches for solving the problem. In addition to the techniques for two-level minimisation, a method for developing recurrent circuits different from the technique described in Chapter 10 is described. It is assumed that the reader is familiar with terminology defined in the text.

Early Methods

Early methods developed for circuit optimization primarily were oriented towards minimization of single-output circuits. Several early techniques include simplification using Boolean axioms and theorems, map-based approaches, and the Quine-McCluskey method.

Boolean Simplification. One of the advantages of digital circuits is that they may be described mathematically by Boolean functions[1] of the two-element Boolean algebra, $\mathbf{B_2} = \{0, 1\}$. Since digital circuits often are called *switching circuits*, the two-element Boolean algebra is called *switching algebra*. Boolean functions in the switching algebra are called *switching functions*. Additionally, the terms *switching* and *logic* often are used interchangeably. (Nagle 75:76) The nodes of a circuit are depicted by Boolean variables; the gates of a circuit are modeled by Boolean operators. Every Boolean formula which represents a switching function has a corresponding switching circuit implementation and vice versa. A conjunction corresponds to an AND gate; a disjunction corresponds to an OR gate; and a complement is implemented by an inverter. The output of a switching circuit for a particular input combination is the same as the value of the corresponding switching

[1] It is assumed that the reader understands the basic terminology of Boolean algebra. See Chapter 2 for a discussion of the basics of Boolean algebra.

function given the same assignment of values to its variables. An example of a sum-of-products formula which represents a three-variable switching function, $f : \mathbf{B}_2^3 \rightarrow \mathbf{B}_2$, is

$$x'z + x'yz' + xy' + xyz. \tag{A.1}$$

The corresponding circuit for this formula is given in Figure A.1. Each term of the formula is implemented by an AND gate. The disjunction of the terms of the formula corresponds to the combination of the outputs of the AND gates with an OR gate. Because only two gates must be traversed between the circuit inputs and the circuit output, this circuit it is called a *two-level* or *two-stage* logic circuit; specifically, it is an AND-OR circuit. The AND gates form the first level; the OR gate forms the second level. The inverters are not said to form a level, because often the input signals and their complements are both available, eliminating the need for inverters. Other two-level logic circuits are NAND-NAND and NOR-NOR circuits. The number of levels of a circuit is defined as the maximum number of gates that must be traversed between the circuit inputs and circuit outputs, less inverters required to complement the input signals. In general, any circuit which has more than two levels is called a *multi-level* or *multi-stage* circuit.

Often when designing a circuit it is necessary to list the output values of the circuit for given combinations of input values. Such values are defined in a table called a *truth table*. Switching functions are also defined by a truth table. Switching circuits and their corresponding switching functions have the same truth table. A truth table for the circuit of Figure A.1 is shown in Table A.1.

A switching circuit may be implemented by different combinations of components and still behave the same. Likewise, a given switching function can be represented by a variety of formulas. In either case, the number of realizations is actually infinite. Different formulas which represent the same function are called *equivalent* formulas; different switching circuits which realize the same

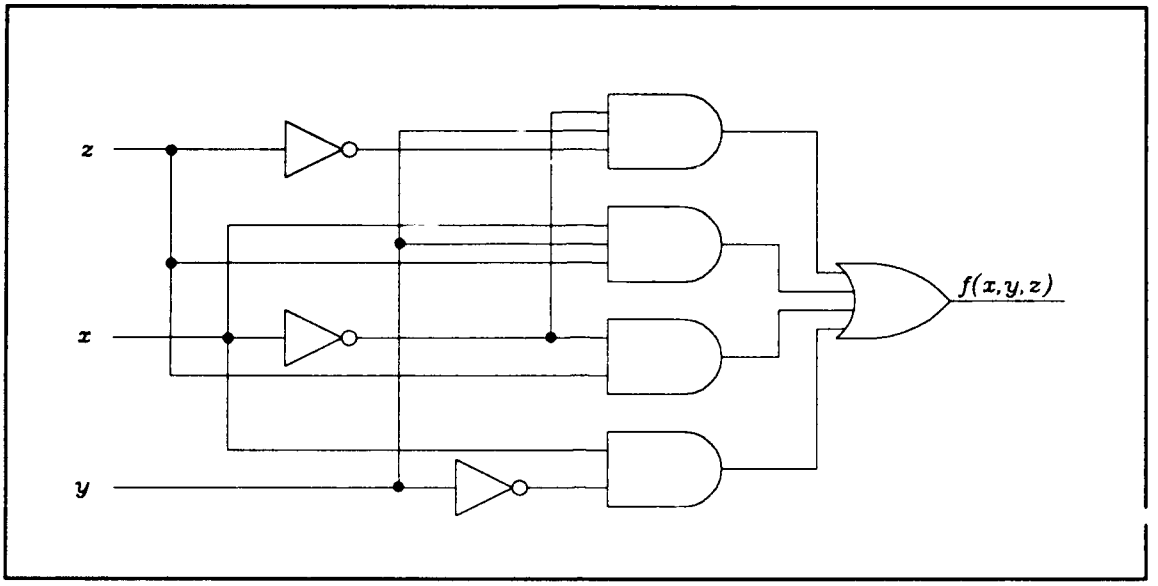


Figure A.1. Circuit Implementation of $x'z + x'yz' + xy' + xyz$

$x\ y\ z$	$f(x,y,z)$
0 0 0	0
0 0 1	1
0 1 0	1
0 1 1	1
1 0 0	1
1 0 1	1
1 1 0	0
1 1 1	1

Table A.1. Truth Table for $x'z + x'yz' + xy' + xyz$

function are called *equivalent* circuits. One of the first techniques developed to minimize a switching circuit was to take the corresponding Boolean formula and use the axioms and theorems of Boolean algebra to produce a simpler equivalent formula. A simpler formula maps into a simpler switching circuit; however, the function remains the same.

A simplification of expression (A.1) using the axioms and theorems of Boolean algebra is given below.

1. Terms three and four of (A.1) are used to form a new term using consensus (2.32):

$$x'z + x'yz' + xy' + xyz + xz. \tag{A.2}$$

2. Term four of (A.2) is eliminated due to absorption (2.25) with respect to term five:

$$x'z + x'yz' + xy' + xz. \tag{A.3}$$

3. Terms one and four of (A.3) form a consensus term:

$$x'z + x'yz' + xy' + xz + z. \tag{A.4}$$

4. Term five of (A.4) absorbs terms one and four. The resulting formula is:

$$x'yz' + xy' + z. \tag{A.5}$$

5. Term three of (A.5) forms a consensus with term one:

$$x'yz' + xy' + z + x'y. \tag{A.6}$$

6. Finally, term four of (A.6) absorbs term one:

$$xy' + z + x'y. \tag{A.7}$$

Once formula (A.7) is developed, a minimized equivalent to the circuit of Figure A.1 may be implemented. This circuit is shown in Figure A.2. The optimized circuit requires two fewer AND gates and one less inverter than the original circuit; additionally, a three-input OR gate is required versus a four-input gate. Hence, there is a substantial decrease in required hardware.

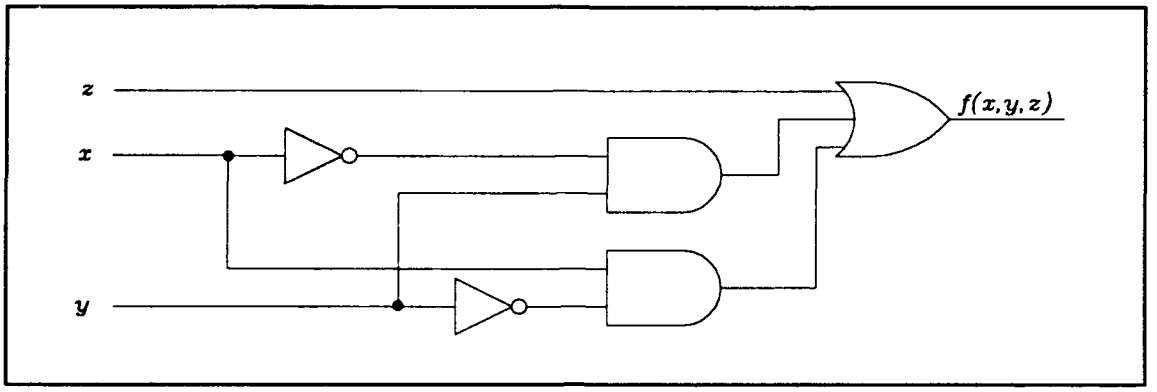


Figure A.2. Circuit Implementation of $x'y + xy' + z$

Map-Based Approaches. Minimization using the axioms and theorems of Boolean algebra is difficult for all but the smallest circuits due to the fact that there are no methods to guide the choice of rule to be applied at each step of the process, i.e., it is an ad hoc technique (Mano 79:72). Hence, other approaches have been developed to perform minimization in a more simplified manner. Map-based approaches consist of graphical techniques used to manually construct simplified sum-of-products formulas to represent two-level switching functions. A method found in most digital logic textbooks is the use of a graphical technique called the Karnaugh map. For a more detailed explanation of the use of Karnaugh maps see (Johns 87), (Mano 79), or (Nagle 75). Other graphical approaches include Marquand diagrams and Svoboda grids; see (Svobo 79) or (Klir 72) for an explanation of these methods. The difficulty with map-based techniques is that they are unwieldy for functions of greater than five or six variables. Additionally, these methods do not guarantee that a minimal formula will be derived. Hence, the use of map-based approaches is confined to relatively simple functions.

Quine-McCluskey Method. Quine (Quine 52, Quine 55) was interested in developing minimal representations for logical expressions. In studying this problem, he developed the substantial part of what is now called the Quine-McCluskey method for logic minimization. McCluskey's (McClu 56) contribution stems from simplifying the bookkeeping entailed using Quine's approach.

The Quine-McCluskey method is a tabular approach to deriving a minimal sum-of-products formula to represent a two-level switching function.

The first step of the technique is to develop the Blake canonical form for a function, i.e., the set of all prime implicants of a function. The determination of a minimal collection of prime implicants required to implement the function is the second step of the Quine-McCluskey method. Using the set of prime implicants, a prime implicant table (discussed in Chapter 6) is constructed which denotes the coverage by the prime implicants of the minterms of the function. The identification of essential prime implicants, and row and column domination are used to reduce the table. Finally, a covering problem[2] is performed using the reduced table to determine a minimal collection of prime implicants required to cover all of the minterms.

Different metrics of cost can be used to guide the solution of the covering problem. One measure of cost, stated by Quine, is to choose a collection of prime implicants which minimizes the total number of literals. On the other hand, McCluskey stated that the choice of prime implicants should be made first to reduce the number of prime implicants to the least number. He called such a collection of PIs a *minimum sum*. Several minimum sums may exist; the selection of one minimum sum is based on the minimum sum which contains the least number of literals. Differing from Quine, McCluskey states that the choice of a minimal collection of prime implicants "is not necessarily the expression containing the fewest total literals" (McClu 56:1419). Nevertheless, the choice of the cost measure may differ based on the intended circuit implementation.

The primary benefit of the Quine-McCluskey method is that it is a systematic method for logic minimisation. Since the method is systematic, it is easily automated. The method does not depend on a designer's ability use the axioms and theorems of Boolean algebra to produce a simplified formula. Likewise, the intuition required to use a map-based method to produce a minimal formula is not required using the Quine-McCluskey method. Additionally, the Quine-

[2]This problem is a variation of the well-known set covering problem (Murog 79:168).

McCluskey method can be used for functions with a larger number of variables than map-based methods. (Nagle 75:141)

Algebraic Techniques for Minimization

Importance of Two-Level Minimization. With the advent of LSI and VLSI in the mid to late 1970s, minimisation of two-level circuits became a vigorous area of research. Two-level circuits are practical in LSI/VLSI due to ease of implementation in the form of *Programmable Logic Arrays* (PLAs). There are a number of advantages of PLA-based implementations (Newto 86:33):

1. It is easy to implement a function in PLA form with a low probability of error. There is a one-to-one correspondence between a symbolic representation of a PLA using 0s and 1s and the physical layout of the function.
2. Computer-aided design (CAD) tools have made it easy to automatically layout PLAs.
3. It is easy to change a PLA once it has been constructed. Often, all that is entailed is to disconnect or connect a transistor.

A PLA is a grid where each input column of the grid is a literal, complemented or uncomplemented, each output column is a function output, and each row is a term of a sum-of-products formula. Since each possible literal is available for every term, the object of PLA-based minimization is strictly to reduce the number of product terms of a formula. By reducing the number of terms, the number of rows of PLA is proportionately decreased. A depiction of a PLA is given in Figure A.3. In this example, a three-variable multiple-output[3] function is implemented. A dot at the intersection of two lines corresponds to a connection. Note that term x_1x_2 is shared between two circuit outputs.

[3] In PLA optimisation, several single-output functions may be implemented; however, a set of single-output functions is treated as a single multiple-output function. This facilitates the sharing of terms between different functions.

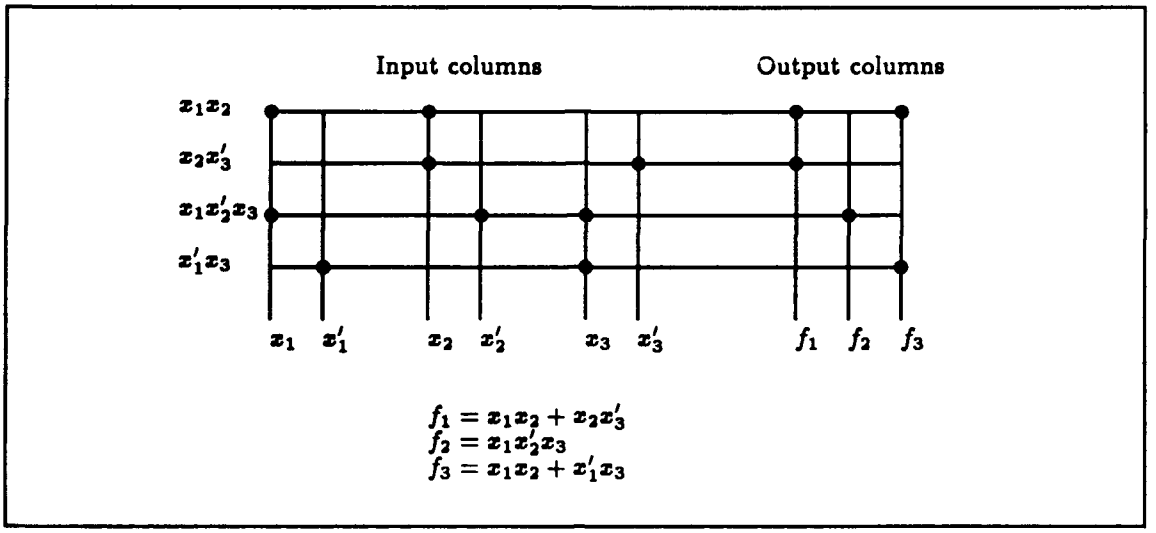


Figure A.3. Depiction of a Programmable Logic Array

Approach of Algebraic Methods. A common methodology is followed in virtually all algebraic methods for developing a minimal SOP formula to represent a function f in an interval $[g, h]$. These steps are:

1. form the set of prime implicants of h;
2. develop a *base* for $[g, h]$;
3. develop *inclusion formulas* representing coverage of the terms of the base by prime implicants of h; and
4. form the product of the inclusion formulas.

The primary differences among algebraic methods are the form of the base for $[g, h]$ and the method for developing inclusion formulas which denote coverage of the terms of the base by subsets of the prime implicants of h.

A key problem in minimization theory is to devise a base for $[g, h]$ and a corresponding method for forming inclusion formulas that is efficient. A number of bases have been used over the years. Petrick (Petri 56) used the minterm canonical form of a function. The Blake canonical form of a function was used in (Ghaza 57), (Mott 60), (Gaine 64), and (Tison 67). Chang and Mott

(Chang 65) employed an irredundant disjunctive form of a function as a base. Reusch (Reusc 75) showed that any disjunctive form which represents a function may be used as a base. A subset of the minterm canonical form of a function called the *abridged minterm base* was devised by Cutler (Cutle 80). Hong used a subset of the abridged minterm base that he called the "epi-eliminated" minterm base (Hong 91); the epi-eliminated minterm base contains only the minterms of the abridged minterm base which are not covered by essential prime implicants of the function. We surveyed a number of these techniques in Chapter 5. In addition, Cutler's approach and the significance of Gaines's contributions were discussed in Chapter 6.

One shortcoming of virtually all of these methods is that a specific technique must be applied to a specific design problem. For example, Cutler (Cutle 80) presented four different algorithms to handle the different combinations of single- and multiple-output circuits, completely- and incompletely-specified specifications.

Heuristic Techniques for PLA Synthesis

General Concepts. Many of the early methods for logic minimization, such as the Quine-McCluskey method, can be applied to PLA minimization. However, the number of inputs and outputs characteristic of functions to be implemented in VLSI, and the resulting number of prime implicants and minterms, makes their use prohibitive (Brayt 84:9). Hence, heuristic techniques have been developed to handle the PLA-based minimization problem which do not require generation of function minterms or prime implicants. Heuristic techniques for PLA synthesis are characterized by mechanisms for the expansion of function implicants and removal from consideration of other implicants covered by newly expanded implicants (Brayt 84:10). Typically, as an implicant is expanded other implicants are reduced correspondingly in what can be viewed as a "molding" process. Implicants are reduced incrementally by the expansion of other implicants until they are

covered completely and then removed from consideration. The result is a near-minimal number of implicants which cover all of the output functions.

In PLA minimisation, use is made of the concept of *cellular n-cubes* (Prath 67:125-128). Using cellular n-cubes an n-variable Boolean function may be plotted on an n-dimensional cube. Each vertex of the n-dimensional cube, or n-cube, corresponds to a minterm of the function. Vertices may be grouped together if they are adjacent to form implicants of the function; vertices are called adjacent when they are connected by arcs. Groupings which are as large as possible depict prime implicants. Multiple-output functions are represented by a set of cubes in which each cube corresponds to an output of the function. Groupings may be formed between various cubes to indicate the sharing of implicants among output functions. Figure A.4 depicts a 3-cube for the three-variable function $f(x, y, z) = y' + xz$. Points at five of the vertices of the cube denote the minterms of the function. The vertices are labeled according to their xyz coordinate, a one indicating an uncomplemented literal, a zero a complemented literal. The grouping of vertices 001 and 101 forms the implicant $y'z$. The grouping of vertices 101 and 111, and vertices 000, 001, 100, and 101 form the prime implicants xz and y', respectively.

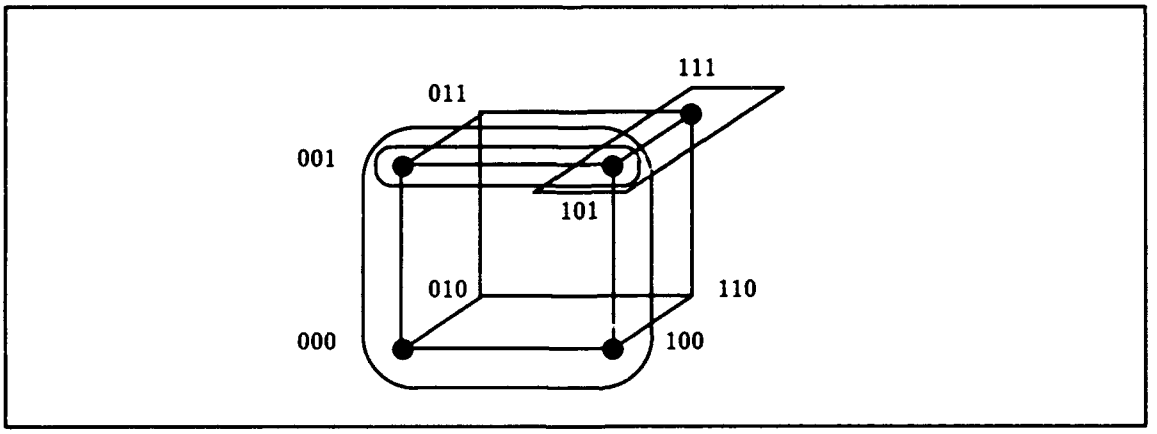


Figure A.4. Boolean 3-cube for $f(x, y, z) = y' + xz$

An example of the molding process of PLA minimization is given in Figures A.5 and A.6. Figure A.5 is an initial multiple-output function $\underline{f}(x, y, z)$, in which

- $\underline{f}(f_1, f_2)$, and
- $f_1(x, y, z) = x'y' + y'z + x'y + xy'z' + x'yz$,
- $f_2(x, y, z) = xy'x' + x'yz + xz$.

A heuristic technique may produce a result as shown in Figure A.6 where

- $f_1(x, y, z) = x' + xy'$, and
- $f_2(x, y, z) = xy' + yz$.

The number of terms is reduced from a total of six to three where the term xy' is shared between the two outputs in the result (two terms are shared between the outputs in the initial function). Of significance is the fact that $f_1(x, y, z)$ could be represented by a simpler formula, i.e., $x' + y'$; however, this would prohibit the sharing of term xy' resulting in a net increase in the number of terms.

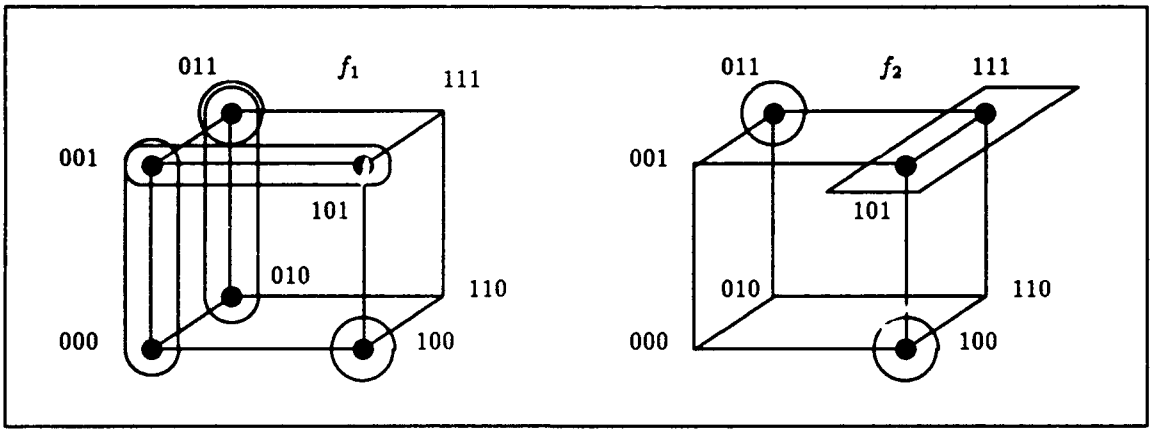


Figure A.5. Multiple-Output Function Prior to PLA Minimization

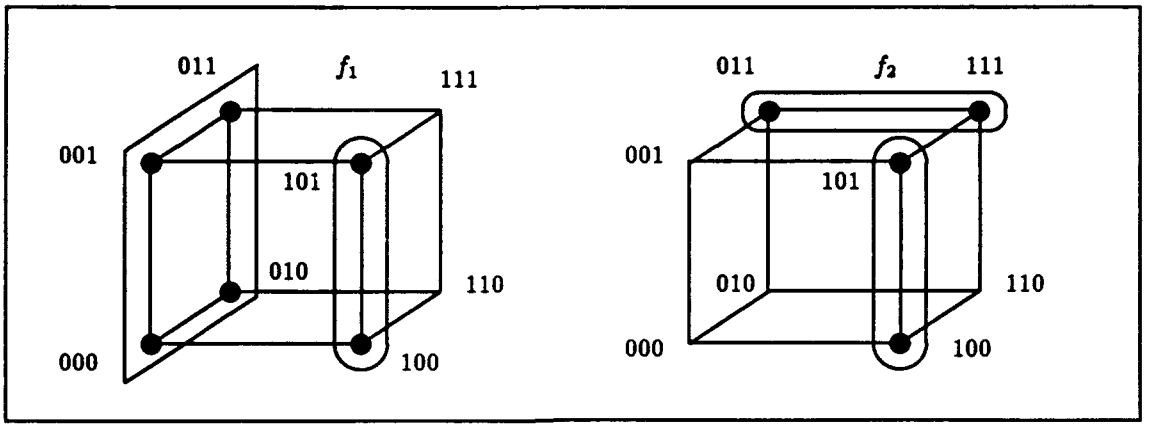


Figure A.6. Multiple-Output Function After PLA Minimization

Simultaneous Identification and Extraction of Implicants. Due to the potentially large number of prime implicants for a logic function, research in the 1970s focused on techniques which did not require the generation of all of the prime implicants of a function. These techniques are characterized by the simultaneous identification and extraction of function implicants. (Rhyne 77) and (Areva 78) describe methods which fit into this category to generate near-minimal two-level circuits.

Rhyne's method (Rhyne 77) requires the initial generation and partitioning of all of the minterms of the function to be minimized. Minterms are then partitioned into the on-set, off-set, and don't-care set.[4] After partitioning of the minterms, one minterm of the on-set is chosen and expanded until all of the prime implicants which cover the minterm are generated. By expansion, we mean that minterms are combined using consensus and absorption to form a prime implicant which covers the minterms used to create it. For example, given a three-variable Boolean function for which the on-set consists only of xyz and xyz' and further suppose that minterm xyz is chosen for expansion. During expansion, minterm xyz is combined with xyz' to form a new term xy using consensus. This new term absorbs the original minterms forming a prime implicant of the original

[4] Rhyne actually calls minterms which belong to the on-set *true forms*, those that belong to the off-set *false forms*, and those that belong to the don't care set *redundancies*. The terminology used here is more common.

function. For non-trivial examples, a minterm selected for expansion will be used to generate a set of PIs, i.e., all of the PIs which cover it. However, only one prime implicant is selected for use. Then all minterms which are covered by the newly-formed prime implicant are removed from consideration. Another minterm is selected and expanded until prime and minterms covered by the next prime implicant selected are removed. The process continues until all minterms in the on-set are covered by PIs. Mechanisms are developed to quickly identify essential PIs. Although the original method could only handle single-output circuits, Rhyne's procedure has been updated to handle multiple-output circuits (Perki 88).

The primary difference between Arevalo's method (Areva 78) and Rhyne's is that only a subset of the prime implicants is generated for each minterm to be expanded. Additionally, only the on-set and don't care-set of minterms are generated and stored. The technique produces irredundant formulas to represent a single-output functions. Reduced formulas–not necessarily irredundant–are generated for multiple-output functions. For multiple-output functions, product functions are formed from multiplying each of the single output functions together. When producing formulas for the multiple-output case, resulting terms may be prime implicants of one of the product functions rather than PIs of individual functions (Areva 78:1032). This technique is faster than Rhyne's, but does not produce as good results (Brayt 84:9).

MINI, PRESTO, and ESPRESSO-II. In the 1970s a variety of heuristic techniques were developed to handle PLA-based multiple-input, multiple-output circuit minimization. Notable examples include MINI, PRESTO, and ESPRESSO-II.

MINI (Hong 74:443) was developed by researchers at IBM to solve the PLA minimization problem. MINI begins by assigning an equal weight to each function implicant. An initial SOP formula is generated, followed by iterative improvement of the solution. MINI uses a three-step process to perform minimization:

1. Each implicant is reduced to the smallest possible size with respect to the number of minterms covered.
2. Implicants are examined in pairs to see if they can by reshaped by reducing one while enlarging the other by the same set of minterms.
3. Each implicant is enlarged to its maximal size with respect to covered minterms and other implicants that are then covered are removed.

The three steps are iterated until no further reduction can be obtained in the size of the solution. The order in which implicants are reduced, reshaped, etc. affects the outcome of the procedure. An advantage of MINI is that all of the minterms of a function do not have to be generated. However, the generation of the complement of a function is required to check if the expansion of an implicant changes the coverage of a function—if a newly expanded implicant intersected with the function complement is null, then coverage is maintained. MINI produces a near-minimal set of implicants, but not necessarily prime implicants, to represent both binary and multi-valued multiple-output functions.

PRESTO, a minimizer developed by Antonin Svoboda, differs somewhat from the MINI approach (Brown 81). In PRESTO, implicants are expanded while implicants covered by the newly-expanded implicants are removed. A final step in PRESTO guarantees that an irredundant cover is generated; however, the cover may include non-prime implicants. Different than MINI, PRESTO does not generate the complement of the input function. Consequently, the expansion process requires a check on whether all minterms covered by the expanded implicant are covered by some other implicant of the cover (Brayt 84:10-11).

After studying the MINI and PRESTO algorithms, researchers developed improved techniques which led to the development of the ESPRESSO-I and ESPRESSO-II procedures for PLA minimisation. The original ESPRESSO-I program was developed as an implementation of the MINI and PRESTO algorithms with switches for controlling the sequence of actions in the program. This allowed experimentation to determine the strengths and weaknesses of each program. Based on the experiments, the authors of the program made two basic conclusions:

- The technique of computing a complement of a function (MINI) was superior to the PRESTO method for checking minterm coverage, and
- Iteration as used by MINI gave a good enough improvement of designs to justify the additional computation. (Brayt 84:11-12)

These conclusions were used to guide the development of the ESPRESSO-I method.

The authors of the ESPRESSO-I program developed improved procedures, many based on Boole's Expansion Theorem, which increased the efficiency of the program. The result of their efforts was the ESPRESSO-II program. The goals of the authors of ESPRESSO-II were:

1. To solve logic minimisation problems with limited computing resources, and
2. To attain results close to a global optimum. (Brayt 84:12)

The sequence of operations in ESPRESSO-II consists of the following:

1. Compute the complement of the function (off-set) in addition to the don't-care set.
2. Expand each implicant into a prime implicant and remove covered implicants.
3. Extract the essential prime implicants and put them into the don't-care set.
4. Find an irredundant cover.
5. Reduce each implicant to a minimum essential implicant.
6. Iterate expansion, irredundant cover, and reduction until no more improvement occurs.
7. Expand, find an irredundant cover, and reduce a final time using a different strategy. If the function can be reduce further, try it again.
8. Include the essential PIs in the cover and make the PLA as sparse as possible. (Brayt 84:12-13)

The ESPRESSO-II algorithm produces an irredundant, prime cover for a PLA-based implementation of a multiple-input, multiple-output circuit. On actual circuits, the authors state that the program produces results which are near-minimal if not a minimal representation of a circuit. Results have been compared to an implementation of the Quine-McCluskey algorithm. However, whereas ESPRESSO-II attained results which were near minimal, the CPU time used

by the Quine-McCluskey method was 10-100 times larger than ESPRESSO-II for large problems (Brayt 84:156).

The ESPRESSO-II algorithm has been implemented and extended in a number of ways. The ESPRESSO-IIC version is a version coded in the C language which is part of the Berkeley VLSI CAD tool environment. ESPRESSO-MV is a version of ESPRESSO designed for use on multi-valued logic. ESPRESSO-MLT extends ESPRESSO to multi-level logic minimization. The techniques contained in ESPRESSO-II are described in a book written by the program developers called *Logic Minimization Algorithms for VLSI Synthesis* (Brayt 84).

Other Methods. Recent work has been performed to further improve PLA minimization techniques. Malik and others have developed a modification for ESPRESSO which does not require the generation of the full off-set (complement) of a function; this algorithm is good for functions where the on-set of a function is small but the off-set is very large (Malik 88). Biswas, et al., have developed several PLA minimization algorithms (Biswa 84, Biswa 86, Gurun 87, Gurun 89). Their latest method includes techniques for fast determination of essential prime implicants without generating all of the prime implicants of a function and does not require generation of a function complement or all of the prime implicants of a function. Finally, PALMINI (Nguye 87) is used to develop minimal solutions without generating all prime implicants through the solution of a graph-coloring problem.

Exact Minimization Methods

Two notable algorithms used to develop minimal two-level designs are the ESPRESSO-EXACT algorithm (Rudel 89, Rudel 86) and McBOOLE (Dagen 86). Both techniques perform the same steps as the Quine-McCluskey method: the generation of all prime implicants of a function and the selection of a minimal set of prime implicants to cover the function. However, the means by which these steps are performed are much different.

McBOOLE uses graph and partitioning techniques to find minimal covers. A directed graph called a *covering graph* is used for determining the relationships among prime implicants of a function. Techniques are provided for determining prime implicants to retain or discard based on the covering graph of a function. Cycles in the graph preclude the selection of prime implicants to retain and discard; a form of search is used to select prime implicants for instances in which cycles appear in the graph. A graph-partitioning technique is used to decompose the problem.

The ESPRESSO-EXACT algorithm is theoretically similar to the ESPRESSO heuristic minimisation technique. Several of the operations used in ESPRESSO, a tautology-based algorithm in particular, are extended for use in ESPRESSO-EXACT. ESPRESSO-EXACT includes techniques for quickly detecting and eliminating from consideration essential prime implicants and selecting a minimum cover using the remaining prime implicants. A branch-and-bound search process is used in the final step of constructing a minimal formula. A technique based on the formation of a maximal independent set is used to control a search process to solve a covering problem involving a reduced form of a prime implicant table.

Both McBOOLE and ESPRESSO-EXACT have proven useful in finding minimal solutions for functions which have up to twenty inputs and twenty outputs and over 9000 prime implicants.

Recursive Realizations of Combinational Logic

A method for generating recursive realizations of combinational logic was developed by Brown (Brown 90) and extended by Knutson (Knuts 90). The intent of this technique is to determine how to reconfigure a design such that outputs are defined in terms of the inputs and other outputs to reduce the circuit cost. One output must be defined solely in terms of the circuit inputs; the next output is defined in terms of the inputs and the previously defined output, etc. This process continues until all outputs are defined, hence, the phrase "recursive" in the name of the method.

A pictorial description of this method is given in Figure A.7. Although the figure depicts that the output z_1 is the output which depends only on the inputs, this choice is arbitrary.

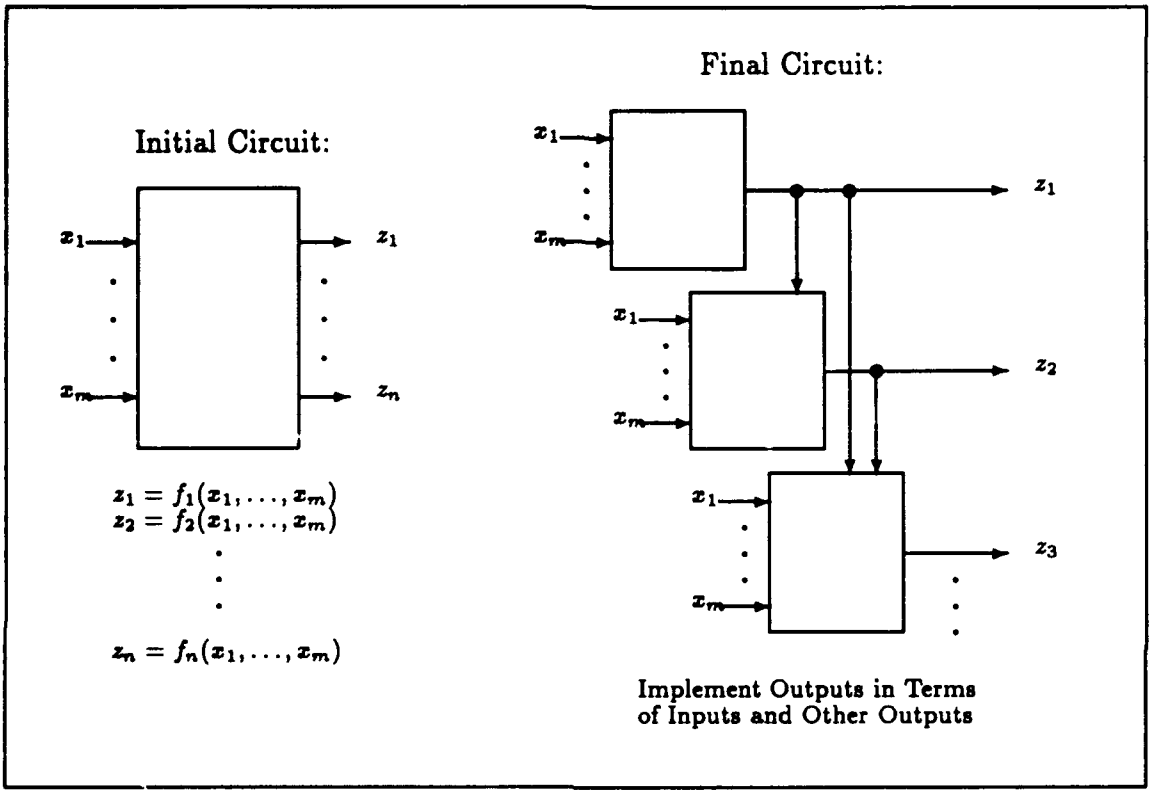


Figure A.7. Recursive Realisations of Combinational Logic

To attain a recursive circuit realization, a two-step process is used. In the first step, a dependency analysis is performed for each output to determine minimal subsets of inputs and other outputs that can be used to generate a given output; these sets are called *minimal determining subsets* (MDS). Typically, various combinations of inputs and outputs may be used to generate an output. Hence, a set of MDSs is developed for each output. Once the set of minimal determining subsets is generated for each output, a sub-minimal formula is developed to represent the output with respect to variables in each subset. A cost is then developed for the formula based on the number of gate inputs required if the formula were to be used as the basis for a design.

The initial specification, given by a system of equations, is reduced to a single Boolean equation $\phi(X, Z) = 1$ which represents the circuit specification. Boolean reasoning is then used to generate minimal determining subsets. After costs are associated with each of the MDSs associated with each output, the second step of the procedure is performed. In this step, a branch-and-bound search is performed to find the combination of MDSs to use to generate each output which produces a least-cost circuit. As an example of this method, consider the specification

$$u = bc + bd + a'cd + a'b'c'd'$$
$$v = a'cd + a'c'd'$$
$$w = a + b'c + b'd + bc'd'.$$

The cost, found by counting the number of gate inputs if the output is implemented in a two-level AND-OR circuit, for this circuit is 34. This cost is determined by counting all literals (the inputs to AND gates) and all terms of the formula (the input to OR gates). However, if a term consists of a single literal, the literal is not counted with the other literals because it is input directly to an OR gate. Additionally, it is assumed that the complement of each input is available. The minimal determining subsets for u, v, and w are

$$u: \ \{\{b, v\}, \{a, b, c, d\}, \{a, c, d, w\}\}$$
$$v: \ \{\{a, b, u\}, \{a, c, d\}\}$$
$$w: \ \{\{a, u\}, \{a, b, v\}, \{a, b, c, d\}\}.$$

Example costs are

- v: $\{a, c, d\}$ costs 8,
- w: $\{a, b, c, d\}$ costs 11,
- w: $\{a, b, v\}$ costs 7, etc.

Using the MDSs and their associated costs, a branch-and-bound search would produce the following least-cost reconfiguration of the circuit function

$$u = b'v + bv'$$
$$v = a'cd + a'c'd'$$
$$w = u' + a$$

The cost of this realisation is 16, significantly cheaper than the original cost of 34. Output v is constructed only in terms of the circuit inputs, output u is constructed using v and a circuit input, etc.

As currently implemented, the system to obtain recursive realizations of combinational circuits uses only the inputs and outputs of a circuit. Intermediate nodes of a circuit are not considered. Additionally, no consideration is given of speed of the resulting circuit. The system is significant due to the use of both Boolean reasoning and search techniques.

Appendix B. Example Functions and Intervals

In this appendix, examples are described to which we apply selected algorithms introduced in this work. The times listed in the tables in this appendix are given in the format:

`hours:minutes:seconds.hundredths of a second.`

The times were obtained using the BORIS toolset. BORIS is an acronym for BOolean Reasoning In Scheme. BORIS consists of a set of procedures for Boolean reasoning programmed in the Scheme dialect of the LISP programming language. All of the procedures described in Chapter 2 of this dissertation are implemented in BORIS.

BORIS was originally developed by Dr. F.M. Brown at the Air Force Institute of Technology and has been revised and extended by the author. The current version of BORIS executes in PC Scheme, a microcomputer-based dialect of Lisp available from Texas Instruments. BORIS was used to prototype selected algorithms presented in this work. Unless otherwise noted, the computer used in these calculations was an 20 Mhz, 80386-based, IBM-compatible computer. PC Scheme was run as a task in the Microsoft Windows environment.

Data Set B

Data set B is a set of randomly-generated switching functions. The definition of this data set is taken from (Cutle 80:208-209).

Given that a function $f(X)$ consists of n variables, we generate a random number r_j between 0 and 1 for each minterm m_j to determine if the corresponding discriminant is equal to 0 or 1:

- if $r_j > 0.25$, then the discriminant which corresponds to m_j is equal to 0;
- otherwise, the discriminant which corresponds to m_j is equal to 1.

The expected number of discriminants of a function which are equal to 1 is $(1/4) \cdot (2^n)$; however, the actual number varies. Statistics on data set B to include the number of variables of $f(X)$, the actual number of discriminants of $f(X)$ which are equal to 1, and the number of prime implicants (PIs) of each function are listed in Table B.1. The last column of Table B.1 lists the number of prime implicants in the IDF representing $f(X)$ which consists of the fewest number of terms.

Function Identifier	No Vars n	Expected No Minterms	Actual No Minterms	Number PIs	Essen PIs	Inessen PIs	Least No PIs - IDF
B1	4	4	4	3	3	0	3
B2	5	8	10	5	4	1	4
B3	6	16	13	11	10	1	10
B4	7	32	28	25	19	3	20
B5	8	64	58	48	26	3	32
B6	9	128	135	127	33	7	67
B7	10	256	231	206	72	11	116
B8	11	512	543	525	112	13	253

Table B.1. Data Set B (Statistics)

The number of terms of the Blake canonical form, a simplified formula, and an irredundant SOP formula (IDF) to represent each function as well as the times required to calculate the respective formulas are given in Table B.2. The method used to generate each Blake canonical form is the recursive multiplication method (Procedure 2.20). Procedure 2.15 is the method used to formulate a simplified formula to represent a given function. Additionally, Procedure 2.31 is used to obtain an IDF; the time required to obtain the irredundant formula includes the time required to first generate the Blake canonical form of the function.

Data Set C

Data set C is a set of randomly-generated n-variable switching functions, in which $n = 12$. The definition of this data set is taken from (Cutle 80:210).

Function Identifier	Number PIs	Time BCF	No Terms Simp Form	Time Simp Form	No Terms IDF	Time IDF
B1	3	0.05	3	0.06	3	0.05
B2	5	0.16	4	0.11	4	0.17
B3	11	0.33	10	0.33	10	0.44
B4	25	1.04	20	1.05	20	1.93
B5	48	3.08	35	2.47	33	7.47
B6	127	14.56	78	7.69	68	58.22
B7	206	40.48	138	20.93	127	3:34.16
B8	525	3:49.54	316	1:29.48	274	35:55.82

Table B.2. Data Set B (Calculation Times)

In this data set an implicant is formed by generating n random numbers, $r_1, \ldots, r_n$, between 0 and 1. For each $i = 1, \ldots, n$:

- if $r_i < 1/3$, then literal x_i' is contained in the implicant;
- if $r_i > 2/3$, then literal x_i is contained in the implicant;
- otherwise, neither x_i' nor x_i is contained in the implicant.

The functions in the data set differ by the number of implicants generated to form the function specification; the number of implicants generated for a given function is listed under the "Number Terms" column in the table. Statistics on data set C are listed in Table B.3.

Function Identifier	Number Terms	Number PIs	Essen PIs	Inessen PIs	Least No PIs - IDF
C1	10	16	10	6	10
C2	20	51	19	29	20
C3	30	113	29	84	29
C4	40	149	37	105	39
C5	50	321	42	186	49
C6	60	407	37	135	58
C7	70	446	44	228	61

Table B.3. Data Set C (Statistics)

Table B.4 states the number of terms of the Blake canonical form, a simplified formula, and an IDF which represents each function as well as the times required to calculate the respective formulas.

Procedure 2.20 (Blake Canonical Form - Recursive Multiplication) is the method used to generate the Blake canonical form. The procedure used to formulate a simplified formula to represent each function is Procedure 2.15. Additionally, Procedure 2.31 is used to obtain an irredundant formula to represent each function. The time required to obtain the irredundant formula includes the time required to first generate the respective function's Blake canonical form.

Function Identifier	Number PIs	Time BCF	No Terms Simp Form	Time Simp Form	No Terms IDF	Time IDF
C1	16	1.37	11	1.37	10	1.81
C2	51	5.00	24	2.85	20	10.60
C3	113	16.36	31	4.83	29	54.49
C4	149	28.34	45	6.37	39	1:38.38
C5	321	2:21.38	87	12.14	49	11:34.48
C6	407	6:35.29	105	16.31	59	26:17.30
C7	446	5:55.15	102	17.03	61	29:40.19

Table B.4. Data Set C (Calculation Times)

Data Set D

Data set D is a set of randomly-generated n-variable switching functions formed in the same fashion as data set C, in which $n = 18$. The definition of this data set is taken from (Cutle 80:210). Statistics on data set D are listed in Table B.5.

The number of terms of the Blake canonical form, a simplified formula, and an irredundant formula which represents each function as well as the times required to calculate the respective formulas are listed in Table B.6. The time required to obtain an irredundant formula to represent a function includes the time required to first generate the Blake canonical form.

Function Identifier	Number Terms	Number PIs	Essen PIs	Inessen PIs	Least No PIs - IDF
D1	10	12	10	2	10
D2	20	25	20	5	20
D3	30	64	30	34	30
D4	40	95	40	55	40
D5	50	134	50	84	50
D6	60	185	60	125	60
D7	70	205	70	135	70
D8	80	299	80	219	80
D9	90	586	84	480	86
D10	100	434	100	334	100
D11	110	564	107	445	108
D12	120	593	119	474	119

Table B.5. Data Set D (Statistics)

Function Identifier	Number PIs	Time BCF	No Terms Simp Form	Time Simp Form	No Terms IDF	Time IDF
D1	12	1.59	10	2.14	10	1.87
D2	25	6.04	21	6.26	20	7.91
D3	64	17.64	30	12.36	30	29.00
D4	95	21.53	40	14.39	40	45.37
D5	134	53.34	51	23.84	50	1:50.40
D6	185	1:13.27	61	27.41	60	3:19.54
D7	205	1:32.76	72	35.10	70	4:26.33
D8	299	3:02.14	85	44.87	80	9:10.96
D9	586	10:38.67	103	58.11	86	1:02:03.89
D10	434	6:08.66	110	59.27	100	28:48.83
D11	564	9:38.54	125	1:13.05	108	*
D12	593	10:31.26	128	1:21.73	119	*

Table B.6. Data Set D (Calculation Times)

Data Set E

Data set E is a set of randomly-generated n-variable switching functions formulated in the same fashion as data set C, in which $n = 24$. The definition of this data set is taken from (Cutle 80:210). Statistics on data set E are listed in Table B.7. A characteristic of this data set is that the disjunction of the essential prime implicants of each function forms the only irredundant SOP formula which may represent the function.

Function Identifier	Number Terms	Number PIs	Essen PIs	Inessen PIs	Least No PIs - IDF
E1	10	10	10	0	10
E2	20	25	20	5	20
E3	30	42	30	12	30
E4	40	53	40	13	40
E5	50	65	50	15	50
E6	60	87	60	27	60
E7	70	109	70	39	70
E8	80	130	80	50	80
E9	90	184	90	94	90
E10	100	165	100	65	100
E11	110	215	110	105	110
E12	120	220	120	100	120
E13	130	225	130	95	130
E14	140	347	140	207	140
E15	150	284	150	134	150
E16	160	362	160	202	160

Table B.7. Data Set E (Statistics)

The number of terms of the Blake canonical form, a simplified formula, and an irredundant SOP formula to represent each function as well as the times required to calculate the respective formulas are given in Table B.8. The time required to obtain an irredundant formula to represent a function includes the time required to first generate the Blake canonical form.

Function Identifier	Number PIs	Time BCF	No Terms Simp Form	Time Simp Form	No Terms IDF	Time IDF
E1	10	2.25	10	2.53	10	2.47
E2	25	6.92	20	6.97	20	8.95
E3	42	16.59	30	14.12	30	23.01
E4	53	24.00	40	21.04	40	34.33
E5	65	33.39	50	29.06	50	47.62
E6	87	56.35	61	43.83	60	1:27.33
E7	109	1:12.56	71	51.46	70	2:08.53
E8	130	1:24.43	80	1:00.26	80	2:34.62
E9	184	3:00.76	91	1:33.54	90	5:57.13
E10	165	2:40.94	100	1:31.17	100	4:58.47
E11	215	3:41.24	111	1:47.93	110	7:56.75
E12	220	3:50.68	120	1:55.95	120	8:56.51
E13	225	4:24.41	130	2:20.61	130	9:36.94
E14	347	8:09.61	142	2:53.35	140	*
E15	284	5:47.57	150	2:46.20	150	*
E16	362	8:47.06	165	3:28.71	160	*

Table B.8. Data Set E (Calculation Times)

Data Set IC

Data set IC is a set of 15 randomly-generated intervals defined by a lower-bound function $g(X)$ and an upper-bound function $h(X)$. The definition of this data set is taken from (Hong 83:106).

Given the number n of variables and a specified minterm density d, we generate a random number r_j between 0 and 1 for each minterm m_j of the 2^n possible minterms to determine if the minterm should be a member of the on-set, off-set, or don't care-set:

- if $r_j > d$, then the minterm m_j is in the off-set;
- if $r_j \leq d/2$, then the minterm m_j is assigned to the on-set;
- otherwise, $d/2 < r_j \leq d$. In this case, the minterm is assigned to the don't care-set.

Statistics on data set IC to include the number of variables, the minterm density d, and the number of minterms in the on-set and dc-set for each function are listed in Table B.9. The number of prime implicants for each function in Table B.9 is the number of prime implicants of the function $h(X)$.

Additionally, the number of terms listed for a simplified formula is the number of terms in a simplified formula representing the lower bound function $g(X)$.

Function Identifier	No Vars n	Density d	Minterms ON-Set	Minterms DC-Set	BCF PIs	Essen PIs	Inessen PIs	Least No PIs - IDF
IC1	6	0.2	3	3	6	3	0	3
IC2	6	0.3	11	4	12	1	0	3
IC3	6	0.4	15	4	14	2	0	3
IC4	7	0.2	15	11	23	7	0	8
IC5	7	0.3	19	18	25	4	0	8
IC6	7	0.4	18	24	34	10	0	14
IC7	8	0.2	26	25	38	18	1	20
IC8	8	0.3	37	43	61	24	5	27
IC9	8	0.4	55	48	96	8	0	29
IC10	9	0.2	51	47	81	24	3	34
IC11	9	0.3	73	73	136	31	7	49
IC12	9	0.4	89	108	183	14	1	57
IC13	10	0.2	109	105	206	45	5	77
IC14	10	0.3	139	170	295	38	2	99
IC15	10	0.4	202	204	398	23	2	112

Table B.9. Data Set IC (Statistics)

The number of terms of the Blake canonical form of $h(X)$, a simplified formula for $g(X)$, and an irredundant SOP formula to represent a function $f(X)$ belonging to each interval $[g(X), h(X)]$ as well as the times required to calculate the respective formulas are given in Table B.10. The method used to generate the Blake canonical form of each $h(X)$ is the recursive multiplication method (Procedure 2.20). Procedure 2.15 is the method used to formulate a simplified formula to represent each $g(X)$. Additionally, Procedure 2.33 is used to obtain an irredundant formula. The time required to obtain each irredundant formula includes the time required to first generate the Blake canonical form for $h(X)$ and a simplified formula for $g(X)$.

Function Identifier	Number PIs	Time BCF	No Terms Simp Form	Time Simp Form	No Terms IDF	Time IDF
IC1	6	0.17	3	0.11	3	0.44
IC2	12	0.28	3	0.16	3	0.88
IC3	14	0.44	3	0.17	3	1.09
IC4	23	0.93	8	0.50	8	5.60
IC5	25	0.93	13	0.88	9	13.02
IC6	34	1.38	17	1.10	15	25.65
IC7	38	2.31	20	1.54	21	52.73
IC8	61	5.06	29	2.69	27	3:15.10
IC9	96	8.57	34	3.07	32	10:27.03
IC10	81	7.80	35	3.74	35	7:45.99
IC11	136	17.47	53	6.59	52	43:02.77
IC12	183	29.33	79	11.04	61	2:08:50.16
IC13	206	39.16	80	12.58	79	2:52:17.04
IC14	295	1:15.58	119	21.31	106	11:07:56.23
IC15	398	2:09.51	141	27.52	123	*

Table B.10. Data Set IC (Calculation Times)

Appendix C. Computational Results

In this appendix, we discuss the computational results of applying two of the algorithms presented in Chapter 6 to the several sets of examples introduced in Appendix B. In the first section, we discuss a number of important issues with respect to the prototypes of the algorithms. The results of applying Algorithms 6.1 and 6.2 are then presented for several data sets; the fewest-gates cost criterion is used for all examples listed in this appendix. The final section describes the results of applying the search strategies discussed in Chapter 9 to a number of example functions.

Prototype Overview

Prototype implementations were developed for Algorithms 6.1 and 6.2 discussed in Chapter 6. The initial program development was accomplished using PC Scheme and a 20 MHz, 80386-based, IBM-compatible computer. After development on a PC, the procedures were ported to the T programming environment, Version 3.1, hosted on a SUN SPARCStation 2. The T language—a dialect of LISP—was developed at Yale University and is very similar to Scheme. The author developed a set of translation routines which allow the use of code developed for PC Scheme in the T environment with very little modification.[1] The results presented in this chapter were developed using the T-based implementation. The ability to use a workstation facilitated the handling of larger problems than was possible using a personal computer, particularly due to memory constraints. Moreover, the T-based implementation running on a SUN SPARCStation 2 is on the average 11-12 times faster than the PC Scheme implementation for a given operation. (The speed-up appears to be invariant no matter what operation is performed.)

For each set of examples, the time that it took to apply each algorithm to each member in the set is given. This information should not treated as a measure of the true utility of the algorithms

[1] The macro facilities for PC Scheme and T differ. We could not determine how to translate macros appearing in Scheme code to macros in T. Hence, different macro routines must be written for the Scheme-based and T-based implementations. However, macros are only used in input and output routines, and not in the implementation of any procedures in this dissertation.

in this work. The implementations are rough prototypes which do not faithfully represent the algorithms presented in the text. Rather, they were developed in the course of research in an attempt to validate the utility of various ideas. Program effectiveness was the primary consideration during the development of the prototypes; program efficiency was not a major concern. After procedures presented in the text were formalized, the implementations were not changed to reflect the theory. Implementations based on the text will likely be "cleaner" than the programs used to develop the data in this appendix.

Another issue that must be considered in examining the speed of these prototypes is the implementation language. Since our concern was to quickly develop working programs, we chose to use a LISP-based environment to implement the procedures. A procedural-based implementation—such as C—would have required a significant time investment and would not have lended itself to experimentation as did a LISP-based implementation. A future task is to recode the procedures presented in this work using C. A procedural implementation of the techniques should provide a gain in speed over the implementations used in this work.

Data Set B Results

Algorithm 6.1 uses the set of all useful, conditionally-eliminable prime implicants as the base for developing inclusion formulas. Table C.1 presents the results of applying Algorithm 6.1 to the functions in Data Set B. In the first column, the number of inclusion formulas developed in the course of the algorithm is listed; this number corresponds to the number of terms in the base. The total number of terms in the resulting set of inclusion formulas is given in the second column. The average number of terms per inclusion formula is found by dividing the number in the second column by the number in the first column.

In the third, fourth, and fifth columns, the number of prime implicants identified for containment in a minimal formula F during different stages of the process is given. The number of essential

prime implicants is listed in the third column. The fourth column gives the number of prime implicants identified during rule reduction. Finally, the fifth column lists the number of prime implicants determined during the search process. The A* search strategy was used during the search process with heuristic function $h_1(n)$, Topology #1, and an explicit node representation.[2] Column six gives the number of prime implicants in a minimal formula F; this number is derived by summing the numbers in columns three through five. Finally, the total time to develop a minimal F is given in the last column.

Function Identifier	Num IFs	Num Terms	Essen PIs	Rul Red PIs	Search PIs	Total PIs	Total Time
B1	-	-	3	-	-	3	0.62
B2	-	-	4	-	-	4	0.69
B3	-	-	10	-	-	10	0.74
B4	3	6	19	1	-	20	0.99
B5	19	33	26	6	-	32	2.16
B6	87	187	33	34	-	67	10.36
B7	123	236	72	37	7	116	28.75
B8	400	1370	112	129	12	253	263.58

Table C.1. Data Set B - Algorithm 6.1

Table C.2 gives the results of applying Algorithm 6.2 to Data Set B. In Algorithm 6.2 an irredundant disjunctive form is developed; the base for each function is the set of conditionally-eliminable prime implicants appearing in the IDF. Hence, the number of inclusion formulas developed for each function is greatly reduced, which decreases memory requirements. However, the times in Table C.2 are greater than in Table C.1 due to the time required to form an IDF. Additionally, the reduction of the number of inclusion formulas shifts effort to the search process. Whereas most CEPIs used to constitute a minimal F are identified during rule reduction in Algorithm 6.1, a search process is important for identifying prime implicants in Algorithm 6.2. The computation

[2]Unless otherwise noted, this search configuration is used for all examples in this appendix, except as discussed in the final section of the appendix.

time is not listed for function B8; data for function B8 will be discussed in the section on search strategies at the end of this appendix.

Function Identifier	Num IFs	Num Terms	Essen PIs	Rul Red PIs	Search PIs	Total PIs	Total Time
B1	-	-	3	-	-	3	0.74
B2	-	-	4	-	-	4	0.74
B3	-	-	10	-	-	10	0.79
B4	1	2	19	-	1	20	1.12
B5	7	11	26	3	3	32	2.32
B6	36	63	33	21	13	67	11.51
B7	54	97	72	18	26	116	39.31
B8	163	551	112	25	116	253	*

Table C.2. Data Set B - Algorithm 6.2

Since the number of terms in Base #3 is comparable to the number of CEPIs in the IDF (See Table 6.1), we speculate that an implementation of Algorithm 6.3 should be faster than the implementation of Algorithm 6.2.

Data Set C Results

The results of applying Algorithm 6.1 to functions in Data Set C are given in Table C.3. Additionally, Table C.4 lists the result of applying Algorithm 6.2 to each of the functions. For this data set, Algorithm 6.2 generally produces results faster than Algorithm 6.1. The reason for this is that the average number of terms per inclusion formula is large for this data set; hence, a significant amount of time is spent deriving inclusion formulas for each term of the base. The time that it takes to form an IDF in Algorithm 6.2 is more than offset by the reduction in the number of inclusion formulas which must be formed.

Function Identifier	Num IFs	Num Terms	Essen PIs	Rul Red PIs	Search PIs	Total PIs	Total Time
C1	-	-	10	-	-	10	1.20
C2	3	3	19	1	-	20	2.49
C3	-	-	29	-	-	29	6.34
C4	7	7	37	2	-	39	10.89
C5	93	1572	42	7	-	49	113.32
C6	235	12471	37	21	-	58	4799.31
C7	174	3591	44	17	-	61	319.13

Table C.3. Data Set C - Algorithm 6.1

Function Identifier	Num IFs	Num Terms	Essen PIs	Rul Red PIs	Search PIs	Total PIs	Total Time
C1	-	-	10	-	-	10	1.19
C2	1	2	19	-	1	20	2.58
C3	-	-	29	-	-	29	6.31
C4	2	4	37	-	2	39	10.75
C5	7	419	42	-	7	49	95.63
C6	22	875	37	3	18	58	1428.54
C7	20	148	44	3	14	61	131.38

Table C.4. Data Set C - Algorithm 6.2

Data Set D Results

Data Set D is characterised by functions which have few conditionally-eliminable prime implicants. Hence, minimal formulas which represent functions in this data set consist primarily of essential prime implicants. Table C.5 contains the results of applying Algorithm 6.1 to functions in Data Set D. Virtually all time spent developing a minimal formula is spent partitioning the prime implicants; in particular, most processing time is devoted to identifying the essential prime implicants. The result of applying Algorithm 6.2 differ little from the data in Table C.5 with the exception that fewer inclusion formulas must be developed for functions D9 and D11.

Function Identifier	Num IFs	Num Terms	Essen PIs	Rul Red PIs	Search PIs	Total PIs	Total Time
D1	-	-	10	-	-	10	1.24
D2	-	-	20	-	-	20	2.52
D3	-	-	30	-	-	30	6.18
D4	-	-	40	-	-	40	9.34
D5	-	-	50	-	-	50	17.64
D6	-	-	60	-	-	60	26.14
D7	-	-	70	-	-	70	35.62
D8	-	-	80	-	-	80	64.31
D9	22	102	84	2	-	86	230.64
D10	-	-	100	-	-	100	139.62
D11	12	60	107	1	-	108	225.19
D12	-	-	119	-	-	119	231.50

Table C.5. Data Set D - Algorithm 6.1

Data Set IC Results

Data Set IC is a set of intervals in which each is defined by a lower-bound function $g(X)$ and an upper-bound function $h(X)$, i.e., don't-care conditions exist. Table C.6 lists the results of applying Algorithm 6.1 to each interval. All prime implicants contained in a minimal formula F representing a function $f(X)$ belonging to the interval $[g(X), h(X)]$ are either essential or are identified during the application of reduction rules. Hence, search is not required using Algorithm 6.1 for members of this data set.

The results of applying Algorithm 6.2 to members of Data Set IC are given in Table C.7. Since a smaller base is used in Algorithm 6.2 than in Algorithm 6.1, fewer inclusion formulas are generated for each interval. However, much effort is shifted to the search process. The formation of an irredundant formula and the use of a search process causes the time required to develop a minimal formula to be greater for Algorithm 6.2 than for Algorithm 6.1. An implementation of Algorithm 6.3 should be an improvement over Algorithm 6.2 since the number of terms in Bases #2 and #3 is comparable and an IDF does not have to be formed in Algorithm 6.3 (See Table 6.3). The total time for developing a minimal formula for intervals IC12, IC14, and IC15 is discussed

Function Identifier	Num IFs	Num Terms	Essen PIs	Rul Red PIs	Search PIs	Total PIs	Total Time
IC1	-	-	3	-	-	3	0.77
IC2	5	10	1	2	-	3	0.71
IC3	4	11	2	1	-	3	0.71
IC4	3	6	7	1	-	8	0.80
IC5	15	45	4	4	-	8	1.54
IC6	13	31	10	4	-	14	1.49
IC7	5	10	18	2	-	20	1.55
IC8	10	20	24	3	-	27	2.63
IC9	69	241	8	21	-	29	10.37
IC10	27	57	24	10	-	34	4.75
IC11	57	163	31	18	-	49	12.24
IC12	147	752	14	43	-	57	90.96
IC13	92	264	45	32	-	77	29.78
IC14	197	762	38	61	-	99	129.51
IC15	324	2573	23	89	-	112	1215.19

Table C.6. Data Set IC - Algorithm 6.1

in the next section; the times given is Tables C.6 and C.7 are for the A* search strategy using heuristic function $h_1(n)$.

Search Results

In this section, the result of applying the search strategies described in Chapter 9 to a number of examples is discussed. The examples dealt with in this section are ones for which the search process is non-trivial, i.e., considerable effort is required during the search process. In each case, we discuss the search process required after the application of reduction rules in Algorithm 6.2. Examples discussed in this section are B8, C6, IC12, IC14, and IC15.

For each example, we present data developed applying the search strategies discussed in Chapter 9 in a number of configurations. For every search strategy, Topology #1 with an explicit node representation was used. Six different search strategy configurations were applied to each example:

Function Identifier	Num IFs	Num Terms	Essen PIs	Rul Red PIs	Search PIs	Total PIs	Total Time
IC1	-	-	3	-	-	3	0.74
IC2	2	4	1	1	1	3	0.81
IC3	1	3	2	-	1	3	0.80
IC4	1	2	7	-	1	8	1.03
IC5	4	14	4	-	4	8	1.45
IC6	5	12	10	3	1	14	1.50
IC7	3	6	18	2	-	20	1.64
IC8	3	7	24	2	1	27	2.51
IC9	24	77	8	9	12	29	19.47
IC10	10	22	24	7	3	34	5.77
IC11	22	58	31	12	6	49	22.50
IC12	47	251	14	13	30	57	743.69
IC13	34	103	45	14	18	77	85.39
IC14	70	272	38	25	36	99	867.16
IC15	105	928	23	22	67	112	*

Table C.7. Data Set IC - Algorithm 6.2

1. A* search using $h_1(n)$;
2. A* search using $h_2(n)$;
3. dynamic-weighting using $h_1(n)$, $\epsilon = 0.1$, and $N = 0.333 \cdot$ number of PIs in IFs;
4. dynamic-weighting using $h_2(n)$, $\epsilon = 0.1$, and $N = 0.333 \cdot$ number of PIs in IFs;
5. beam search using $h_2(n)$ and a width $w = 4$; and
6. static-weighting using $h_2(n)$ and a weight $W = 2$.

Each search strategy is measured based on the following criteria:

1. number of nodes generated;
2. number of nodes expanded;
3. number of nodes found to be over the upper bound (if applicable);
4. cost of the solution found; and
5. time of the search.

Example C6. Search data for function C6 is given in Table C.8. The first and second columns of the table are the number of nodes generated and expanded, respectively. The number of nodes generated which are greater than or equal to a previously-determined upper bound is

given in the third column; an upper bound calculation is not developed for either the dynamic or static-weighting methods. The cost of the resulting solution is in the fourth column. The fifth column lists the amount of time spent in Algorithm 6.2 up to the point prior to the search process. The time spent in the search process is given in the sixth column, followed by the total time in the last column.

The minimal cost of a formula for C6 is 58. Each search strategy used produces a least-cost result. The use of the heuristic function $h_2(n)$ rather than $h_1(n)$ yields a significant improvement in the efficiency of the algorithm. For this example, the A* search using function $h_2(n)$ produces a result quicker than any other method, while the A* search using the admissible heuristic function $h_1(n)$ requires the most processing time.

	Nodes Gen	Nodes Expand	Number $\geq UB$	Cost of Result	Prior Time	Search Time	Total Time
A* - $h_1(n)$	28	14	15	58	1130.91	284.87	1415.78
A* - $h_2(n)$	4	2	3	58	1130.91	169.97	1300.88
DW - $h_1(n)$	72	36	-	58	1130.91	244.35	1375.26
DW - $h_2(n)$	36	18	-	58	1130.91	183.84	1314.75
Beam - $w = 4$ - h_2	2	1	2	58	1130.91	184.54	1315.45
SW - $W = 2$ - h_2	36	18	-	58	1130.91	184.65	1315.56

Table C.8. C6 (Search Data)

Example IC12. Search data for interval IC12 is given in Table C.9. The least-cost formula for IC12 consists of 57 terms. As is the case with C6, the use of function $h_2(n)$ rather than $h_1(n)$ yields a significant improvement in the efficiency of the algorithm. Other than the static-weighting method which yields a formula of 58 terms, each search strategy produces a least-cost result. However, the static-weighting method produces a result faster than other methods. Hence, there is a trade-off in this example between the quickness with which we attain a result versus the minimality of the result.

	Nodes Gen	Nodes Expand	Number $\geq UB$	Cost of Result	Prior Time	Search Time	Total Time
A* - $h_1(n)$	480	240	0	57	250.79	492.90	743.69
A* - $h_2(n)$	54	27	8	57	250.79	54.20	304.99
DW - $h_1(n)$	276	138	-	57	250.79	208.02	458.81
DW - $h_2(n)$	68	34	-	57	250.79	47.62	298.41
Beam - $w = 4$ - h_2	154	77	35	57	250.79	83.58	334.37
SW - $W = 2$ - h_2	54	27	-	58	250.79	28.29	279.08

Table C.9. IC12 (Search Data)

Example IC14. Table C.10 lists search data for interval IC14. All of the search strategies with the exception of static-weighting produce a minimal formula consisting of 99 terms. Static-weighting yields a formula containing 100 terms; however, the static-weighting method also produces a result faster than other search strategies. Additionally, as in previous examples, the heuristic function $h_2(n)$ when used with A* and dynamic-weighting significantly increases the efficiency of the search without a degradation in the quality of the solution. Beam search requires more effort than several other strategies due to the fact that we are requiring that four nodes be expanded at each level of the search tree; in other strategies fewer nodes are expanded at each level. However, for functions which are highly complex, beam search may be used to develop a solution that is unattainable using other methods.

Most of the time spent in the production of a minimal formula is spent developing an irredundant disjunctive form when forming a base in Algorithm 6.2. The use of Algorithm 6.3 should significantly decrease the computational time required to develop a solution.

Function IC15. Search data for interval IC15 is given in Table C.11. For this example, a solution could only be developed in the search process using the beam search, static weighting, and dynamic weighting search strategies. Using either heuristic function, the A* search did not yield a solution after over 90 minutes of runtime and was not progressing towards a solution. Additionally, a result was not developed in 90 minutes of runtime using $\epsilon = 0.1$ for the dynamic weighting

	Nodes Gen	Nodes Expand	Number $\geq UB$	Cost of Result	Prior Time	Search Time	Total Time
A* - $h_1(n)$	279	146	28	99	735.18	131.98	867.16
A* - $h_2(n)$	39	20	16	99	735.18	20.97	756.15
DW - $h_1(n)$	311	162	-	99	735.18	146.96	882.14
DW - $h_2(n)$	61	31	-	99	735.18	18.32	753.50
Beam - $w = 4$ - h_2	55	28	26	99	735.18	30.44	765.62
SW - $W = 2$ - h_2	50	25	-	100	735.18	10.99	746.17

Table C.10. IC14 (Search Data)

strategy (the setting used for other examples). However, after increasing ϵ to 0.5, a solution was attainable. Increasing ϵ to 1 significantly increased the speed of the search process, without an increase in the cost of the solution. As in other examples, a solution was developed faster using static weighting than when using other strategies.

For each of the search strategies, the best solution attainable is a formula with 114 terms. A minimal-cost formula, developed using Algorithm 6.1, is 112 terms. Thus, although a minimal formula was not attained using Algorithm 6.2, one reasonably close to a least-cost result was developed. An irredundant formula for IC15 developed using Procedure 2.33 outlined in Chapter 2 consists of 123 terms (see B.10); hence, the result developed using Algorithm 6.2 and a search strategy such static weighting is better than one attained using a more simplistic approach.

	Nodes Gen	Nodes Expand	Number $\geq UB$	Cost of Result	Prior Time	Search Time	Total Time
DW - $h_2(n)$ - $\epsilon = 0.5$	199	100	-	114	3499.73	1144.54	4644.27
DW - $h_2(n)$ - $\epsilon = 1$	122	61	-	114	3499.73	412.2	3911.93
Beam - $w = 4$ - h_2	225	113	19	114	3499.73	1678.63	5178.36
SW - $W = 2$ - h_2	112	56	-	115	3499.73	390.55	3890.28

Table C.11. IC15 (Search Data)

Example B8. Search data for function B8 is given in Table C.12. Similar to IC15, a result could not be developed in a reasonable amount of time for B8 using A* search. However, we also were unable to produce a result using dynamic weighting for B8. Using beam search a result

consisting of 257 terms was developed, compared to a least-cost formula of 253 terms constructed using Algorithm 6.1. An IDF developed using Procedure 2.31 in Chapter 2 consists of 274 terms (see B.2). Thus, we were able to develop a cheaper formula than attainable using a simpler method.

We conjecture that examples IC15 and B8 will be more easily handled once the graph-based decomposition strategy outlined at the end of Chapter 9 is implemented.

	Nodes Gen	Nodes Expand	Number $\geq UB$	Cost of Result	Prior Time	Search Time	Total Time
Beam - $w = 4$ - h_2	344	172	26	257	387.05	3979.56	4366.61

Table C.12. B8 (Search Data)

Appendix D. Procedures

In this appendix, the procedure format used throughout this dissertation is introduced. Procedures and algorithms are generally listed at the point in the text at which the theory for the method is described. However, for the sake of brevity, this appendix contains the procedures for which the theoretical basis is described in Chapters 2 and 8, which are primarily background chapters.

We make a distinction between what are called "procedures" and "algorithms" in this dissertation. *Procedures* are simple techniques that are used as the "building blocks" for larger methods. We designate as *algorithms* the methods used to produce minimal formulas. In this distinction, procedures compose algorithms. Procedures and algorithms are written in a manner that should facilitate easy computer implementation.

Procedure Format

Each procedure or algorithm is given in a step-by-step format. We have attempted to form each procedure so that a specific action is taken in a step. However, substeps are sometimes contained in a step. We denote such a condition by the form:

Step X.

1. first substep;
2. second substep; and
3. third substep.

Steps for which only one of several actions is to be performed—similar to a case statement—are denoted by the form:

Step Y.

- action one;
- action two; or
- action three.

Additionally, **Step 0** in procedures is reserved for initialization steps such as setting an accumulator to some initial value.

Chapter 2 - Fundamentals of Boolean Reasoning

Procedure 2.1 (Boolean Addition): Given two Boolean SOP formulas F and G which represent Boolean functions f and g, $f + g$ is computed in the following manner:

Step 1. If F consists of no terms, return G. Otherwise, continue.

Step 2. If the first term in F is absorbed by any term in G, remove the term from F and repeat Step 2. Otherwise, continue.

Step 3. Remove from G any terms absorbed by the first term in F. Remove the first term from F, append it to G, and return to Step 1.

Procedure 2.2 (Cross-Product): Given two Boolean SOP formulas F and G which represent Boolean functions f and g, $f \times g$ is computed in the following manner:

Step 0. Initialise an accumulator ACC to empty.

Step 1. If F consists of no terms, return ACC. Otherwise, continue.

Step 2. Multiply the first term in F successively by each term in G. The term-by-term multiplication is performed in the following manner:

- If the term in F and the term in G have any opposed literals, the result is 0 by the complements axiom (2.9).
- If the term in F and the term in G have duplicate literals, the terms are appended together and the duplicates are removed.
- Otherwise, the terms simply are appended together.

Resulting terms which are not 0 are added to ACC.

Step 3. Remove the first term from F, and return to Step 1.

Procedure 2.3 (Unate Cross-Product): Given two Boolean SOP formulas F and G which represent relatively unate Boolean functions f and g, $f \times g$ is computed in the following manner:

Step 0. Initialise an accumulator ACC to empty.

Step 1. If F consists of no terms, return $ABS(ACC)$. $ABS(ACC)$ is the minimum-term formula representing $f \times g$. Otherwise, continue.

Step 2. Multiply the first term in F successively by each term in G. The term-by-term multiplication is performed in the following manner:

- If the term in F and a term in G have duplicate literals, the terms are appended together and the duplicates are removed.
- Otherwise, the terms simply are appended together.

All resulting terms are added to ACC.

Step 3. Remove the first term from F, and return to Step 1.

Procedure 2.4 (Boolean Division): Given an SOP formula F representing a function f and a term t, f/t is computed as follows:

Step 0.

- If $t = 1$, return F. ($f/1$ is defined as the function f.)
- Otherwise, continue to Step 1.

Step 1.

- If t consists of no literals, return F.
- Otherwise, continue to Step 2.

Step 2. Divide each term in F by the first literal of t. This is performed in the following manner:

- If the literal exists in a term in F, form a new term by removing the literal from the term. Replace the previous term in F by the new term.
- If the literal is opposed in a term in F, delete that term in F.
- If the literal does not exist in a term in F, keep the term and do nothing.

Step 3. Remove the first literal from t, and return to Step 1.

Procedure 2.5 (Splitting-Variable Heuristic): Given an n-variable SOP formula F:

Step 1. For each variable x_i which exists in F, determine:

1. n_0^i: the number of terms of F in which the literal x_i' appears; and
2. n_1^i: the number of terms of F in which the literal x_i appears.

Step 2. For each i, calculate

$$\alpha \cdot \min(n_0^i, n_1^i) + \beta \cdot (n_0^i + n_1^i). \tag{D.1}$$

Step 3.

- If $\max_i \min(n_0^i, n_1^i) > 0$, then at least one variable is binate. If so, then return the associated x_i for which (D.1) is maximal. x_i is the splitting variable. α and β are constants which are usually set equal to 1 and 2, respectively.
- If $\max_i \min(n_0^i, n_1^i) = 0$, then all variables in F are unate. Hence, apply the unate version of the operation on the given function(s).

Procedure 2.6 (Merge Operation): Given two SOP formulas H_0 and H_1 representing functions h_0 and h_1, the merge operation to construct formulas to represent functions $\widehat{h_0}$, $\widehat{h_1}$, and h_2 is performed as follows:

Step 0. Initialise accumulators H_{0-ACC} and H_2 to empty.

Step 1. If H_0 consists of no terms:

1. Return H_{0-ACC} as the formula representing $\widehat{h_0}$;
2. Return H_1 as the formula representing $\widehat{h_1}$; and
3. Return H_2 as the formula representing h_2.

Otherwise, continue to Step 2.

Step 2. Initialise an accumulator H_{1-ACC} to empty.

Step 3. If H_1 is empty, place all of the terms of H_{1-ACC} in H_1. Remove the first term in H_0 and place it in H_{0-ACC}. Return to Step 1.

Step 4. Compare the first term in H_0 to the first term in H_1.

- If the term in H_0 is both the subset and superset of the term in H_1, i.e., $s_i = t_j$, then delete the terms from both H_0 and H_1. Place the term in H_0 in H_2. Append the remaining terms of H_1 to H_{1-ACC} and call it H_1. Return to Step 1.
- If the term in H_0 is the superset of the term in H_1, i.e., $s_i \leq t_j$, then delete the term from H_0 and place the term in H_2. Append H_1 to H_{1-ACC} and call it H_1. Return to Step 1.
- If the term in H_0 is the subset of the term in H_1, i.e., $t_j \leq s_i$, then delete the term from H_1 and place the term in H_2. Return to Step 3.
- If the term in H_0 is neither the subset nor the superset of the term in H_1, then remove the first term from H_1 and place it in H_{1-ACC}. Return to Step 3.

Procedure 2.7 (Complementation): Given an SOP formula F which represents a Boolean function f, f' is computed as follows:

Step 0.

- If $F \equiv 0$, then return a value of 1.
- If $F \equiv 1$, then return a value of 0.
- Otherwise, continue to Step 1.

Step 1. Determine a good splitting variable x using Procedure 2.5.

- If a binate x does not exist, then apply the unate complementation algorithm given by Procedure 2.8.
- Otherwise, continue to Step 2.

Step 2. Apply Boole's Expansion Theorem using the splitting variable found in Step 1. Determine the complements of leaf functions f/x' and f/x.

Step 3. Using the formulas which represent $(f/x')'$ and $(f/x)'$, apply the merge operation given by Procedure 2.6. Three formulas are returned by the merge operation. One consists of the remaining terms of the formula representing $(f/x')'$, the second consists of the remaining terms of the formula representing $(f/x)'$, and the third consists of the terms taken from the original two formulas.

Step 4.

1. Prefix the remaining terms of the formula representing $(f/x')'$ with the literal x'.
2. Prefix the remaining terms of the formula representing $(f/x)'$ with the literal x.

Step 5. Append the two formulas created in Step 4 to the third formula returned by the merge operation (the terms taken from $(f/x')'$ and $(f/x)'$). The resulting formula represents the function f'.

Procedure 2.8 (Unate Complementation): Given an SOP formula F which represents a unate Boolean function f, f' is found as follows:

Step 0.

- If $F \equiv 0$, then return a value of 1.
- If $F \equiv 1$, then return a value of 0.
- If F consists of a single term, calculate the complement of f using DeMorgan's Law (2.29) and return it.
- Otherwise, continue to Step 1.

Step 1. Find the term with the fewest number of literals in the formula F. Of the literals in this term, determine the literal x which appears most frequently in F.

Step 2. Partition the terms of F into terms which include the literal x and those which do not include x.

Step 3. The terms of F which do not include x represent the function f/x'. Since f/x' is a unate function, determine the complement of f/x' using the unate complementation algorithm.

Step 4. Divide the terms of F which include x by the literal x. Append the result to the terms which do not include x. The resulting formula represents the function f/x. Since f/x is a unate function, determine the complement of f/x using the unate complementation algorithm.

Step 5. Prefix every term of the formula representing $(f/x')'$ found in Step 3 with the complement of literal x. Append the resulting formula to the formula representing $(f/x)'$ found in Step 4. The resulting formula represents the function f'.

Procedure 2.9 (Product): Given SOP formulas F and G which represent Boolean functions f and g, $f \cdot g$ is computed as follows:

Step 0.

- If $F \equiv 0$ or $G \equiv 0$ then return a value of 0.
- If $F \equiv 1$, then return $ABS(G)$.
- If $G \equiv 1$, then return $ABS(F)$.
- Otherwise, continue to Step 1.

Step 1. Given formulas F and G, determine a good splitting variable x using Procedure 2.5.

- If a binate x does not exist, then multiply f and g with the unate cross-product algorithm given by Procedure 2.3.
- Otherwise, continue to Step 2.

Step 2. Apply Boole's Expansion Theorem using the splitting variable found in Step 1. Recursively apply the product operation to form $(f/x' \cdot g/x')$ and $(f/x \cdot g/x)$.

Step 3. Using the formulas which represent $(f/x' \cdot g/x')$ and $(f/x \cdot g/x)$, apply the merge operation given by Procedure 2.6. Three formulas are returned by the merge operation. One consists of the remaining terms of the formula representing $(f/x' \cdot g/x')$, the second consists of the remaining terms of the formula representing $(f/x \cdot g/x)$, and the third consists of the terms taken from the original two formulas.

Step 4.

1. Prefix the remaining terms of the formula representing $(f/x' \cdot g/x')$ with the literal x'.
2. Prefix the remaining terms of the formula representing $(f/x \cdot g/x)$ with the literal x.

Step 5. Append the two formulas created in Step 4 to the third formula returned by the merge operation (the terms taken from $(f/x' \cdot g/x')$ and $(f/x \cdot g/x)$). The resulting formula represents the function $f \cdot g$.

Procedure 2.10 (Subtraction): Given SOP formulas F and G which represent Boolean functions f and g, $f - g$ is computed as follows:

Step 0.

- If $F \equiv 0$ or $G \equiv 1$, return a value of 0.
- If $G \equiv 0$, return $ABS(F)$.
- If $F \equiv 1$, return g'. Use Procedure 2.7 to determine g'.
- Otherwise, continue to Step 1.

Step 1. Given formulas F and G, determine a good splitting variable x using Procedure 2.5. If a binate x exists, continue to Step 2. Otherwise:

1. Form g' using the unate complementation algorithm of Procedure 2.8.
2. Multiply f by g' with the cross-product algorithm given by Procedure 2.2.
3. Form $ABS(F - G)$; it is the formula which represents $f - g$.

Step 2. Apply Boole's Expansion Theorem using the splitting variable found in Step 1. Recursively apply subtraction to form $(f/x' - g/x')$ and $(f/x - g/x)$.

Step 3. Using the formulas which represent $(f/x' - g/x')$ and $(f/x - g/x)$, apply the merge operation given by Procedure 2.6. Three formulas are returned by the merge operation. One consists of the remaining terms of the formula representing $(f/x' - g/x')$, the second consists of the remaining terms of the formula representing $(f/x - g/x)$, and the third consists of the terms taken from the original two formulas.

Step 4.

1. Prefix the remaining terms of the formula representing $(f/x' - g/x')$ with the literal x'.
2. Prefix the remaining terms of the formula representing $(f/x - g/x)$ with the literal x.

Step 5. Append the two formulas created in Step 4 to the third formula returned by the merge operation (the terms taken from $(f/x' - g/x')$ and $(f/x - g/x)$). The resulting formula represents the function $f - g$.

Procedure 2.11 (Exclusive-OR): Given SOP formulas F and G which represent Boolean functions f and g, $f \oplus g$ is found as follows:

Step 0.

- If $F \equiv 0$, return g.
- If $G \equiv 0$, return f.
- If $F \equiv 1$, then use Procedure 2.7 to determine g'. Return g'.
- If $G \equiv 1$, then use Procedure 2.7 to determine f'. Return f'.
- Otherwise, continue to Step 1.

Step 1. Given the formulas F and G, determine which formula has the fewest number of terms. Arbitrarily pick a variable which appears in the smallest formula. This is the splitting variable x.

Step 2. Apply Boole's Expansion Theorem using the splitting variable found in Step 1. Recursively apply XOR operations to form $(f/x' \oplus g/x')$ and $(f/x \oplus g/x)$.

Step 3.

1. Prefix the terms of the formula representing $(f/x' \oplus g/x')$ with the literal x'.
2. Prefix the terms of the formula representing $(f/x \oplus g/x)$ with the literal x.

Step 4. Append the two formulas created in Step 3. The resulting formula represents the function $f \oplus g$.

Procedure 2.12 (Exclusive-OR): Given Boolean formulas F and G which represent Boolean functions f and g, $f \oplus g$ is found as follows:

Step 0.

- If $F \equiv 0$, then return g.
- If $G \equiv 0$, then return f.
- If $F \equiv 1$, then use Procedure 2.7 to determine g'. Return g'.
- If $G \equiv 1$, then use Procedure 2.7 to determine f'. Return f'.
- Otherwise, continue to Step 1.

Step 1. Given formulas F and G, determine a good splitting variable x using Procedure 2.5. If a binate x exists, continue to Step 2. Otherwise:

1. Form f' and g' using the unate complementation algorithm of Procedure 2.8.
2. Using the cross-product algorithm given by Procedure 2.2, multiply f by g' and f' by g.
3. Make the formula which represents $f \cdot g'$ absorptive. Likewise, form the equivalent absorptive formula for the formula representing $f' \cdot g$. Form a new formula by appending the resulting absorptive formulas; it represents $f \oplus g$.

Step 2. Apply Boole's Expansion Theorem using the splitting variable found in Step 1. Recursively apply the XOR operation to form $(f/x' \oplus g/x')$ and $(f/x \oplus g/x)$.

Step 3. Using the formulas which represent $(f/x' \oplus g/x')$ and $(f/x \oplus g/x)$, apply the merge operation given by Procedure 2.6. Three formulas are returned by the merge operation. One consists of the remaining terms of the formula representing $(f/x' \oplus g/x')$, the second consists of the remaining terms of the formula representing $(f/x \oplus g/x)$, and the third consists of the terms taken from the original two formulas.

Step 4.

1. Prefix the remaining terms of the formula representing $(f/x' \oplus g/x')$ with the literal x'.
2. Prefix the remaining terms of the formula representing $(f/x \oplus g/x)$ with the literal x.

Step 5. Append the two formulas created in Step 4 to the third formula returned by the merge operation (the terms taken from $(f/x' \oplus g/x')$ and $(f/x \oplus g/x)$). The resulting formula represents the function $f \oplus g$.

Procedure 2.13 (Exclusive-NOR): Given SOP formulas F and G which represent Boolean functions f and g, $f \odot g$ is computed as follows:

Step 0.

- If $F \equiv 0$, then use Procedure 2.7 to determine g'. Return g'.
- If $G \equiv 0$, then use Procedure 2.7 to determine f'. Return f'.
- If $F \equiv 1$, then return g.
- If $G \equiv 1$, then return f.
- Otherwise, continue to Step 1.

Step 1. Given the formulas F and G, determine which formula has the fewest number of terms. Arbitrarily pick a variable which appears in the smallest formula. This is the splitting variable x.

Step 2. Apply Boole's Expansion Theorem using the splitting variable found in Step 1. Recursively apply $XNOR$ operations to form $(f/x' \odot g/x')$ and $(f/x \odot g/x)$.

Step 3.

1. Prefix the terms of the formula representing $(f/x' \odot g/x')$ with the literal x'.
2. Prefix the terms of the formula representing $(f/x \odot g/x)$ with the literal x.

Step 4. Append the two formulas created in Step 3. The resulting formula represents the function $f \odot g$.

Procedure 2.14 (Exclusive-NOR): Given Boolean formulas F and G which represent Boolean functions f and g, $f \odot g$ is found as follows:

Step 0.

- If $F \equiv 0$, then use Procedure 2.7 to determine g'. Return g'.
- If $G \equiv 0$, then use Procedure 2.7 to determine f'. Return f'.
- If $F \equiv 1$, then return g.
- If $G \equiv 1$, then return f.
- Otherwise, continue to Step 1.

Step 1. Given formulas F and G, determine a good splitting variable x using Procedure 2.5. If a binate x exists, continue to Step 2. Otherwise:

1. Form f' and g' using the unate complementation algorithm of Procedure 2.8.
2. Using the unate cross-product algorithm given by Procedure 2.3, multiply f' by g' and f by g.
3. Append the formulas together which represent $f' \cdot g'$ and $f \cdot g$. It represents $f \odot g$.

Step 2. Apply Boole's Expansion Theorem using the splitting variable found in Step 1. Recursively apply the $XNOR$ operation to form $(f/x' \odot g/x')$ and $(f/x \odot g/x)$.

Step 3. Using the formulas which represent $(f/x' \odot g/x')$ and $(f/x \odot g/x)$, apply the merge operation given by Procedure 2.6. Three formulas are returned by the merge operation. One consists of the remaining terms of the formula representing $(f/x' \odot g/x')$, the second consists of the remaining terms of the formula representing $(f/x \odot g/x)$, and the third consists of the terms taken from the original two formulas.

Step 4.

1. Prefix the remaining terms of the formula representing $(f/x' \odot g/x')$ with the literal x'.
2. Prefix the remaining terms of the formula representing $(f/x \odot g/x)$ with the literal x.

Step 5. Append the two formulas created in Step 4 to the third formula returned by the merge operation (the terms taken from $(f/x' \odot g/x')$ and $(f/x \odot g/x)$). The resulting formula represents the function $f \odot g$.

Procedure 2.15 (Simplification): Given an SOP formula F which represents a Boolean function f, an equivalent formula to represent f is formed as follows:

Step 0.

- If $F \equiv 0$, then return a value of 0.
- If $F \equiv 1$, then return a value of 1.
- Otherwise, continue to Step 1.

Step 1. Given formula F, determine a good splitting variable x using Procedure 2.5.

- If a binate x does not exist, form $ABS(F)$. It is the simplified formula for f.
- Otherwise, continue to Step 2.

Step 2. Apply Boole's Expansion Theorem using the splitting variable found in Step 1. Simplify the leaf functions f/x' and f/x.

Step 3. Using the formulas which represent f/x' and f/x, apply the merge operation given by Procedure 2.6. Three formulas are returned by the merge operation. One consists of the remaining terms of the formula representing f/x', the second consists of the remaining terms of the formula representing f/x, and the third consists of the terms taken from the original two formulas.

Step 4.

1. Prefix the remaining terms of the formula representing f/x' with the literal x'.
2. Prefix the remaining terms of the formula representing f/x with the literal x.

Step 5. Append the two formulas created in Step 4 to the third formula returned by the merge operation (the terms taken from f/x' and f/x). The result is a simplified formula for the function f.

Procedure 2.16 (Simplification): Given a Boolean formula F which represents a Boolean function f, an equivalent formula to represent f is formed as follows:

Step 0. Initialise an accumulator ACC by removing the first term from F and placing it in ACC.

Step 1.

- If F is empty, form $ABS(ACC)$. It is a simplified formula which represents f.
- If the first term in F is absorbed by any term in ACC, then remove the term from F and repeat Step 1.
- Otherwise, continue to Step 2.

Step 2. Remove the first term F and call it T. Initialise an accumulator $\widehat{ACC}$ by assigning it the contents of ACC. Initialize a second accumulator $ACC_{consens}$ to empty. Continue to Step 3.

Step 3.

- If $\widehat{ACC}$ is empty, no consensus terms (if any) formed by comparing T to the terms originally in $\widehat{ACC}$ absorb T. Therefore, add T to ACC. Append the contents of $ACC_{consens}$ to F and return to Step 2.

- Otherwise, continue to Step 4.

Step 4. Using the first term in $\widehat{ACC}$ and T, form a consensus term if one can be formed and the consensus term absorbs at least one of the parent terms. Depending on the result, take one of the following actions:

- If no consensus term was formed, remove the first term from $\widehat{ACC}$ and return to Step 3.
- If the consensus term is identically equal to 1, then the original function f is also identically equal to 1. Return a formula which represents a function which is identically equal to 1 as the result of the procedure.
- If the consensus term absorbs the term T, add the consensus term to $ACC_{consens}$. Append the contents of $ACC_{consens}$ to F and return to Step 2.
- Otherwise, a consensus term was formed and it absorbs only the first term in $\widehat{ACC}$. In this case add the consensus term to $ACC_{consens}$. Remove the first term from $\widehat{ACC}$ and return to Step 3.

Procedure 2.17 (Relative Simplification): Given a formula F which represents a Boolean function f and a formula G which represents a Boolean function g, formula G is simplified relative to F forming a new function $\hat{g}$ as follows:

Step 1.

- If F is empty, return the current formula G. It is the formula $\widehat{G}$ which represents the new function $\hat{g}$.
- Otherwise, remove the first term from F and call it T. Initialize an accumulator ACC to empty and continue to Step 2.

Step 2.

- If G is empty, assign it the contents of ACC. Return to Step 1.
- Otherwise, continue to Step 3.

Step 3. Using the first term in G and T, form a consensus term if one can be formed and the consensus term absorbs at least one of the parent terms. Depending on the result, take one of the following actions:

- If the consensus term is identically equal to 1, then the new function $\hat{g}$ also is identically equal to 1. Return a formula which represents a function which is identically equal to 1 as the result of the procedure.
- If no consensus term was formed, remove the first term from G and place it in ACC. Return to Step 2.
- If the consensus term absorbs the first term in G, add the consensus term to ACC. Remove the first term from G and return to Step 2.
- Otherwise, remove the first term from G and place it in ACC. Return to Step 2.

Procedure 2.18 (Blake canonical form - Successive Extraction): Given an SOP formula F which represents a Boolean function f, we generate the Blake canonical form for f as follows:

Step 0. By examining the formula F, form the set OPP of opposed variables for the function f.

Step 1.

- If OPP is empty, then f is unate. Return $ABS(F)$ as the Blake canonical form of f.
- Otherwise, continue to Step 2.

Step 2.

- If OPP is empty, return F. It is the Blake canonical form of f.
- Otherwise, continue to Step 3.

Step 3. Remove the first variable from the set OPP of opposed variables. Call this variable x.

1. Form a formula G which consists of all of the terms of F in which x is positive. Divide G by x using Procedure 2.4.
2. Form a formula H which consists of all of the terms of F in which x is negative. Divide H by x' using Procedure 2.4.
3. Using the cross-product operation (Procedure 2.2), multiply G/x by H/x'. The resulting terms are all of the consensus terms that can be formed in which the variable x is opposed. Call the resulting formula $\tilde{F}$.

Step 4. Add $\tilde{F}$ to F. Form $ABS(F+\tilde{F})$ and replace the contents of F with $ABS(F+\tilde{F})$. Return to Step 2.

Procedure 2.19 (Modified Splitting-Variable Heuristic): Given an n-variable SOP formula F:

Step 1. For each variable x_i which exists in F, determine:

1. n_0^i: the number of terms of F in which the literal x_i' appears; and
2. n_1^i: the number of terms of F in which the literal x_i appears.

Step 2. For each i, calculate

$$n_0^i * n_1^i. \tag{D.2}$$

Step 3.

- If $\max_i(n_0^i * n_1^i) > 0$, then at least one variable is binate. If so, then return the associated x_i for which (D.2) is maximal. x_i is the modified splitting variable.
- If $\max_i(n_0^i * n_1^i) = 0$, then all variables in F are unate. Hence, apply the unate version of the operation on the given function(s).

Procedure 2.20 (Blake canonical form - Recursive Multiplication): Given an SOP formula F which represents a Boolean function f, we generate the equivalent formula which is the Blake canonical form for f as follows:

Step 0.

- If $F \equiv 0$, then return a formula which represents 0.
- If $F \equiv 1$, then return a formula which represents 1.
- Otherwise, continue to Step 1.

Step 1. Determine a modified splitting-variable x using Procedure 2.19.

- If a modified splitting-variable x does not exist, then f is a unate function represented by a formula which only consists of unate variables. Form $ABS(F)$ and return it; it is the Blake canonical form of f.
- Otherwise, continue to Step 2.

Step 2. Form $BCF(f/x)$ and $BCF(f/x')$.

Step 3.

1. Form $BCF(f/x) \times BCF(f/x')$ using the cross-product operation given by Procedure 2.2. Call the result H.
2. Form $ABS(H)$.

Step 4. Form $ABSREL(BCF(f/x), ABS(H))$ and $ABSREL(BCF(f/x'), ABS(H))$.

Step 5.

1. Prefix the terms of the formula representing $ABSREL(BCF(f/x), ABS(H))$ with the literal x.
2. Prefix the terms of the formula representing $ABSREL(BCF(f/x'), ABS(H))$ with the literal x'.

Step 6. Append the two formulas developed in Step 5 with $ABS(H)$. The resulting formula is the Blake canonical form of f.

Procedure 2.21 (Least-Binate Argument): Given a Boolean function f represented by an SOP formula F and a set T of variables which appear in F, the least binate variable x relative to the variables in T is calculated as follows:

Step 1. For each variable $x \in T$ determine:

1. n_0: the number of terms of F in which x' appears; and
2. n_1: the number of terms of F in which x appears.

Step 2. For each x, calculate

$$\gamma * (n_0 * n_1) - \max(n_0, n_1) \tag{D.3}$$

where γ is a large constant (≥ 10).

Step 3. Return the associated x for which (D.3) is the smallest value. It is the least binate variable in T relative to the other variables in T.

Procedure 2.22 (Conjunctive Eliminant - ECON): Given an SOP formula F which represents the Boolean function f and a set T of literals, the conjunctive eliminant of f with respect to T, $ECON(f,T)$, is constructed as follows:

Step 1.

- If T is empty, then return F. It is $ECON(f,T)$.
- Of the variables in set T, determine the least binate variable using Procedure 2.21. Call this variable x.

Step 2.

- If $F \equiv 0$, then return a formula F which represents 0. It is $ECON(f,T)$.
- If $F \equiv 1$, then return a formula F which represents 1. It is $ECON(f,T)$.
- Otherwise, continue to Step 3.

Step 3. Partition the terms of F into the following sets:

- P, the terms of F which include the literal x', with the literal divided out;
- Q, the terms of F which include the literal x, with the literal divided out; and
- R, the terms of F which include neither x nor x'.

Using Procedure 2.9, multiply p by q. Append the result to r. The resulting formula represents $ECON(f,\{x\})$.

Step 4. Using Procedure 2.15, simplify the formula generated in Step 3. Replace the contents of F with the simplified formula and return to Step 1.

Procedure 2.23 (Test for Tautology): Given a function f which is represented by the SOP formula F, we test whether f is a tautology in the following manner:

Step 1.

- If $F \equiv 0$, then return a value of FALSE. f is not a tautology.
- If $F \equiv 1$, then return a value of TRUE. f is a tautology.
- Otherwise, continue to Step 2.

Step 2. Determine if any variables in F are unate.

- If so, delete all terms in F which include as a literal any variable which is unate. The revised formula $\hat{F}$ represents a new function $\hat{f}$; f is a tautology if and only if $\hat{f}$ is a tautology. Return to Step 1 to determine if $\hat{f}$ is a tautology.
- Otherwise, continue to Step 3.

Step 3. Determine a good splitting variable x using Procedure 2.5. Form f/x and f/x'.

Step 4. Determine if f/x and f/x' are tautologies. f is a tautology if and only if both f/x and f/x' are tautologies.

- If both f/x and f/x' are tautologies, then return a value of TRUE.
- If at least one of f/x and f/x' is not a tautology, then return a value of FALSE.

Procedure 2.24 (Test for Tautology): Given a function f which is represented by the SOP formula F, we test whether f is a tautology in the following manner:

Step 1.

- If $F \equiv 1$, then return a value of TRUE. f is a tautology.
- Otherwise, continue to Step 2.

Step 2. Determine if F includes a term consisting of a single literal x.

- If so, then determine whether f/x' is a tautology. f is a tautology if and only if f/x' is a tautology.
- Otherwise, continue to Step 3.

Step 3. Determine whether F has a binate variable. If F has any binate variables, arbitrarily choose one and call it x.

- If a binate variable x exists, determine whether f/x' and f/x are tautologies. f is a tautology if and only if both f/x and f/x' are tautologies. If both f/x and f/x' are tautologies, then return a value of TRUE; otherwise, return a value of FALSE.
- Otherwise, continue to Step 4.

Step 4. If we reach this step, then f is not identically equal to 1. Nor does F include any opposed variables; hence, f is a unate function. Thus, f is not a tautology. Return a value of FALSE.

Procedure 2.25 (Test for Inclusion): Given two Boolean functions g and h and SOP formulas G and H which represent them, we determine if $g \leq h$ in the following manner:

Step 1.

- If G consists of no terms, then g is included in h. Return TRUE.
- Otherwise, continue to Step 2.

Step 2. Remove the first term from G and call it t. Apply (2.171) to determine if t is included in h.

- If $t \not\leq h$, then $g \neq h$. Return FALSE.
- Otherwise, return to Step 1.

Procedure 2.26 (Test for Inclusion): Given two Boolean functions g and h and SOP formulas G and H which represent them, we determine if $g \leq h$ in the following manner:

Step 1. Using the subtraction operation specified by Procedure 2.10, subtract function h from function g.

Step 2. Test the result $g - h$ obtained in Step 1 for equality to 0.

- If $g - h = 0$, then $g \leq h$. Return TRUE.
- If $g - h \neq 0$, then $g \not\leq h$. Return FALSE.

Procedure 2.27 (Test for Equivalence): Given two Boolean functions g and h and SOP formulas G and H which represent them, we determine if $g = h$ in the following manner:

Step 1. Using the test for inclusion given by Procedure 2.25, determine if $g \leq h$.

- If $g \not\leq h$, then $g \neq h$. Return FALSE.
- If $g \leq h$, then continue to Step 2.

Step 2. Use Procedure 2.25 to determine if $h \leq g$.

- If $h \not\leq g$, then $g \neq h$. Return FALSE.
- Otherwise, $h \leq g$. It follows that $g = h$. Return TRUE.

Procedure 2.28 (Test for Equivalence): Given two Boolean functions g and h and SOP formulas G and H which represent them, we determine if $g = h$ in the following manner:

Step 1. Using the Exclusive-OR operation specified by Procedure 2.11, form $g \oplus h$.

Step 2. Test the result $g \oplus h$ obtained in Step 1 for equivalence to 0.

- If $g \oplus h = 0$, then $g = h$. Return TRUE.
- If $g \oplus h \neq 0$, then $g \neq h$. Return FALSE.

Procedure 2.29 (Test for Membership in Interval): Given a Boolean function f and two Boolean functions g and h which specify an interval $[g, h]$, we determine if $g \leq f \leq h$ in the following manner:

Step 1. Using Procedure 2.25, determine if $g \leq f$.

- If $g \not\leq f$, then f is not a member of interval $[g, h]$. Return FALSE.
- If $g \leq f$, then continue to Step 2.

Step 2. Use Procedure 2.25 to determine if $f \leq h$.

- If $f \not\leq h$, then f is not a member of interval $[g, h]$. Return FALSE.
- Otherwise, $g \leq f \leq h$. Return TRUE.

Procedure 2.30 (Test for Membership in Interval): Given a Boolean function f and two Boolean functions g and h which specify an interval $[g, h]$, we determine if $g \leq f \leq h$ in the following manner:

Step 1. Using Procedure 2.26, determine if $g - f = 0$.

- If $g - f \neq 0$, then f is not a member of interval $[g, h]$. Return FALSE.
- If $g - f = 0$, then continue to Step 2.

Step 2. Use Procedure 2.26 to determine if $f - h = 0$.

- If $f - h \neq 0$, then f is not a member of interval $[g, h]$. Return FALSE.
- Otherwise, $g - f = 0$ and $f - h - 0$. Therefore, $g \leq f \leq h$. Return TRUE.

Procedure 2.31 (Irredundant Formula - Completely-Specified Function): Given a Boolean function f and its corresponding Blake canonical form $BCF(f)$, a sub-minimal formula representing f is obtained in the following manner:

Step 0. Sort the terms of $BCF(f)$ such that the terms with the greatest number of literals appear before terms of fewer literals. Call the resulting formula F. Initialize an accumulator ACC to empty.

Step 1.

- If F is empty, then ACC is a sub-minimal formula which represents f. Return ACC.
- Otherwise, continue to Step 2.

Step 2. Remove the first term from F and call it T.

Step 3. Apply (2.171) to determine if T is included the formula formed by appending the remaining terms of F to ACC, i.e., $T \leq F + ACC$.

- If $T \leq F + ACC$, then T is a redundant term. Return to Step 1.
- Otherwise, T is a required term. Place T in ACC and return to Step 1.

Procedure 2.32 (Irredundant Formula - Completely-Specified Function - Essentials Identified): Given a Boolean function f, its corresponding Blake canonical form $BCF(f)$, and the function's essential prime implicants, a sub-minimal formula is obtained to represent f in the following manner:

Step 0. Initialise an accumulator ACC to the set of essential prime implicants of f. Sort the remaining primes of $BCF(f)$ such that the terms with the greatest number of literals appear before terms of fewer literals. Call the resulting formula F.

Step 1.

- If F is empty, return ACC. ACC is a sub-minimal formula which represents f.
- Otherwise, continue to Step 2.

Step 2. Remove the first term from F and call it T.

Step 3. Apply (2.171) to determine if T is included the formula formed by appending the remaining terms of F to ACC, i.e., $T \leq F + ACC$.

- If $T \leq F + ACC$, then T is a redundant term. Return to Step 1.
- Otherwise, T is a required term. Place T in ACC and return to Step 1.

Procedure 2.33 (Irredundant Formula - Interval): Given an interval $[g, h]$ consisting of Boolean functions g and h and the corresponding Blake canonical form $BCF(h)$, an irredundant formula representing an f in $[g, h]$ is derived as follows:

Step 0. Sort the terms of $BCF(h)$ such that the terms with the greatest number of literals appear before terms of fewer literals. Call the resulting formula H. Initialize an accumulator ACC to empty.

Step 1.

- If H is empty, return ACC. ACC is a sub-minimal formula which represents f.
- Otherwise, continue to Step 2.

Step 2. Remove the first term from H and call it T.

Step 3. Apply Procedure 2.25 to determine if g is included in the function represented by the formula formed by appending the remaining terms of H to ACC, i.e., $g \leq H + ACC$.

- If $g \leq H + ACC$, then T is a redundant term. Return to Step 1.
- Otherwise, T is a required term. Place T in ACC and return to Step 1.

Procedure 2.34 (Irredundant Formula - Interval - Essentials Identified): Given an interval $[g, h]$ consisting of Boolean functions g and h, the corresponding Blake canonical form $BCF(h)$, and the set of essential prime implicants of h, an irredundant formula representing an f in $[g, h]$ is derived as follows:

Step 0. Initialise an accumulator ACC to the set of essential prime implicants of h. Sort the remaining primes of $BCF(h)$ such that the terms with the greatest number of literals appear before terms of fewer literals. Call the resulting formula H.

Step 1.

- If H is empty, return ACC. ACC is a sub-minimal formula which represents f.
- Otherwise, continue to Step 2.

Step 2. Remove the first term from H and call it T.

Step 3. Apply Procedure 2.25 to determine if g is included in the function represented by the formula formed by appending the remaining terms of H to ACC, i.e., $g \leq H + ACC$.

- If $g \leq H + ACC$, then T is a redundant term. Return to Step 1.
- Otherwise, T is a required term. Place T in ACC and return to Step 1.

Chapter 8 - An Introduction to Search

Procedures listed in this section are similar to those in *Artificial Intelligence* (Winst 84). These procedures should be viewed as general outlines of each approach, since two considerations are ignored which are usually handled in search processes. First, as nodes are generated, they are not compared to existing nodes in the OPEN list to determine duplicates (a queue Q, stack S, or list L corresponds to the OPEN list of nodes.). Hence, duplicate nodes may exist on the OPEN list. Second, a CLOSED list is not maintained. All newly-generated nodes are treated as never having been expanded.

Procedure 8.1 (Breadth-First Search): Given a root node, breadth-first search is performed in the following manner:

Step 1. Form a one-element queue Q consisting of the root node.

Step 2.

- If Q is empty, then no solution exists. Announce the failure of the procedure.
- Otherwise, continue to Step 3.

Step 3. Remove the first node n from Q and determine if it is the goal node.

- If n is the goal node, a solution has been found. Return the solution.
- Otherwise, continue to Step 4.

Step 4.

1. Expand n by generating all of n's children, if any.
2. Add n's children to the back of Q.

Return to Step 2.

Procedure 8.2 (Depth-First Search): Given a root node, depth-first search is performed in the following manner:

Step 1. Form a one-element stack S consisting of the root node.

Step 2.

- If S is empty, then no solution exists. Announce the failure of the procedure.
- Otherwise, continue to Step 3.

Step 3. Remove the first node n from S and determine if it is the goal node.

- If n is the goal node, a solution has been found. Return the solution.
- Otherwise, continue to Step 4.

Step 4.

1. Expand n by generating all of n's children nodes, if any.
2. Add n's children to the top of S.

Return to Step 2.

Procedure 8.3 (Branch-and-Bound Search): Given a root node, branch-and-bound search is performed in the following manner:

Step 1. Form a one-element list L consisting of the root node.

Step 2.

- If L is empty, then no solution exists. Announce the failure of the procedure.
- Otherwise, continue to Step 3.

Step 3. Remove the first node n from L and determine if it is the goal node.

- If n is the goal node, a solution has been found. Return the solution.
- Otherwise, continue to Step 4.

Step 4.

1. Expand n by generating all of n's children nodes, if any.
2. For each of the children, determine the cost $g(n)$.
3. Insert each of the children into L in a manner that maintains a node ordering in L such that nodes appear in L in ascending order of their associated cost $g(n)$.

Return to Step 2.

Procedure 8.4 (Greedy Method): Given a root node, the greedy method is performed in the following manner:

Step 1. Denote the root node as the current node n.

Step 2. Expand the current node n by generating all of n's children.

- If n has no children, then no solution was found. Announce the failure of the procedure.
- Otherwise, continue to Step 3.

Step 3.

- For each of the children, determine if any is a goal node. If so, then return the node associated with the least-cost solution.
- Otherwise, determine the cost $g(n)$ for each of the children. Then, discard all but the cheapest cost node; denote it as the new current node n.

Return to Step 2.

Procedure 8.5 (Hill-Climbing): Given a root node, hill-climbing is performed in the following manner:

Step 1. Denote the root node as the current node n.

Step 2. Expand the current node n by generating all of n's children.

- If n has no children, then no solution was found. Announce the failure of the procedure.
- Otherwise, continue to Step 3.

Step 3.

- For each of the children, determine if any is a goal node. If so, then return the node associated with the least-cost solution.
- Otherwise, determine the estimated distance $h(n)$ between each child and a goal node for each of the children. Then, discard all but the node with the smallest $h(n)$; denote it as the new current node n.

Return to Step 2.

Procedure 8.6 (Best-First Search): Given a root node, best-first search is performed in the following manner:

Step 1. Form a one-element list L consisting of the root node.

Step 2.

- If L is empty, then no solution exists. Announce the failure of the procedure.
- Otherwise, continue to Step 3.

Step 3. Remove the first node n from L and determine if it is the goal node.

- If n is the goal node, a solution has been found. Return the solution.
- Otherwise, continue to Step 4.

Step 4.

1. Expand n by generating all of n's children nodes, if any.
2. For each of the children, determine the estimated distance $h(n)$ between the child and a goal node.
3. Insert each of the children into L in a manner that maintains a node ordering in L such that nodes appear in L in ascending order of their associated value $h(n)$.

Return to Step 2.

Procedure 8.7 (Beam Search): Given a root node and a width w, beam search is performed in the following manner:

Step 1. Form a one-element list L consisting of the root node.

Step 2.

- If L is empty, then no solution exists. Announce the failure of the procedure.
- Otherwise, continue to Step 3.

Step 3. Remove all nodes from L and expand each node by generating the children of each node.

- If any of the children are goal nodes, then return the node associated with the least-cost solution.
- Otherwise, continue to Step 4.

Step 4.

1. Evaluate each of the children using an evaluation function $f(n)$.
2. Sort the children in ascending order of the value $f(n)$.
3. Discard all but the w best children.
4. Place the remaining w nodes on L.

Return to Step 2.

Procedure 8.8 (A* Search): Given a root node, A* search is performed in the following manner:

Step 1. Form a one-element list L consisting of the root node.

Step 2.

- If L is empty, then no solution exists. Announce the failure of the procedure.
- Otherwise, continue to Step 3.

Step 3. Remove the first node n from L and determine if it is the goal node.

- If n is the goal node, a solution has been found. Return the solution.
- Otherwise, continue to Step 4.

Step 4.

1. Expand n by generating all of n's children nodes, if any.
2. For each of the children, determine the cost $g(n)$ of the path between the root node and the child.
3. For each of the children, determine the estimated distance $h(n)$ between the child and a goal node.
4. For each of the children, form $f(n) = g(n) + h(n)$.
5. Insert each of the children into L in a manner that maintains a node ordering in L such that nodes appear in L in ascending order of their associated value $f(n)$.

Return to Step 2.

Procedure 8.9 (Dynamic Weighting): Given a root node and constants ϵ and N, the dynamic-weighting method is implemented in the following manner:

Step 1. Form a one-element list L consisting of the root node.

Step 2.

- If L is empty, then no solution exists. Announce the failure of the procedure.
- Otherwise, continue to Step 3.

Step 3. Remove the first node n from L and determine if it is the goal node.

- If n is the goal node, a solution has been found. Return the solution.
- Otherwise, continue to Step 4.

Step 4.

1. Expand n by generating all of n's children nodes, if any.
2. For each of the children, determine the cost $g(n)$ of the path between the root node and the child.
3. For each of the children, determine the estimated distance $h(n)$ between the child and a goal node.
4. For each of the children, determine their depth $d(n)$ in the search tree.
5. For each of the children, form $f(n) = g(n) + h(n) + \epsilon[1 - d(n)/N]h(n)$.
6. Insert each of the children into L in a manner that maintains a node ordering in L such that nodes appear in L in ascending order of their associated value $f(n)$.

Return to Step 2.

Procedure 8.10 (Static Weighting): Given a root node and a constant w, the static-weighting method is implemented in the following manner:

Step 1. Form a one-element list L consisting of the root node.

Step 2.

- If L is empty, then no solution exists. Announce the failure of the procedure.
- Otherwise, continue to Step 3.

Step 3. Remove the first node n from L and determine if it is the goal node.

- If n is the goal node, a solution has been found. Return the solution.
- Otherwise, continue to Step 4.

Step 4.

1. Expand n by generating all of n's children nodes, if any.
2. For each of the children, determine the cost $g(n)$ of the path between the root node and the child.
3. For each of the children, determine the estimated distance $h(n)$ between the child and a goal node.
4. For each of the children, form $f(n) = g(n) + w \cdot h(n)$.
5. Insert each of the children into L in a manner that maintains a node ordering in L such that nodes appear in L in ascending order of their associated value $f(n)$.

Return to Step 2.

Bibliography

[Areva 78] Arevalo, Zosimo and Jon G. Bredeson. "A Method to Simplify a Boolean Function into a Near Minimal Sum-of-Products for Programmable Logic Arrays," *IEEE Transactions on Computers, C-27:* 1028-1039 (November 1978).

[Barr 81] Barr, Avron and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence, Volume I.* Reading, Massachusetts: Addison-Wesley, 1981.

[Barte 61] Bartee, Thomas C. "Computer Design of Multiple-Output Logical Networks," *IRE Transactions on Electronic Computers, EC-10:* 21-30 (March 1961).

[Biswa 86] Biswas, Nripendra N. "Computer-Aided Minimisation Procedure for Boolean Functions," *IEEE Transactions on Computer-Aided Design, CAD-5:* 303-304 (April 1986).

[Biswa 84] Biswas, Nripendra N. "Computer-Aided Minimization Procedure for Boolean Functions," *Proceedings of the Twenty-First Design Automation Conference,* 699-702. Washington, D.C.: IEEE Computer Society Press, 1984.

[Blake 37] Blake, Archie. *Canonical Expressions in Boolean Algebra,* PhD Dissertation. Department of Mathematics, University of Chicago, Chicago, Illinois, 1937.

[Boole 54] Boole, George. *An Investigation of the Laws of Thought.* Originally published in 1854 by Macmillan, London. Reprinted by Dover Publications in 1958.

[Brayt 84] Brayton, Robert K., Gary D. Hachtel, Curtis T. McMullen, and Alberto L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Boston: Kluwer Academic Publishers, 1984.

[Brayt 82] Brayton, R.K., J.D. Cohen, G.D. Hachtel, B.M. Trager, and D.Y.Y. Yun. "Fast Recursive Boolean Function Manipulation," *Proceedings of the IEEE International Symposium on Circuits and Systems,* 58-62. Washington, D.C.: IEEE Computer Society Press, 1982.

[Brown 81] Brown, Douglas W. "A State-Machine Synthesizer-SMS," *Proceedings of the Eighteenth Design Automation Conference,* 301-305. Washington, D.C.: IEEE Computer Society Press, 1981.

[Brown 90] Brown, Frank M. *Boolean Reasoning: The Logic of Boolean Equations.* Boston: Kluwer Academic Publishers, 1990.

[Chang 65] Chang, D.M.Y. and T.H. Mott, Jr. "Computing Irredundant Normal Forms from Abbreviated Presence Functions," *IEEE Transactions on Electronic Computers, EC-14:* 335-342 (June 1965).

[Cutle 87] Culter, Robert Brian and Saburo Muroga. "Derivation of Minimal Sums for Completely-Specified Functions," *IEEE Transactions on Computers, C-36:* 277-292 (March 1987).

[Cutle 80] Cutler, Robert Brian. *Algebraic Derivation of Minimal Sums for Functions of a Large Number of Variables,* PhD Dissertation. Department of Computer Science, University of Illinois, Urbana-Champaign, Illinois, 1980.

[Dagen 86] Dagenais, Michel R., Vinod K. Agarwal, and Nicholas C. Rumin. "McBOOLE: A New Procedure for Exact Logic Minimisation," *IEEE Transactions on Computer-Aided Design, CAD-5:* 229-238 (January 1986).

[Darri 81] Darringer, John A., William H. Joyner, Jr., C. Leonard Berman, and Louise Trevillyan. "Logic Synthesis Through Local Transformations," *IBM Journal of Research and Development, 25:* 272-280 (July 1981).

[deGeu 89] de Geus, Aart J. "Logic Synthesis Speeds ASIC Design," *IEEE Spectrum, 26:* 27-31 (August 1989).

[deGeu 85] de Geus, Aart J. and William Cohen. "A Rule-Based System for Optimizing Combinational Logic," *IEEE Design and Test of Computers, 2:* 22-32 (August 1985).

[Enomo 85] Enomoto, Kiyoshi, Shunichiro Nakamura, Takuji Ogihara, and Shinichi Murai. "LORES-2: A Logic Reorganization System," *IEEE Design and Test of Computers, 2:* 35-41 (October 1985).

[Fiduc 82] Fiduccia, C.M. and R.M. Mattheyses. "A Linear-Time Heuristic for Improving Network Partitions," *Proceedings of the Nineteenth Design Automation Conference,* 175-181. Washington, D.C.: IEEE Computer Society Press, 1982.

[Gaine 64] Gaines, R.S. "Implication Techniques for Boolean Functions," *Proceedings of the Fifth Annual Symposium on Switching Theory and Logical Design,* Princeton, N.J., 174-182. 1964.

[Garey 79] Garey, Michael R. and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York: W.H. Freeman, 1979.

[Ghasa 57] Ghasala, M.J. "Irredundant Disjunctive and Conjunctive Forms of a Boolean Function," *IBM Journal of Research and Development, 1:* 171-176 (April 1957).

[Givon 70] Givone, Donald D. *Introduction to Switching Circuit Theory.* New York: McGraw-Hill, 1970.

[Gurun 89] Gurunath, B. and Nripendra N. Biswas. "An Algorithm for Multiple Output Minimisation," *IEEE Transactions on Computer-Aided Design, CAD-8:* 1007-1013 (September 1989).

[Gurun 87] Gurunath, B. and Nripendra N. Biswas. "An Algorithm for Multiple Output Minimisation," *Proceedings of the IEEE International Conference on Computer-Aided Design,* 74-77. Washington, D.C.: IEEE Computer Society Press, 1987.

[Hardi 89] Harding, Bill. "Logic Synthesis Forces Rethinking of Design Methods," *Computer Design, 28:* 51-57 (1 December 1989).

[Hong 74] Hong, S.J., R.G. Cain, and D.L. Ostapko. "MINI: A Heuristic Approach to Logic Minimization," *IBM Journal of Research and Development, 18:* 443-458 (September 1974).

[Hong 91] Hong, Sung Je and Saburo Muroga. "Absolute Minimisation of Completely Specified Switching Functions," *IEEE Transactions on Computers, C-40:* 53-65 (January 1991).

[Hong 83] Hong, Sung Je. *Design of Minimal Programmable Logic Arrays,* PhD Dissertation. Department of Computer Science, University of Illinois, Urbana-Champaign, Illinois, 1983.

[Hunti 04] Huntington, E.V. "Sets of Independent Postulates for the Algebra of Logic." *Transactions of the American Mathematical Society, 5:* 288-309 (1904).

[Johns 87] Johnson, E.L and M.A.Karim. *Digital Design: A Pragmatic Approach.* Boston: PWS Engineering, 1987.

[Kerni 70] Kernighan, B.W. and S. Lin "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell System Technical Journal, 49:* 291-307 (January 1970).

[Klir 72] Klir, George J. *Introduction to the Methodology of Switching Circuits.* New York: D.Van Nostrand, 1972.

[Knuts 90] Knutson, Eric J. *Recursive Optimization of Digital Circuits,* MS Thesis AFIT/GCE/ENG/90D-03. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1990.

[Lau 89] Lau, H.T. *Algorithms on Graphs.* Blue Ridge Summit, PA: Tab Books, 1989.

[Ledle 60] Ledley, Robert Steven. *Digital Computer and Control Engineering.* New York: McGraw-Hill, 1960.

[Lipsc 76] Lipschutz, Seymour. *Discrete Mathematics.* New York: McGraw-Hill, 1976.

[Malik 88] Malik, Abdul A., Robert K. Brayton, A.Richard Newton, and Alberto L. Sangiovanni-Vincentelli. "A Modified Approach to Two-Level Logic Minimization," *Proceedings of the IEEE International Conference on Computer-Aided Design,* 106-109. Washington, D.C.: IEEE Computer Society Press, 1988.

[Mano 79] Mano, M. Morris. *Digital Logic and Computer Design.* Englewood Cliffs: Prentice-Hall, 1979.

[McClu 56] McCluskey, E.J., Jr. "Minimization of Boolean Functions," *Bell System Technical Journal, 35:* 1417-1444 (November 1956).

[Mitch 83] Mitchell, O.H. "On a New Algebra of Logic," *Studies in Logic,* edited by C.S.Pierce. Boston: Little, Brown, 1883.

[Mott 60] Mott, T.H. Jr., "Determination of the Irredundant Normal Forms of a Truth Function by Iterated Consensus of the Prime Implicants," *IEEE Transactions on Electronic Computers, EC-9:* 245-252 (June 1960).

[Murog 79] Muroga, Saburo. *Logic Design and Switching Theory.* New York: John Wiley, 1979.

[Nagle 75] Nagle, H.Troy Jr., B.D. Carroll, and J.David Irwin. *An Introduction to Computer Logic,* Englewood Cliffs: Prentice-Hall, 1975.

[Newto 86] Newton, A.R. "Techniques for Logic Synthesis," *VLSI '85: VLSI Design of Digital Systems,* edited by E. Horbst. New York: Elsevier Science Publishers, 1986.

[Nguye 87] Nguyen, Loc Bao, Marek A. Perkowski, and Nahum B. Goldstein. "PALMINI-Fast Boolean Minimizer for Personal Computers," *Proceedings of the Twenty-Fourth Design Automation Conference,* 208-214. Washington, D.C.: IEEE Computer Society Press, 1987.

[Paton 71] Paton, K. "An Algorithm for the Blocks and Cutnodes of a Graph," *Communications of the ACM, 14:* 468-475 (1971).

[Paton 69] Paton, K. "An Algorithm for Finding a Fundamental Set of Cycles in a Graph," *Communications of the ACM, 12:* 514-518 (1969).

[Pearl 84] Pearl, Judea. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Reading, Massachusetts: Addison-Wesley, 1984.

[Perki 88] Perkins, Sharon R. and Tom Rhyne. "An Algorithm for Identifying and Selecting the Prime Implicants of a Multiple-Output Boolean Function," *IEEE Transactions on Computer-Aided Design, CAD-7:* 1215-1219 (November 1988).

[Petri 59] Petrick, S.R. "On the Minimisation of Boolean Functions," *Proceedings of the International Conference on Information Processing,* 422-423. Munich, Germany: R. Oldenbourg, 1960.

[Petri 56] Petrick, S.R. *A Direct Determination of the Irredundant Forms of a Boolean Function From a Set of Prime Implicants.* AFCRC TR 56-110. Cambridge, Massachusetts: Air Force Cambridge Research Center, April, 1956.

[Prath 67] Prather, Ronald E. *Introduction to Switching Theory: A Mathematical Approach.* Boston: Allyn and Bacon, 1967.

[Quine 59] Quine, W.V. "On Cores and Prime Implicants of Truth Functions," *American Mathematical Monthly, 66:* 755-760 (October 1959).

[Quine 55] Quine, W.V. "A Way to Simplify Truth Functions," *American Mathematical Monthly, 62:* 627-631 (November 1955).

[Quine 52] Quine, W.V. "The Problem of Simplifying Truth Functions," *American Mathematical Monthly, 59:* 521-531 (October 1952).

[Reusc 75] Reusch, Bernd. "Generation of Prime Implicants from Subfunctions and a Unifying Approach to the Covering Problem," *IEEE Transactions on Computers, C-24:* 924-930 (September 1975).

[Rhyne 77] Rhyne, V. Thomas, Philip S.Noe, Melvin H. McKinney, and Udo W. Pooch. "A New Technique for the Fast Minimisation of Switching Functions," *IEEE Transactions on Computers, C-26:* 757-764 (August 1977).

[Rich 91] Rich, Elaine and Kevin Knight. *Artificial Intelligence.* (Second Edition.) New York: McGraw-Hill, 1991.

[Rich 83] Rich, Elaine. *Artificial Intelligence.* New York: McGraw-Hill, 1983.

[Rudea 74] Rudeanu, Sergiu. *Boolean Functions and Equations.* Amsterdam: North Holland, 1974.

[Rudel 89] Rudell, Richard L. *Logic Synthesis for VLSI Design,* PhD Dissertation. Department of Electrical Engineering and Computer Science, University of California, Berkeley, California, 1989.

[Rudel 86] Rudell, Richard and Alberto Sangiovanni-Vincentelli. "Exact Minimisation of Multiple-Value Functions for PLA Optimisation," *Proceedings of the IEEE International Conference on Computer-Aided Design,* 352-355. Washington, D.C.: IEEE Computer Society Press, 1986.

[Samso 54] Samson, E.W. and B.E. Mills. *Circuit Minimization: Algebra and Algorithms for New Boolean Canonical Expressions.* AFCRC TR 54-21. Cambridge, Massachusetts: Air Force Cambridge Research Center, April, 1954.

[Schrõ 90] Schröder, E. *Vorlesungen über die Algebra der Logik.* Leipzig: Volume I, 1890. Reprinted by Chelsea Publications, Bronx, N.Y., 1966.

[Shann 49] Shannon, C.E. "The Synthesis of Two-Terminal Switching Circuits," *Bell System Technical Journal, 28:* 59-98 (January 1949).

[Svobo 79] Svoboda, Antonin and Donnamaie White. *Advanced Logical Circuit Design Techniques.* New York: Garland STPM Press, 1979.

[Tison 67] Tison, Pierre. "Generalisation of Consensus Theory and Application to the Minimisation of Boolean Functions," *IEEE Transactions on Electronic Computers, EC-16:* 446-456 (August 1967).

[Ullma 84] Ullman, Jeffrey D. *Computational Aspects of VLSI.* Rockville, MD: Computer Science Press, 1984.

[Wilso 79] Wilson, Robin J. *Introduction to Graph Theory.* (Second Edition.) London: Longman Group, 1979.

[Winst 84] Winston, Patrick Henry. *Artificial Intelligence.* Reading, Massachusetts: Addison-Wesley, 1984.

[Zakre 69] Zakrevskii, A.D. "Testing for Identities in Boolean Algebra," *LYaPAS: A Programming Language for Logic and Coding Algorithms,* edited by M.A. Gavrilov and A.D. Zakrevskii. New York: Academic Press, 1969.

Vita

Captain James J. Kainec was born on 8 July 1960 in Garfield Hts., Ohio. Following graduation from high school in Bedford (Cleveland), Ohio in 1978, he received an appointment to the United States Military Academy at West Point, New York. Upon graduation from West Point in May 1982 with a degree of Bachelor of Science, he received a commission as a Second Lieutenant in the United States Army Signal Corps. After the completion of the Signal Officer Basic Course at Fort Gordon, Georgia, Captain Kainec was assigned to the 25th Infantry Division, Schofield Barracks, Hawaii. There he served as the Communications Platoon Leader for Headquarters, 1st Infantry Brigade, the Communications-Electronics Staff Officer for the 1st Battalion, 27th Infantry "Wolfhounds", and the Battalion Maintenance Officer for the 125th Signal Battalion. In 1986, Captain Kainec attended the Signal Officer Advanced Course at Fort Gordon, Georgia. Before beginning a master's program, he completed the US Army Teleprocessing Operations Officer Course at the Air Force Institute of Technology. Captain Kainec received the degree of Master of Science in Electrical Engineering from the Air Force Institute of Technology in 1988. Captain Kainec's next assignment will be as an instructor in the Department of Electrical Engineering and Computer Science, at the United States Military Academy, West Point, New York.

Permanent address: 374 Union Street
Bedford, Ohio 44146

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1992	PhD Dissertation

4. TITLE AND SUBTITLE

BOOLEAN REASONING AND INFORMED SEARCH IN THE MINIMIZATION OF LOGIC CIRCUITS

5. FUNDING NUMBERS

6. AUTHOR(S)

James J. Kainec, Captain, U.S. Army

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

School of Engineering
Air Force Institute of Technology (AU)
Wright-Patterson AFB, OH 45433

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/DS/ENG/92-02

9. SPONSORING MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Dr. John Hines
WL/ELED
WPAFB, OH 45433-6543 AVN 785-4448

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The minimization of logic circuits has been an important area of research for more than a half century. The approaches taken in this field, however, have for the most part been ad hoc. Boolean techniques have been employed to manipulate formulas, but not to perform symbolic reasoning. Boolean equations are employed principally as icons; they are never solved. The first objective of this dissertation is to apply Boolean reasoning systematically and uniformly to the minimization problem. Boolean reasoning entails the reduction of systems of Boolean equations to a single equation; the single equation is an abstraction, independent of the form of the original equations, upon which a variety of reasoning operations may be performed. The second objective is to apply informed search, which has arisen from research in Artificial Intelligence, to the minimization problem. A circuit specification is reduced to a single equivalent equation called a 1-normal form. It is shown that forming a particular solution for the equation corresponds to deriving a design.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Boolean Algebra; Boolean Sums; Logic Circuits Artificial Intelligence; Heuristic Methods Boolean Functions	608
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL