This is to certify that the

dissertation entitled

CANDIDATE SUBCIRCUIT ENUMERATION
FOR MODULE IDENTIFICATION IN DIGITAL
CIRCUITS

presented by

Jennifer Lynn White

has been accepted towards fulfillment
of the requirements for

Doctoral degree in Computer Science
& Engineering

Anthony J. Wojcik
Major professor

Date June 23, 2000

DATE DUE	DATE DUE	DATE DUE

CANDIDATE SUBCIRCUIT ENUMERATION FOR MODULE IDENTIFICATION IN DIGITAL CIRCUITS

By

Jennifer Lynn White

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science and Engineering

2000

Abstract

CANDIDATE SUBCIRCUIT ENUMERATION FOR MODULE IDENTIFICATION IN DIGITAL CIRCUITS

By

Jennifer Lynn White

Reverse engineering involves the transformation of system specifications to a description that contains fewer details but presents a more abstract view of the system. Within design automation, reverse engineering can be applied to low-level circuit descriptions, such as the geometric description of a chip, to recover a higher level of abstraction, such as transistor- or gate-level specifications.

Transforming a circuit specification from a lower-level specification (gates) to a higher-level specification (modules) is the focus of the *Module Identification Problem.* Transforming a specification from the gate to module level provides a conceptual description of the circuit, facilitating the reverse engineering effort. The Module Identification Problem has two subproblems: Candidate Subcircuit Enumeration and Subcircuit Identification. Candidate Subcircuit Enumeration locates clusters of gates within the target circuit that may be equivalent to a known high-level module, and Subcircuit Identification tests them for equivalence.

All gate clusters within the target circuit that may be functionally equivalent to a known high-level module must be enumerated to effectively allow Subcircuit Indentification. This thesis presents a technique for enumerating all candidate subcircuits. Each subcircuit is enumerated exactly once, to ensure complete coverage, and only subcircuits that perform a viable function in the context of digital logic are enumerated.

The number of subcircuits in a circuit can be exponential with respect to the number of devices in the circuit, so a technique is presented to partition the subcircuits into structural equivalence classes by generating an identifier for each subcircuit that describes the subcircuit structure. Only one instance of each structural equivalence class must be tested for semantic equivalence to known high-level modules; all other members of the class inherit the equivalence results. This classification reduces the number of times that semantic subcircuit identification must be applied, and thus improves the performance of module identification without loss of efficacy.

To everyone who believed that I would create this thesis,

especially my family,

who raised me to believe that I could.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Anthony S. Wojcik, for his guidance, encouragement, and patience throughout the process of researching and writing this dissertation. I would also like to thank my committee members, who provided excellent suggestions to improve this work, particularly Dr. Moon-Jung Chung for spending so much time considering the theoretical aspects, and Dr. Betty Cheng for thoroughly editing the dissertation itself.

This work was first conceived at Argonne National Laboratory under the advisement of Gregory Chisholm. I would like to extend my appreciation to him and the other members of the Reverse Engineering Team: Ken Dritz, Steve Eckmann, Chris Lain, and Bob Veroff.

My family has believed in me through all of my endeavors, and for that I thank them. They have always been willing to expend any effort necessary to help me during this educational process. I received encouragement, emotional support, and technical assistance from my dissertation support group: Mark Brehob, Scott Connell, Travis Doom, Dave Paoletti, Delia Raymer, and Mike Raymer. I would especially like to thank Mike and Delia, who happily welcomed me into their home for my frequent visits to Michigan State University during the completion of this dissertation.

Most importantly, I would like to thank Travis Doom, who encouraged me to begin work on a doctoral degree, applauded as each theory was formed, and supported me through each step of the process. His personal and professional support made it possible for me to travel this path.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

> "Reverse engineering is considered as the process of developing a set of specifications for a complex hardware system by an orderly examination of specimens of that system." [97]

Design automation is concerned with the computer-aided transformation of a circuit design from one level to another. The usual focus of design automation is *forward engineering* or *synthesis*, that is, the transformation of high-level design specifications to a geometric layout which realizes the design. The inverse of this process, *reverse engineering*, is a less extensively explored topic.

Reverse engineering involves the transformation of a lower-level specification to a higher-level specification. Simply put, reverse engineering is the investigation of the component parts of a system to discern the nature of the whole. This technique has been applied in environments such as computer software [24, 30, 46, 49], mechanical parts [101, 107, 108], and digital circuits [32, 77, 97, 105], and may be motivated by *reeengineering* for the purposes of modifying functionality or implementation, in-

formation retrieval, or replication of a component to extend the lifetime of a system. Reverse engineering is typically applied to an existing specification to recover a higher-level description of a system, but in the extreme case it may require the transformation of the physical implementation into a high-level specification of the system's behavior. These varied applications have led to many approaches to reverse engineering.

The research presented herein is directed toward developing an approach to reverse engineering in situations in which no design information is available about the component under study. The main focus of the research is the transformation from a description at the logic gate level to a functional specification, comprised of implementation independent functional modules.

Several techniques are presented in this dissertation to facilitate understanding of digital circuits by abstracting away from the details (gate-level description) and providing a more general overview (module-level description).

Section 1.1 discusses conceptual representations of circuits, including levels of abstraction, and the forward engineering process. Section 1.2 discusses why reverse engineering is currently an important topic in design automation. The formal definition of the Module Identification Problem is defined in Section 1.3. The scope of this research into the problem solution is discussed in Section 1.4. Section 1.5 presents the contributions made by this research. Finally, Section 1.6 presents an outline of this dissertation.

1.1 Background

Design automation is primarily concerned with developing techniques to facilitate the synthesis of an implementation that realizes a design specification. The synthesis process consists of several steps, each of which defines the circuit at an increasingly detailed level. These *levels of abstraction* are described in more detail in Section 1.1.1.

An integral part of developing a method for reverse engineering is understanding the forward engineering process. Section 1.1.2 describes the process of designing a hardware component, from the initial concept to the implementation specifications.

1.1.1 Conceptual Circuit Representation

Levels of Abstraction

A circuit may be described at many different levels by increasing or decreasing the amount of detail in the description. For instance, an arithmetic logic unit may contain several functional modules, such as adders and comparators, which are built of logic gates, and each logic gate is built of transistors. Each step in the progression is an example of one of the *levels of abstraction* [48] that may be used to describe digital circuits.

- *Processor Level.* The processor level is the highest tangible level of abstraction for circuits. It consists of microchip components such as processors, memories, controllers, and interfaces, as well as application-specific integrated circuits (ASICs).

- *Register Level.* The register level contains modules that are used to build the processor components, such as adders, registers, comparators, counters, etc.

- *Gate Level.* The gate level describes how the register level components are built from primitive logic elements. Only gates and flip-flops are present at the gate level.

- *Transistor Level.* The transistor level is the lowest level of abstraction, describing all gates and flip-flops of the gate level by the transistors necessary to implement them.

Design automation focuses on transformations between levels of abstraction. For example, from the register level, technology mapping (Section 2.4.3) may be used to locate appropriate gate-level implementations of necessary functionality. The research in this dissertation focuses on transforming a gate-level specification to an equivalent register-level specification.

Views

There are three major *views* that are encountered during the engineering or reverse engineering of a digital circuit: behavioral, structural, and physical [48, 84].

- *Behavioral View.* The behavioral view describes the functionality of the circuit to be engineered without considering the implementation. The behavior may be described as a function of the input values and elapsed time.

- *Structural View.* The structural view breaks the behavioral design into interconnected components. This view specifies the organization of the circuit and

some of its implementation.

- *Physical View.* The physical view includes the physical specifications of the components defined in the structural view, such as dimension and location. The physical view is generated for the purpose of manufacturing the design.

Level	View		
	Behavioral Forms	Structural Components	Physical Objects
Transistor	Differential equations, current-voltage diagrams	Transistors resistors, capacitors	Analog and digital cells
Gate	Boolean equations finite-state machines	Gates, flip-flops	Modules, units
Register	Algorithms, flowcharts, instruction sets, generalized FSM	Adders, comparators, registers, counters, register files, queues datapaths	Microchips
Processor	Executable specification, programs	Processors, controllers, memories, ASICs	Printed-circuit boards, multichip modules

Table 1.1: Hardware Specification Levels [48]

Each view may be described at any of the four levels of abstraction, as summarized in Table 1.1. By locating functional modules, datapath and control units may become clearly delineated, allowing an architectural-level description of the device to be recovered.

1.1.2 The Forward Engineering Process

The engineering process begins with a concept which is translated into a behavioral specification. This specification provides details about the interface of the circuit with the environment and the functionality that the circuit will perform. Behavioral-level specifications are often written in natural language, although executable specifications are becoming more common because they may be verified, analyzed, and synthesized more easily. Executable specifications include some hardware description languages (HDLs), such as VHSIC Hardware Description Language (VHDL) [6] and Verilog [15]. These HDLs may be used to describe the circuit at various levels, as well as perform simulation and synthesis.

This overview of the circuit's functionality is then partitioned into blocks, each of which has a well-understood functionality that can be described by a mathematical formula or an algorithm, or possibly still in natural language, to be refined later. This represents the first layer (processor) in the structural view of the circuit. The processor level representation is then refined to include more information about implementation by determining which register-level modules will be used to provide the functionality for the processor-level modules. In a similar step, the register-level modules are replaced by equivalent gate-level implementations. This task is often automated, and focuses on determining the implementation of each register-level module that optimizes the circuit for quality measures such as area, cost, or performance. The gates are then replaced by their transistor representation, and the design may be verified for correctness against the original behavioral design. The design is finally

manufactured by replacing the structural components in the specifications by their physical counterparts.

1.1.3 Relevance

The overall goal of reverse engineering is to raise the level of abstraction of the circuit specification to recover a conceptual understanding of the circuit, based on functional modules. The levels described in Section 1.1.1 (processor, register, gate, transistor) are the four levels at which the circuit may be represented. The objective of the Module Identification Problem (Section 1.3) is to transform a gate-level specification to a register- or processor-level specification.

It is apparent by examining the process of forward engineering that module identification is a useful tool for reverse engineering. This top-down methodology in which large modules are implemented by connecting smaller modules or gates results in clusters of gates that perform interesting and identifiable functions.

Further, when synthesizing gate-level specifications from the register transfer level description, the engineer may use implementations from a standard cell library, or create new implementations for the register modules. In either case, it is likely that the circuit will contain gate clusters that may be identified as functional modules.

1.2 Motivation

Reverse engineering was defined in 1985 by M.G. Rekoff as "the act of creating a set of specifications for a piece of hardware by someone other than the original de-

signers, primarily based upon analyzing and dimensioning a specimen or collection of specimens." [97]. This landmark paper was the first to formally discuss reverse engineering of hardware, its difficulties, and to present a methodology for performing reverse engineering of complex hardware systems. Rekoff identified two primary motivations for reverse engineering: the gaining of information and the reproduction of a part.

1.2.1 Information Retrieval

Reverse engineering of a part to gain information has many applications. Large corporations apply reverse engineering to the products of their competitors to maintain a competitive edge. With the development of design watermarking techniques [25, 63, 64], a chip manufacturer can determine via reverse engineering whether its intellectual property has been stolen. It is likely that military powers apply reverse engineering to captured weapons and military equipment. When correct and current specifications of a component are not available, information retrieval becomes the first step in the process toward Rekoff's second motivation for reverse engineering: the reproduction of parts.

1.2.2 Reengineering

> "An electronic product may need reengineering for a number of reasons related explicitly or implicitly to the passage of time. It needs reengineering if its requirements (what it does) change over time or if its specifications

(how it does it) change." [77]

Reengineering or *redesign* is the modification of a component to extend or modify its functionality or implementation. This activity requires that an accurate specification of the component is available so that appropriate changes can be made and subsequently re-implemented in hardware.

The reproduction of a part by reengineering often drives reverse engineering of a device. Technology improvements may allow a faster, smaller, cheaper, or more efficient implementation to be designed. Unexpected design flaws may be detected after the unit is in service, requiring the implementation of an error-free replacement. A device in a large system may degrade and require replacement. If no spares are available, then the device may be reverse engineered and reproduced, forestalling the replacement of the entire system.

There are two basic types of replacement parts [97]. A *clone* is an exact reproduction of the original device, including details such as the location and material of each internal primitive element. A *surrogate* is a re-implementation of the device, with identical functionality and input/output performance, but the internal design and implementation may be completely different. Clones of digital logic devices are seldom possible because the rate at which fabrication technology advances obviates a design within 18 months. Fabrication facilities may cease to exist for older technologies within a short span of time, causing cloning to be financially infeasible. For most reverse engineering motivations, clones are not necessary; the identical functionality provided by surrogates is sufficient.

EPOI [21, 105]

The United States Air Force is currently developing a solution to their inability to obtain needed parts. Just 20 years ago, before integrated circuits became ubiquitous in the consumer market, the military accounted for over 15% of integrated circuit purchases. Since integrated circuits began appearing in personal computers and other household devices, the military has lost its influence over the market, and can no longer expect manufacturers to maintain obsolete technology fabrication facilities to provide parts indefinitely. Consequently, the Air Force has developed the Electronic Parts Obsolescence Initiative (EPOI) to explore other options.

EPOI has identified four alternatives to handle parts obsolescence: life-of-time buys, complete system upgrade, part substitution, and board redesign. Life-of-time buys involve purchasing sufficient stock to support replacement throughout a system's expected lifetime. A complete system upgrade is so expensive that it is seldom warranted. Part substitution and board redesign have been determined to be the most cost-effective solution to parts obsolescence. Both of these alternatives involve reverse engineering of the existing components.

The current cost of a replacement that has the same interface and functionality of the original part can range from $10,000 to $30,000, primarily due to the fact that the engineers must manually recover the design from incomplete original specifications. This situation has lead the Air Force to investigate automated tools for reverse engineering, with the help of commercial and research companies such as Litton-TASC, TRW, Raytheon, Northrop Grumman and Aspect Development Inc.

Obtaining Specifications

Reengineering frequently requires information retrieval. The amount of effort that is necessary to reengineer a component is directly related to the amount of design information that is available. It is obvious that when a specification that has been proven to be accurate is available, then no reverse engineering is necessary and the redesign process can begin immediately. All too frequently, however, the original specifications for the component are not available. The original specifications may have been lost or damaged. A more common problem, particularly when dealing with older (legacy) systems, is that the company originally contracted to design and manufacture the device is no longer in business. Design knowledge and specifications have disappeared as well. Such situations require extensive effort to recover the design from the hardware.

If a complete and provably accurate design is available or can be obtained, then it is possible to re-implement the hardware from that specification. If some original design information is available, then it can be used to guide the reverse engineering process, thereby greatly simplifying the effort.

Even when specifications are available, it is unlikely that they represent a reliable description of the device. Changes made further along during the design or implementation process may not be accurately reflected in the documentation. For this reason, it is prudent to treat information gathered from design documents carefully. The only completely accurate information is contained within the device itself.

There are two types of specifications that are necessary to completely describe

complex hardware products: functional and dimensional [97]. The functional specification describes how the hardware works and what capabilities it performs. Dimensional specifications describe the details about the fabrication of the component, including materials, dimensions, distances between constituent components, coefficients of friction, etc.

Depending on the goal of the reverse engineering, it is likely that only the functional specification is required. If the intention is to redesign to take advantage of new manufacturing technology, then it is not necessary to have the obsolete dimensional specifications. Only the functional information is required, and the redesign engineers or computer aided design (CAD) utilities can create appropriate dimensional specifications.

Dimensional specifications may be helpful, however, if the goal is to correct a design error. Rather than re-implement the entire component, only the erroneous section must be repaired and re-implemented. Dimensional information from the faulty component can reduce reengineering time.

Legacy Systems

The most frequent candidates for reverse engineering and reengineering are not systems that were designed last month or last year, but rather those that were designed many years ago with now obsolete design tools and technology. These are referred to as *legacy systems.*

As the number and age of existing legacy systems increases, so does the demand for automated methods of design recovery. The problem of extracting design information

from hardware is gaining importance as many of these existing systems are reaching a state at which they need to be reengineered to meet changing design specifications, overcome defects, or replace failing parts.

Legacy systems are particularly likely to require reverse engineering. Original design information is seldom available [39], and the designers or manufacturers of these older systems no longer exist, so the specifications are no longer available or are obsolete.

Legacy systems are often complex systems that perform a very specific task. Designing a new system to take over the functionality is usually prohibitively expensive. The most practical way to maintain a legacy system is in an evolutionary manner, incrementally redesigning and replacing portions of the system as necessary [78].

1.2.3 Alternatives to Reverse Engineering

The reverse engineering of hardware is a difficult task, and it is worthwhile to consider other options before initiating the reverse engineering process. An obvious alternative is to simply design the circuit again. If some of the original design specifications are available, then this may be an appropriate solution. An expenditure of sufficient man-hours could result in an acceptable circuit. This approach may not be cost-effective if original design information is not available.

Additionally, if the component to be redesigned interacts with other components within the system, it is necessary to have a comprehensive understanding of the interfaces with those other components. Interface information is frequently not available,

rendering this approach difficult or impossible. In this case, the entire system may need to be replaced.

1.2.4 Reverse Engineering Focus

The research presented in this dissertation focuses upon the reverse engineering of digital hardware in the absence of design information toward the goal of redesign or replacement of the hardware component. The resulting specification will contain a functional register-level description of the component. The techniques presented in this thesis utilize only the information that can be derived from the component itself, such as a transistor-level netlist, transistor position information, circuit power lines, and clock lines.

1.3 Problem Description

Reverse engineering can be described as abstracting away details to obtain a more general understanding. In digital circuits, the details involve low-level components such as transistors, resistors, logic gates, and buffers. To create a more easily understood description of the circuit, those components are subsumed into functional modules such as arithmetic logic units, multiplexors, and adders.

The recovery of a gate-level specification (*netlist*) from a physical piece of hardware involves physically examining each layer of silicon composing the device to create a low-level geometric description [31, 107], which can then be converted into a transistor-level netlist by pattern matching or visual inspection [31]. Transistor implementations

of logic gates are generally small in number and can be easily identified by graph isomorphism and pattern matching techniques [17, 72, 91]. Techniques to perform these three steps toward abstraction have been developed, and recovering a design to the gate-level is considered to be an achievable step in the process. Section 2.2 presents more detailed descriptions of the applicable techniques.

The same cannot be said of a subsequent transformation from that netlist to a functional module-level netlist. The transformation of a gate-level netlist describing a combinational logic circuit into a modular-level representation of the circuit in terms of functional components and glue logic is the focus of the general *Reverse Engineering Project*, first proposed at Argonne National Laboratory by Eckmann and Chisholm [43].

To illustrate the difference between a gate-level description and a module-level description, a 2-bit adder is shown in Figures 1.1, 1.2, 1.3 and 1.4. Figure 1.1 shows the adder described at the gate-level, which is divided into two 1-bit full adders in Figure 1.2. Those two 1-bit full adders are functional modules, so the same 2-bit adder can also be described by the modular description in Figure 1.3. Moving to a higher level of abstraction within the modular description describes the 2-bit full adder as a single module in Figure 1.4.

1.3.1 Syntactic and Semantic Matching

Many existing approaches to the transformation between gate-level and register-level specifications use specialized solutions to the classic graph-theoretic subgraph iso-

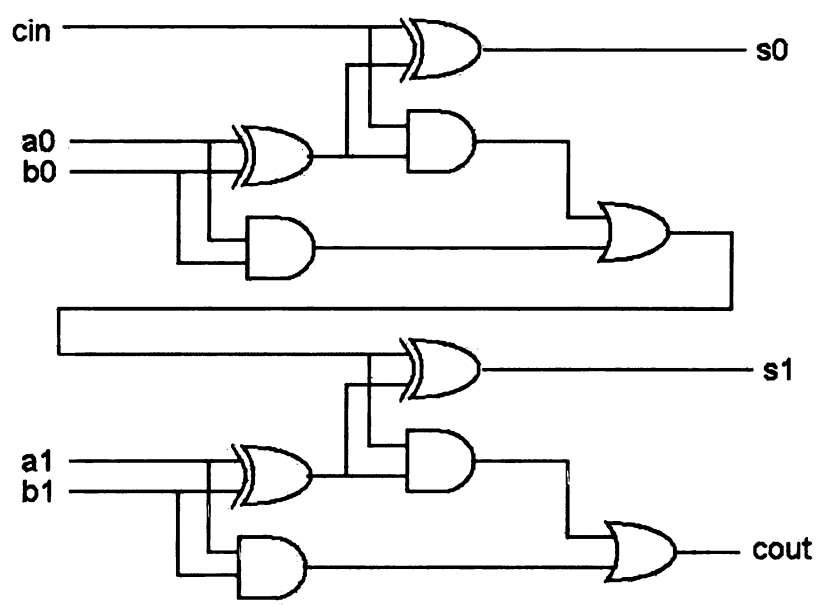


Figure 1.1: Gate-level 2-bit Full Adder.

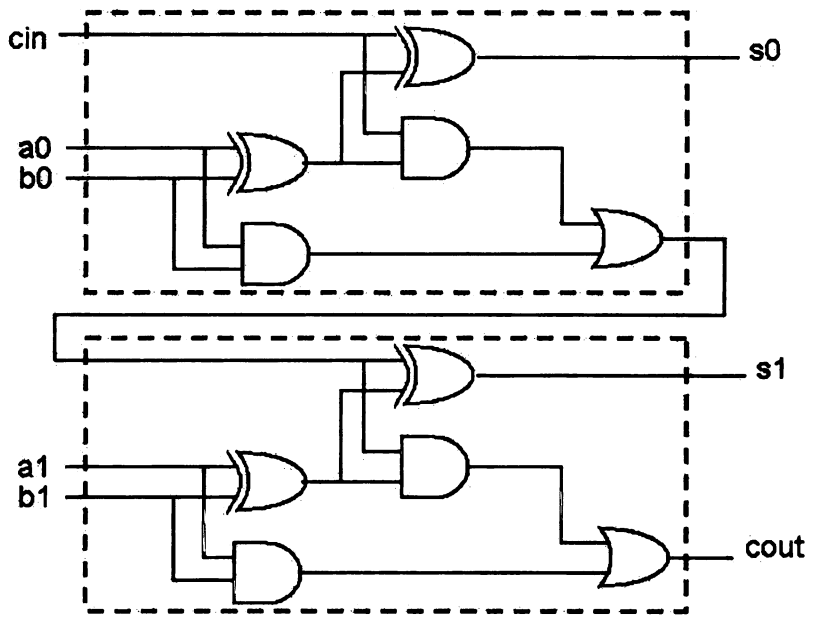


Figure 1.2: Gate-level 2-bit Adder with Two 1-bit Full Adders Identified.

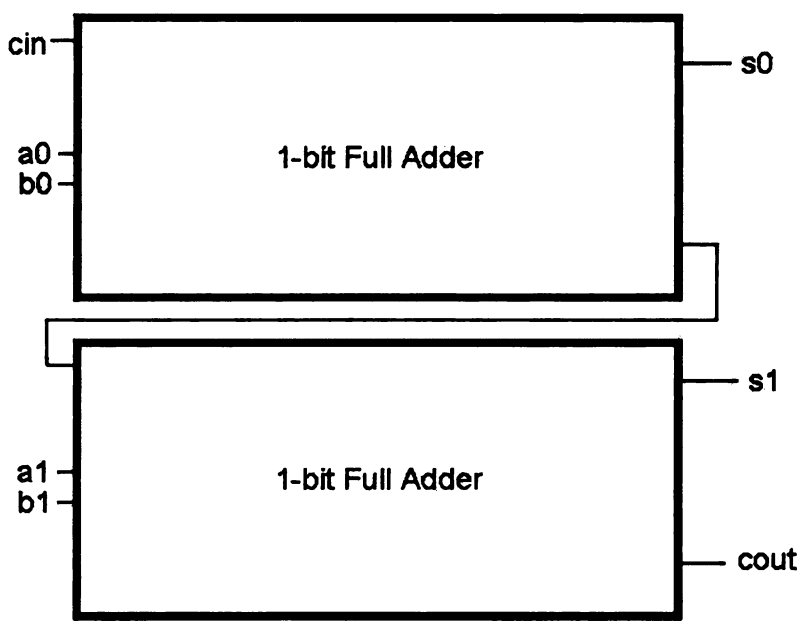


Figure 1.3: 2-bit Adder as Two 1-bit Full Adder Modules.

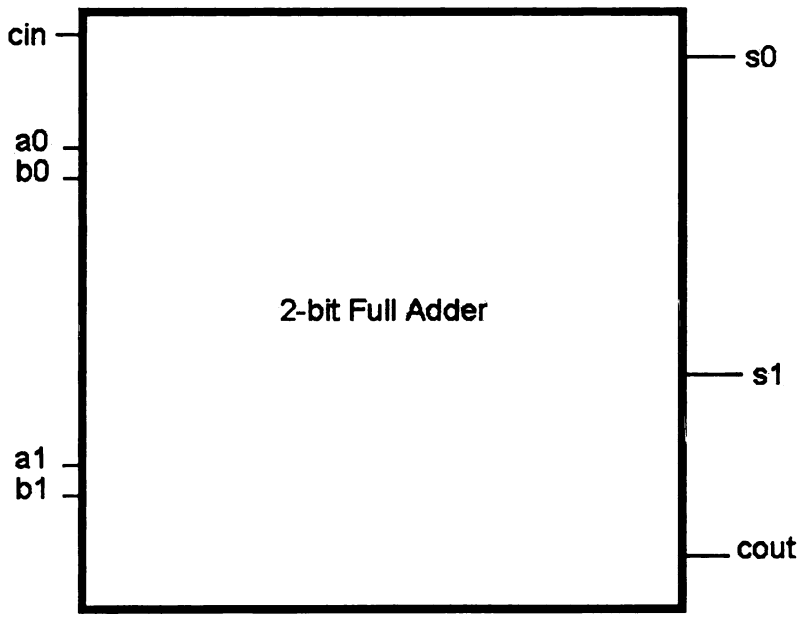


Figure 1.4: 2-bit Adder as One 2-bit Full Adder Module.

morphism problem [110]. Given a specific implementation of a functional module, these approaches can locate all instances of that circuit structure within the circuit under examination, the *target circuit.* This is referred to as *syntactic matching* [17, 72, 76, 91, 93] because it matches circuit structure (syntax) rather than meaning.

Systems employing syntactic matching are usually very fast, capable of locating all instances within linear time with respect to the size of the target circuit [91]. Unfortunately, not all functionally equivalent circuits are structurally equivalent. As described above, a 2-bit adder may be built by connecting two 1-bit adders implemented by NAND gates, but it may also be implemented completely differently, in XOR-AND format, for example. These two implementations will perform exactly the same function, but will bear no resemblance to one another.

Therefore, the drawback of the syntactic approach is that it is only capable of locating known implementations of the functional modules. To locate all instances of 2-bit adders in a circuit by applying a syntactic matching technique, it would be necessary to create a library of all possible implementations of a 2-bit adder. Even with extensive libraries, it is unlikely that all needed implementations of a functional module will be present. In addition, the matching time to search for all implementations would be prohibitive.

Syntactic matching approaches can provide valuable information, particularly when the circuit contains many stock implementations from a known library. This is likely in circuits that were designed by using glue logic to connect modules from a library to implement the desired functionality. However, reverse engineering is often applied to obsolete circuits or circuits designed by someone other than the reverse

engineer, so it is unlikely that the original implementation library used to design the circuit is known and available.

A more powerful matching technique is *semantic matching* [32, 75, 92]. Semantic matching compares the logical functionality of the circuits rather than their structure. When applying semantic matching, the module library need only contain one implementation of each functional module. All gate clusters in the target circuit that perform the same function, regardless of their structure, will be identified by semantic matching.

In an effort to provide a general solution for reverse engineering, the research in this dissertation focuses upon a problem formulation for module identification that applies semantic matching instead of syntactic matching.

1.3.2 The Module Identification Problem

The following formal description has been developed to describe the gate-level to module-level transformation of a digital circuit specification.

Module Identification Problem [37, 38, 114]. Given a gate-level logic description (*netlist*) of a target circuit, efficiently identify all gate clusters (*subcircuits*) that perform the function of a known library module.

- **Candidate Subcircuit Enumeration Problem.** Identification of gate clusters (*candidate subcircuits*) within the netlist that may comprise a functional module.

- **Subcircuit Identification Problem.** Proving functional equivalence between a candidate subcircuit and a known standard library module.

The research presented in this dissertation provides an approach to the first half of the Module Identification Problem: Candidate Subcircuit Enumeration.

Subcircuit Identification The second part of the Module Identification Problem is determining whether the subcircuits produced by candidate subcircuit enumeration are equivalent to a functional module. Several solutions to this problem have been proposed in the literature, and are discussed in detail in Section 2.8.

Candidate Subcircuit Enumeration A *candidate subcircuit* is defined as a cluster of gates that possesses a well-defined functionality in the context of digital systems. To provide the most complete coverage of a circuit with functional modules, all candidate subcircuits should be enumerated.

If the circuit is viewed as a graph, in which vertices represent gates and edges represent wires, enumerating all subgraphs is a very complex problem because the number of subgraphs is not polynomially bound [50]. Even in a graph that conforms to the structure of a planar binary tree, the number of subtrees of height n that contain the root vertex is described by the doubly exponential recurrence relation:

$$x_{n+1} = x_n^2 + 1, n \geq 0; x_0 = 1 \tag{1.1}$$

which generates the sequence 1, 2, 5, 26, 677, 458330, 210066388901, ... [1].

The structure of a circuit is not as well-defined as that of a binary tree, so it is not

possible to develop a simple equation to obtain an exact count of the subgraphs in an arbitrary circuit. It is obvious that enumeration of all subgraphs is a computationally difficult problem. However, for the purpose of reverse engineering, not all subgraphs need to be enumerated. Only those subgraphs that represent viable circuits need to be considered. A viable *subcircuit* is represented by a subgraph in which each gate represented is fully-specified with respect to the subgraph. If the subgraph contains a 2-input AND gate, the subgraph may not contain only one of the gate's inputs. Either both or neither of the inputs must be present to fully-specify the gate. Limiting enumeration to subcircuits reduces the subgraphs to be enumerated considerably. For instance, in a 1-bit adder composed of eight logic gates, there are 114 subgraphs, but only 18 subgraphs that represent subcircuits. Only those 18 need to be enumerated.

To further limit the number of subgraphs that must be enumerated, the class of subcircuits may be restricted to include only those subcircuits that do not share any functionality with neighboring subcircuits. These subcircuits are known as self-contained subcircuits. Of the 18 subcircuits in the 1-bit adder, only 6 of those are self-contained.

In addition to defining the types of subgraphs to enumerate, it is important to ensure that each of the subgraphs is enumerated exactly once. A naïve algorithm for subgraph enumeration, which iteratively adds neighboring vertices to an existing subgraph, will produce each subgraph many times, an unnecessary expenditure of computational effort. If each subgraph produced, including duplicates, is then tested for semantic equivalence, module identification becomes completely impractical.

Even when only a restricted class of subgraphs, subcircuits, is enumerated, the number of subcircuits in the original circuit may still be unwieldy. Several heuristics have been developed in this research to guide the enumeration without interfering with the effectiveness of the overall technique.

1.4 Problem Scope

1.4.1 Reverse Engineering Focus

The process of reverse engineering for the purpose of transformation between gate-level and register-level circuit specifications is presented in Figure 1.5. The process begins with a gate-level netlist, which is partitioned into smaller segments to improve performance. The current solution to the Module Identification Problem does not apply to sequential circuits (because current semantic matching approaches are limited to combinational logic), so the combinational logic is extracted from these segments and passed to the Candidate Subcircuit Enumeration module. All of the subcircuits generated are placed into Structural Equivalence Classes, then matched to known modules. A circuit covering technique may then be applied to produce a register-level description of the circuit.

The work presented in this dissertation focuses on the steps outlined in bold, Candidate Subcircuit Enumeration and Structural Equivalence classes.

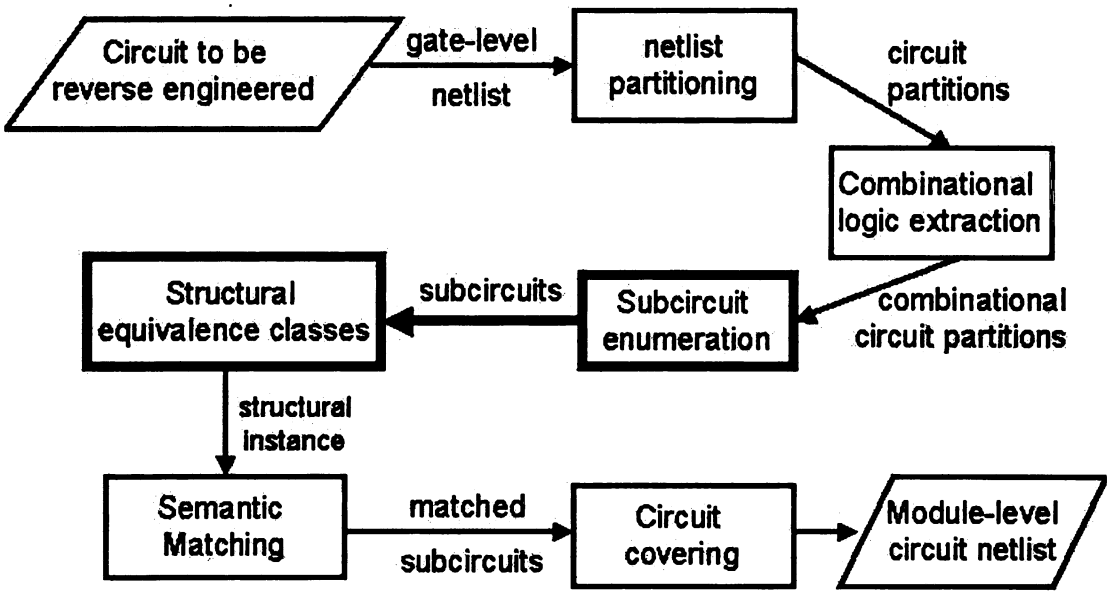


Figure 1.5: Overview of Gate Level to Register Level Transformation

1.4.2 Approach Requirements and Limitations

The approach to the Module Identification Problem (Section 1.3) described in this dissertation focuses on performing gate-level to register-level transformations on combinational logic circuits. The solution to Candidate Subcircuit Enumeration presented in Chapter 3 has been designed to work on general directed graphs, including those with cycles. The current solutions [32, 36] available for Subcircuit Identification perform combinational logic matching only, and cannot operate effectively on sequential circuits.

Thus, this approach may depend upon the following information:

1. The circuit represents combinational logic only.

2. The functionality of the candidate subcircuit is assumed to have the same number of inputs and outputs as that of the library module. No output can be "ignored"; every output of any library entity must either be a primary output or be used as the input at some other point in the circuit.

3. The functional modules to be located are expected to be a single connected subcircuit, not several disjoint subcircuits. If a module is built of disjoint subcircuits then each subcircuit should be defined as a module.

4. The functionality of the candidate subcircuit and the library module must match exactly; *don't care* sets are not considered at this time.

As research progresses in semantic matching of sequential circuits, new approaches to subcircuit identification can easily be incorporated into the module identification

system. It is also possible to perform preliminary partitioning into combinational blocks [23], thus allowing reverse engineering of the combinational portions of the design.

In addition, it is currently necessary that a library of functional modules be available. The technique presented in this thesis applies semantic matching to the Module Identification Problem, so only one implementation of each module need be included in the library. The library modules may be represented in any format, including circuit netlist, circuit graph, or in a pure functional format, described by Binary Decision Diagrams [19].

1.5 Contributions

The research presented in this dissertation provides both theoretical and practical contributions to the field of design automation. Techniques have been developed to allow the reverse engineering of a gate-level circuit specification to a more conceptual module-level specification. This module-level specification can be more easily utilized for the purposes of reengineering, reuse, replacement, and repair.

1.5.1 Overview

- *Problem formulation:* The Module Identification Problem, as discussed above, has been developed to provide a framework for this research. Details presented in Chapter 1.

- *Unique Subgraph Enumeration:* A technique for uniquely enumerating each subgraph of a general graph has been developed by indexing the vertices and subgraphs and applying ordering rules to guide subgraph expansion. Details presented in Chapter 3.

- *Restrictive Subgraph Types:* Two types of subgraphs that are interesting within the Module Identification Problem have been defined: subcircuits and contained subcircuits. The enumeration of subgraphs can be focused solely on these subgraphs to reduce the effort required to enumerate candidate subcircuits. Details presented in Chapter 3.

- *Structural Circuit Identifiers:* A technique for developing structural circuit identifiers to place subcircuits into structural equivalence classes has been defined. This reduces applications of semantic matching. Details presented in Chapter 4.

- *Heuristics for Practical Application:* The techniques cannot be applied 'as is' to circuits of arbitrary size, so a set of heuristics has been defined to allow effective candidate subcircuit enumeration to be performed for any circuit. Details presented in Chapter 5.

1.5.2 Details

Module Identification Problem A formal problem, the Module Identification Problem, has been defined to describe the reverse engineering of digital circuits. The Module Identification Problem consists of two subproblems: Candidate Subcircuit

Enumeration and Subcircuit Identification. The research in this dissertation focuses on the problem of Candidate Subcircuit Enumeration.

Unique Subgraph Enumeration The unique and comprehensive enumeration of subgraphs of an arbitrary graph is a theoretical result that has many applications within design automation and graph theory. The most important contribution is vertex indexing and associated rules that ensure that each subgraph will be created exactly once. This indexing prevents unnecessary computation and improves execution time.

Unique enumeration may be applied within both syntactic and semantic solutions to the Candidate Enumeration part of the Module Identification Problem. When coupled with standard partitioning heuristics, it can be used to improve bottom-up partitioning methods. The vertex ordering that prevents duplication of subgraphs may easily and efficiently be applied to netlist clustering and vertex clustering. Clustering appears in the solution to many problems, so widespread benefits can result from improving its execution.

Restricted Subgraph Types The enumeration of all subgraphs of a graph is intractable because the number of subgraphs is not polynomially bound. To focus enumeration upon subgraphs that may be functionally equivalent to a module, and to reduce enumeration complexity, two classes of subgraphs have been defined.

A *subcircuit* is a subgraph that possesses well-defined functionality. It is necessary that all gates in the subcircuit are accompanied by either all or none of its inputs.

Gates that are only partially specified are not allowed. Restricting enumeration to subcircuits provides considerable reduction of effort without reducing effectiveness.

Further, a *contained subcircuit* is a subcircuit that does not share functionality with neighboring subcircuits. These subcircuits are likely to represent a complete functional module, and thus their enumeration and identification represent an efficient step in the reverse engineering process.

Structural Circuit Identifiers A structural representation of circuits was developed to implement the concept of subcircuit equivalence classes. This *structural identifier* completely describes the structure of the circuit, so the subcircuits may be partitioned into structural equivalence classes. Structurally equivalent subcircuits are necessarily functionally equivalent, so only one instance of each equivalence class must be tested for equivalence to register-level modules.

Although several other string representations for graphs and circuits are available (see Section 2.6), they require extensive processing to determine equivalence of graphs and circuits. The method presented in Chapter 4 is based on efficiently determining a vertex ordering that can be used to generate a structural identifier. This technique allows fast and effective partitioning of the subcircuits into structural equivalence classes. Although the application is simplified by the vertex labels, indicating gate type or functionality, it also may be applied to unlabeled directed graphs.

Heuristics for Practical Application The number of candidate subcircuits in a circuit can grow exponentially with the number of devices in the circuit. Therefore,

the practicality of performing candidate subcircuit enumeration decreases as the size of the circuit grows. Therefore, several heuristics have been defined that allow module identification to be performed on circuits of any size. These heuristics include reduction of the circuit size by preliminary partitioning, limitation of candidate subcircuit order, and hierarchical module replacement.

1.5.3 Summary

To solve the Module Identification Problem for a digital circuit, each cluster of gates in a circuit that may be equivalent to a known module must be enumerated. An algorithm is presented in this dissertation that uniquely enumerates each and every subcircuit that may be functionally equivalent to a known module. The number of these subcircuits may be inherently intractable, so several guidelines have been designed to effectively reduce the number of subcircuits enumerated. Properly applied, these heuristics will permit circuits of arbitrary size to be analyzed, without reducing the accuracy of the module identification.

In addition, the concept of structural equivalence classes and a technique for determining a efficient representation for comparison of subcircuit structure are discussed to further improve the efficiency of the solution to the Module Identification Problem.

The combination of unique enumeration of subcircuits and structural circuit representation provides an effective solution to the first part of the Module Identification Problem, Candidate Subcircuit Enumeration. When combined with a semantic matching approach to Subcircuit Identification, the basis for a powerful reverse engi-

neering tool is created.

1.6 Dissertation Outline

Chapter 1 has discussed motivations for reverse engineering, presented the specific formulation that is the focus of this dissertation, and stated the scope of the problem. Chapter 2 reviews the current literature on reverse engineering methods and solutions to related problems. Concepts, terminology and notation that will be used throughout this dissertation are also presented.

Chapter 3 discusses the Subcircuit Enumeration Problem and presents an algorithm that enumerates each and every candidate subcircuit of a circuit exactly once. Chapter 4 introduces the idea of subcircuit equivalence classes, which serves as an effective method of reducing the number of necessary semantic comparisons. The application of these theoretical techniques is discussed in Chapter 5, with several methods to extend their application to circuits of any size. Chapter 6 contains a summary of the contributions of this research as well as future work in this area.

Chapter 2

Background and Related Work

As the need for reverse engineering increases, so does the motivation for research into better methods and techniques to accomplish reverse engineering tasks. Previous work has provided important background to the topics discussed in this dissertation, and many of these techniques may be applied in conjunction with the algorithm and heuristics presented here to achieve improved results and useful contributions to the reverse engineering effort.

The formulation of the Module Identification Problem in Section 1.3 emphasizes its relation to problems such as technology mapping, design validation, circuit clustering, and regularity extraction. These problems are described and contrasted to the Module Identification Problem, and current research into their solutions is presented in this chapter.

This research represents digital circuits as directed graphs. Section 2.1 presents the general concepts and terminology of digital logic and graph theory that are necessary to understand the discussions in the remainder of this dissertation. The techniques

presented in this dissertation focus on the transformation of a gate-level netlist to a module-level netlist, where the module level may contain register- or processor-level modules. Techniques developed by others that perform the transformation from the physical-level to a gate-level netlist are discussed in Section 2.2.

Related work is described in the remainder of the chapter, beginning with a discussion of other approaches to reverse engineering in Section 2.3. There are several application areas in which similar problems are being researched. These application areas are presented in Section 2.4. Discussion of techniques used to solve candidate enumeration within these domains follows in Section 2.5, including circuit partitioning in Section 2.5.1 and netlist clustering in Section 2.5.2.

Techniques for representing circuits for comparison are presented in Section 2.6. The two approaches to Subcircuit Identification, syntactic and semantic matching, are discussed in Sections 2.7 and 2.8, respectively. The reverse engineering of software shares many goals with reverse engineering, such as the abstraction away from details and location of functional modules. Section 2.9 discusses several approaches to software reverse engineering. Section 2.10 summarizes this chapter.

2.1 Terminology

The following section provides background for understanding the Module Identification Problem and its subproblems, as well as the related work discussed in the following sections.

2.1.1 Design Automation and Digital Circuits

Design automation is the study of the computer-aided transformation of a digital design from one level of abstraction to another, ending with the final synthesis of a digital circuit. The research described here provides an approach to reverse engineer a low-level (gate-level netlist) description of a circuit to a high-level (functional-module netlist) specification. The following terms are from digital logic and are used throughout this dissertation [111].

Definition 2.1 *A* **gate** *or* **logic gate** *is an electrical component that has one or more inputs and produces an output that is a function of the current input values. Examples: AND, XOR, NOT.*

Definition 2.2 *A* **module** *(***functional module** *or* **high-level module***) is a collection of interconnected gates that perform a known and well-defined functionality. Examples: ALU, multiplexor, adder (Figure 1.3).*

Definition 2.3 *A* **circuit** *is an arrangement of logic gates that are interconnected to perform a specified function. A circuit may include many modules.*

Definition 2.4 *The depth of a device g in a circuit C is the length of the maximum path between g and any input of C.*

Definition 2.5 *A* **specification** *is a description of a circuit or a digital design. Specifications may exist in many different levels of abstraction (Section 1.1.1). Specifications are written in hardware description languages, such as VHDL and Verilog.*

Definition 2.6 *A* **gate-level netlist** *is a circuit specification containing only logic gates.*

Definition 2.7 *A* **module-level netlist** *is a circuit specification containing high-level modules and glue logic (gates).*

Definition 2.8 *A* **flip-flop** *is a device that stores either a 0 or a 1. The value of a flip-flop may change only at times determined by a clock input. Examples: D flip-flop, S-R flip-flop.*

Definition 2.9 *A* **combinational** *logic circuit is a circuit that contains no storage or memory capability.*

Definition 2.10 *A* **sequential** *logic circuit is a circuit that contains storage or memory capabilities. Sequential logic circuits include flip-flops.*

Definition 2.11 *A* **synchronous** *logic circuit is a sequential circuit whose state changes only at a specified point on a triggering input called the clock.*

Definition 2.12 *A* **library** *is a collection of high-level modules. Details are in Section 1.4.*

Definition 2.13 *A* **partition** *of a circuit is a contiguous portion of the circuit.*

2.1.2 Graph Theory

The technique of subcircuit enumeration presented in this dissertation represents circuits as directed graphs. This section presents graph theory terminology [28, 34] that will be used during the discussion of the techniques developed in this research.

Graphs

Definition 2.14 *A* **graph** G *is a finite nonempty set* $V(G)$ *of objects called* **vertices** *and a (possibly empty) set* $E(G)$ *of two element subsets of* $V(G)$ *called* **edges**. *The set* $V(G)$ *is called the* **vertex set** *of* G *and* $E(G)$ *its* **edge set**.

Definition 2.15 *The number of vertices in a graph* G *is called its* **order**, *denoted* $|V(G)|$, *and the number of edges is its* **size**, *denoted* $|E(G)|$.

Definition 2.16 *A* **trivial graph** *is a graph consisting of a single vertex and no edges.*

Definition 2.17 *Vertices* u *and* v *are* **adjacent** *to each other in a graph* G *if* $uv \in E(G)$. *If* u *and* v *are adjacent, then* u *is a* **neighbor** *of* v *and* v *is a* **neighbor** *of* u.

Definition 2.18 *For a vertex* v *in a graph* G, *its* **neighborhood** *is defined by* $N(v) = \{u \in V(G) \mid vu \in E(G)\}$. *The* **degree** *of a vertex* v *is the number of vertices adjacent to* v, *that is,* $|N(v)|$.

Definition 2.19 *A graph* H *is a* **subgraph** *of a graph* G *if* $V(H) \subseteq V(G)$ *and* $E(H) \subseteq E(G)$.

Definition 2.20 *A subgraph* H *of* G *is an* **induced subgraph** *of* G *if* $E(H)$ *contains precisely those edges joining two vertices in* $V(G)$.

Definition 2.21 *A* **walk** *in a graph* G *is an alternating sequence of vertices and edges,* $v0, e1, v1, e_2, v_2, ..., e_n, v_n$ *for* $n \geq 0$, *beginning and ending with vertices, such that* $e_i = v_{i-1}v_i$ *for* $i = 1, 2, ..., n$.

Definition 2.22 *A* **path** *in a graph G is a walk in which no vertex is repeated.*

Definition 2.23 *A graph G is* **connected** *if there exists a path between u and v for every pair u, v of vertices of G. Otherwise, it is* **disconnected**.

Definition 2.24 *A vertex v is a* **cut-vertex** *of G if $G - v$ is disconnected.*

Directed Graphs

Definition 2.25 *A* **directed graph** *or* **digraph** *D is a finite, nonempty set $V(D)$ of vertices and a (possibly empty) set $E(D)$ of ordered pairs of distinct vertices. The elements of $E(D)$ are called* **arcs**.

Definition 2.26 *If uv is an arc of D, then u is said to be* **adjacent to** *or a* **parent of** *v, and v is said to be* **incident to** *or a* **child of** *u.*

Definition 2.27 *The* **outdegree** *of a vertex v in a digraph D is the number of vertices incident to v. The* **indegree** *of v is the number of vertices adjacent to v. The* **degree** *of v is outdegree(v) + indegree(v).*

Definition 2.28 *A* **semiwalk** *in a directed graph D is an alternating sequence of vertices and arcs, $v0, e1, v1, e_2, v_2, ..., e_n, v_n$ for $n \geq 0$, such that either $e_i = \{v_{i-1}, v_i\}$ or $e_i = \{v_i, v_{i-1}\}$ for each $i (1 \leq i \leq n)$.*

Definition 2.29 *A digraph D is* **connected** *if there exists a semiwalk between u and v for every pair u, v of vertices of D.*

2.2 Transistor- to Gate-Level Transformation

Beginning the reverse engineering process with a digital device and no external information is the most difficult type of reverse engineering. However, the hardware itself is the most reliable information that is available, because specifications often become incorrect as changes are made during the implementation process (Section 1.2.1).

The research presented in this thesis assumes that the hardware has already been reverse engineered to a gate-level specification. This section gives an overview of the steps necessary to recover the gate-level netlist from a physical circuit. Table 2.1 presents the steps of design recovery for digital circuits, as developed at the 1998 Reverse Engineering Workshop [103] and presented in [36]. The well-defined techniques for transforming a circuit from the physical device implementation to the gate-level implementation (Levels 0 - 4) are described in the following paragraphs.

Sample Preparation (Level 0) The first step involves extraction of the design layout of the device. A destructive technique (chemical etching or mechanical slicing) is used to expose each layer of the device [31].

Image Acquisition (Level 1) High-resolution images of each level are obtained by a scanning electron microscope. Many images are necessary to capture the entire layer at a high enough resolution to allow further recovery. These images are then assembled to present an image of the entire layer [31].

The problem inherent in these steps is the destructive nature of this etching process. Care must be taken to ensure that the images are collected accurately. If only

Level	Technologies	Technical Challenges
0	*Sample Preparation*	
	Etching	Etching
1	*Image Acquisition*	
	SEM Image processing BMP to GDS-II	Accuracy Geometry Staging Unconventional technology
2	*Geometric Description*	
	Postprocessing Design rule checkers	Process information
3	*Transistor Netlist*	
	Syntactic matching Semantic matching Semantic matching	Exact models of gates Unconventional technology
4	*Gate Netlist*	
4a	*Layout*	
	Pattern matching Syntactic matching Semantic matching Contextual matching Optimization tools	Combinatorics Optimizations Library support Incomplete information Clustering Function-centric naming
4b	*Timing*	
	Simulation and modeling Technology-specific information	Unconventional technology
5	*Register Transfer*	
	Model generation Sequential functionality Timing Domain-specific information	Complexity Validation Automating process
6	*Behavioral*	

Table 2.1: **Reverse engineering technologies and challenges, 1998. [36]**

one instance of the device to be reverse engineered exists, then the images will be the only information remaining about the device after image acquisition.

Geometric Description (Level 2) The high-resolution images obtained from the Image Acquisition stage must be converted into a geometric data stream, a description of the elements on each layer including position information. This can be accomplished by standard pattern recognition techniques, provided that knowledge about the implementation technology is available [8, 47, 55].

Transistor-Level Description (Level 3) The transformation from the geometric description to a transistor-level description can be performed by commercially available CAD tools that recognize physical structures in the geometric description such as transistors, wires, and resistors [31].

Gate-Level Description (Level 4) The final step to bring the design to a gate-level description uses isomorphism techniques to locate transistor implementation of logic gates. This step requires an extensive library of gate implementations. Because the number of transistor implementations of gates is relatively small, syntactic library matching is usually sufficient for this purpose. Many tools are available to perform this transformation, such as *SubGemini* [91], LOGEX [17], GateMaker [72], and BLEX [76], among others.

2.3 Reverse Engineering

> "Reverse engineering is the inverse of the design process. The design process begins with an abstract description of a target device and, through a succession of refinements, produces an implementable design. Reverse engineering begins with the disassembly of a manufactured device and ends with an abstract description of its functionality." [32]

The reverse engineering of a digital design has many purposes. It may be necessary to change the specifications of the product to add new features or to meet changing client needs. Portions of an existing system may be reverse engineered so they may be used within another system [99]. With improvements in available technology, hardware may be re-implemented to improve power, size, or speed constraints. The availability of cheaper technology or materials may drive reverse engineering for redesign of a functionally equivalent but more cost effective part.

Technology is improving at such a fast rate that it is necessary to perform reverse engineering and redesign quickly so the new part is not obsolete before it is released. Thus, reverse engineering techniques must produce results quickly and effectively so that the redesign process may begin.

The most practical way to implement recovery of the gate-level netlist is to incorporate recovery within the design process itself. A system that allows simple transitions between design levels (higher or lower) produces designs that can be redesigned within the system. Unfortunately, most hardware was not designed in such an interactive system, so reverse engineering is a "blind" process. Some design information

may be available, but it is not guaranteed to be current.

The following sections describe several approaches to reverse engineering. The first two, Module Identification and FINES, are the techniques described in the literature that are the most similar to the approach described in this dissertation. The last approach, Hybrid Reverse Engineering, describes a general methodology for module identification that provides many interesting techniques for simplifying reverse engineering. These techniques span both automated and manual identification techniques.

2.3.1 Module Identification

The technique that is most similar to the research presented in this dissertation is proposed by Chisholm *et al* [32]. Two parts of the module identification problem are identified:

- Partitioning - generate candidate subcircuits by partitioning the subject netlist.
- Matching - determine matchings between candidate subcircuits and library modules.

This technique considers both syntactic and semantic matching. The syntactic approach is used initially to locate subcircuits whose functionality is already understood. Assuming that module identification is for the purposes of verification of one's own design, the library modules for this step can be built from the design documentation.

The syntactic matching technique used for this approach is the *SubGemini* algorithm [91]. Details on this algorithm can be found in Section 2.7.2. Recognizing that syntactic matching has limitations, such as requiring a large module library and an

inability to match optimized modules, the researchers now apply a semantic technique to locate all implementations of the library modules.

The semantic equivalence problem is solved by using Binary Decision Diagrams (Section 2.8.1) as a canonical form for the circuits, then comparing those canonical representations as necessary. The difficulty with using Binary Decision Diagrams as a canonical form is that it is necessary to determine a correspondence between the inputs and outputs of the pattern circuit and those of the target circuit.

The candidate subcircuits are identified by applying partitioning to the circuit. Details on this technique are given in Section 2.5.1. Once the candidate subcircuits have been determined, the canonical BDD description of their logic is generated for each output. These canonical forms for the candidate outputs are compared to the canonical forms for the outputs of the library elements. If all outputs of the candidate match all outputs of a library element, a match has been found.

Discussion

The focus for the technique presented by Chisholm *et al* [32] is on reducing the complexity of the second step of the Module Identification Problem (Matching or Subcircuit Identification) by identifying possible outputs of modules in the target circuit and attempting to match those with outputs of library modules. This ensures that the matching is done between smaller circuits with only one output, which simplifies each individual matching test considerably, though it requires a greater number of equivalence tests because each output must be matched.

The approach to the Module Identification Problem described in this thesis per-

forms matching between full subcircuits with multiple inputs and outputs. Candidate Subcircuit Enumeration (Chapter 3) concentrates on reducing the complexity of the first step (Candidate Enumeration). By combining these improvements to each of the Module Identification steps, a powerful tool for module identification may be developed.

2.3.2 FINES

FINES (*Functional INformation Extraction System*) [92] is a prototype system designed to extract logic functions from combinational systems for the purpose of design verification and developing functional simulation models and test patterns. The eventual goal of the project is to enable automated development of design documents and manuals for logic circuits.

The interesting feature of FINES is that it focuses on extracting arithmetic functions instead of logic formulas because they are sufficient for the purposes of design verification and functional simulation and are generally simpler and more compact than logic formulas.

The goal of FINES is to represent a circuit with a function table, which is an intermediate form between logic formulas and truth tables. A function table is a desirable representation because it is compact and is frequently used to describe commercial integrated circuits in data books, so it is a familiar representation to many engineers. Binary Decision Diagrams (presented in Section 2.8.1) are used to create the function table, because they allow well-defined manipulation to determine

a reduced BDD, which translates into a smaller but equivalent function table.

FINES operates under the assumption that additional information is available from an engineer to guide the extraction process. This additional information can take the form of inputs, outputs, and control data of important modules.

The input to FINES is a logic circuit description, a library of arithmetic functions, and some additional information. The output is a function table that describes the functionality of the logic circuit. The additional information is used to determine the ordering of BDD variables that results in the most compact function table. The authors determined by experimentation that by ordering the control inputs first, the resulting ROBDD (and thus the function table) was smaller.

Once the function table has been determined for a logic circuit, the known arithmetic functions are extracted. FINES compares the BDDs of the library entities (arithmetic functions) with the function table created for the circuit. The method of determining correct input correspondences between the library entity BDD and a possible match BDD is not discussed in [92].

Discussion

The interesting feature of this approach is that it describes the entire target circuit by a table describing its functionality in logic formulas, then attempts to locate library entities in this function table so that an algebraic function table may be created.

The function table is a more easily understood representation of a circuit, but its readability and compactness is dependent upon additional information provided by an engineer. Without this additional information, the function table is equivalent to

a truth table, which would be quite large for circuits with realistic functionality.

A limitation of this system is that it is designed to locate only arithmetic functions. However, this approach can be easily extended by expanding the library to contain more complex modules. The real difficulty with FINES is that the entire circuit must be transformed into a BDD, which can quickly become prohibitively large. This requirement severely limits the size of circuits to which FINES may be applied.

2.3.3 Hybrid Module Identification

Hansen, Yalcin and Hayes [53] describe the reverse engineering of several of the ISCAS-85 benchmark circuits to illustrate a hybrid module identification technique as well as the hierarchical nature of the ISCAS benchmarks.

The technique applied in this case study uses many different module identification techniques to identify all meaningful modules in the target circuit. These techniques are as follows:

- *Library modules.* Location of known library modules, such as multiplexors, decoders and adders. These modules can be found in integrated circuit manufacturers data books and common cell libraries.

- *Repeated modules.* Multiple instances of a subcircuit (with or without identified functionality) may occur in the target circuit. These repeated elements are most common in datapath circuits, and are the focus of regularity extraction techniques (discussed in Section 2.4.1.

- *Expected global structures.* After identifying several modules, a reverse engineer

can use that information as a guide to locate larger modules that may use the known modules.

- *Computed functions.* For small subcircuits (no more than four or five signals), the logic function may be computed and aid in identification of local modules.

- *Control Functions.* Key control signals may be identified that partition the functionality of a subcircuit into simpler subcircuits on various control line values.

- *Bus structures.* After repeated modules have been identified, their outputs may be grouped together into buses, allowing circuit partitioning based upon that information.

- *Common names.* If names are known for elements in the netlists, elements with similar or identical names can be grouped together, in hope of sparking further structural insight.

- *Black boxes.* If a group of gates cannot be identified by any of the above techniques, they may be encapsulated within a black box.

Discussion

This method requires a high-level of engineer input. It does apply a syntactic matching technique to locate the library modules and repeated modules identified by the engineer. This paper provides an excellent overview of module identification techniques, many of which may be applied by an engineer at any stage of the module

identification process to guide an automated system.

A significant drawback to the methodology is that the library module identification applies syntactic matching. However, the approach assumes that an engineer is guiding the entire process, which provides a large degree of flexibility. Once a module has been identified by any of the methods listed above, it may be placed in the library so other instances may be identified.

2.4 Application Areas

Although reverse engineering or design recovery is the stated application of Module Identification in this dissertation, there are several other problems in the digital logic domain to which this technique may be applied. Three of these applications are described here. Although the problem description may differ, the underlying focus is the same. All of these problems can be solved by the location of functional modules within a circuit netlist.

2.4.1 Regularity Extraction

Circuits are commonly designed to optimize a set of features, such as footprint, performance, power consumption, etc. One of the powerful tools available for this optimization is *regularity extraction* [5, 33, 95]. Regularity extraction involves the identification of *templates*: gate clusters within the circuit that may re-occur multiple times. By identifying these templates, designers may be able to redesign the system to exploit the duplication, rather than replicating the technology many times. In addition, the

designers can optimize a template for a specific factor (speed, size, etc), and that optimization effort applies to all instances of the template. Thus, regularity extraction can improve the efficiency of the layout as well as reduce the necessary design effort.

The process of regularity extraction can be described by three steps:

1. *Template Library Identification*: Identify a set of templates, the *template library*.

   Many of the techniques currently available assume that the template library was designed and provided by engineers [96]. Several techniques provide a template library identification step [33]. Essentially, this problem involves the enumeration of subcircuit equivalence classes under isomorphism. However, due to the potentially exponential number of subcircuit equivalence classes, restrictions are placed on the type of subcircuit that qualifies as a template. For instance, one might only consider subcircuits that occur more than once within the circuit, or subcircuits with only one output, or subcircuits that are not subcircuits of another template.

2. *Template Location*: Locate all instances of each of the templates.

   Using the templates provided by Step 1, locate all instances of each template. This problem is equivalent to the subgraph isomorphism problem. The identification of the subcircuits for comparison to the templates is accomplished by applying clustering methods [96] or by a syntactic matching technique [33].

3. *Determine Template Cover*: Position templates to cover circuit.

   The goal of this step is to identify the templates that cover the largest portion of the circuit. The problem of determining the optimal cover is difficult, so many

regularity extraction techniques rely on designer input to facilitate the process. Common heuristics used are Largest-Fit-First, in which the largest templates are applied first, then smaller templates are applied to cover the remaining area, and Most-Frequent-Fit-First, in which the template with the largest number of instances is applied first. The remaining logic components comprise a module that must be custom designed.

Relation to Module Identification Problem

This problem is directly related to the Module Identification Problem, except that regularity extraction is commonly performed to locate syntactic matches. There are many techniques that can be used to locate these syntactic matches that are more efficient for this purpose than the technique presented in this dissertation.

However, the generation of the template library and identification of isomorphisms could be performed by the techniques presented in this thesis. Regularity extraction methods require that the template subcircuit occur at least once, and in fact attempt to locate the templates with the most occurrences in the graph. This count can easily be maintained while the structural equivalence classes (Chapter 4), which represent templates, are developed by Candidate Subcircuit Enumeration (Chapter 3). The result would be an exhaustive listing of the templates in the circuit, as well as a count and identification of their instances, which could then be used as an input to the template covering step.

2.4.2 Formal Verification

An important step in circuit synthesis is design verification. Once the circuit has been implemented from its specifications, it is necessary to prove that the intended functionality was correctly realized. It is common for the verification process to consume 70% of the design time [2].

In the past, simulation has provided the basis of design verification for digital circuits [44]. Due to increasing circuit complexity, performing extensive simulation tests of the implementation is no longer a reasonable approach. Exhaustive simulation of a design requires a test set that represents all possible inputs to a system along with expected outputs. This verification technique is infeasible for most circuits, so determining a set of input vectors to be tested is a possible approach, but may leave some input vectors untested so cannot guarantee equivalence.

Consequently, many current verification strategies take advantage of formal verification methods [61]. By applying formal verification to the designs, it is no longer necessary to exhaustively test all input vectors because the implementation can be proved to be correct for all possibilities.

In a survey of current formal verification methods for combinational circuits, Jain *et al* [61] describe the available approaches as belonging to either functional or structural approaches to verification:

- *functional*: Circuits are represented by a functional canonical form and are considered to be equivalent if their canonical forms are equivalent. Lai, Pedry, and Sastry [74] use BDDs, which are a very common canonical representation

in verification, to represent the circuits to be compared.

- *structural*: Key equivalent vertices are located within the Boolean network extracted from the implementation and in the Boolean network representing the original design. These starting points are used to simplify and focus the equivalence checking process. The Gemini [42] and GeminiII [41] systems both follow structural approaches to verification.

  Pelz and Roettcher [93] and Batra and Cooke [9] both present hierarchical structural techniques for formal verification that successively match library modules to higher and higher levels of abstraction. Many structural circuit equivalence techniques have been derived for transistor-level implementations [60, 100, 106].

The two types of formal verification, structural and functional, correlate directly to the syntactic and semantic reverse engineering techniques identified in this dissertation. The design recovery technique applied by *SubGemini* [91] to identify high-level modules is based directly on the structural formal verification techniques first used in Gemini.

Relation to Module Identification Problem

There is one primary difference between formal verification of circuit designs and reverse engineering of circuits. In formal verification, the modules being matched, the mapping between devices, and the overall functionality of the circuit are all known *a priori* or can be easily derived.

The equivalence checking process can be effectively guided with this information.

In addition, the correspondences between input and output variables are generally available (unlike reverse engineering), which simplifies the semantic comparison process. Therefore, formal verification of circuits is a much simpler task than reverse engineering an unknown device.

The solution to the Module Identification Problem presented in this thesis is similar to the functional approaches to logic verification. However, formal verification, because it is working with two essentially known circuits, does not need to perform the first step of the Module Identification Problem, Candidate Subcircuit Enumeration. Therefore, the unique enumeration of subgraphs is not a technique that should be applied within formal verification. However, the structural identifier presented in Chapter 4 could prove very useful within structural verification techniques for proving structural equivalence.

2.4.3 Technology Mapping

After the design for a new digital device is complete, it is transformed into a technology independent logic description and optimized. That specification is commonly stored as a Boolean network. The next step in the design process is to identify logical modules (standard cells) that provide the best cover (based on design goals such as footprint, power, and speed) of the functionality represented by the Boolean network. This step is called *technology mapping.*

Technology mapping is a particularly important step in the design of a circuit because the quality of the design is greatly affected by the choice of cells. The cell

selection can optimize the realized circuit in many ways, including area, speed, and power.

There are two steps inherent in technology mapping:

- Step 1 - *matching*: locating logical modules that perform the necessary functionality.
- Step 2 - *covering*: identifying an optimal cover of the Boolean network with the logical modules.

The matching step in technology mapping has typically been handled by Boolean matching [115], although pattern matching of logic functions has also been applied [81]. Many older techniques used tree matching, restricting matched modules to those with only one output [91].

A typical technique for solving the technology mapping problem is to represent the logic function to be performed in a canonical form, such as NAND2 and NOT1 gates exclusively. The cells are also represented by a canonical form, and the goal is to determine an optimized cover of the canonical function graph with the canonical primitive cells.

Tree covering approaches are capable of operating in linear time when both the target graph and the subject graph are trees, by applying dynamic program techniques. When the functions are not restricted to representation by trees, directed acyclic graph covering techniques [71] may be applied.

Wu, Chen, and Acken present an interesting alternative to NAND2-NOT1 canonical form that involves manipulation of a function's truth table to develop an input

ordering that permits a canonical description [115]. This approach, although designed for application to single-output functions, has possible applications to multiple-output functions as well.

Relation to Module Identification Problem

Technology mapping involves determining an optimal cell cover of an optimized logic function. The cells available are generally small (20 logic gates) and have only one output. This simplicity allow the application of more focused techniques to be applied to solve technology mapping problems. The matching applied within the Module Identification Problem must be able to determine equivalence between complex modules with multiple inputs and outputs.

2.5 Candidate Enumeration

The first part of the Module Identification Problem, Candidate Subcircuit Enumeration, can be approached in several ways. The first method involves partitioning the netlist into the candidate subcircuits. This approach to subcircuit enumeration and details of some research in this area are discussed in Section 2.5.1. The other common approach is to build clusters of vertices. The details of netlist clustering and some related approaches are given in Section 2.5.2.

2.5.1 Partitioning

Many graph partitioning algorithms have been modified specifically to operate on circuit netlists [3]. The goal of these general algorithms is typically to divide a system specification into clusters such that the number of inter-cluster connections is minimized for use in circuit board layout design. This section will discuss several partitioning techniques that have been applied within module identification to identify candidate subcircuits.

Although they appear similar, there are several factors which distinguish the Module Identification Problem from the partitioning problem. Partitioning implies that the clusters are disjoint, but that assumption cannot be made for the Module Identification Problem. Functionality may be shared among several modules, so they may overlap and share gates. Relying on a strict partitioning algorithm could result in unidentified modules.

In solving the Module Identification Problem, it is necessary to identify *all* subcircuits that are semantically identical to the high-level module. An approximate partition will not suffice. If a NOT gate is appended to the end of a subcircuit that is equivalent to a functional module, the function that that subcircuit performs is drastically altered. The subcircuit will no longer be able to be identified as the functional module.

Several approaches to the Module Identification Problem use partitioning to identify candidate subcircuits for comparison. Details on the heuristics and techniques used to guide the partitioning are described in the following sections.

Partitioning Heuristics

The design recovery approach by Chisholm *et al* (Section 2.3) [32] uses three heuristics to guide the partitioning of the circuit into candidate subcircuits:

- *Logic signal analysis.* Two devices that have equivalent signals at their control ports are more likely to be part of the same module. Clock trees of flip-flops are developed, and the constituent gates are correlated to determine whether or not they are likely to belong to the same module.

- *Relative position analysis.* Devices that lie close to each other are more likely to be part of the same module. Designer input can also be used to define an area of the netlist that looks (to the human eye) like it may be a module.

- *Graph analysis.* Identification of articulation points, path length between vertices, and cycles all provide hints regarding functional modules. For example, two vertices separated by only a few edges are more likely to be a part of a module than if they were separated by hundreds of edges.

Key Vertex Location

The *SubGemini* [91] algorithm identifies a *key vertex* in the library module, then locates all of the instances of that vertex within the circuit. This set of vertices is called the *candidate vector*, and each represents a possible partition of the circuit. Each of the vertices in the candidate vector are then investigated to determine whether or not they may match the key vertex.

2.5.2 Subgraph Clustering

The goal of netlist clustering in this application is to locate the subcircuits of the gate-level netlist that may be equivalent to library modules. Netlist clustering has been applied to many areas of digital design, particularly module placement in physical design [59, 65, 98].

Typically, the goal of clustering a netlist is to group cells into disjoint clusters. By clustering a netlist, the complexity of the netlist can be reduced, thus improving the execution time of partitioning and searching algorithms.

The two basic approaches to clustering are top-down and bottom-up. Top-down approaches recursively partition the netlist into subclusters by considering global information about the netlist. When applied to the Module Identification Problem, this paradigm is similar to the partitioning discussed in Section 2.5.1. Top-down approaches, although they are unlikely to make bad decisions, are computationally complex and are seldom applied alone.

The bottom-up approach is more similar to the candidate subcircuit enumeration research presented in this dissertation. Initially, each cell (gate) is assigned to a trivial cluster. The clusters are then merged together by considering local information about neighboring clusters, until a satisfactory clustering of the netlist has been achieved [4, 52, 65, 96, 98].

Discussion

Standard clustering approaches develop clusters that are disjoint. In module identification, it is possible for the modules to share gates, a situation that often occurs when the circuit is optimized. The Candidate Subcircuit Enumeration approach presented in Chapter 3 will produce all gate clusters, so that overlapping modules may be identified.

Many clustering concepts may be applied to subcircuit enumeration, such as heuristics to determine whether or not a gate may be added to a subcircuit. If a gate is physically located "far" from the current cluster, it is unlikely that it is part of the same functional module. If a gate is connected to a subcircuit by a single wire, it is less likely (though obviously not impossible) that it is a part of the module. These heuristics may be added to the implementation of the general solution to the candidate subcircuit enumeration problem to improve execution time in real-world applications.

Similarly, clustering approaches may be improved by applying the vertex indexing technique and rules described in Chapter 3 to avoid duplication of gate clusters.

2.6 Circuit Representations for Structural Comparison

There are several approaches described in the literature that develop a label for the circuit or graph and then compare these labels via string matching [13, 58]. This

approach was first applied to representing the molecular structure of chemical compounds, which are very similar to circuit graphs in that each vertex has additional information (atom type, gate type) that may be used to simplify equivalence comparisons.

This technique has several advantages. String comparison is much more efficient than comparing graphs vertex by vertex, and strings serve as an excellent hash index, allowing isomorphic graphs to be quickly located. The disadvantage of this technique is that it is often difficult to develop the label itself. However, for module identification via structural isomorphism, circuit labeling can be very useful, because the labels for the circuits in the library may be calculated before execution begins.

A circuit labeling technique, applying a vertex weighting approach similar to that described in Section 2.6.1 is applied in Chapter 4 to partition the subgraphs into equivalence classes.

2.6.1 Structural Labeling

Benecke, Kerber, and Laue present a technique for canonically numbering chemical molecules to facilitate searches in an on-line chemical identification system (MolGen [51]) [11]. This approach considers structural properties of the molecules to develop classes to define the order in which a nested structure can be built to describe the molecule.

This technique does not elegantly perform canonical element ordering. If two elements cannot be simply ordered, a backtracking procedure is required to evaluate

the resulting nested structures to determine the correct canonical one. Developing a technique for canonical element ordering is possible, though computationally costly.

2.6.2 K-Formulas

A *K-formula* is a representation of directed graphs that is defined by a set of linear formulas. This string preserves the structure of the graph, and the graph may be recreated from its K-formula representation [10].

K-formulas were first proposed by Krider [69] and further explored by Berztiss [13, 14], and have been applied to the problem of graph isomorphism [14]. The K-formulas for two graphs may be evaluated for equivalence by attempting to determine a matching between the vertices in the two graphs.

K-formulas are based on representing arcs of the directed graphs as the K-operator ('*') followed by the arc's endpoints. An arc connecting vertex u to vertex v is represented as $*uv$. A K-formula for a vertex v is written with a K-operator for each arc, then the vertex name, followed by the endpoints of all arcs leaving v. The K-formula for a vertex v that has neighbors u, x, and y is $***vuxy$. The symbols for the endpoints of the arcs may be written in any order, so $v's$ K-formula may also be written $***vxyu$ or $***vyux$, for example.

The K-formula for a directed graph may be derived by combining the formulas of the vertices. Vertex labels for endpoints in the K-formula for the vertex v may be replaced by the endpoint's K-formula. Details on this substitution process are available in [13].

The problem with using K-formulas to represent directed graphs is that there may be more than one K-formula for a single graph, because the K-formula is dependent on the order in which the graph was traversed during the K-formula creation. Multiple K-formulas for a single graph structure means that determining isomorphism of two graphs is a complicated process, involving determining the correct matching between vertices in the graphs.

The technique presented in Chapter 4 provides a method of ordering the vertices of a directed graph to allow a particular set of K-formulas to be created for a structure. Thus, each graph structure will result in only a few K-formulas, allowing more efficient comparison for determining graph isomorphism results.

2.6.3 Circuit Matrix Manipulation

Huang and Overhauser [58] present a circuit labeling technique that uses manipulation of a circuit matrix to develop a unique circuit label for each subcircuit structure. This technique was designed to operate on bipolar junction transistors, which have three different vertex terminals. A circuit matrix is an adjacency matrix for the circuit graph that also represents the type of terminal connecting the vertices.

Once the circuit matrix is built, it is arranged via partitioning to develop a final fully partitioned matrix. This matrix defines a unique ordering of the subcircuit vertices and elements. The partitioning involves four steps:

1. Partition by vertex size of partitioning groups.

2. Partition by types of vertices and elements.

3. Partition by connection numbers in the matrix.

4. Arrange the unpartitioned portion to determine maximum matrix code.

The matrix code is generated by concatenating the integer value of each row in the partitioned circuit matrix. The value of a row is the binary number created when each element in the row is considered to be 0 if the element value is 0 and 1 otherwise.

The circuit code itself is generated by concatenating the matrix code, the numbers in the matrix, the ordered list of element types, and the ordered list of vertex types. This code uniquely defines each subcircuit, allowing comparison of the codes, rather than the subcircuits. This technique is designed to operate on graph vertices with exactly three neighbors (transistors). The research presented in Chapter 4 derives a similar result for circuits containing vertices with an arbitrary number of neighbors.

2.7 Syntactic Matching

The problem of locating gate clusters syntactically equivalent to a given circuit has been addressed in several different domains, including reverse engineering [17, 72, 91], technology mapping [68, 94], regularity extraction [33, 87, 88, 96], and validation [41]. The problem itself is related to the classic graph theoretic problem, subgraph isomorphism, which is known to be NP-complete [50].

There are two structural isomorphism problems that play a part in this research.

- *graph isomorphism:* Given two graphs $G1$ and $G2$, determine whether or not they are structurally identical. In the research presented in this dissertation,

this problem arises when the subcircuits are divided into equivalence classes.

- *subgraph isomorphism:* Given two graphs $G1$ and $G2$, determine whether or not there is a subgraph within $G2$ that is structurally identical to $G1$. This problem is equivalent to locating a subcircuit within a circuit that is structurally equivalent to a template subcircuit (structural matching).

Both the graph isomorphism and subgraph isomorphism problems also have applications in other fields, most notably pattern recognition [22, 29, 83] and chemical analysis [12, 40, 104, 109]. Accordingly, much effort has been expended to develop efficient techniques for solving these difficult problems.

Although graph and subgraph isomorphism are difficult problems, their application in other domains is frequently simplified by additional information. In digital circuits, vertices represent logic devices with specific functionality, which can be used to facilitate these techniques.

2.7.1 Graph Isomorphism

> "Two finite graphs are isomorphic if there exists a bijective map between the vertex sets of the two graphs that preserves adjacency." [57]

Graph isomorphism has been a standing problem for many years because it holds interest both for the theory and applications communities. This problem has not been proven to be NP-Complete, nor has it been proven *not* NP-Complete [7, 57]. Assuming that $P \neq NP$, it can be considered to be $NP - intermediate$ [57], along

with factorization of integers and computing discrete logarithms [7]. This ambiguous status appeals to theoreticians. Graph isomorphism has been solved for many restricted types of graphs, such as planar graphs, which are linear in the graph size, and trivalent graphs, which require $O(n^4)$ steps to test isomorphism [57].

There are three primary approaches to graph isomorphism represented in the current literature: topological, group-theoretic, and algorithmic.

- *Topological Approach*: The topological approach involves embedding the graphs to be compared onto a surface of minimum genus [1]. Each graph is drawn onto a surface without any edges crossing. For example, a sphere allows only planar graphs to be embedded onto it. For non-planar graph, a surface with more sides is necessary, such as a torus, so all edges may be drawn without crossing. Once the graphs are embedded, they may be dissected into planar components, and those planar components may be compared in time linear to the number of vertices in one of the graphs.

  However, the problem with this approach is that determining the genus of a graph is a difficult problem in itself, as is the dissection into planar components [57].

- *Group-Theoretic Approach*: The group theoretic approach involves determining the group of all automorphisms for the graph. An automorphism is any mapping or function that, when applied to a graph G, produces an identical graph G'.

[1] A surface is a sphere in which a certain (finite) number of holes have been inserted. Equivalently, a surface is a sphere on which a number of handles have been placed. The number of holes or handles is called the genus of the surface. The surface of genus 0 is, therefore, the sphere (or plane)[28].

By determining the automorphisms of the two graphs to be compared, the automorphism groups may instead be compared.

The difficulty with group-theoretic approaches is that there is large overhead associated with them and therefore they usually have a large constant factor within their complexity [83].

- *Algorithmic Approach*: A more practically oriented approach to graph and subgraph isomorphism is the algorithmic approach, in which algorithms are designed to investigate the search space, generally applying depth first search and backtracking techniques.

There are several approaches presented in the literature to solve the graph isomorphism problem within the specific domain of circuit netlists, including partitioning, and canonical circuit labeling (discussed in Section 2.6). Chapter 4 presents a solution to the circuit isomorphism problem that uses canonical structural circuit identifiers.

Partitioning

The circuit validation approaches *SubGemini* [91], *Gemini* [42] and *GeminiII* [41] use a partitioning approach to solve the circuit isomorphism problem.

Given two circuits $C1$ and $C2$, the goal is to prove them isomorphic or non-isomorphic. The two circuits are partitioned by vertex invariants such as gate type and degree with the intention of developing partitions that contain equivalent vertices. Once an initial partitioning has been accomplished, the partitions for each vertex are refined based on information about neighboring vertices. If the circuits are

isomorphic, the partitions developed through this partitioning will be identical.

The partitioning in the Gemini algorithms is done by applying a label to each vertex that classifies the vertex into a partition. Initially, the label represents the vertex invariants (device type, net degree). For example, a likely partition after the first partitioning of a gate-level netlist would contain all NAND gates that have 2 inputs and 1 output. As the partitions are refined, the vertices are relabeled with the previous vertex label and with the labels of the neighboring vertices. An isomorphism has been found when a matching of singleton partitions has been identified for the circuits.

2.7.2 Subgraph Isomorphism

Subgraph isomorphism is an NP-Complete problem [7, 50] consisting of two subproblems: locating subgraphs that may be equivalent to the pattern subgraph and determining structural equivalence. As with graph isomorphism, the additional information obtained from the circuit graph can be used to derive practical solutions to locating isomorphic subgraphs.

Backtracking

A successful algorithm for determining subgraph isomorphism was proposed by Ullman [110], and is still considered to be one of the most efficient approaches to the problem. It improved upon the original backtracking procedure for graph isomorphism presented by Corneil and Gotleib [35]. The algorithm is based on the fact that subgraph isomorphism may be determined by applying a brute force enumera-

tion algorithm to perform a tree search simultaneously on the pattern subgraph and the target graph, in the same way as [14].

The difficulty with this technique, in general, is the large number of vertices that must be searched to identify an isomorphism. Ullman prunes the number of vertices that must be considered by applying a refinement step after each vertex has been added to the potentially isomorphic subgraph. This step eliminates successor nodes that will not lead to discovery of an isomorphism, improving the average execution time to a time proportional to n^3, where n is the number of vertices in the pattern graph.

Decision Trees for Multiple Pattern Graphs

Bunke and Messmer [83] present an approach to subgraph isomorphism that can simultaneously search for multiple pattern graphs within the target graphs. Multiple pattern graphs arise in many applications, including syntactic matching for gate- to register- level transformation, so this technique has practical appeal.

The technique involves the generation of all permutations of the adjacency matrix of a pattern graph, and then a decision tree representing these permutations is built. The adjacency matrix permutations for any number of pattern graphs may be added to the decision tree. The decision tree is applied to the adjacency matrix of the target graph to identify all pattern graphs within.

Multi-place Graph Weighting

Within the field of verification, Luellau, Hoepken, and Barke [76] present BLEX, a tool that employs syntactic matching to identify components in a transistor-level netlist. As defined in this context, multi-place graphs have an additional vertex type: multi-nets. Edges represent device pins (base, collector, emitter, cathode, anode) and are distinguishable as such. Multi-nets represent intersections between edges in the graph.

Each edge in the graph is weighted with a primary number representing its pin type. To obtain the weight of a vertex or multi-net, the weights of its adjacent edges are multiplied together. A weighted incidence matrix can be built to show adjacency of multi-net vertices and device vertices. The values in the matrix product of all edges connecting the device with the multi-net. Two circuits are identical if their weighted incidence matrices are identical.

It would be possible to interchange rows and columns of the adjacency matrix until they proved to be identical or all possibilities had been tested. However, with n vertices and m multi-nets, there are $n! \times m!$ permutations to test. Instead, BLEX partitions the incidence matrix by sorting devices and multi-nets by their weights, and then gradually matching the circuits by locating a starting vertex and growing from there. It matches pins and vertices adjacent to the subcircuit and gradually exploring the matrix until a match has been found.

2.7.3 Discussion

The disadvantage of all syntactic techniques is that they can identify only the set of specific implementations of a functional component that are contained in its library. Nonstandard or intentionally obfuscated implementations will never be recognized. Furthermore, any optimization that modifies the implementation of the entity (such as optimizations for *don't care* conditions) renders the entity unfit for recognition by structural techniques. Structural matching cannot reliably recognize all functional components that exist in a circuit.

Although syntactic techniques are useful in applications such as converting a transistor-level netlist into a gate-level netlist, techniques that rely on exact structural matching have limited application to higher levels of design, because high-level functional components generally have many valid implementations. The execution and memory costs associated with maintaining a library of many implementations for each module are not reasonable, so this method is not general enough to reliably solve the Module Identification problem.

Any solution to the general Module Identification Problem should include the use of structural matching to recognize standard implementations of functional components within the circuit. When the technique of syntactic matching is applied first, the effective complexity of a circuit can be significantly reduced before more complex approaches are applied.

2.8 Semantic Matching

In contrast to syntactic matching, semantic matching bases equivalence on functionality rather than structure. Two subcircuits may have completely different structures, yet perform identical functions. This is a powerful matching technique.

A general solution to the Subcircuit Identification Problem requires the identification of high-level components that are more complex then those dealt with in Boolean matching but that lack the input/output correspondences between the logic design and the library components that verification techniques require. Because the functionality of the high-level component may be represented in any number of structural forms, it is necessary to identify the subcircuit by proving semantic equivalence [43].

Semantic matching provides a structurally independent solution to the problem of identifying meaningful subcircuits. By using a semantic technique, it is possible to identify subcircuits that are equivalent to a high-level component in many situations for which syntactic techniques fail. The structural changes imposed by new implementations, design optimizations for area and power, and many other complicating factors cause purely syntactic techniques to fail, but they are amenable to semantic matching techniques. Details about the semantic matching approach are presented in [36, 38].

2.8.1 Functional Canonical Form

Canonical representation is a technique that ensures that equivalent instances of a circuit are represented identically. A canonical semantic representation of a circuit

describes the circuit's functionality in a set of logical formulas. Any functionally equivalent implementation of that circuit produces an identical set of logical formulas.

Semantic matching can be reduced to a two-step algorithm if canonical representations of the circuits are used:

- Convert each of the circuits to be compared to a canonical form.
- Compare the canonical forms of the circuits.

The difficulty, of course, is developing a canonical representation. Binary Decision Diagrams are the most effective option [32, 115], but they still require that input and output names of the circuits are matched, which requires trying all permutations of the input and output ordering.

Binary Decision Diagrams

Since their initial presentation in 1986 [19], Binary Decision Diagrams (BDDs) have become a widely used representation of Boolean functions. The advantage of BDDs is that they can be manipulated and generated by efficient graph algorithms. A variant of BDDs, Reduced Ordered Binary Decision Diagrams (ROBDDs) [20] can represent Boolean functions in canonical form. A further extension of ROBDDs, partitioned ROBDDs, can be exponentially more compact than ROBBDs and are more easily manipulated [85, 86].

BDDs represent functions as directed, acyclic graphs, in which each nonterminal vertex is labeled by a function variable. Each vertex has two outgoing edges, one corresponding to the case in which the variable evaluates to 0 (generally represented

by a dashed line), the other corresponding to the case in which the variable evaluates to 1 (generally represented by a solid line). The terminal vertices in the graph are labeled either 0 or 1, representing the value of the entire function. For any variable assignment, the function value can be determined by following a path through the graph, following the appropriate branch for the value of the variable.

Each decision variable must appear at most once on a path from the root vertex to a terminal vertex. The order in which the variables are visited is determined by a total ordering on the variables. When this total ordering is applied, a BDD becomes an ordered binary decision diagram (OBDD). By eliminating and sharing isomorphic subgraphs within the OBDD, the BDD may be reduced to a compact form, a reduced ordered binary decision diagram (ROBDD), and can present a canonical form of the function.

The difficulty in deriving this canonical form is the ordering of the decision variables. Two identical functions will present two different ROBDDs if their variables are given different orderings. So, when attempting to use this canonical representation to compare the functionality of two circuits, it is necessary to determine the correct variable ordering for each so that the OBDD representations of identical circuits are also identical.

ROBDDs are a valuable tool for function comparison. The representation is compact for the majority of functions that appear in logic circuits, and manipulation and comparison is simple and fast. The focus, therefore, must be on reducing the variable ordering permutations that must be tested to determine equivalence.

The difficulty when attempting to determine equivalence of two ROBDDs is that

the ordering of the variables can result in vastly different ROBDDs. To reduce the situations in which all ordering permutations must be tested, it is useful to immediately disregard functions that cannot be equivalent. This non-equivalence can be determined by deriving and comparing Boolean signatures [36, 79].

Boolean Signatures

A signature of a Boolean function is a representation of a property of the function [36]. Signature functions are applied to a function and return a characteristic signature for that function. The signature function must be dependent on just the behavior of the function; variable orders and labels may not be considered by the signature function.

Functions with equivalent signatures are said to belong to the same *signature class.* For two functions to be equivalent, they must share a signature class. Typical signature functions are designed to be very quick to return signatures, so they may be efficiently applied to reduce the possible correspondences that must be tested.

2.8.2 Cones of Logic

Rather than performing multiple-output semantic matching, outputs may be considered individually by building a *logic cone* for each output of a subcircuit [32]. This technique has also been applied in partitioning techniques to improve the mapping of partitions to FPGAs [18].

A logic cone is built by starting at one of the outputs of the subcircuit and adding gates that are inputs to gates already within the cone until the primary inputs to the subcircuit are reached. This cone will contain all of the gates that contribute to the

determination of that output value.

The current technique to solve the Module Identification Problem with logic cones involves locating a single output of a known library module (the *target output*). Logic cones are built from gates within the circuit. Each of these cones is semantically compared to the function of the target output.

Expansion stops when the logic cone is semantically equivalent to the function of the target output or the primary inputs to the circuit have been reached. Once the target output has been located in the circuit, an attempt is made to expand the cone to locate the functionality to produce the other outputs of the known module.

The advantage of this technique is that single-output semantic matching is less intensive than matching with multiple-outputs. The disadvantage is that the number of matches that must be performed is larger. The candidate subcircuit enumeration technique presented in Chapter 3 can enumerate just these logic cones, so logic cone matching can be performed within module identification by providing an appropriate output functionality library.

2.8.3 Hierarchical Semantic Matching

In an approach to regularity extraction, Chowdhary *et al* [33] perform tree subgraph matching. Each tree, or template, is stored as a root node and a set of templates that represent its children. To view an entire tree, the template for each child may be expanded.

The technique simultaneously builds a list of templates, and it is against these

that the expanding subtrees are matched. The result is a list of logic functions. For each logic function, there is a list of the subcircuits of the circuit that perform that function. The details of the logic function matching were not discussed in [33].

The obvious drawback to this technique is that it operates on only a small subset of the subcircuits, subtrees. Subtrees represent single output functions with no reconvergent fan-in. Some extensions for multiple outputs have been researched, but the focus is primarily on single-output functions.

2.8.4 Pseudo-Semantic Matching

Methods that fall between the strict definitions of syntactic and semantic matching are permutation matching and rule-based structure manipulation.

The approaches that have applied permutation matching effectively have operated exclusively on transistor-level netlists, which have a simpler structure and fewer reorganization options than gate-level netlists. For instance, YNCC [100] uses permutation to iteratively gather information about possible bindings until a match can be located.

A circuit comparison system presented by Takashima *et al* [106] uses a rule-based system to identify small subcircuits corresponding to any of the implementations that occur in the rule base. That subcircuit may then be replaced by a block or element representing the appropriate module.

There are several systems that perform "functional matching" of components but rely upon an extensive library and syntactic techniques to perform matching [93], or

use rules or permutations to manipulate simple structures until a match is found [100, 106]. These systems are useful and efficient in circuits for which high-level modules are known or can be easily identified, but they do not perform true semantic matching.

2.8.5 Discussion

The techniques discussed in this section are approaches to performing the most difficult type of matching: semantic matching. For module identification, it is much more powerful than syntactic matching because it can locate all instances of a functional module, regardless of implementation. However, it is a more complex technique, requiring more computation than syntactic matching. The goal of the research in this thesis is to reduce the number of applications of semantic matching to make the Module Identification Problem more tractable.

2.9 Software Reverse Engineering

Unlike in hardware, in which a digital design is synthesized, implemented, and manufactured, then left static until it is replaced, software is often subject to constant extension and modification. In fact, the majority of effort applied to software development is not in creation and implementation, but in the extension and modification of existing software systems [112]. This is not surprising, because the reengineering of software systems is frequently driven by outside influences. As the available technology of digital hardware improves and drives replacement of the underlying computer systems, the software must be modified to operate on the new platform. The ac-

cessibility of software leads to greater user expectations with regard to functionality, performance, and ease of use.

Legacy systems exist in the software domain as well as the hardware domain, and their reverse engineering is desirable for the same reasons: complete replacement is too expensive or infeasible due to complex features and the lack of accurate original specifications. In some cases, a "wrapper" may be developed that runs on the new hardware platform, but interfaces with the legacy system on its original platform. An example of this is developing a front-end client application to allow access to a mainframe software system. This solution is safe because no change has been made to the underlying functionality, and can be implemented cheaply and quickly. The disadvantage is that the system does not take advantage of the performance improvements gained by the new platform [90].

Support and evolution of systems programmed in older languages like COBOL are suffering from a lack of knowledgeable software engineers to upgrade, modify, and troubleshoot these systems. Transforming useful systems that are implemented in a obsolete language into more current language not only increases the chances that they can be modified as needs change, but also allows them to be more easily ported to newer hardware and operating platforms.

A compromise between replacement of the legacy system and continuing to use obsolete technology is to reengineer the system by extracting and reusing critical portions of the code. In this way, complex business rules, for instance, can be ported "as is", without requiring an in-depth understanding of their operation. This technique of reengineering incorporates functional module identification, also referred to

as *reusable component recovery* [89].

2.9.1 Functional Module Identification

Just as in reverse engineering of hardware, one of the focuses of software reverse engineering is the identification of logical blocks of code that perform a well-defined and potential useful piece of functionality. Once these modules have been identified, they may be reused or redesigned.

Modularization

A functional module identification technique called *modularization* [80] was developed by a team of engineers at Boeing to improve the maintainability of large payroll systems implemented in COBOL. A large compilation unit, the *source*, may be broken up into a collection of smaller units *targets*, each containing functionally related subroutines. In effect, this develops an object-oriented realization of the program, and the targets represent self-contained libraries of subroutines that are more easily modified and may be easily applied within other systems.

The critical tasks of this modularization process include:

- *Splitting*: the partitioning of the procedure source into the targets according to a plan designed by a project leader. After modularization, the source contains calls to the subroutines in the targets.

- *Linkage determination*: the determination of how the targets interact with each other, as well as the proper handling of their data elements.

The splitting step is the most similar to the module identification problem presented in this thesis. The library modules correspond to the targets defined by the project leader. The source must be searched to locate potential subroutines and determine their proper target placement. This functional identification can be determined by analyzing the control flow graph, call graphs (which represent the subroutine interaction and parameter information), and abstract syntax trees.

Concept Assignment

The *concept assignment problem* [16] is a formulation of the transformation between levels of abstraction in software. It is the study of how to abstract away from variables and keywords to realize a human-level understanding of the software functionality. The goal is to recognize concepts implemented by the portions of the code and relate them to each other to understand the system.

Several techniques are combined to develop an approach to this problem. Syntax-based approaches, which formally consider the structure of the code, are necessary but not sufficient to perform effective concept recovery. These approaches look for patterns of features within the code that can be used to form a signature to identify the function of the code. These approaches may employ a bottom-up approach, locating elemental concepts and gradually subsuming them within coarser grained concepts.

The Concept Assignment Problem bears a strong resemblance to the Module Identification Problem presented in this thesis. It uses a semantic module identification approach to identify elemental concepts within the code and then abstracts away from

the details by identifying larger modules that include those simple concepts.

Formal Methods

Formal methods provide techniques for specifying software by representing the software by a well-defined specification language [56]. Inference rules can be applied to this specification to provide correctness and consistency, and also to extract information about the software functionality. By defining and applying formal transformations to the source code, the code may be transformed into a higher-level description of the code, hiding the programming details to reveal the intent of the program.

2.9.2 Discussion

The conceptual approach to reverse engineering of software and hardware is very similar. Both involve the abstraction away from the details (e.g., low-level netlist, source code) to derive a higher-level understanding of the system. However, the difficulty of software reverse engineering surpasses hardware engineering due to the difficulty of accurately representing the concepts implemented in software. Formal methods may be applied to develop descriptions of the concepts, but the syntax of software is not as rigidly defined as is that of hardware, which complicates the reduction and transformation into a syntactic or semantic representation that may be applied within module identification.

Reverse engineering in both the software and hardware domains applies syntactic and semantic matching techniques to perform transformation between levels of abstraction. Syntactic approaches in software involve parsing of source code to extract

patterns for comparison and identification. In hardware, syntactic approaches involve pattern matching of structures in a netlist representation of the circuit.

However, although the goals and abstract approaches of reverse engineering of hardware and software reverse engineering are nearly identical, the methods of performing tasks toward these goals must be guided by the environment in which they are applied. In software, a semantic technique capable of comparing complex software routines is necessary, possibly by representation in a theoretical framework, exhaustive simulation, or bottom-up functional extraction. Hardware can more easily be represented by a canonical functional representation, such as ROBDDs or a logical formula.

Therefore, the focus of reverse engineering in these two fields should concentrate on solving the problems specific to the domain. This dissertation presents techniques for approaching the two parts of the Module Identification Problem with regard to hardware: candidate subcircuit enumeration (Chapter 3) and semantic matching (Section 2.8).

2.10 Summary

This chapter has presented background information on reverse engineering and current approaches to reverse engineering. In addition, an overview was presented of areas in which the research presented here may also be applied. Each of these application areas can be reduced to the same problem: the Module Identification Problem. Due to the disparity of the application areas, many different approaches have been

proposed for the solution of this problem. Those techniques have been categorized and presented, along with their relation to the solutions proposed in this dissertation.

The next chapter, Chapter 3, presents a solution to the first half of the Module Identification Problem: Candidate Subcircuit Enumeration.

Chapter 3

Candidate Subcircuit Enumeration

Raising the level of abstraction of a circuit specification by identifying all of the high-level modules within the circuit requires that all of the *potential* high-level modules within the circuit are identified. This is a challenging task because a section of the circuit that functionally corresponds to a library module will not necessarily appear to be similar in structure, as measured by size, order, gate types, connectivity ratio, or physical layout. Additionally, a cluster of gates that is functionally "close to" being equivalent to a library module may not exhibit any traits that would indicate this near-equivalence. This lack of a relation between structural features and functionality impedes any attempt to guide the search for library modules with simple heuristics.

A straightforward approach to the Module Identification Problem enumerates all of the gate clusters within the circuit, then checks each for functional equivalence to known functional modules. This chapter presents a technique to enumerate all *interesting* gate clusters to ensure that all library modules can be located within the target circuit. Interesting gate clusters are those that perform a well-defined function;

enumerating only these gate clusters reduces the number of clusters to be enumerated.

The netlist of the target circuit can be represented by a directed graph; subgraphs of this graph represent the gate clusters. Each vertex represents a gate, and each arc represents a wire. The gate clusters are represented by *induced, connected, subgraphs.* Within the context of the Module Identification Problem, each of the gates within a gate cluster must be reachable from every other gate. Put simply, the gate cluster is represented by a connected subgraph. This is a reasonable restriction on the gate clusters to be enumerated because each library module is represented by a single subcircuit, not several disjoint subcircuits (discussed in Section 1.4). Further, the subgraphs are induced because all of the arcs connecting the vertices in a subgraph are also considered to be included within that subgraph. All subgraphs referred to within this thesis are assumed to be induced and connected.

In a completely connected graph [1] of order n, there are n cliques [2] of order 1, C_2^n cliques of order 2, C_3^n cliques of order 3, and so on. Therefore, the formula $\sum_{i=1}^{n} C_i^n = 2^n$ can be derived to describe the number of (not necessarily disjoint) subgraphs, where n is the order of the graph. Digital circuits are seldom (if ever) completely connected, so the number of subgraphs in a circuit graph will be significantly smaller. However, the above formula does give an upper bound and succinctly characterizes the enormity of this problem.

[1] a graph in which there is an edge from every vertex to every other vertex
[2] a subgraph in which there is an edge from every vertex to every other vertex

3.0.1 Uniqueness

When developing subgraphs for enumeration, it is possible to derive a single subgraph in several different ways, depending on the order in which the vertices are grouped together. This results in unnecessary duplication of effort. When applied within the Module Identification Problem, not only is extraneous computation expended to produce duplicate subgraphs, but also in unnecessary applications of Subcircuit Identification or the detection and pruning of duplicate subgraphs. The enumeration technique presented in this chapter guarantees that each subgraph is produced exactly once.

3.0.2 Focused Enumeration

Subcircuits

Not all of the subgraphs represent interesting gate clusters within the target circuit. It is important that the subgraph represent a functional *subcircuit* of the circuit. No gate in the subcircuit may be accompanied by some but not all of its inputs; that situation leaves the functionality of the gate undefined. For example, a 2-input AND gate in a subcircuit must be accompanied in the subcircuit by both of its inputs. A subgraph containing a gate with undefined functionality does not have functional meaning, so the subgraph does not represent a meaningful subcircuit.

In a subcircuit, any vertex with an input arc connecting it to another vertex in the subcircuit may not have an input arc connecting it to a vertex outside of the subcircuit. In graph theory terminology, no vertex in the subgraph may be

accompanied in the subgraph by a proper subset of its parents.

Contained Subcircuits

To further optimize performance of Module Identification, a subset of the subcircuits, *contained subcircuits*, may be enumerated. A contained subcircuit is the most likely subgraph type to match a library module because its logic is entirely self-contained. Circuit designers often use a library of modules, connecting them together with glue logic to create the final circuit. Each of the modules in a circuit designed in this manner will be represented by a contained subcircuit.

For example, each of the 1-bit full adders that comprise the 2-bit full adder in Figure 1.1 on page 16 are contained. No logic is shared with other modules. One of the outputs from the first adder is an input to the second adder, but the two adders themselves are disjoint.

In a contained subcircuit, any vertex with an arc that exits the subcircuit may not have another arc that is an input to a vertex in the subcircuit. All outputs from a contained subcircuit are primary outputs of that subcircuit. In graph theory terminology, no vertex in the subgraph may be accompanied in the subgraph by a proper subset of its children vertices.

It is not sufficient to enumerate only the contained subcircuits of the target circuit, but contained subcircuit enumeration can be performed much more quickly than subcircuit enumeration. Therefore, it can be applied as a preliminary module identification step to reduce the complexity of the target circuit, thus improving the overall performance of module identification.

Candidate Subcircuits

With regard to the Module Identification Problem, a *candidate subcircuit* may refer to either a subcircuit or a contained subcircuit, depending on the application. The candidate subcircuits enumerated by the technique presented in this chapter must be subcircuits. However, by applying an additional restriction, only contained subcircuits may be enumerated as candidate subcircuits. Enumerating and identifying only the contained subcircuits is useful as an initial technique for reducing the complexity of the target circuit. For a more detailed description of subcircuits and contained subcircuits, see Section 3.1.1.

3.0.3 Overview of Technique

The naïve solution to candidate subcircuit enumeration enumerates all subgraphs of the graph representing the circuit, then simply discard all duplicate subgraphs and subgraphs that are not subcircuits. A more efficient technique enumerates each subgraph exactly once. However, the number of subgraphs in an arbitrary circuit can be prohibitive ($\mathcal{O}(2^n)$ where n is the number of primitive devices in the circuit netlist), so a more restrictive technique is necessary. The focused enumeration algorithm presented and discussed in Section 3.2.3 generates unique candidate subcircuits only, eliminating excess computation effort spent on the enumeration of unnecessary subgraphs. This algorithm also enumerates each candidate subcircuit exactly once, guaranteeing that all interesting subcircuits are available for examination while wasting as little effort as possible.

Although the effort to generate only the subcircuits or contained subcircuits requires some overhead, the benefit of solely enumerating these subsets far outweighs this cost. For example, there are 98,922 unique subgraphs in a 3-bit adder, but only 522 subcircuits, and only 66 contained subcircuits. Only the *subcircuits* need to be considered for equivalence to a known module during module identification, so focusing the enumeration effort on this class of subgraphs provides significant reduction of computational effort. The improvement gained by enumerating just these subsets of the possible subgraphs is presented in Section 3.3.

Generating all of the subgraphs of a graph cannot be accomplished in polynomial time because the number of subgraphs within an arbitrary directed graph is exponential, as described on page 3. However, the algorithms presented here enumerate only a specific subset of the subgraphs, and can generate all of these candidate subcircuits quickly for small circuits (< 100 gates) by taking advantage of the fact that the target circuit is represented by a labeled, directed, graph. This information can be used to focus the efforts on subcircuits that are likely to match library modules.

3.0.4 Heuristics

Due to the inherent difficulty of subgraph enumeration, it is necessary to provide heuristics to allow this technique to be effectively applied to larger circuits (> 100 gates). The most important of these is preliminary partitioning. The number of potential subcircuits may increase exponentially with each additional logic device, so by intelligently reducing the size of the circuit to be explored, significant computational

improvement may be gained.

Additionally, within the specific domain of the Module Identification Problem, it is reasonable to limit the order of the subgraphs that are created. Although a functional module may be realized by any number of logic devices, design constraints such as minimization of area and power consumption make it unlikely that the number of gates in a specific implementation will greatly exceed that of a standard implementation of that module. Therefore, the order of the subcircuits that must be enumerated may be bounded, providing great improvement in the computational complexity of subgraph enumeration.

There are several other computation reducing techniques that may be applied within candidate subcircuit enumeration along with preliminary partitioning and subgraph order limiting. These heuristics are discussed in Chapter 5.

3.0.5 Chapter Outline

Section 3.1 presents background information necessary for understanding the subcircuit enumeration technique presented in this chapter. The technique itself, including the algorithm, explanatory examples, proof of correctness, and complexity analysis, is presented in Section 3.2. Proof of concept results and discussion appear in Section 3.3, and Section 3.4 summarizes and considers the ramifications of this research.

3.1 Background

3.1.1 Representation

The first step in formalizing our approach to the Subcircuit Enumeration Problem is the representation of the circuit as a directed graph. Graph theory is a well-explored area, and by working with graphs rather than circuits, advantage can be taken of previously derived results, algorithms, and tools. These graphs, transformed in a well-defined manner from target circuits, are referred to as *circuit graphs.*

There are several graphical representations of digital circuits, including a method that represents both connections and gates as vertices, resulting in a bipartite graph [41, 42, 91]. A vertex representing a connection contains information about each of its endpoints as well as any additional information that is available, such as length and the physical position of its endpoints. The vertex representing a gate includes a list of the connection vertices that are adjacent to it. An extension of this technique is *multiplace graphs* or *hypergraphs*, in which the edge vertices may connect more than two gate vertices [26, 27, 66, 76]. This method provides flexibility in the implementation of procedures on the converted graph. For the purpose of subcircuit enumeration a simpler type of circuit graph is sufficient.

The circuit graphs used in this thesis represent connections as arcs and gates as vertices, resulting in a directed graph [33, 58, 71, 81, 95]. In the elementary case, an arc begins at the vertex that has a lower depth (distance from the circuit inputs) and terminates at the vertex that has a higher depth. When sequential logic is encountered, arcs can connect vertices in either direction. Primary inputs to the

circuit are represented as vertices as well. Primary outputs are considered to be represented by the logic gate that produces them.

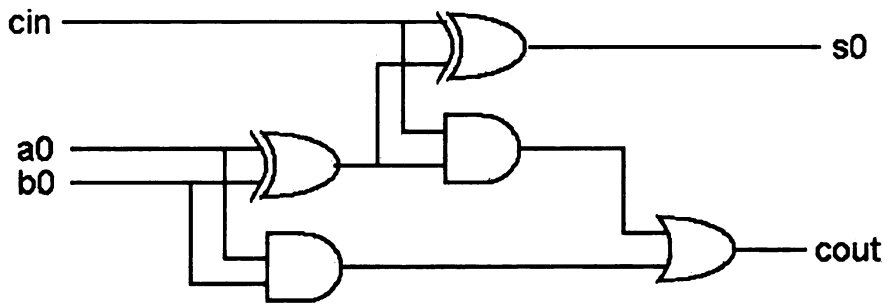
A 1-bit adder is shown in Figure 3.1a. To transform this circuit into a circuit graph, each gate and circuit input is replaced by a vertex. Wires connecting gate inputs to gates are replaced by arcs pointing from the vertex representing the input to the vertex representing the gate. The transformed circuit is presented as the circuit graph in Figure 3.1b. Dashed vertices represent input and output vertices.

Definition 3.1 *A* **circuit graph** *G representing a digital circuit C is a directed graph with $V(G)$ containing a vertex for each logic gate in G. $E(G)$ contains an arc for each connection within C. A* **parent** *of a vertex v is a vertex that represents an input of the gate represented by v. The* **child** *of a vertex is a vertex that represents an output of the gate represented by v.*

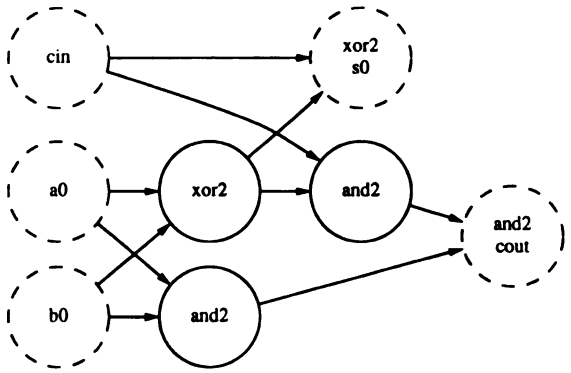
To facilitate the development and application of rules and algorithms applied to the graph, a unique numerical *index* is applied to each vertex in the circuit graph. Similarly, each subgraph also has an index associated with it. By exploiting the uniqueness of the index, a total ordering is defined for the set of vertices. This ordering is the key to the technique to enumerate each subgraph exactly once.

Definition 3.2 *The* **index** *of a vertex v, $v.index$, is a unique integer assigned to a vertex. The* **index** *of a subgraph H, $H.index$, is equivalent to the highest index of its constituent vertices.*

An induced, connected subgraph H may be enumerated by creating a trivial subgraph H' containing one of H's vertices and iteratively adding a vertex adjacent to



(a)



(b)

Figure 3.1: Circuit to Circuit Graph Transformation Example

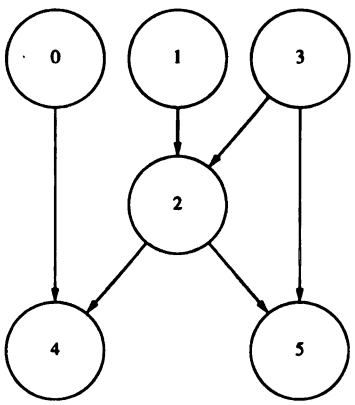
H' until H' is identical to H. The order of the vertices added to create the subgraph is known as its *creation path.* Each vertex in H occupies a specific position within the creation path. Theorem 3.3 proves that when each vertex has a unique integer index and the rules described in Section 3.2.1 are applied, there is exactly one creation path for each subgraph.

Definition 3.3 *The* **creation path** $P(H)$ *of a subgraph* H *of order* n *is a list of the vertices of* H *in the order in which they were added to create* H*. It consists of the vertices* $P_0, P_1, ..., P_{n-1}$*, where* P_0 *is the* **initial vertex** *of* H *and* P_{n-1} *is the* **final vertex** *of* H*. The intermediate subgraphs are denoted* H_x*, where* x *is the position within the creation path of the most recently added vertex.*

For a given subgraph H, the vertices that are adjacent to H are the vertices that may be considered for addition to the subgraph during subgraph expansion. Each of these vertices must be no more than one arc away from some vertex within H. Formally, these vertices are referred to as the subgraph's *neighborhood.*

Definition 3.4 *The* **neighborhood** $N(H)$ *of a subgraph* H *within a graph* G *consists of all vertices of* G *adjacent to at least one member of* H*. Formally,* $N(H) = \{v \mid v \notin V(H) \wedge \exists u \in V(G)$ *such that* $uv \in E(G) \vee vu \in E(G)\}$.

We will be using a simple circuit graph G (Figure 3.2) to clarify the definitions, theorems, and rules presented here. Vertices with a bold outline are constituent vertices of the subgraph being displayed and dashed vertices represent vertices available for expansion; all other vertices and edges represent vertices of the underlying circuit



(a)

Figure 3.2: Circuit Graph G

graph and are not part of the subgraph (e.g., Figure 3.3). The number in the vertex is the index of the vertex, which for simplicity will also serve as its name. If a vertex belongs to the subgraph, its position x on the creation path is displayed under the vertex name as PX.

Focused Enumeration

The naïve solution to the subcircuit enumeration problem is subject to exponential explosion depending on the connectivity of the graph. However, the complexity of the enumeration can be significantly reduced by exploiting the fact that only some subgraphs are of interest in the domain of reverse engineering. Consider a subgraph H composed of a gate vertex v and two vertices representing two of its three inputs (either gates or primary inputs). The vertex v does not represent a complete gate in the context of the subcircuit represented by the subgraph H because it is missing an input. The functionality that it performs is therefore undefined because it is not fully

specified.

For the purposes of module identification the enumeration expense is greatly reduced by enumerating only those subgraphs that exclusively contain vertices representing fully specified gates. These subgraphs are referred to as *subcircuits.* This definition has also been independently developed and presented as *feasible subgraphs* [33].

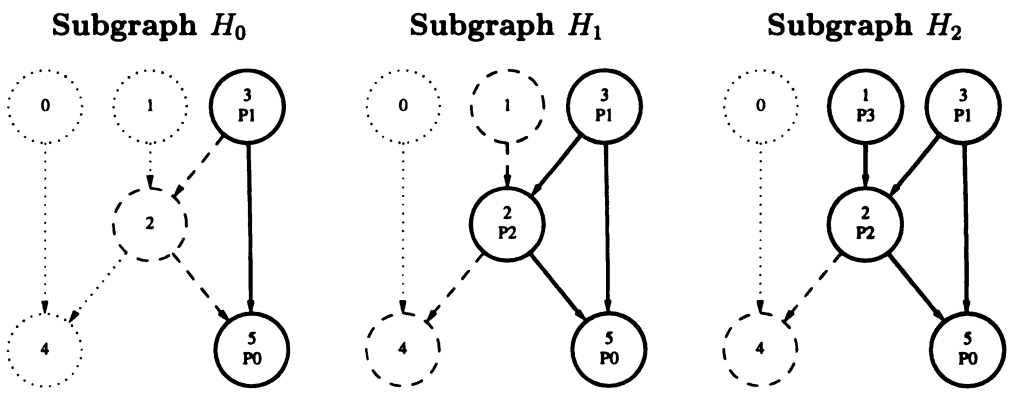
A *fully specified vertex* represents a gate that is joined within the subgraph by either *all* of the vertices representing its inputs or *none* of those vertices.

Definition 3.5 *In a subgraph H of a circuit graph G, a vertex $v \in V(H)$ is a* **fully specified vertex** *if* $(\forall u \mid u \in V(G) \wedge uv \in E(G) \rightarrow u \in V(H)) \vee (\forall u \mid u \in V(G) \wedge uv \in E(G) \rightarrow u \notin V(H))$.

Definition 3.6 *A subgraph H of a circuit graph G is a* **subcircuit** *of G if and only if it is connected and each vertex in H is fully specified.*

In Figure 3.3, the tables below the subgraphs present lists of the vertices in the subgraph (V), and the neighborhood of the subgraph (N). In Subgraph H_0, Vertex 5 is not fully specified because one of its two inputs, Vertex 2, is not a member of H_0, so H_0 is not a subcircuit. In Subgraph H_1, Vertex 5 is fully specified by the addition of Vertex 2. However, Vertex 2 is not fully specified, so H_1 is not a subcircuit. In Subgraph H_2, the addition of Vertex 1 has made all four vertices in $V(H_2)$ fully specified, so H_2 is a subcircuit of the circuit graph G.

A further refinement of this process takes into consideration the fact that most hardware is designed using CAD synthesis tools that utilize a library of ready-made



H_0 Properties	
V	5, 3
N	2

(a)

H_1 Properties	
V	5, 3, 2
N	1, 4

(b)

H_2 Properties	
V	5, 3, 2, 1
N	4

(c)

Figure 3.3: Examples of Subcircuits and Fully Specified Vertices

modules. To reduce the synthesis effort, these modules are often simply connected together to provide the desired functionality. In these cases, the gate clusters representing these modules will be completely contained, with no arcs leaving or entering the subgraph except for the primary inputs and outputs to the module. The vertices in these subgraphs represent gates with fully-specified inputs and contained outputs.

A *contained vertex* represents a gate that is fully specified and joined within the subgraph by either *all* of the vertices representing its outputs or *none* of those vertices.

Definition 3.7 *In a subgraph H of a circuit graph G, a vertex $v \in V(H)$ is a* **contained vertex** *if v is fully specified and $((\forall u \mid u \in V(G) \wedge vu \in E(G) \rightarrow u \in V(H)) \vee (\forall u \mid u \in V(G) \wedge vu \in E(G) \rightarrow u \notin V(H)))$.*

Definition 3.8 *A subcircuit H of a circuit graph G is a* **contained subcircuit** *of G if and only if each vertex in H is contained.*

The first subgraph in Figure 3.4, H_0, containing only Vertex 5, is a contained subcircuit. None of the inputs of Vertex 5 are in the subgraph, so it is fully specified, and it does not have any outputs, so it is contained. The subgraph H_1 is created by adding Vertices 3, 2, and 1 to H_0. It is *not* a contained subcircuit because although all of the vertices are fully specified, Vertex 2 is not a contained vertex because only one of its two children is present in the subgraph. The only contained subgraph of the original circuit graph G is the subgraph H_2, because after Vertex 4 is added to contain Vertex 2, Vertex 4 is not fully-specified, so Vertex 0 must also be added.

There are considerably fewer contained subcircuits than subcircuits. To improve the module identification process, a preliminary search to locate and match only

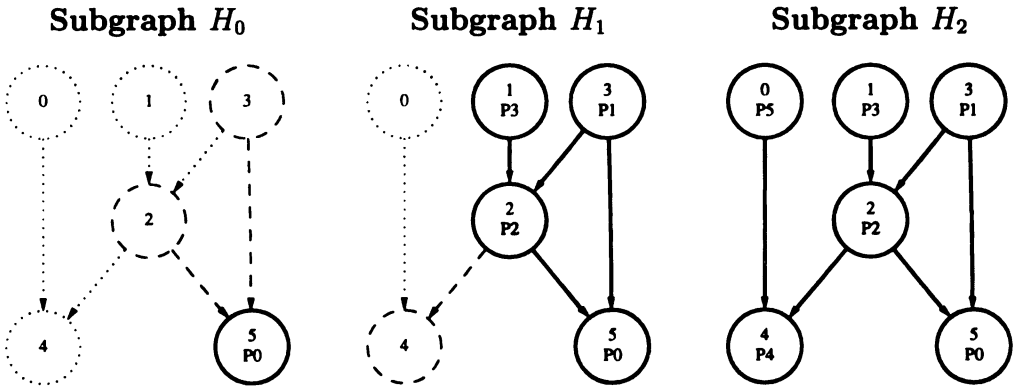


Figure 3.4: Examples of Contained Subcircuits and Vertices

contained modules can provide a considerable reduction in computation, because any vertices within identified modules would no longer be considered for inclusion in another module, thus reducing the order of the circuit graph being explored.

3.2 Enumeration of Candidate Subcircuits

Developing a naïve technique to enumerate the candidate subcircuits is straightforward. By enumerating all of the subgraphs of the circuit graph, all candidate subcircuits must be enumerated. A simple recursive algorithm can accomplish this, but the complexity of such an approach makes it applicable to only the smallest circuits (< 20 gates). It is more difficult to enumerate *only* the candidate subcircuits, but even this task is relatively easy. Following a few simple rules will focus the recursive algorithm on the enumeration of only the candidate subcircuits. A challenging problem is to

generate only the candidate subcircuits and generate each *exactly once.*

The unique generation of each subcircuit is very important. Computation time is wasted by generating duplicate subcircuits. More importantly, this enumeration technique is designed to solve the Module Identification Problem in partnership with a solution to Subcircuit Identification. The current techniques to perform the necessary semantic matching evaluation are computationally intensive. Performing a semantic match between two subcircuits is complex and requires significant processing time. Therefore, by enumerating each subcircuit only once, unnecessary semantic matching can be avoided.

There are three ways to handle the duplicate subcircuit problem within the Module Identification Problem. First, perform semantic matching for each subcircuit that is created. This approach is computationally undesirable because the amount of processing time needed to perform each semantic match is considerable and can significantly impact the overall execution time. Second, maintain a hash table of subcircuits. This approach allows a preliminary check to be performed to determine whether or not the current subcircuit has already been generated and evaluated for equivalence. This approach is highly inefficient because the memory consumption of such a hash table can be prohibitive, and the time to search it for an identical subcircuit can be considerable.

The third approach is to avoid the problem entirely. Generation and enumeration of duplicate subcircuits can be avoided by applying the rules presented in Section 3.2.1. These rules impose a strict ordering on the addition of vertices to subgraphs, thus guaranteeing that each subgraph will be generated once and only once. This is

the solution described in Section 3.2.1.

It is possible to enumerate all subgraphs of a graph by creating a subgraph to contain a single gate of the original graph, recursively adding neighboring vertices until it is equivalent to the original circuit, and then doing the same for every other vertex of the original graph. Each intermediate subgraph would be enumerated by this process. This naïve algorithm for subcircuit enumeration is described in Figure 3.5 and diagrammed in Figure 3.6. With this method there are many paths to the creation of each subgraph, and each will be explored in the procedure. In a completely connected graph, which contains 2^n subgraphs, this algorithm would produce $n!$ subgraphs, though only 2^n of them would be unique. This naïve algorithm provides a framework for refinement. By applying slight modifications to the naïve algorithm, unique focused enumeration of candidate subcircuits may be performed.

The research presented in this chapter provides a technique to enumerate each subgraph of a graph exactly once. A set of well-defined rules that govern the addition of vertices to subgraphs has been developed. Using these rules, it is possible to ensure that there is exactly one path to the creation of each subgraph. To permit the application of these rules, each vertex is assigned a unique integer *index* (Definition 3.2). This index provides an ordering between each pair of vertices in relation to the subgraph being created. This ordering results in exactly one path of vertex addition to the creation of any subgraph and, consequently each subgraph is enumerated once and only once.

In addition, within module identification, this enumeration may be focused on a specific subset of the subgraphs. Only subgraphs with functional meaning are

Naïve Subgraph Enumeration Algorithm

0. Transform circuit into circuit G.
1. **foreach** $v \in V(G)$:
   2. Create a subgraph H_0 with $V(H_0) \leftarrow v$.
   3. Determine $N(H_0)$.
   4. expandSubgraph(H_0).
5. End **foreach**.

subroutine expandSubgraph(graph H_i)

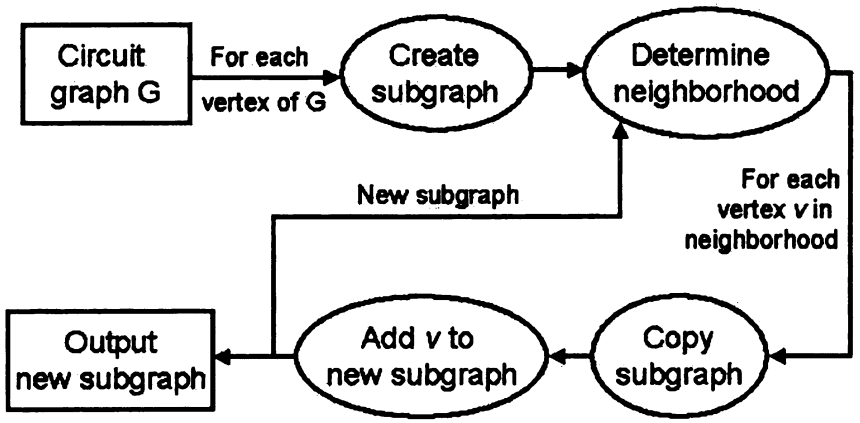
0. **foreach** vertex $v_{v+1} \in N(H_i)$:
   1. Create a subgraph H_{i+1} such that $V(H_{i+1} \leftarrow H_i + v_{i+1}$.
   2. Determine $N(H_{i+1})$.
   3. Output H_{i+1}.
   4. expandSubgraph(H_{i+1}).
5. End **foreach**.

Figure 3.5: Algorithm for Naïve Subgraph Enumeration.

interesting, so only these *subcircuits* (Definition 3.6) are enumerated. The candidate subcircuits enumerated may be further reduced to the subset including only contained subcircuits. The next section presents the definition and rules that are applied to focus the naïve algorithm on unique enumeration of candidate subcircuits.

3.2.1 Rules

The following rules are applied in the algorithm presented in Section 3.2.3 and serve several purposes within the enumeration process. Rule 1 prevents any subgraph from being created more than once. Rule 2 and Rule 3 focus subgraph generation on specific types of subgraphs, subcircuits (Definition 3.6) and contained subcircuits (Definition 3.8), respectively.



(a)

Figure 3.6: Algorithm for Naïve Subgraph Enumeration (Diagram).

Unique Enumeration

To ensure that there is only one creation path for each subgraph created, an order must be defined for each pair of vertices with respect to their addition to a specific subgraph. The naïve algorithm in Figure 3.5 iteratively adds each vertex in the neighborhood of a subgraph H_i to H_i, creating H_{i+1} (Steps 0 and 1 of **expandSubgraph**), then expands H_{i+1}.

The first step toward unique enumeration is recognizing that all of the vertices of the subgraph's neighborhood are not necessarily viable candidates for expansion. A simple step toward eliminating duplicate subgraph generation is to disallow addition of vertices that have an index higher than that of the subgraph. This subset of the neighborhood is referred to as the *frontier* of the subgraph.

Definition 3.9 *The* **frontier** *of a subgraph H, $\mathcal{F}(H)$, consists of all v such that $v \in N(H)$ and $v.index < H.index$.*

In Figures 3.7, 3.8, and 3.9, the final subgraphs are identical, containing vertices [5, 3, 2], but the creation paths are different. In Figure 3.7, the creation path is {3, 2, 5}. In Figure 3.8, the creation path is {5, 2, 3}. By restricting viable expansion vertices to those on the frontier of a subgraph, the former creation path is disallowed.

Restricting expansion of a subgraph H to members of $\mathcal{F}(H)$ does not enforce a single creation path for each subgraph. The creation paths in Figures 3.8 and 3.9 both create the subgraph containing vertices [5, 3, 2] when the frontier is considered. It is therefore necessary to further specify the ordering of vertex addition to ensure that no duplicate creation paths may exist. This is accomplished by limiting the

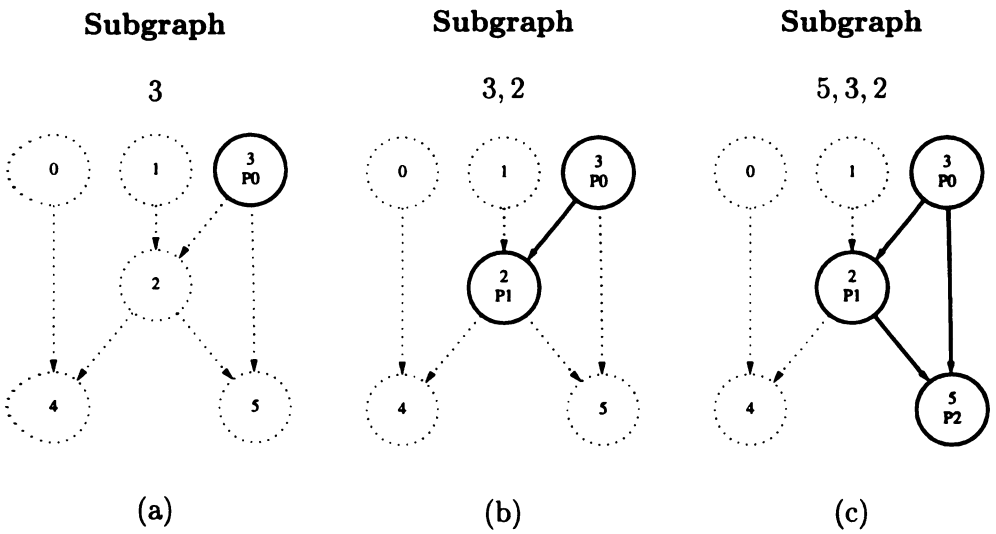


Figure 3.7: Subgraph containing vertices [5, 3, 2] created by $P(H) = \{3, 2, 5\}$.

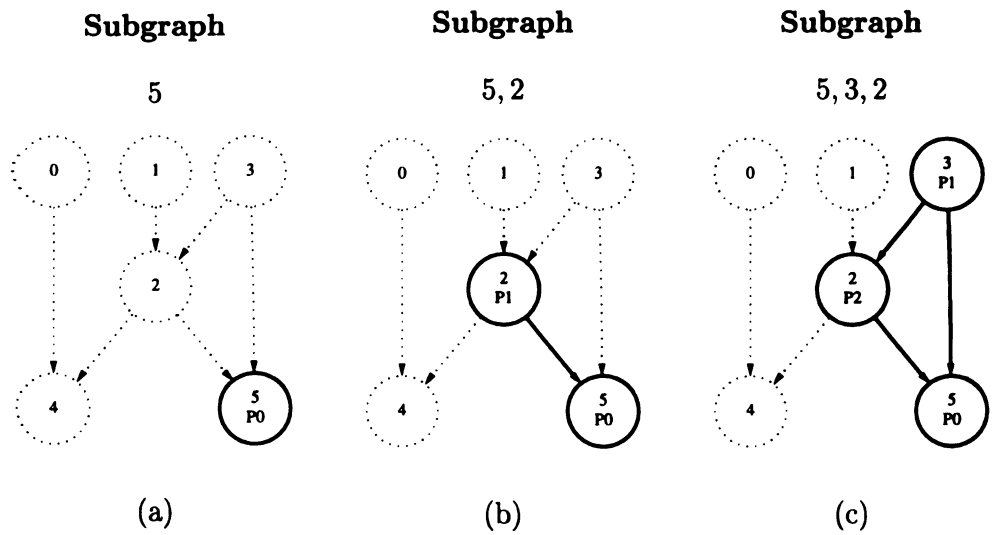


Figure 3.8: Subgraph containing vertices [5, 3, 2] created by $P(H) = \{5, 2, 3\}$.

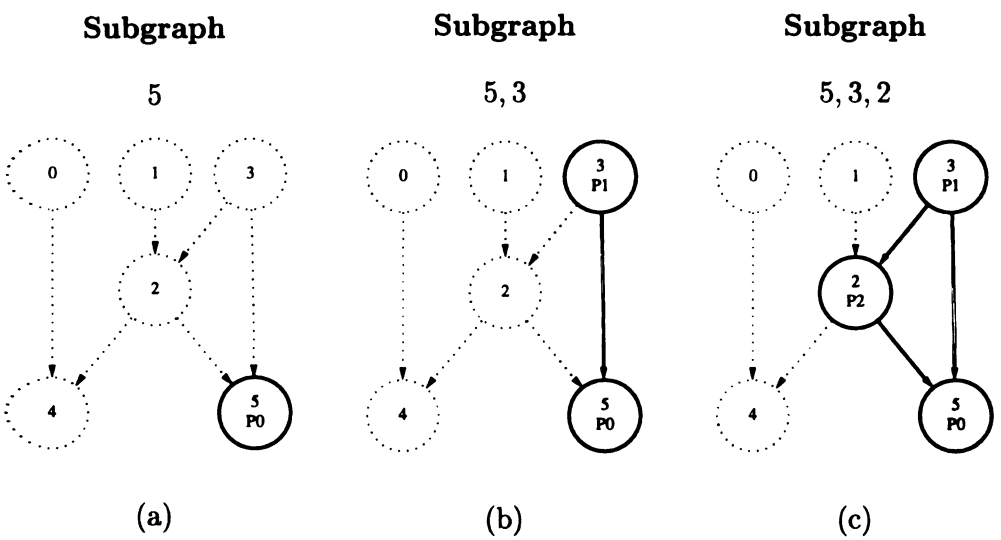


Figure 3.9: Subgraph containing vertices [5, 3, 2] created by $P(H) = \{5, 3, 2\}$.

viable expansion vertices to a subset of the frontier, called the *reachable frontier.*

The reachable frontier of a subgraph H consists of each vertex of the frontier whose addition has not yet been forbidden by the addition of some vertex already on the creation path for H. There are two reasons why a vertex v may be on the reachable frontier. First, it may have just become a member of the frontier due to the addition of an intermediate vertex that put it into the neighborhood of H. Every vertex on the frontier is considered for addition to the reachable frontier exactly once. Second, it may have been on the reachable frontier for a subgraph earlier in the creation path, but has always had an index lower than the vertices added so has not been removed.

When a vertex v is a member of the reachable frontier of a subgraph H_0, it may be added to create a new subgraph H_1. At that point, the reachable frontier of H_1 will contain any vertices that are new to the frontier of H_1, as well as any vertices

from the reachable frontier of H_0 that have an index less than $v.index$. Thus, a local ordering for addition with respect to the subgraph H_1 exists for each pair of vertices.

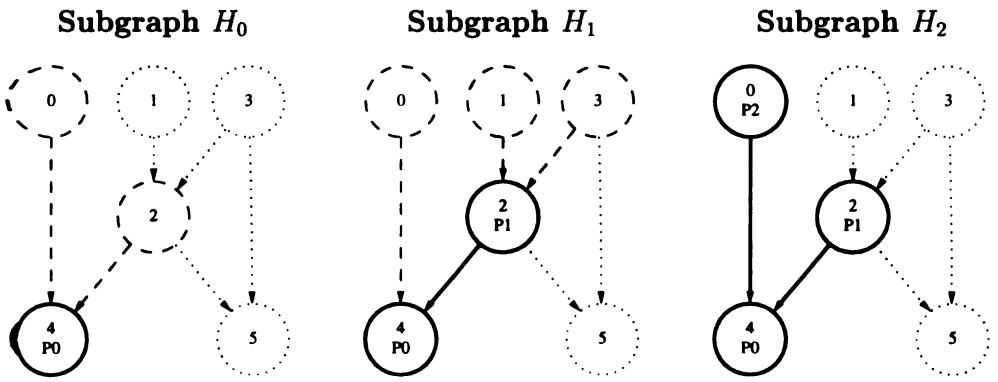
The ordering between vertices is only with respect to the subgraph currently being expanded. The ordering is affected by the order in which vertices are encountered and added to the neighborhood. Until a vertex enters the neighborhood of a subgraph, its ordering with respect to other vertices is not yet defined.

Definition 3.10 The **reachable frontier** of a subgraph H, $\mathcal{F}^R(H)$, consists of all of the vertices v that may be added to H. For a subgraph $H_i = H + P_i$, $\mathcal{F}^R(H_i)$ consists of all v such that:

$v \in \mathcal{F}(H_i)$ and either $(v \notin \mathcal{F}(H_{i-1}))$ or $(v \in \mathcal{F}^R(H_{i-1})$ and $v.index < P_i.index)$.

In Figure 3.10a, the only vertex in the subgraph is Vertex 4. The vertices in the neighborhood are Vertices 0 and 2, and those also comprise the frontier and the reachable frontier of H_0. In H_1, Vertex 2 has been added. All of the vertices in H_1's neighborhood are on the frontier and reachable frontier except Vertex 5, because it violates the $v.index < H.index$ property of the definition of the frontier, because $5.index > H_1.index$. Because it is not a member of the frontier, it cannot be a member of the reachable frontier. When Vertex 0 is added to the subgraph, all of the vertices in $\mathcal{F}^R(H_1)$ with indices greater than the index of Vertex 0 are removed from the reachable frontier and will never again appear on it.

The first rule of vertex addition prevents the creation of any duplicate subgraphs. It requires that any vertex added to a subgraph H must be a member of $\mathcal{F}^R(H)$, the reachable frontier of H.



H_0 Properties	
V	4
N	0, 2
F	0, 2
F^R	0, 2

(a)

H_1 Properties	
V	4, 2
N	0, 1, 3, 5
F	0, 1, 3
F^R	0, 1, 3

(b)

H_2 Properties	
V	4, 2, 0
N	1, 3, 5
F	1, 3
F^R	$\emptyset$

(c)

Figure 3.10: Frontier and Reachable Frontier

Definition 3.11

Rule 1: Only add vertices to H that are members of $\mathcal{F}^R(H)$.

Unique Subgraph Enumeration Algorithm

0. Transform circuit into circuit G with unique vertex indices.
1. **foreach** $v \in V(G)$:
   2. Create a subgraph H_0 with $V(H_0) \leftarrow v$.
   3. Determine $\mathcal{F}^R(H_0)$.
   4. expandSubgraph(H_0).
5. End **foreach**.

subroutine expandSubgraph(graph H_i)
0. **foreach** vertex $v_{v+1} \in \mathcal{F}^R(H_i)$ (Rule 1):
   1. Create a subgraph H_{i+1} such that $V(H_{i+1} \leftarrow V(H_i) + v_{i+1}$.
   2. Determine $\mathcal{F}^R(H_{i+1})$.
   3. Output H_{i+1}.
   4. expandSubgraph(H_{i+1}).
5. End **foreach**.

Figure 3.11: Algorithm for Unique Subgraph Enumeration.

Figure 3.11 shows the algorithm for unique subgraph enumeration, modified from the naïve algorithm in Figure 3.5 on page 103 by the application of Rule 1 in Step 0 of the **subgraphExpand** subroutine. Figure 3.12 presents a graphical representation of the algorithm.

Examples of Rule 1 Application To illustrate the application of Rule 1, consider the example circuit graph G (Figure 3.2).

Example 1 The subgraphs discussed in this example are shown in Figure 3.13.

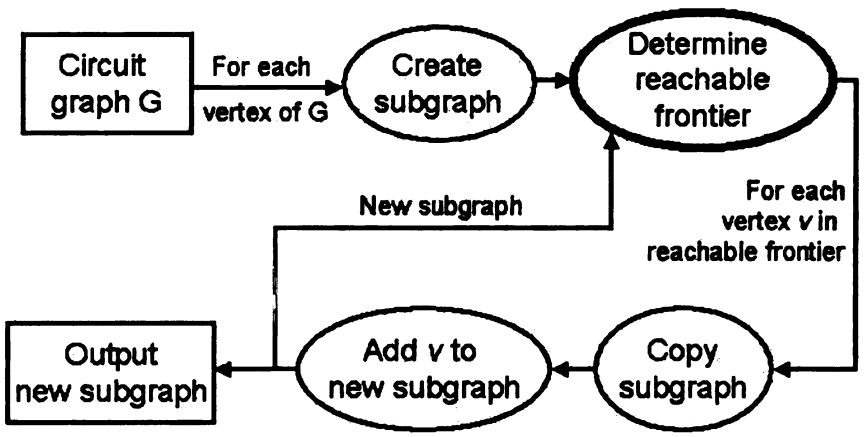
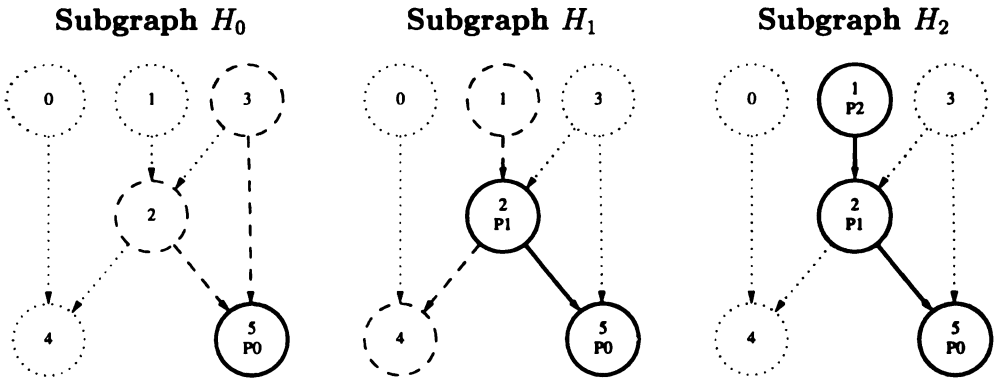


Figure 3.12: Algorithm for Unique Subgraph Enumeration (Diagram).



H_0 Properties	
V	5
N	2, 3
F	2, 3
F^R	2, 3

(a)

H_1 Properties	
V	5, 2
N	1, 3, 4
F	1, 3, 4
F^R	1, 4

(b)

H_2 Properties	
V	5, 2, 1
N	3, 4
F	3, 4
F^R	$\emptyset$

(c)

Figure 3.13: Rule 1 (Example 1).

Subgraph H_0 is composed of a single vertex, Vertex 5. The frontier of H_0, $\mathcal{F}(H_0)$ (Definition 3.9), contains all of the vertices that are no more than one arc away and have an index less than that of the subgraph, in this case, Vertices 2 and 3. The reachable frontier of H_0, $\mathcal{F}^R(H_0)$ (Definition 3.10), is simple to calculate in this case, because there are no previous subgraphs in the path; the reachable frontier is equal to the frontier.

Rule 1 states that any vertex $v \in F^R(H_0)$ may be added. The addition of Vertex 2 to H_0 creates the subgraph H_1. Vertices 1, 3, and 4 are all members of $\mathcal{F}(H_1)$ now, but Vertex 3 is no longer reachable because its index is greater than that of the final vertex of $\mathcal{F}(H_1)$, Vertex 2. The index of Vertex 1 is less than that of Vertex 2, but Vertex 1 is newly reachable (by the addition of Vertex 2), so it is currently reachable. Thus, $\mathcal{F}^R(H_1)$ contains Vertices 1 and 4.

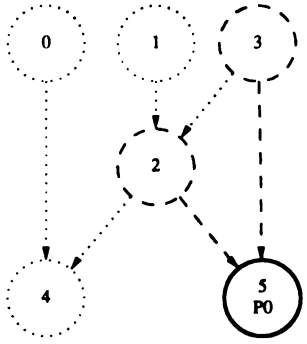
Vertex 1 is added to H_1 to create the subgraph H_3. $\mathcal{F}^R(H_2)$ is empty, so this path has been exhausted. The creation path (Definition 3.3) for this subgraph, $P(H_3)$, is $\{5, 2, 1\}$.

Example 2 The subgraphs discussed in this example are shown in Figure 3.14.

The example above illustrates that there is an ordering between the vertices with respect to the current subgraph. The addition of Vertex 2 before Vertex 3 prevented Vertex 3 from ever appearing in a subgraph based on the expansion of H_1 (Figure 3.13b).

H_1 (Figure 3.14b) is the result of adding Vertex 3 to H_0, instead of Vertex 2. Vertex 2 is the only vertex on the reachable frontier of H_1, so it is added next to

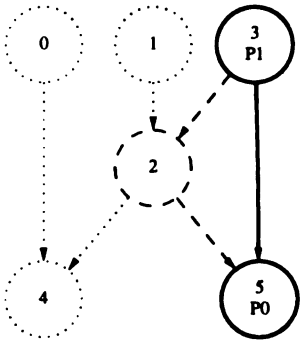
Subgraph H_0



H_0 Properties	
V	5
N	2, 3
F	2, 3
F^R	2, 3

(a)

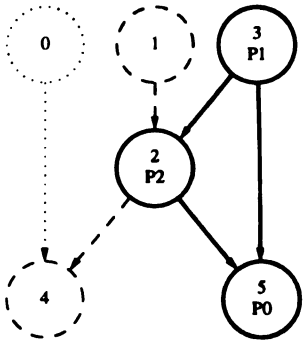
Subgraph H_1



H_1 Properties	
V	5, 3
N	2
F	2
F^R	2

(b)

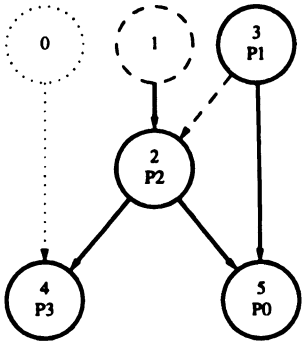
Subgraph H_2



H_2 Properties	
V	5, 3, 2
N	1, 4
F	1, 4
F^R	1, 4

(c)

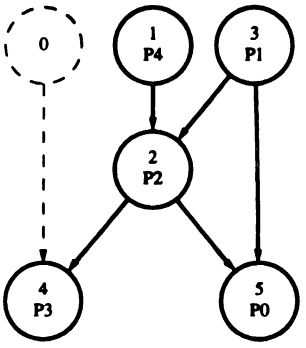
Subgraph H_3



H_3 Properties	
V	5, 3, 2, 4
N	0, 1
F	0, 1
F^R	1

(d)

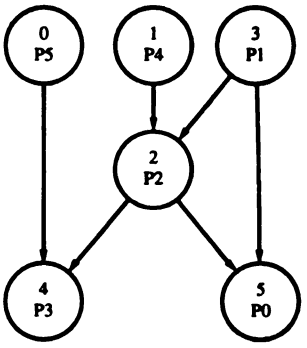
Subgraph H_4



H_4 Properties	
V	5, 3, 2, 4, 1
N	0
F	0
F^R	0

(e)

Subgraph H_5



H_5 Properties	
V	5, 3, 2, 4, 1, 0
N	$\emptyset$
F	$\emptyset$
F^R	$\emptyset$

(f)

Figure 3.14: Rule 1 (Example 2).

create H_2. Either of the two vertices on the reachable frontier of H_2 may be added next. If Vertex 1 is added then the expansion will end immediately because the only remaining neighbor, Vertex 4, would not be reachable. Vertex 4 can be added before Vertex 1 to create H_3, allowing the addition of Vertices 1 and 0 in that order. This effectively creates a subgraph equivalent to the original graph. That creation path, $P(H_5) = \{5, 3, 2, 4, 1, 0\}$, is the only path that will reproduce circuit graph G.

Additional rules are needed to ensure that only subcircuits or contained subcircuits are enumerated. Rule 2 will cause only subcircuits to be produced, while Rule 3 will cause only contained subcircuits to be generated. Note that Rule 3 requires the simultaneous application of Rule 2.

Subcircuits

Rule 2 is designed to ensure that only subcircuits (Definition 3.6) are enumerated. Each vertex in the subgraph is tested to ensure that it is fully specified. If a vertex is only partially specified, then the vertices necessary to make it fully specified are added to the subgraph (if they are members of the reachable frontier).

When all of the vertices are fully-specified, then the subgraph is a subcircuit. If the necessary vertices may not be added because they are not members of the reachable frontier of the subgraph (Rule 1), then the subgraph is discarded.

Definition 3.12

Rule 2: Output only subcircuits

while $(\exists v \mid v \in V(G) \wedge (\exists u \mid uv \in E(G) \wedge u \in V(H)) \wedge (\exists w \mid wv \in E(G) \wedge w \notin$

$V(H)))$

- *If* $w \in \mathcal{F}^R(H)$ *then* $V(H) = V(H) + w$.

- *Else discard* H.

end while

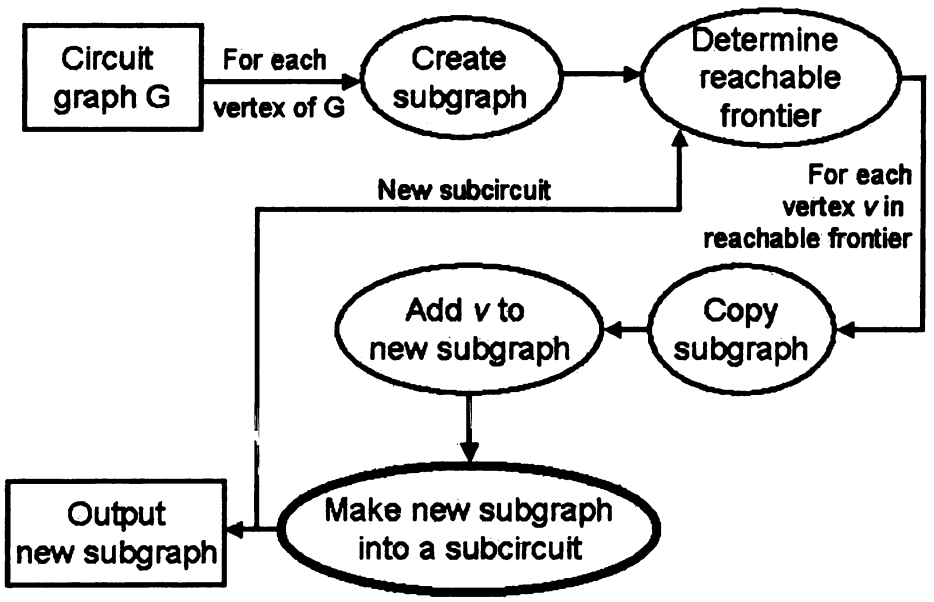
Unique Focused Enumeration Algorithm

0. Transform circuit into circuit G with unique vertex indices.
1. **foreach** $v \in V(G)$:
   2. Create a subgraph H_0 with $V(H_0) \leftarrow v$.
   3. Determine $\mathcal{F}^R(H_0)$.
   4. expandSubgraph(H_0).
5. End **foreach**.

subroutine expandSubgraph(graph H_i**)**
0. **foreach** vertex $v_{v+1} \in \mathcal{F}^R(H_i)$ (Rule 1):
   1. Create a subgraph H_{i+1} such that $V(H_{i+1} \leftarrow V(H_i) + v_{i+1}$.
   2. Determine $\mathcal{F}^R(H_{i+1})$.
   3. Add vertices necessary to make H_{i+1} into a subcircuit (Rule 2).
   4. Output H_{i+1}.
   5. expandSubgraph(H_{i+1}).
6. End **foreach**.

Figure 3.15: Algorithm for Unique Focused Enumeration.

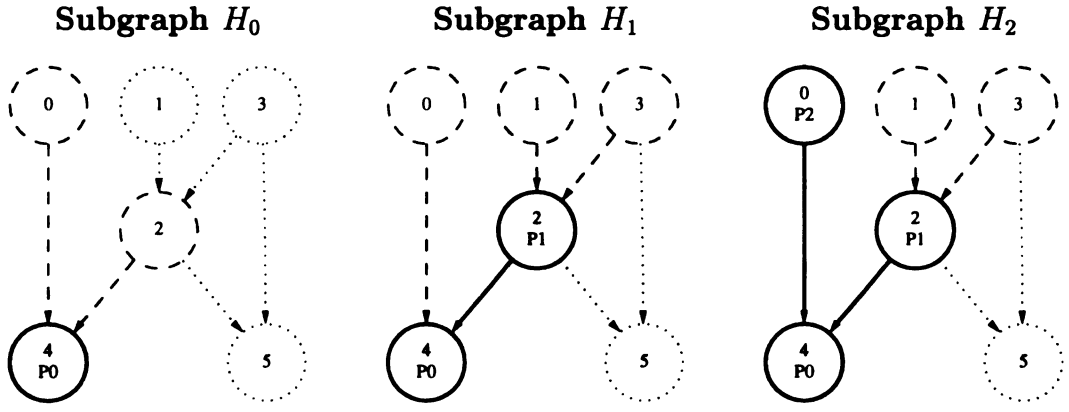
The algorithm in Figure 3.15 presents the unique subgraph enumeration algorithm of Figure 3.11 on page 110 modified by the addition of Rule 2 in Step 3 within the **subgraphExpand** subroutine. A graphical representation of the algorithm is presented in Figure 3.16.



(a)

Figure 3.16: Algorithm for Focused Subgraph Enumeration (Diagram).

Examples of Rule 2 Application To illustrate the application of Rule 2, its application will be demonstrated on the example circuit graph G (Figure 3.2).



H_0 Properties	
V	4
N	0, 2
F	0, 2
F^R	0, 2

(a)

H_1 Properties	
V	4, 2
N	0, 1, 3, 5
F	0, 1, 3
F^R	0, 1, 3

(b)

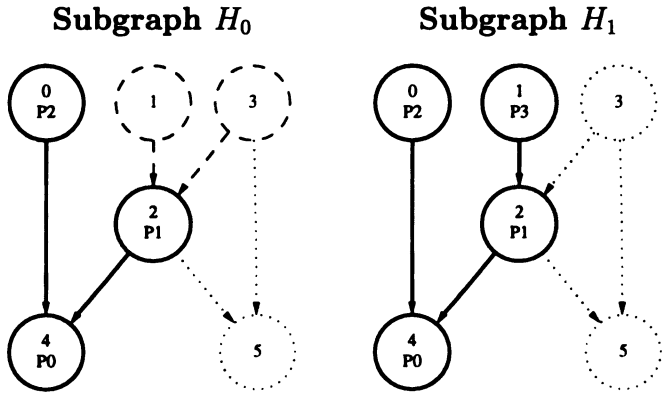
H_2 Properties	
V	4, 2, 0
N	1, 3, 5
F	1, 3
F^R	1, 3

(c)

Figure 3.17: Rule 2 (Example 1).

Example 1 The subgraphs discussed in this example are shown in Figure 3.17.

The trivial subgraph H_0 may be expanded by adding either of its neighbors, Vertex 2 or Vertex 0. The addition of Vertex 2 results in H_1, in which Vertex 4 is not fully specified. Rule 2 requires that any vertices necessary to fully specify each partially specified vertex are added immediately. Therefore, Vertex 0 is added, making Vertex 4 fully-specified and H_2 a subcircuit.



H_0 Properties	
V	4, 2, 0
N	1, 3, 5
F	1, 3
F^R	1, 3

(a)

H_1 Properties	
V	4, 2, 0, 1
N	3, 5
F	3
F^R	$\emptyset$

(b)

Figure 3.18: Rule 2 (Example 2).

Example 2 The subgraphs discussed in this example are shown in Figure 3.18.

Beginning with H_0, either Vertex 1 or Vertex 3 may be added. H_1 is the result of adding Vertex 1. Vertex 2 is only partially specified in H_1 because Vertex 3 is not a member. Rule 2 dictates that Vertex 3 must be added immediately, but it is not on the reachable frontier. Therefore, this subgraph is not a subcircuit and is discarded.

Contained Subcircuits

Rule 3 guarantees that only contained subcircuits (Definition 3.8) are created. It operates similarly to Rule 2, except that as a vertex is added, Rule 2 is applied, then the subcircuit is evaluated to ensure that it is contained. If not, the vertices necessary to fulfill that condition are added to the subgraph, if possible.

When all of the vertices are fully specified and contained, then the subcircuit is marked as contained. If the necessary vertices may not be added because they are not members of the reachable frontier of the subgraph (Rule 1), then the subgraph is discarded.

Definition 3.13

Rule 3: Output only contained subcircuits.

while $(\exists v \mid v \in V(G) \wedge (\exists u \mid vu \in E(G) \wedge u \in V(H)) \wedge (\exists w \mid vw \in E(G) \wedge w \notin V(H)))$

- *If* $w \in \mathcal{F}^R(H)$ *then* $V(H) = V(H) + w$.
- *Else discard* H.

end while

Example of Rule 3 Application

Example 1 This example again illustrates the creation of the subgraph that covers the entire circuit graph, but within the context of Rule 3 application. Subgraphs H_0 - H_5 are shown in Figure 3.19

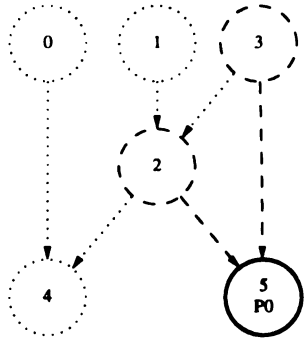
When Vertex 3 is added to the initial subgraph H_0, Vertex 2 must be added immediately, both to fully specify Vertex 5 and to contain Vertex 3. The result is H_2, in which Vertex 2 is only partially specified and non-contained. Rule 2 applies first, so Vertex 1 is added to H_2 to fully specify Vertex 3 and create H_3. The reachable frontier of H_3 is not affected by the addition of Vertex 1 because all of the input vertices of a vertex are treated as a single unit if the highest non-member vertex is in the reachable frontier when Rule 2 is applied. Expansion proceeds from H_3 by adding Vertex 4 to create H_4. Vertex 0 is added last to reproduce the entire circuit graph as H_5. All vertices are fully specified and contained, so H_5 is a contained subcircuit.

The application of Rule 3 guided the generation along this creation path to most efficiently create the contained subcircuit.

Example 2 The subgraphs for this example are displayed in Figure 3.20. When Vertex 2 is added to trivial subgraph H_0, it is non-contained, because its second parent (Vertex 5) is not a member of H_1. Furthermore, Vertex 5 is not a member of the frontier or reachable frontier of H_1, so Vertex 2 could never become contained. H_1 is therefore discarded.

If contained subcircuit enumeration is desired, Rule 3 may be added to the focused

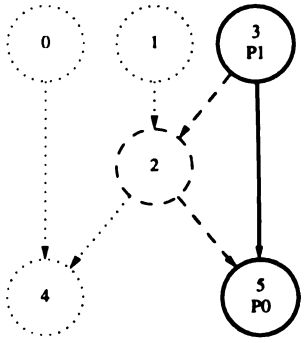
Subgraph H_0



H_0 Properties	
V	5
N	2, 3
F	2, 3
F^R	2, 3

(a)

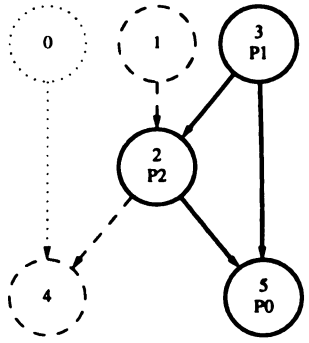
Subgraph H_1



H_1 Properties	
V	5, 3
N	2
F	2
F^R	2

(b)

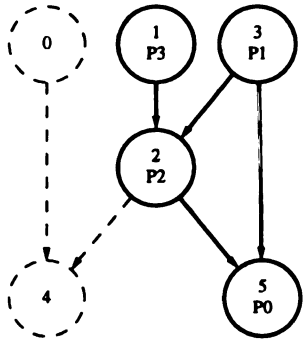
Subgraph H_2



H_2 Properties	
V	5, 3, 2
N	1, 4
F	1, 4
F^R	1, 4

(c)

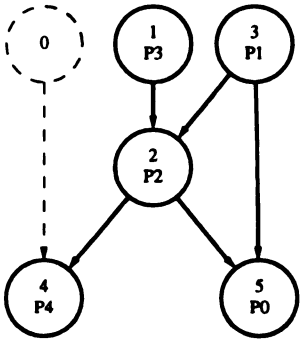
Subgraph H_3



H_3 Properties	
V	5, 3, 2, 1
N	4
F	4
F^R	4

(d)

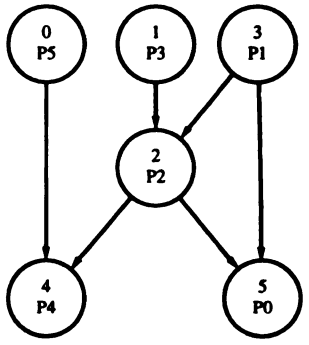
Subgraph H_4



H_4 Properties	
V	5, 3, 2, 1, 4
N	0
F	0
F^R	0

(e)

Subgraph H_5

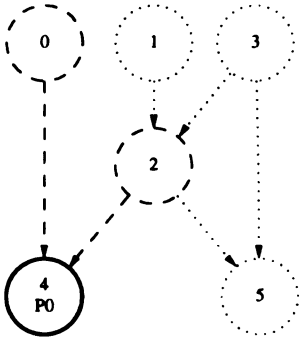


H_5 Properties	
V	5, 3, 2, 1, 4, 0
N	$\emptyset$
F	$\emptyset$
F^R	$\emptyset$

(f)

Figure 3.19: Rule 3 (Example 1).

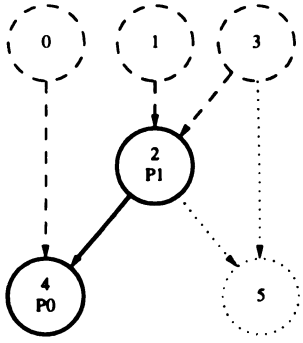
Subgraph H_0



H_0 Properties	
V	4
N	0, 2
F	0, 2
F^R	0, 2

(a)

Subgraph H_1



H_1 Properties	
V	4, 2
N	0, 1, 3, 5
F	0, 1, 3
F^R	0, 1, 3

(b)

Figure 3.20: Rule 3 (Example 2).

enumeration algorithm in Figure 3.15 by changing Step 3 of the subgraphExpand subroutine to apply both Rule 2 and Rule 3.

3.2.2 Correctness of Rules

To establish the correctness of the rules presented in Section 3.2.1, several lemmas are presented to establish that the following three major objectives have been accomplished correctly [113].

- When Rule 1 is applied, each subgraph has precisely one creation path.
- When Rule 2 is applied, only subcircuits are enumerated.
- When Rule 3 is applied, only contained subcircuits are enumerated.

1. $P_0.index$ **is the highest index in the subgraph** H**.** To begin with, it is necessary to establish that the initial vertex of a subgraph is the vertex with the highest index. Rule 1 requires that each vertex v added to a subgraph H must be a member of $\mathcal{F}^R(H)$. From that, its membership in $\mathcal{F}(H)$ may be assumed, which requires that $v.index < H.index$. Because no vertex may be added that has an index higher than that of the subgraph, the initial subgraph must contain the initial vertex by Definition 3.3. Therefore the initial vertex must be the vertex with the highest index. This discussion establishes:

2. $P_{i+1} \in \mathcal{F}(H_i)$**.** Next, the membership of each vertex v on the creation path of H must be established. If v is the next vertex to be added to the subgraph, it must be

in the neighborhood, because otherwise there is no arc between v and H and it could not be accessed. Further, if v will ever be a member of H, it must have an index less than that of H. $v's$ membership in the subgraph is known, so $v.index < H.index$, and therefore it must be in the frontier of H.

Lemma 3.1 *Given a subgraph of H, H_i, containing the vertices of $P(H)$, $P_0, P_1, ..., P_i$, then the next vertex to be added, P_{i+1}, is a member of $\mathcal{F}(H_i)$.*

Proof: Assume $P_{i+1} \notin \mathcal{F}(H_i)$. P_{i+1} is a member of $V(H)$ so it must be a member of $\mathcal{F}^R(H_x)$ at some position x (by Rule 1). Therefore, $P_{i+1} \in \mathcal{F}(H_x)$ (Definition 3.10). So, the vertices between P_i and P_x must be added before P_{i+1} may be added. However, P_{i+1} is by definition the next vertex added, which contradicts the assumption that there are additional vertices between P_i and P_x that must be added. Because no other vertices may be added before P_{i+1} and $P_{i+1} \in \mathcal{F}(H_x)$, $x = i$, contradicting the assumption that $P_{i+1} \notin \mathcal{F}(H_i)$.

3. $P_{i+1} \in \mathcal{F}^R(H_i)$. Lastly, it must be established that when a vertex v was added along the creation path of the subgraph H, it was a member of $\mathcal{F}^R(H)$. Lemma 3.1 asserted that $v \in \mathcal{F}(H)$, so either it must be newly reachable or it must not have been excluded from the reachable frontier by the previous addition of a vertex with a higher index.

The first part is clear: if a vertex is newly reachable, then it must have at least one opportunity to become a member of the graph, so it is automatically on the reachable frontier.

The second part is more complicated. If a vertex is not newly reachable, then it must have been on the reachable frontier of the previous subgraph, H_{i-1}. Therefore, the addition of the vertex that was added to create H_i may have invalidated v's membership on $\mathcal{F}^R(H)$ if its index was less than v's. However, if that occurred, then v would never be able to become a member of H. Its membership is already established, so it must be a member of $\mathcal{F}^R(H_{i-1})$.

Lemma 3.2 *Given a subgraph of H, H_i, containing the vertices of $P(H)$, $P_0, P_1, ..., P_i$, then the next vertex to be added, P_{i+1}, is a member of $\mathcal{F}^R(H_i)$.*

Proof: $P_{i+1} \in \mathcal{F}(H_i)$ by Lemma 3.1

- Suppose $P_{i+1} \notin \mathcal{F}(H_{i-1})$, meaning that P_{i+1} is newly reachable and has not yet been considered for addition.

  Because $P_{i+1} \in \mathcal{F}(H_i)$ and $P_{i+1} \notin \mathcal{F}(H_{i-1})$, then P_{i+1} need not meet any criteria to be a part of $\mathcal{F}^R(H_i)$ because it has not had a chance to be considered for addition yet, so therefore $P_{i+1} \in \mathcal{F}^R(H_i)$.

- Suppose $P_{i+1} \in \mathcal{F}(H_{i-1})$.

  - $P_{i+1} \in \mathcal{F}^R(H_{i-1})$

    Proof: Assume $P_{i+1} \notin \mathcal{F}^R(H_{i-1})$. Because $P_{i+1} \in \mathcal{F}(H_{i-1})$ and $P_{i+1} \notin \mathcal{F}^R(H_{i-1})$, P_{i+1} will not appear on $\mathcal{F}^R(H_x)$ for any $i \leq x \leq n$ (by Definition 3.10), and will never become a member of $V(H)$. However, P_{i+1} **is** a member of $V(H)$, so therefore $P_{i+1} \in F^R(H_{i-1})$.

– $P_{i+1}.index < P_i.index$

Proof: Assume $P_{i+1}.index \geq P_i.index$. We have already proven that $P_{i+1} \in \mathcal{F}^R(H_{i-1})$. Hence, when P_i is added to create the subgraph H_i, P_{i+1} will not be a member of $\mathcal{F}^R(H_i)$, and therefore will never become a member of $V(H)$. However, P_{i+1} is a member of $V(H)$, so $P_{i+1}.index < P_i.index$.

Therefore, by Definition 3.10, $P_{i+1} \in \mathcal{F}^R(H_i)$.

Rule 1: Unique Creation Paths

Theorem 3.1 *Rule 1 implies that for any induced, connected subgraph H of a circuit graph G, there is a creation path $P(H)$ that results in the subgraph H.*

Proof: Assume that there is no creation path $P(H)$. Therefore, for some P_x, Rule 2 must forbid its addition to P_{x-1}, so $P_x \notin \mathcal{F}^R(H_{x-1})$. Lemma 3.2 states that $P_i \in \mathcal{F}^R(H_{i-1})$, which contradicts the assumption, thus proving that there is a creation path to H.

Theorem 3.2 *Rule 1 guarantees that for any induced, connected subgraph H of a circuit graph G, there is no more than one creation path $P(H)$.*

Proof: Assume that there are two or more creation paths for an arbitrary subgraph H. Denote the subgraph created along one of these paths as H^1, and the vertices along H^1's path as $u_1, u_2, ..., u_n$; denote the subgraph created along the other path as H^2 and the vertices along H^2's path as $v_1, v_2, ..., v_n$. Let x be the first position in which $u_x \neq v_x$. By Lemma 3.2, $u_x \in \mathcal{F}^R(H^1_{x-1})$ and $v_x \in \mathcal{F}^R(H^2_{x-1})$. $\mathcal{F}^R(H^1_{x-1})$ must be equivalent to $\mathcal{F}^R(H^2_{x-1})$ because $u_i = v_i$ for $0 \leq i < x$.

All indices are unique, and the labels u and v were assigned arbitrarily, so let $u_x.index < v_x.index$. If $P_x = u_x$, then $v_x \notin \mathcal{F}^R(H_x^2)$, by Definition 3.10, and therefore could not become a member of H. However, we know that $v_x \in V(H)$, so $P_x \neq u_x$. If $P_x = v_x$, then $u_x \in \mathcal{F}^R(H_x^1)$ and it could become a member of H.

Therefore, for any two vertices, there is only one order in which they may be added to H. $u_x = v_x$ for $0 \leq x \leq n$. This contradicts our assumption, thus proving that there is a single unique creation path $P(H)$ to the subgraph H.

Rule 2: Only Subcircuits Created

When Rule 2 is applied to a subgraph, each vertex in the subcircuit is examined to determine whether or not it is fully specified. If a vertex v is found to be only partially specified, that is, at least one but not all of v's input vertices are present, then the vertices necessary to fully specify v are added to the subgraph if they are on the reachable frontier.

If all of the input vertices may not be added because they are not members of the reachable frontier, then the subgraph is discarded. Therefore, the only subgraphs that may be enumerated when Rule 2 is applied are subcircuits.

Rule 3: Only Contained Subcircuits Created

When Rule 3 is applied to a subgraph, each vertex in the subcircuit is examined to determine whether or not it is contained. If a vertex v is found to be *not* contained, that is, at least one but not all of v's output vertices are present, then the vertices necessary to fully specify v are added to the subgraph if they are on the reachable

frontier.

If all of the output vertices may not be added because they are not members of the reachable frontier, then the subgraph is discarded. Therefore, the only subgraphs that may be enumerated when Rule 3 is applied are contained subcircuits.

3.2.3 The Algorithm

The algorithm in Figure 3.21 (repeated from page 116) performs candidate subcircuit enumeration and guarantees the three objectives stated at the beginning of Section 3.2.2. Vertex ordering is implemented so that the algorithm may apply the rules described in Section 3.2.1. Rule 1 prevents the generation of duplicate subgraphs by ensuring that there is only one expansion path that creates each subgraph. The application of Rule 2 guarantees that each of the subgraphs created by the algorithm is a functional subcircuit. Rule 3 may also be applied to guarantee that each of the subgraphs is also a contained subgraph.

Algorithm Details

- Step 0: **Transform circuit into circuit graph G with vertex indices.**

  Create a vertex to represent each logic element in the circuit. These vertices contain necessary information about the logic element, including number of inputs and outputs, distance from primary inputs, and function performed. Each vertex is assigned a unique integer index as it is visited.

  This operation is $\mathcal{O}(n)$, and will halt because the number of logic elements in

Unique Focused Enumeration Algorithm

0. Transform circuit into circuit G with unique vertex indices.
1. **foreach** $v \in V(G)$:
   2. Create a subgraph H_0 with $V(H_0) \leftarrow v$.
   3. Determine $\mathcal{F}^R(H_0)$.
   4. expandSubgraph(H_0).
5. End **foreach**.

subroutine expandSubgraph(graph H_i)
0. **foreach** vertex $v_{v+1} \in \mathcal{F}^R(H_i)$ (Rule 1):
   1. Create a subgraph H_{i+1} such that $V(H_{i+1} \leftarrow H_i + v_{i+1}$.
   2. Determine $\mathcal{F}^R(H_{i+1})$.
   3. Add vertices necessary to make H_{i+1} into a subcircuit (Rule 2).
   4. Output H_{i+1}.
   5. expandSubgraph(H_{i+1}).
6. End **foreach**.

Figure 3.21: Algorithm for Unique Focused Enumeration.

any circuit is finite.

- Step 1: **foreach** $v \in V(G)$:

  This loop is performed n times, once for each vertex in the circuit graph, so the complexity is $\mathcal{O}(n)$.

- Step 2: **Create a subgraph H_0 with $V(H_0) \leftarrow v$.**

  Creating a subgraph containing v is an $\mathcal{O}(1)$ operation. The subgraph information (index, order, depth, etc.) is initialized.

- Step 3: **Determine $\mathcal{F}^R(H_0)$.**

  The reachable frontier of a trivial subgraph is equivalent to the frontier, which consists of all of the neighboring vertices with index less than that of the sub-

graph. All vertices may be neighbors of the subgraph, so this step is $\mathcal{O}(n)$.

- Step 4: **expandSubgraph(H_0)**.

  Send the newly created subgraph H_0 to the **expandSubgraph** subroutine.

- Step 5: **end foreach**.

  This outer loop is performed exactly n times, once for each vertex in the circuit graph.

subroutine expandSubgraph(subgraph H_i)

- Step 0: **Rule 1: foreach vertex** $v_{i+1} \in \mathcal{F}^R(H_i)$:

- Step 1: **Create a subgraph** $H_{i+1} \leftarrow H_i + v_{i+1}$.

  The subgraph H_i is copied. All information is copied to the new subgraph, including the list of vertices, the order, the depth, the index, etc. Copying the subgraph H_i and adding a vertex to the new subgraph H_{i+1} is an $\mathcal{O}(n)$ operation because it is possible that all vertices of the circuit graph belong to the new subgraph.

- Step 2: **Determine** $\mathcal{F}^R(H_{i+1})$.

  For each vertex x adjacent to $H_i + 1$: if x is not a member of $V(H)$ or $F^R(H_i$ and has an index less than $H.index$ and has not been previously explored, then add x to $\mathcal{F}^R(H_{i+1})$.

  For each vertex x in $\mathcal{F}^R(H_i)$: if $x.index < v_{i+1}.index$ then add x to $\mathcal{F}^R(H)$.

Each vertex in $V(G)$ may be a neighbor of v_{i+1}, so this step is $\mathcal{O}(n)$ and will halt when each vertex adjacent to x has been considered.

- Step 3: **Rule 2: Add vertices necessary to make H_{i+1} a subcircuit or discard.**

  The addition of vertex v_{i+1} can only affect vertices adjacent to it. For each of those vertices, if a proper subset of its parent vertices are members of $V(H_{i+1})$, then add necessary vertices until H_{i+1} is a contained subcircuit. If a necessary vertex is not a member of $\mathcal{F}^R(H_{i+1})$ then discard the subgraph.

  Because this step requires a nested loop to check all (possibly $n-1$) inputs of internal vertices (possibly n) to determine if they are fully specified each time a vertex is added, its complexity is $\mathcal{O}(n^3)$.

  This step is performed more efficiently by sorting the inputs of each vertex to ensure that the expansion proceeds with the fewest subcircuits discarded due to Rule 1 enforcement.

- Step 4. **Output H_{i+1}.**

  Output the newly created subgraph.

- Step 5. **expandSubgraph(H_{i+1}).**

  Send the newly created subgraph H_{i+1} to be expanded.

- Step 6. **End foreach.**

Algorithm Complexity

The creation of each unique candidate subcircuit as described in Figure 3.21 has a worst-case complexity of $\mathcal{O}(n^3)$. The algorithm was presented in this manner for clarity of presentation, though an actual implementation of this technique can combine the determination of the reachable frontier and the application of Rule 2 (and Rule 3, if desired) and reduce the complexity to $\mathcal{O}(n^2)$. Therefore, the computational complexity of the focused enumeration algorithm is $\mathcal{O}(kn^2)$, where k is the number of candidate subcircuits in the circuit graph (see Section 3.3).

Correctness of Algorithm

To address the correctness of the focused enumeration algorithm, several theorems are needed. It must be demonstrated that the three rules developed to restrict the enumeration of subgraphs provide the desired results when applied within the focused enumeration algorithm (Figure 3.21).

Rule 1: Unique Subgraphs

Theorem 3.3 *When Rule 1 is applied, the focused enumeration algorithm produces each induced, connected subgraph H of a circuit graph G exactly once.*

Proof: Theorems 3.1 and 3.2 state that there is one and only one creation path $P(H)$ for any induced, connected, subgraph H of a connected graph G. Therefore, by applying Rule 1, for each induced, connected, subgraph H, the enumeration may follow $P(H)$ to the unique creation of H.

Rule 2: Subcircuits

Theorem 3.4 *When Rule 2 is applied, all subcircuits are produced by the focused enumeration algorithm exactly once.*

Proof: Based on Rule 1, Theorem 3.3 states that each induced, connected, subgraph of a connected graph G will be produced by the algorithm exactly once. Rule 2 does not interfere with the application of Rule 1, so Theorem 3.3 remains true for focused enumeration with application of Rule 2.

Theorem 3.5 *When Rule 2 is applied, only subcircuits are produced by the focused enumeration algorithm.*

Proof: When Rule 2 is applied to a subgraph H, the subgraph is discarded if it cannot be made into a subcircuit. When the algorithm terminates, only the subcircuits have been produced. Therefore, the application of Rule 2 ensures that only subcircuits are produced by the focused enumeration algorithm.

In essence, these two theorems ensure that when focused enumeration is executed with the application of Rule 2, it will produce all and only subcircuits.

Rule 3: Contained Subgraphs

Theorem 3.6 *When Rule 3 is applied, all contained subcircuits are produced by the focused enumeration algorithm exactly once.*

Proof: Based on Rule 1, Theorem 3.3 states that each induced, connected, subgraph of a connected graph G will be produced by the algorithm exactly once. Rule

3 does not interfere with the application of Rule 1, so Theorem 3.3 remains true for focused subcircuit enumeration with the application of Rule 3.

Theorem 3.7 *When Rule 3 is applied, only contained subcircuits are produced by the focused enumeration exactly once.*

Proof: When Rule 3 is applied to a subgraph H, the subgraph is discarded if it cannot be made into a contained subcircuit. When the algorithm terminates, only the contained subcircuits have been produced. Therefore, the application of Rule 3 ensures that only contained subcircuits are produced by the focused enumeration algorithm.

In essence, these two theorems ensure that when focused enumeration is executed with the application of Rule 3, it will produce all and only contained subcircuits.

3.3 Results

Experiments were run on a Sun Ultra 2 Model 2300 (dual 300mhz processors) with 1024MB of memory. As a proof of concept, results are shown in Table 3.1 for several small circuits from the LogicSynth93 benchmark suite [82]. Times are given in CPU seconds. "N/A" indicates that the enumeration results could not be obtained in a reasonable amount of time. To perform subcircuit enumeration on larger circuits, the techniques discussed in Chapter 5 must be applied. Results for a parallel implementation are presented in Section 5.1, and results for an implementation that enforces an order limit are presented in Section 5.3.

Original Circuit	Gates	Unique Subgraphs		Candidate Subcircuits		Contained Subcircuits	
		number	time	number	time	number	time
1-bit Adder	8	114	0.01	18	<0.01	6	<0.01
2-bit Adder	15	3,408	0.14	114	<0.01	24	<0.01
3-bit Adder	22	98,922	4.79	522	0.04	66	<0.01
C17	24	40,729	1.79	3,951	0.16	1,199	0.05
majority	24	147,366	7.41	12,171	0.45	993	0.05
b1	25	1,066,434	56.60	19,980	0.75	143	0.01
cm138a	33	N/A	-	726,032	27.64	26,652	1.05
cm152a	35	N/A	-	59,962	2.27	6,158	0.24
z4ml3	40	N/A	-	185,196	14.64	281	0.07
x2	54	N/A	-	104,178,928	4282.80	98,322	3.93
rd53	89	N/A	-	N/A	-	207,149	23.88

Table 3.1: Focused Enumeration Results Demonstrating Reduction of Interesting Subgraphs.

These results clearly illustrate the advantage of using the focused subcircuit algorithm. The effort spent on enumeration and the number of subgraphs that must be tested for semantic equivalence is greatly reduced, resulting in significant improvements to overall execution time for solutions to the Module Identification Problem.

3.3.1 Implementation

The algorithm described in Section 3.2.3 has been implemented in C. Code to read a circuit from a BLIF file and produce a network of gates is derived from the Colorado University Decision Diagram package [102].

Correctness

To evaluate the correctness of the focused enumeration algorithm and the rules presented as the solution to the Candidate Subcircuit Enumeration Problem, the naïve

algorithm for subgraph enumeration (Figure 3.5) was implemented.

Uniqueness Each subgraph H produced by the implementation of the naïve algorithm was placed in a hash table with a hash index defined by a lexicographical ordering of the names of the constituent vertices. If a collision occurred, a vertex by vertex comparison was performed for H and each subgraph in that hash table location. If any were found to be identical, H was discarded. If H was found to be unique, it was inserted into the hash table at that location.

The subgraphs in the hash table after a full exploration of the circuit graph correspond exactly to the subgraphs produced by the algorithm that included the application of Rule 1. The results for the implementations of the naïve and the focused enumeration algorithms were compared for a range of circuit graphs to evaluate the correctness of the implementation of the unique enumeration algorithm (Rule 1).

Subcircuits and Contained Subcircuits After enumerating all of the unique subgraphs as described above, each subgraph was tested for membership in the subcircuit and contained subcircuit classes of subgraphs. Each vertex in the subgraph was tested to determine whether or not it was fully specified. If all vertices were fully specified, then the subgraph was marked as a subcircuit and saved. If not, the subgraph was discarded. Each vertex in the subcircuit was then tested for containment. If all vertices were contained, then the subgraph was also marked as a contained subcircuit.

The saved subcircuits of this trivial solution were compared to the results of the

implementation of the focused enumeration algorithm. The subcircuits produced by the two solutions were identical. This provides a great deal of confidence in the correctness of the implementation of the focused enumeration algorithm.

3.4 Summary

This chapter has presented a technique for uniquely enumerating all of the subgraphs of a graph. By applying this approach within the domain of digital circuits, the subgraphs that must be enumerated can focus on specific classes of subgraphs, reducing the computation cost considerably.

The problem of enumerating each subgraph exactly once was addressed. By assigning an index to each vertex in the circuit graph, a local ordering may be enforced between each pair of vertices with respect to the subgraph currently under expansion (Rule 1). This ordering ensures that each subgraph is created and enumerated exactly once.

The enumerated subgraphs were restricted to those subgraphs whose underlying circuit represented a functional subcircuit of the circuit. Rule 2 guides the creation of each subgraph so fewer non-subcircuits are explored, and none are produced. The subcircuits were restricted yet further to those subcircuits that are self-contained. These contained subcircuits are more likely to match library entities, so they provide a valuable starting place for module identification. Rule 3 produces contained subcircuits.

The results show the improvement that focused enumeration achieves over the

naïve enumeration of subgraphs. Only the subcircuits and contained subcircuits need to be considered for functional equivalence to library modules. This reduces the overall execution time of the solution to the Module Identification Problem significantly.

Chapter 4 describes a technique to avoid performing costly semantic matching on more than one instance of subgraphs that represent subcircuits that are structurally identical. Structurally identical subcircuits are functionally identical, so it is only necessary to perform semantic matching on one member of each of these *structural equivalence classes.*

Chapter 4

Subcircuit Equivalence Classes

The overall goal of the research presented in this dissertation is to facilitate the reverse engineering process by identifying all of the subcircuits of a circuit that may be equivalent to a library module. These subcircuits undergo a semantic matching process, Subcircuit Identification, to determine this equivalence. Currently available semantic matching techniques [36] are computationally intensive, so reducing the number of times that subcircuit identification must be performed is desirable.

Many of the candidate subcircuits generated by the focused enumeration algorithm described in Chapter 3 are likely to be structurally identical to other candidate subcircuits. Structurally identical subcircuits are necessarily functionally equivalent, so the number of subcircuits to which Subcircuit Identification must be applied may be significantly reduced by applying it to only a few instances of each unique subcircuit structure.

Despite the large number of uniquely labeled candidate subcircuits, there is a much smaller number of structurally unique subcircuits. In other words, if arbitrary labels

on the vertices, such as names, are removed and replaced with a label that identifies the type of gate they represent in the original circuit, many of the subcircuits become equivalent.

Each unique subcircuit structure defines a *structural equivalence class.* Each class contains only subcircuits whose structures, with respect to gates, modules, and interconnects, are identical. Because the subcircuits in an equivalence class are identical, both structurally and functionally, any matching information gained about one instance of that class provides the same information for all other subcircuits in the class.

To facilitate the partitioning of subcircuits into equivalence classes, a technique for generating *structural identifiers* has been developed. These identifiers describe the structure of a circuit with respect to structural invariants such as the functionality of the individual vertices and the interconnection of the vertices.

Each structural identifier represents a single structural equivalence class. The identifier is created for each subcircuit generated by subcircuit enumeration. All subcircuits with the same identifier are members of the same equivalence class, and all subcircuits within an equivalence class are structurally identical. This chapter focuses on the presentation of the structural identifier generation technique.

4.0.1 Overview of Technique

As with the approach to Candidate Subcircuit Enumeration presented in Chapter 3, structural identifiers and subcircuit equivalence classes apply to circuit graphs,

graphical representations of logic circuits.

An identifier that uniquely describes the structure of a circuit may be generated by first determining an ordering for the vertices, determined by structural invariants such as distance from circuit inputs, functionality of the vertex, number of inputs and outputs, etc. Once the vertices have been sorted, a unique integer label, a *weight*, may be assigned to them to represent their position in the ordering.

Information local to the vertices and their neighbors is not always sufficient to resolve ordering of all vertices. In this case, a second pass through the circuit graph may be necessary to resolve these conflicts. This pass efficiently utilizes information about the surrounding circuit graph to sort the remaining vertices.

Occasionally, two vertices represent identical functionality in identical positions within the circuit. These vertices are *indistinguishable*. This situation can occur, for example, when two circuit inputs are both inputs to a single gate, or when the circuit is structurally symmetrical. In that case, the vertices are completely interchangeable and may be ordered arbitrarily. All other vertices are *distinguishable*, because they are different with respect to functionality or position within the circuit.

It may not be possible to order two distinguishable vertices based solely on the information that can be efficiently obtained during two passes through the circuit graph. These vertices will also be arbitrarily ordered, and this allows the possibility of multiple identifiers that describe the same circuit structure. However, the number of identifiers for a single structure will generally be small (one or two).

The identifier is built after the vertex ordering has been determined. The vertex ordering is applied during identifier construction to ensure that there are only a few

identifiers for any circuit graph structure. Many forms of identifiers may be derived for a circuit graph in which the vertices have been labeled in this way. The method presented in this thesis produces logic functions because logic formulae are familiar to most engineers and provide an easily comprehended representation.

Vertices with more than one output arc will appear more than once in the logic formula identifier, so they must be handled differently than vertices with only one output if all structural information is to be accurately represented. These vertices are called *nets* because they are the interface to a network of gates within the circuit. To ensure that the interconnection described by the vertex is represented correctly in the identifier, nets are assigned special labels. When applied within Candidate Subcircuit Enumeration, these structural identifiers provide an equivalence relation to partition the circuit into structural equivalence classes.

4.0.2 Chapter Outline

Section 4.1 describes the structural identifiers and their generation. Vertex weighting is the first step in the process, and it is discussed in Section 4.2 along with related topics such as vertex functionality labeling, vertex weighting, and nets. The algorithm for vertex weighting is presented and discussed in Section 4.2.4. The next step is the generation of the identifier for the weighted circuit graph, which is described along with an example algorithm in Section 4.3.

Section 4.4 discusses how these structural identifiers are used within candidate subcircuit enumeration to partition the subcircuits into structural equivalence classes.

Section 4.5 shows experimental results justifying the application of these equivalence classes to candidate subcircuit enumeration, and also covers implementation details and complexity. Section 4.6 summarizes the chapter.

4.1 Structural Identifiers

Structural identifiers describing the structure of a circuit can be built by relying only on gate function and interconnects. Labels given to devices or wires are not considered, because they will not be the same between two different circuits, though the underlying structure may be identical.

A structural identifier may take many forms; the only requirement is that the identifier represent the interconnects between the devices and accurately describe the device functionality. The underlying form of the structural identifier can be viewed as an *augmented adjacency matrix.*

Definition 4.1 *An augmented adjacency matrix is an adjacency matrix for a circuit graph with an additional column whose row elements are the functionality label assigned to the vertex represented by that row.*

The generation of a circuit identifier is a two-step process. First, a weighting of the vertices in the circuit graph is performed, based upon structural invariants such as functionality, level (distance from inputs), number of inputs and outputs, etc. To allow effective definition of structural equivalence classes, the weighting algorithm must be designed such that any two identical subcircuits result in as few vertex weightings as possible.

The second step is the generation of the identifier. Once the vertex ordering has been correctly performed, this step may be designed to produce many types of structural identifiers, such as an augmented adjacency matrix, a *K-formula* representation, or a logic function representation. The algorithm described in Section 4.3 produces the latter.

The technique presented in this dissertation for vertex weighting may result in more than one identifier for a single circuit graph structure, though the number of identifiers is typically very small (one or two). It would be possible to extend the technique to ensure that each circuit graph structure produced a single unique ordering, but the complexity of that task (which is equivalent to graph isomorphism) outweighs the computational improvement to the Module Identification Problem that would result from that extension.

4.2 Vertex Weighting

There are two steps necessary to determine an ordering of the vertices so that an applicable structural identifier may be developed.

1. *Establish vertex functionality labels*: determine a unique description for each of the functionalities that are performed by vertices in the circuit graph.

2. *Weight vertices*: assign a unique weight to each vertex based upon its functionality and other structural invariants. Weight values range from 0 to $n-1$, where n is the number of vertices in the circuit graph.

4.2.1 Vertex Functionality Labels

To ensure that the identifier accurately describes the structure of the circuit, each vertex functionality must have a canonical description. Considering that a vertex in the circuit graph may describe a simple logic gate or a complex library module, the vertices are described and identified based on the functionality that they perform, rather than logic gate name or label. The function may be represented by any canonical functional representation, such as ROBDDs or truth tables that have been manipulated by a well-defined procedure to become canonical.

Once canonical descriptions of the vertex functions are available, a unique integer may be assigned to each function. These integer representations induce an ordering on the vertex functionalities. This ordering applies to the entire circuit and may be arbitrarily assigned such that each integer represents exactly one function, and each function is represented by exactly one integer. By using integers instead of the truth table or the BDD itself, vertex function comparison is simple and the size of the eventual identifier is more compact, without loss of structural information.

For the purpose of clarity, gate types will be described in this thesis by natural language labels, such as "AND2" (2-input AND gate), "adder1" (1-bit adder), etc. The vertex function ordering that applies throughout this thesis is alphabetical by the natural language label. For example, "AND3" has higher precedence than "AND2" and "OR3". Table 4.1 shows the precedence of all vertex functions appearing in the examples in this chapter.

Gate Type	Label	Precedence
AND3	6	highest
AND2	5	
NOR2	4	
NOT1	3	
OR2	2	
XOR2	1	
inputs	0	lowest

Table 4.1: Vertex Functionality and Precedence Example

4.2.2 Vertex Precedence

To develop an effective structural description of a circuit, each vertex of the circuit graph must be given an order of precedence, a *weight*, that is dependent solely upon the structure (including vertex functionality) of the circuit.

There are three steps to weight assignment:

1. *Level assignment*: Determine the level (maximum path length from circuit inputs) of each vertex.

2. *First pass (inputs to outputs)*: sort the vertices at each level based on structural invariants of the circuit graph and assign each vertex an integer weight based on that ordering.

3. *Second pass (outputs to inputs)*: resolve conflicts between vertices that could not be ordered based on local information alone by considering global structural information.

Level assignment is accomplished by starting at the inputs and assigning them to Level 0. The procedure traverses up from the inputs in a breadth-first manner and assigns each vertex the level value one higher than the maximum level of its inputs. If a cycle is encountered, i.e., a vertex has already been assigned a weight, then the weight is not modified.

During the first pass, weight assignment begins at the circuit graph inputs, and proceeds up through the levels until all vertices have been weighted. By weighting the levels in ascending order, information about the vertices closer to the circuit inputs can be used during sorting of vertices on higher levels.

Vertices that cannot be sorted by the first pass because not enough information was available are assigned the same weight, and the weights that must be assigned to those vertices to ensure a unique weighting are saved. The second pass resolves these conflicts by beginning at the outputs of the circuit graph, and proceeding down through the levels until all vertices have been assigned a unique weight.

With regard to vertex ordering, each vertex considers only those vertices that are directly adjacent to it. These vertices are the inputs and outputs to the gate that the vertex represents. By restricting the ordering comparisons to this local information, the sorting can be performed quickly and efficiently. Considering only adjacent vertices does not exclude other information about the rest of the circuit, because the weights assigned to those vertices encapsulate relevant information about the surrounding circuit structure. This encapsulated information is referred to as a vertex's *input cone* or *output cone.*

Input Cones

The input cone of a vertex v represents information about the circuit structure from the circuit's primary inputs to v. As vertices are ordered during the first pass, if two vertices have the same functionality, number of inputs, and number of outputs, then their input weight lists are sorted and compared. In this way, the information about the circuit between the inputs and the current vertex may be efficiently taken into consideration.

In digital circuits, the input cone is equivalent to the logic cone described in Section 2.8.2 of this dissertation, so the input cone of a vertex includes all of the vertices that represent gates that affect the output value of the gate. Figure 4.1 presents a conceptual diagram of an input cone.

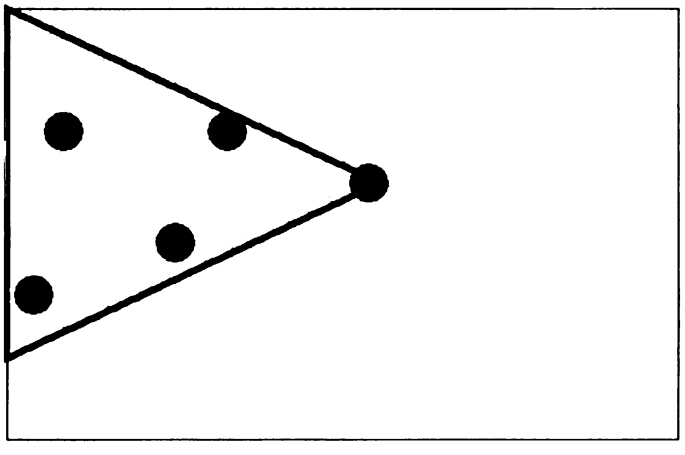


Figure 4.1: Conceptual Description of Input Cone

Output Cones

The output cone of a vertex v is defined by all of the vertices between v and the circuit's primary outputs. By ordering vertices based on their sorted output weight lists, the output cone is efficiently considered when ordering two vertices. If the output cones of the vertices differ in any way, that will be reflected in the weight that is assigned to the vertices.

Figure 4.2 presents a conceptual diagram of an output cone. Note that the output cone actually considers the input cones of each of the vertices in the output cone. This provides extensive information for ordering vertices.

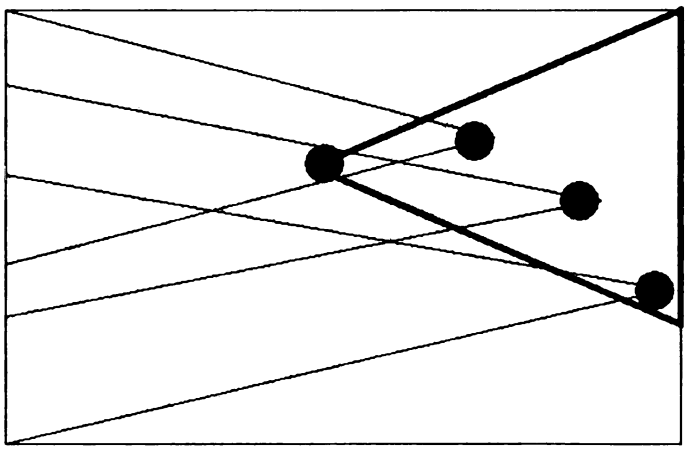


Figure 4.2: Conceptual Description of Output Cone

The following ordering measures are applied to determine sorting precedence.

First Pass Vertex Precedence Rules:

1. *vertex information*: local information about the vertex, considered in the following order:

   (a) vertex functionality

   (b) number of inputs

   (c) number of outputs

2. *input weights*: the weights of the inputs to the vertices under comparison. This information succinctly considers all vertices between the circuit inputs and the current vertex (the *input cone*), because all lower-level information is encapsulated in the weight of a vertex.

After the first pass of vertex weighting, each vertex has a weight that represents its place in the circuit. Example 1 demonstrates the weighting of a circuit graph in which only vertex information is necessary to order and weight all vertices in the circuit graph. Example 2 involves a circuit graph that requires that input weights be considered to accomplish the weighting. In cases in which the local information is not sufficient to sort a set of vertices, they are all assigned the same weight, and the other weights that need to be assigned to those vertices are "saved" until the second pass.

Second Pass Vertex Precedence Rules:

1. *output weights*: the weights that were assigned in the first pass are now used to resolve sorting conflicts. This comparison encapsulates the comparison of the

output cones of the vertices.

2. *arbitrary ordering*: vertices that are indistinguishable with respect to global circuit structure are arbitrarily sorted and weighted.

The two situations that are resolved during the second pass are examined in two examples. Example 3 illustrates resolving conflicts by investigating the output weights of the unsorted vertices. Example 4 includes two vertices that are precisely identical within the circuit, so it is necessary to arbitrarily order them to ensure that each vertex has a unique weight.

Example 5 demonstrates a situation in which two vertices are indistinguishable by only two passes through the circuit graph, though the vertices are actually different within the circuit graph. This situation can result in more than one identifier for the same structure, as discussed in Section 4.1.

Example 1: First Pass Weighting with Local Vertex Information Consideration

The circuit graph in Figure 4.3a has a simple structure with an easily defined vertex ordering. This example demonstrates the use of vertex functionality to sort the vertices and the use of the number of outputs to resolve conflicts between gates of the same functionality.

Level 0 First, the inputs (all vertices at Level 0) are assigned weights. The three vertices, *a*, *b*, and *c*, perform identical functionality; they are all inputs to the circuit. Therefore, they cannot be sorted by functionality but are instead sorted in ascending

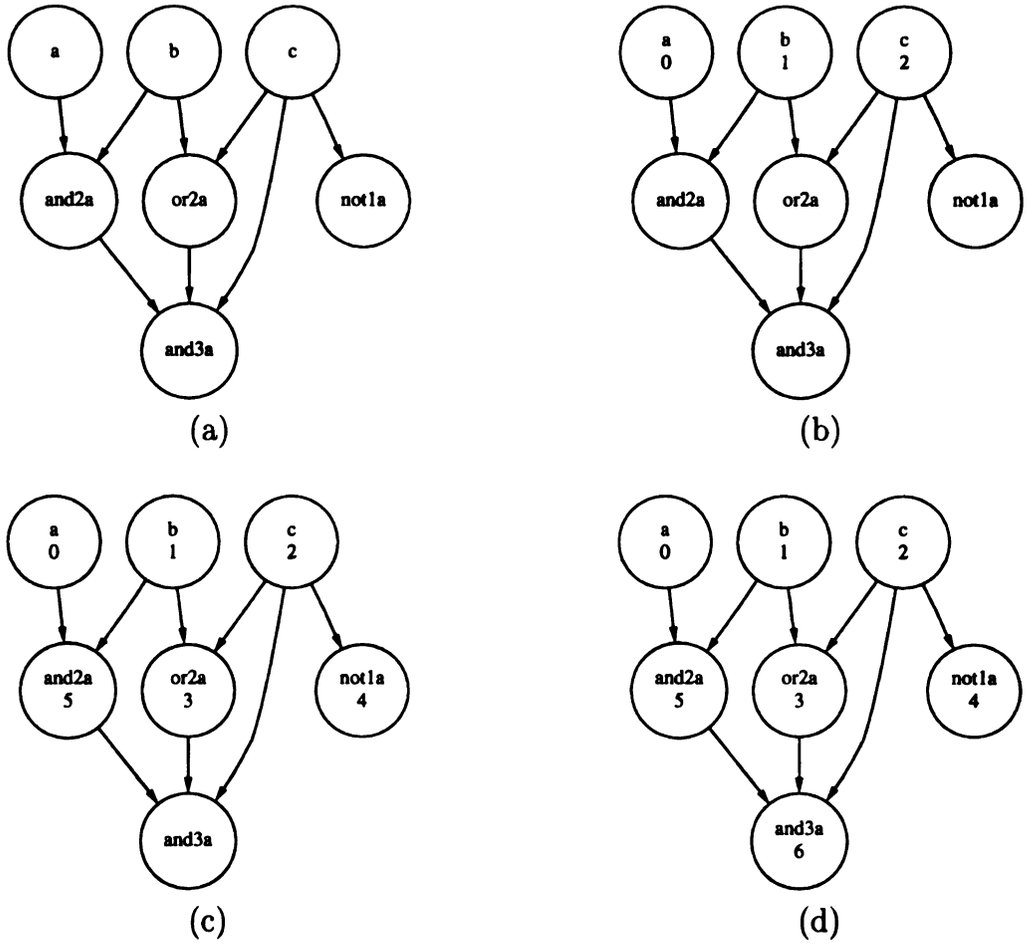


Figure 4.3: Example 1: First Pass Weighting with Local Vertex Information Consideration

order by number of outputs. Weights are assigned to the sorted list of vertices, so Vertex *a* is given weight 0, Vertex *b* weight 1, and Vertex *c* weight 2. Figure 4.3b shows the circuit graph after Level 0 weight assignment.

Level 1 At Level 1 of the circuit graph, the three vertices have three different functionalities, so sorting them is trivial. Table 4.1 shows the vertex functionality precedence ordering. Vertex *or2a* has the lowest precedence so it is assigned weight 3. Vertex *not1a* has the next highest precedence and is assigned weight 4, and Vertex *and2a* is given weight 5 because it has the highest precedence. Figure 4.3c shows the circuit graph after assigning weights to the vertices at Level 1.

Level 2 The top level of the circuit graph contains only one vertex, so sorting is not necessary. Vertex *and3a* is assigned the last weight, weight 6. Figure 4.3d shows the weighting of all of the vertices in the circuit graph.

Example 2: First Pass Weighting with Input Weight Consideration

If the circuit graph in Figure 4.3a is modified slightly so that two of the Level 1 gates are "OR2" gates, then it is necessary to consider input weights to sort those vertices. The modified circuit graph is presented in Figure 4.4a. The vertices on Level 0 are assigned the same weighting as in the previous example, so the result of that weighting is shown in Figure 4.4b.

Level 1 The functionality of Vertex *not1a* has a lower precedence than either Vertex *or2a* or Vertex *or2b*, so it is given weight 3.

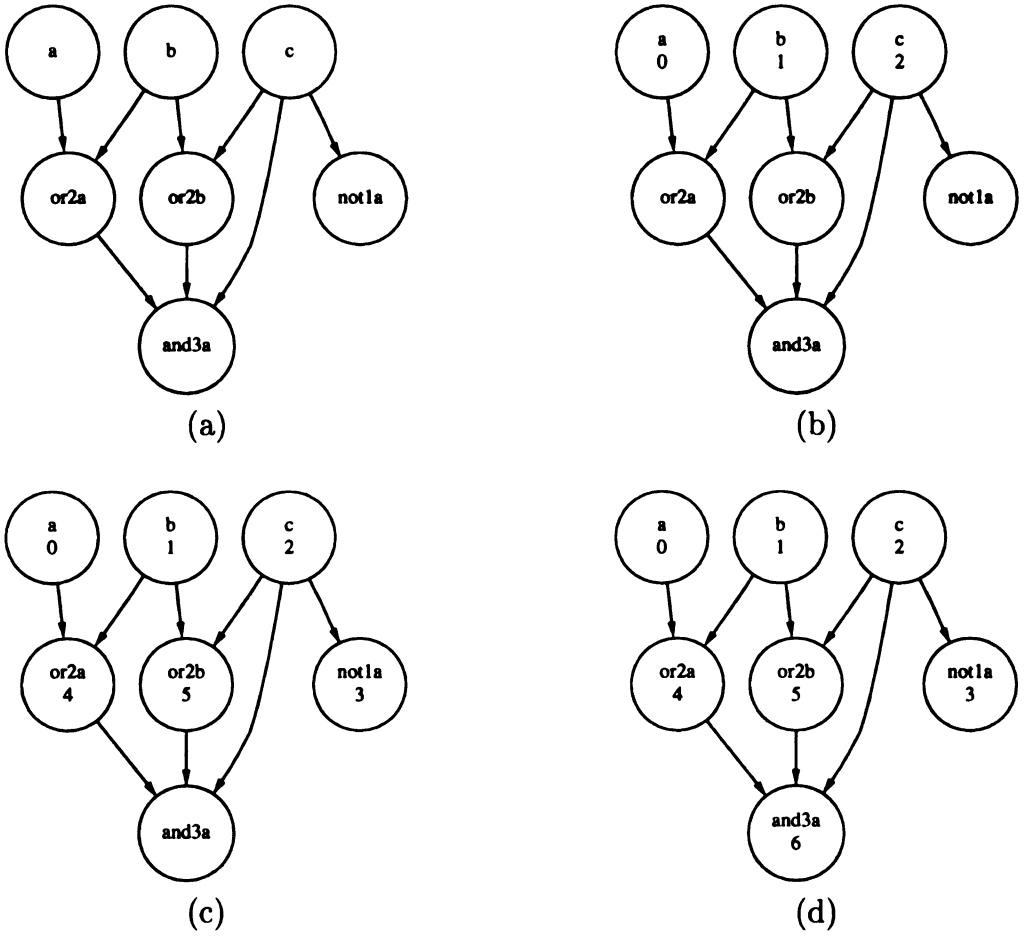


Figure 4.4: Example 2: Second Pass Weighting With Input Weight Consideration

When Vertex *or2a* and Vertex *or2b* are compared by vertex functionality (including number of inputs) and number of outputs, no difference is found that can guide the sorting of one over the other. The second vertex ordering criterion, input weights, must be applied.

The inputs of each vertex are sorted by weight, giving Vertex *or2a* the weight list [0, 1] and Vertex *or2b* the weight list [1, 2]. An iterative comparison of these lists terminates on the first element because $1 > 0$, so *or2a* is assigned the next weight, 4, and *or2b* is assigned weight 5. The result of Level 1 weighting is shown in Figure 4.4c.

Level 2 Again, with only one vertex on Level 2, no sorting is necessary, so Vertex *and3a* is assigned weight 6. Figure 4.4d shows the final weighting of the circuit graph.

Example 3: Second Pass Weighting with Output Weight Consideration

When the information local to the vertices is not sufficient to order them, it is necessary to consider the structure of the surrounding circuit graph. Starting at the outputs, unsorted vertices are ordered by sorting and comparing their output weights.

The circuit graph in Figure 4.5a is an example of a graph which requires the application of this precedence criterion. Figure 4.5b shows the vertex weights after the first pass through the circuit graph. There are two sets of vertices that have identical weights: *a* and *c*, and *xor2a* and *xor2b*.

Second Pass: Output Weight Consideration Resolution of conflicts begins at the outputs of the circuit graph, and all conflicts at each level are resolved before

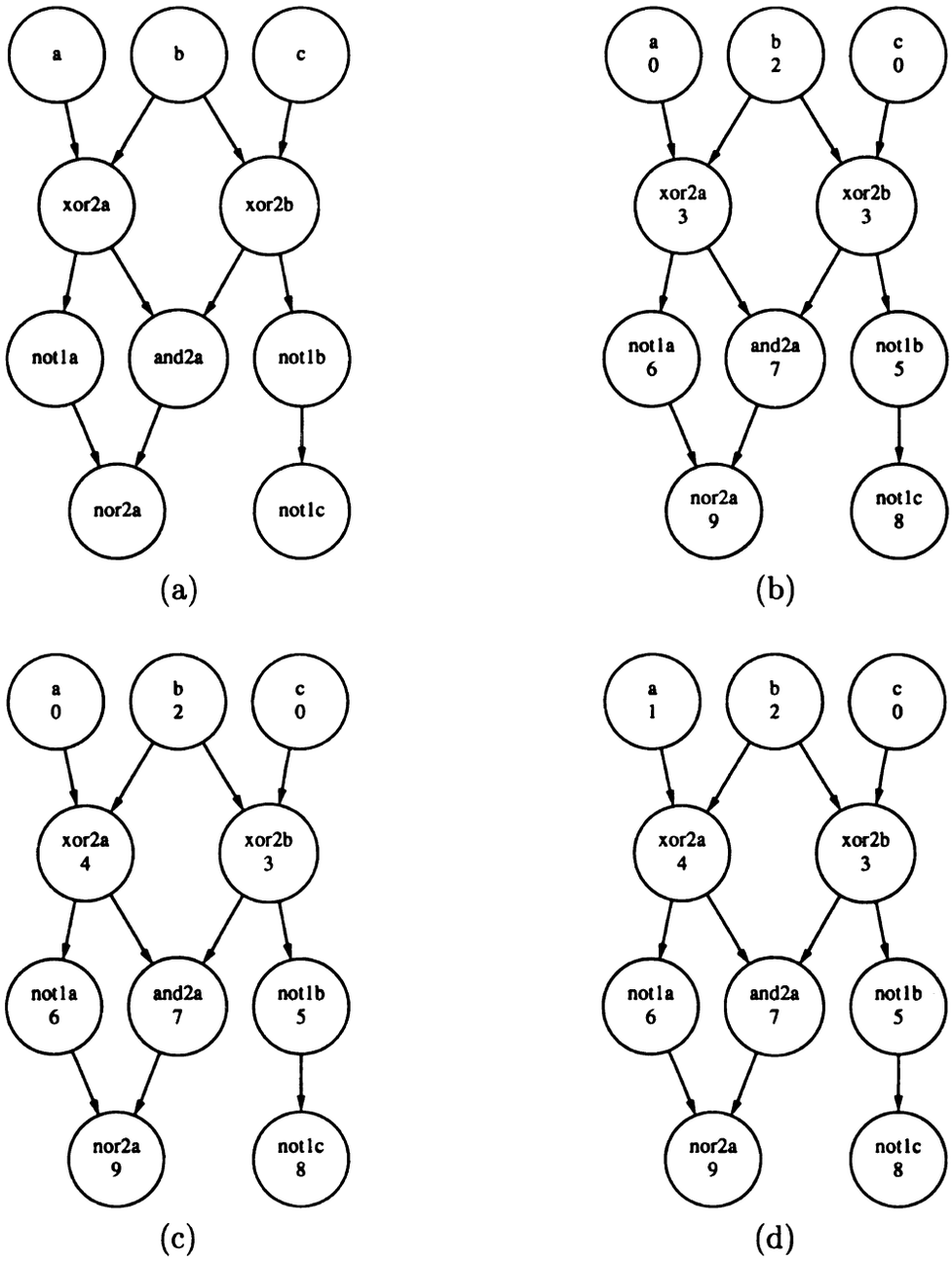


Figure 4.5: Example 3: Second Pass Weighting with Output Weight Consideration.

proceeding any further toward the inputs. In Figure 4.5b, all vertices on Levels 3 and 2 have been sorted by the first pass, so the second pass proceeds to Level 1, where *xor2a* and *xor2b* have both been assigned weight 3.

Level 1 Vertices *xor2a* and *xor2b* can be ordered by sorting and comparing the weights of their outputs. *xor2a*'s sorted output weights are [6, 7] and *xor2b*'s sorted output weights are [5, 7], so *xor2a* is assigned the higher weight of 4. Figure 4.5c shows the circuit graph with the new Level 1 weights.

Level 0 The next set of identically-weighted vertices to resolve is on Level 0: *a* and *c*. The weights of the vertices at Level 1 now allow an unambiguous sorting and vertex weighting, assigning *a* the higher weight of 1. Figure 4.5d shows the completely weighted circuit graph after the second pass.

Example 4: Second Pass Weighting with Arbitrary Ordering of Identical Vertices

Occasionally, it is not possible to sort vertices deterministically because they perform identical functionality in isomorphic locations within the circuit graph. For instance, the circuit graph in Figure 4.6a has identical halves connected in the center at *xor2b*. As a result of this symmetry, there are several sets of vertices that will have identical weights after the first pass. The weights after the first pass can be seen in Figure 4.6b.

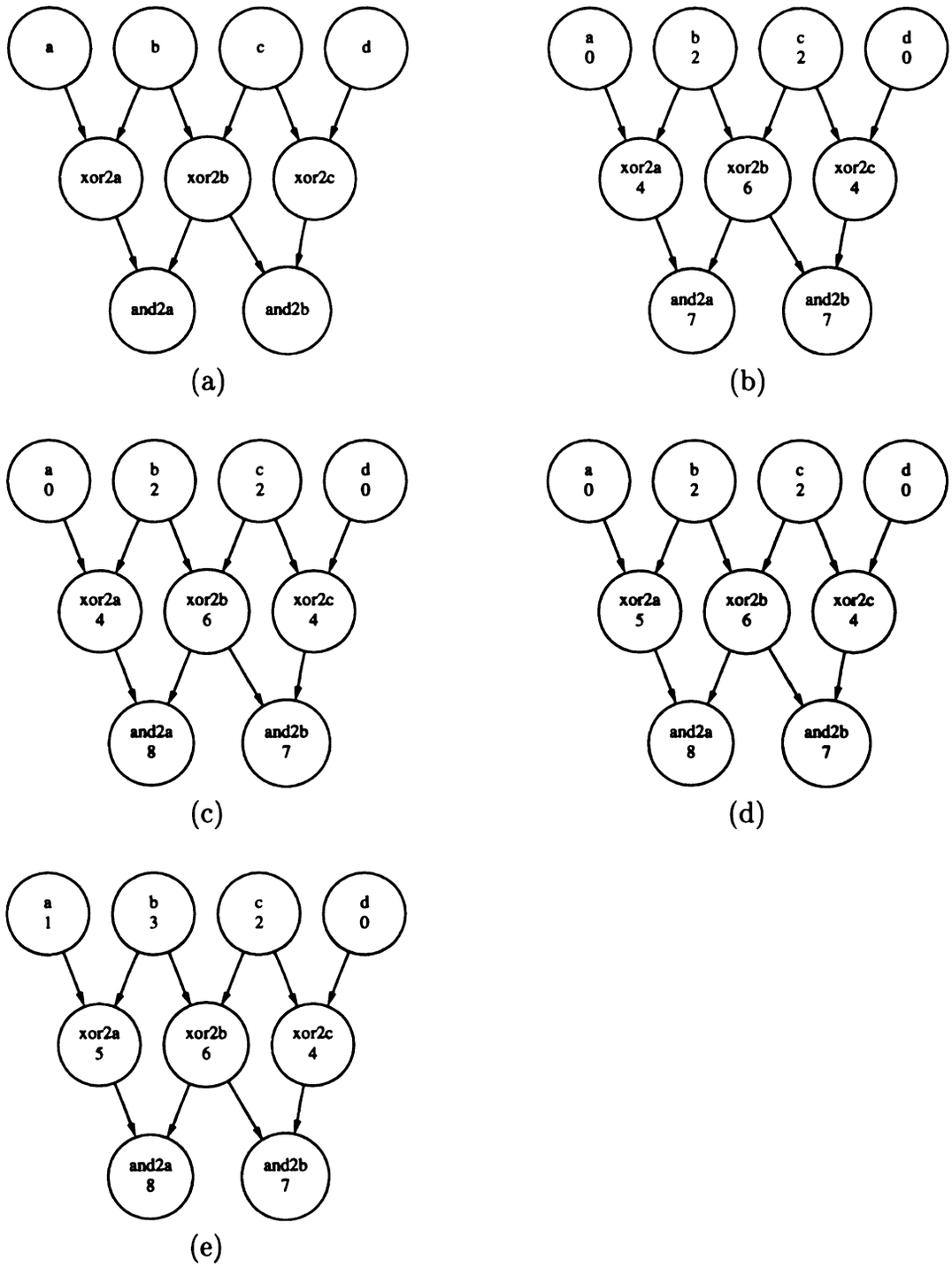


Figure 4.6: Example 4: Second Pass Weighting with Arbitrary Vertex Ordering Required

Second Pass: Arbitrary Ordering

Resolution of identically weighted vertices begins at the outputs (Level 2) and proceeds toward the inputs.

Level 2 The two vertices on Level 2 cannot be sorted because they perform identical functionality and occupy equivalent locations within the circuit graph. Therefore, the order in which their weights are applied is not significant. Arbitrarily, Vertex *and2a* may be assigned the higher weight of 8 (Figure 4.6c).

Level 1 and 0 Sorting the vertices at Levels 1 and 0 proceeds as it did in Example 3, because now the vertices may be sorted based on output weight. In this way, the arbitrary decision made at the highest level of the circuit graph propagates throughout to maintain consistency. The weighted circuit graphs after the weighting of Levels 1 and 0 are shown in Figures 4.6d and 4.6e, respectively.

Example 5: Second Pass Weighting with Arbitrary Ordering of Distinct Vertices

The two passes encompass only the information that is "visible" from each of the vertices, which includes the input cone and the output cone for each of them. If the circuit graph is very shallow, it may be necessary to arbitrarily order two distinguishable vertices.

For instance, the circuit graph in Figure 4.7a demonstrates this difficulty. Figure 4.7b shows the circuit graph after weighting by the first pass. There are two vertices

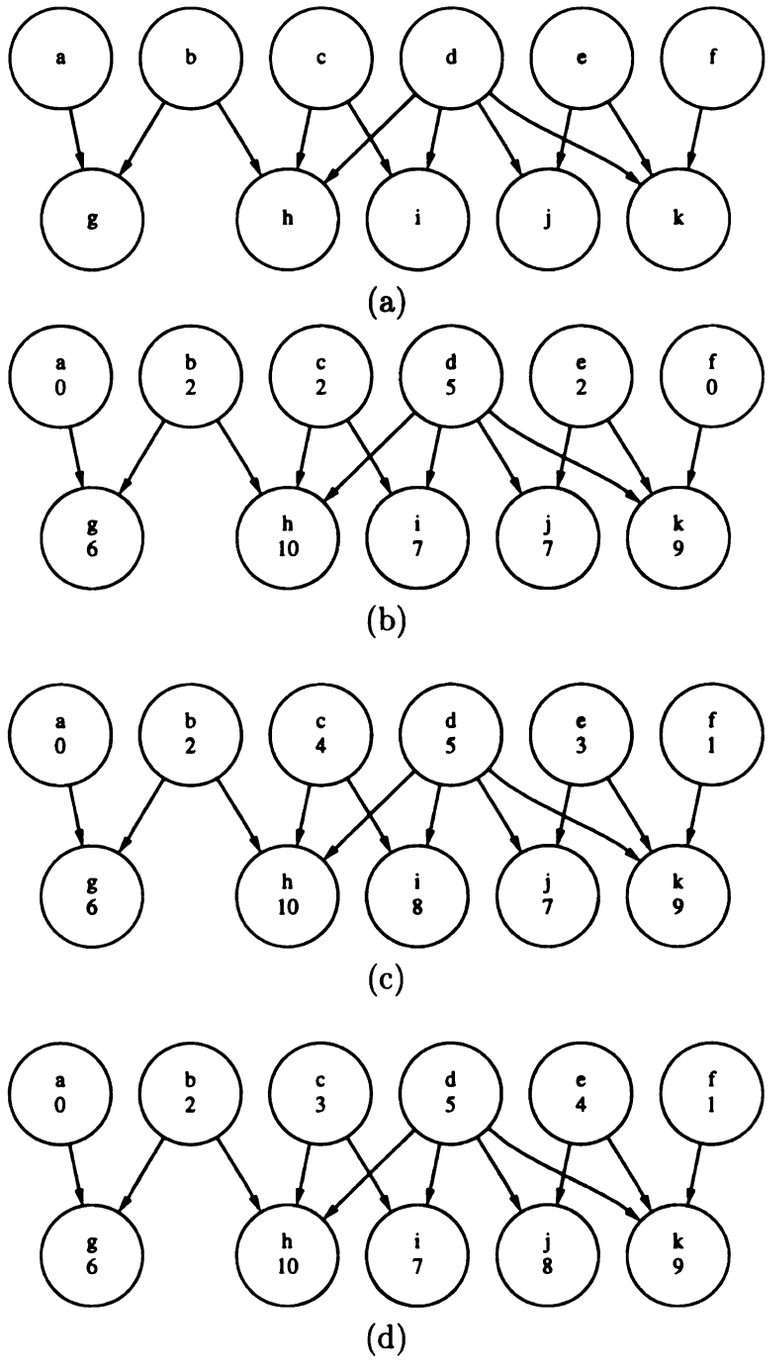


Figure 4.7: Example 5: Second Pass Weighting with Arbitrary Ordering of Distinguishable Vertices

(i, j) that currently have the same weight, so those must be resolved during the second pass.

Second Pass: Arbitrary Ordering

Resolution of identically weighted vertices begins at the outputs (Level 1) and proceeds toward the inputs.

Level 1 The two unordered vertices on Level 2 cannot be deterministically sorted because they perform the same functionality. Although the surrounding circuit graph distinguishes them, the difference cannot be detected by only two passes through the circuit graph. Therefore, the arbitrary decision will result in one of the weightings presented in Figure 4.7c or Figure 4.7d.

This will result in two different identifiers that represent the same circuit graph structure, as discussed in Section 4.1.

4.2.3 Correctness of Vertex Weighting

Case Analysis for Vertex Weighting

If two arbitrary vertices u and v on a level i are distinguished by local (vertex) or surrounding (circuit) structure, then they may be ordered by the feature that distinguishes them. This ordering is the basis for the vertex weighting technique for developing a structural identifier for circuits or unlabeled directed acyclic graphs.

First Pass The following five cases are sufficient to order two arbitrary vertices u and v on a level i for the purpose of weight assignment during the first pass. The relationships between u and v are based on local information about u and v, such as functionality, number of inputs, number of outputs, and global information about the structure of the circuit between the inputs to the circuit and Level i.

When two arbitrary vertices u and v are distinguishable by the first or second pass, let u indicate the vertex that is ordered less significantly without loss of generality. The following cases can be seen applied in the vertex weighting Examples 1, 2, 3, 4, and 5.

First Pass Case Analysis:

- Case 1.1: Vertices u and v have different functionality labels and may be sorted in ascending order by functionality labels. $u.weight < v.weight$.

- Case 1.2: Case 1.1 is false and vertices u and v do not have the same number of inputs, then they may be sorted in ascending order by number of inputs. $u.weight < v.weight$.

- Case 1.3: Cases 1.1 and 1.2 are both false and vertices u and v have unequal numbers of outputs then they may be sorted in ascending order by number of outputs. $u.weight < v.weight$.

- Case 1.4: Cases 1.1, 1.2, and 1.3 are all false and u and v have different sorted input weight lists then they may be sorted in ascending order by the first unequal input weight. $u.weight < v.weight$.

- Case 1.5: Cases 1.1, 1.2, 1.3, and 1.4 are all false. $u.weight = v.weight$.

If local information (Cases 1.1, 1.2, and 1.3) is not sufficient to order u and v, then more comprehensive information about the circuit structure must be applied (Case 1.4). The *input cone* describes all structural information between the inputs (Level 0) of the circuit and the vertex v. Input cones are efficiently compared by comparing the sorted input weight lists of u and v. If the structure of the input cone differs, then the sorted input weight lists will differ and the vertices will be weighted in Case 1.4. If the input cones are not sufficient to order the vertices (Case 1.5), then they are assigned the same weight and will be resolved during the second pass.

Second Pass Resolution of vertex weights during the second pass begins at the vertices of the circuit at the highest level. Case 2.1 considers vertices that have already been deterministically assigned an order by the first pass. Case 2.2 considers the ordering of the vertices by considering their *output cones*. The output cone represents global circuit structure that affects vertex v because the input cone of v is necessarily included in the input cone of every vertex to which v is an input.

When Level i is being considered during the second pass, all vertices on Level $i+1$ have been assigned a unique weight that considers their input and output cones. Thus, any structural information that affects the vertex v has been encapsulated and can be used to resolve identical weights remaining after the first pass. The following cases can be seen applied in Examples 3, 4, and 5.

Second Pass Case Analysis:

- Case 2.1: $u.weight < v.weight$.

- Case 2.2: Case 2.1 is false and u and v have different sorted output weight lists. $u.weight < v.weight$. Consider Figure 4.5.

- Case 2.3: Cases 2.1 and 2.2 are both false. $u.weight < v.weight$ (arbitrarily). Consider Figure 4.6.

Unique Vertex Weighting

Theorem 4.1 *Each vertex in the circuit graph C is assigned a unique weight based on the structure of the circuit graph.*

Proof by induction:

Base case: the vertices at Level n whose local information and input cones are identical are indistinguishable. Therefore, their weights are interchangeable without loss of uniqueness. Thus an arbitrary assignment provides each vertex with a unique weight.

Inductive Hypothesis: For Level i assume that every vertex from Level $i + 1$ *to Level n has been assigned a unique weight.*

Inductive Step: Each vertex at Level i can be assigned a unique weight based upon the structure of the circuit.

For any two vertices at Level i which are not already ordered (by the first pass), if the sorted output weight lists are different then the vertices are distinguishable and their weighting is deterministic. Otherwise, the vertices are indistinguishable with respect to local information, input cones, and output cones, and are arbitrarily ordered. Thus, a unique weight is assigned to each vertex.

$\therefore$ *Each vertex in the circuit graph is assigned a unique weight based on the structure of the circuit graph.*

4.2.4 Algorithm

Figure 4.8 describes the vertex weighting algorithm.

Details and Complexity

First Pass for Vertex Weighting

- Step 0: $weight \leftarrow 0, curWeight \leftarrow 0$

  Initialize the weight counter (*weight*) and the current weight counter (*curWeight*). The current weight counter allows "saving" of weights for assignment during the second pass of the algorithm.

- Step 1: **foreach level i of the circuit graph G**

  Comparison of vertices is performed on a level-by-level basis to reduce the size of the set to be ordered. There are at most n levels, so this step is $\mathcal{O}(n)$.

- Step 2: **sort inputs of each vertex on Level i by weight**

  The second sorting criterion requires that the inputs must be sorted by weight so the comparison may be done. There may be n inputs to each vertex, so this step is order $\mathcal{O}(n \log n)$ using the Quicksort algorithm.

- Step 3: **sort vertices in Level i in increasing order by the following criteria:**

Algorithm for Circuit Graph Vertex Weighting

First Pass for Vertex Weighting

0. $weight \leftarrow 0, curWeight \leftarrow 0$
1. **foreach** level i of the circuit graph G (increasing i):
   2. sort inputs of each vertex on level i by weight.
   3. sort vertices in level i in increasing order by the following criteria:
      local vertex information (functionality, number of inputs/outputs)
      input weights
   4. $level[i].vertex[0].weight \leftarrow weight$
   5. **for** $(j = 1; j < level[i].order; j{+}{+})$
      6. **if** $(level[i].vertex[j] \neq level[i].vertex[j-1])$ **then** $curWeight \leftarrow weight$
      7. $level[i].vertex[j].weight \leftarrow curWeight$
      8. $weight \leftarrow weight + 1$
   9. End **for** j.
   10. $curWeight \leftarrow weight$
11. End **foreach** level.

Second Pass for Identical Vertex Weighting

0. **foreach** level i of the circuit graph G (decreasing i):
   1. sort outputs of each vertex on level i by weight.
   2. $weight \leftarrow net[0].weight$
   3. sort vertices with identical weight in increasing order by the following criterion:
      output weights
   4. **for** $(j = 1; j < level[i].order; j{+}{+})$
      5. **if** $(level[i].vertex[j].weight \leq level[i].vertex[j-1].weight)$
         6. $(level[i].vertex[j].weight \leftarrow level[i].vertex[j-1].weight) + 1$
      7. End **if**.
   8. End **for** j.
9. End **foreach** level.

Figure 4.8: Algorithm for Circuit Graph Vertex Weighting.

1. vertex information: functionality, number of inputs/outputs

2. input weights

An encoding of this information can be derived such that a sorting of that encoding may occur. To prevent excess computation, these criteria may be applied in order and comparison may stop when the vertices have been ordered.

Therefore, the complexity of this step is $\mathcal{O}(n \log n)$ using the Quicksort algorithm.

- Step 4: $level[i].vertex[0].weight \leftarrow weight$

  Set the weight of the vertex with lowest precedence to be value of *weight*.

- Step 5: **for** $(j = 1; j < level[i].order; j++)$

  Assign a weight to each vertex on the level. A vertex may only belong to one level, so on average, this step will apply to n/i vertices, but worst case analysis is $\mathcal{O}(n)$.

- Step 6: **if** $(level[i].vertex[j] \neq level[i].vertex[j-1])$ **then** $curWeight \leftarrow weight$

  If the vertices are not equivalent (based on the criteria in Step 2), then the $curWeight$ variable is set to the *weight* variable, which has been incremented as each vertex has been visited. If they are equivalent, then $curWeight$ is not changed, thus saving the *weight* value for assignment during the second pass.

- Step 7: $level[i].vertex[j].weight \leftarrow curWeight$

  Assign the weight of the vertex to be the value of $curWeight$.

- Step 8: $weight \leftarrow weight + 1$

  Increment the value of *weight*.

- Step 9: **End for loop of vertices on Level i.**

  The possible number of vertices on any level is bounded by the number of vertices in the circuit graph, n. The operation performed within this loop was $\mathcal{O}(n)$, so the complexity of this step is $\mathcal{O}(n)$.

- Step 10: $curWeight \leftarrow weight$

  Reassign *curWeight* to the current value of *weight*, so vertices are assigned the correct weight. At the beginning of each level, *curWeight* and *weight* are always equivalent.

- Step 11: **End foreach level.**

  The vertices on each level were sorted only within that level. The worst case situation would occur if all of the vertices were on a single level, so the complexity of each level (possibly n) is the same as the complexity of the sorting step, $\mathcal{O}(n \log n)$. Overall, the complexity of the first pass is $\mathcal{O}(n^2 \log n)$.

Second Pass for Identical Vertex Weighting

- Step 0: **foreach level i of the circuit graph G (decreasing i):**

  To resolve the conflicts between vertices that appear to be identical when information is only available about the vertex itself and its neighbors (during the first pass), it is now possible to resolve these conflicts with the weighting information of the outputs.

- Step 1: **sort outputs of each vertex on Level i by weight**

  To compare the output weights of two vertices, the weights must be sorted. The possible number of outputs to be considered is potentially n, and their sorting is $\mathcal{O}(n \log n)$ using the Quicksort algorithm.

- Step 2: $weight \leftarrow net[0].weight$

  Set the current weight to be the same as the weight of all of the vertices in the set.

- Step 3: **sort vertices with identical weight in increasing order by the following criterion:**

  - output weights

  The output weights have already been sorted for each vertex, so a sorting of the vertices by considering these weights is $\mathcal{O}(n \log n)$ using the Quicksort algorithm.

- Step 4: **for** $(j = 1; j < level[i].order; j{+}{+})$

  Each vertex v on the level must be assigned a unique weight, according to the sorted order.

- Step 5: **if** $(level[i].vertex[j].weight \leq level[i].vertex[j-1].weight)$

  Step 6: $(level[i].vertex[j].weight \leftarrow level[i].vertex[j-1].weight) + 1$

  Step 7: **End if**

  If the current vertex has a weight that is less than that of the previous vertex, then set the weight of the current vertex to be one higher than the previous

vertex. This ensures that every vertex has a unique weight. The weight values that will be assigned were "saved" during the first pass, so no duplicate weights may occur.

- Step 8: **End for** j

  The **for** loop started in Step 5 assigns a weight to each vertex on the level (if necessary). There may be no more than n vertices on the level, so the complexity of this simple comparison and assignment loop is $\mathcal{O}(n)$.

- Step 9: **End foreach level.**

  The sorting step (Step 3) is the most complex, requiring $\mathcal{O}(n \log n)$ time for the sorting. There may be at most n levels, so the overall complexity is $\mathcal{O}(n^2 \log n)$.

Overall, the complexity of determining an appropriate ordering and weighting the vertices in a circuit graph is dominated by the cost of sorting the vertices on each level which must be done one for each of possibly n levels. The complexity of the first and second pass is equivalent and additive, so the overall complexity of the algorithm is $\mathcal{O}(n^2 \log n)$.

Correctness This algorithm considers each of the five distinguishable cases discussed in the case analysis performed in Section 4.2.3. Step 3 of the first pass considers each of Cases 1.1, 1.2, and 1.3, ensuring that vertices are ordered based on the first pass precedence criteria. Step 3 of the second pass sorts the vertices based on the second pass precedence criteria, Cases 2.1, 2.2, and 2.3. Because this algorithm implements the cases exactly, it correctly performs vertex weighting.

4.3 Structural Identifier Generation

The generation of the actual structural identifier is the final step in the process. This step is relatively straightforward because the ordering of the vertices has already been established. All that is necessary is to recursively build the string identifier. For the purposes of this dissertation, a prefix-form logic formula structural identifier is defined.

Nets

Unless the circuit graph is described by a pure tree structure, in which each vertex has only a single output, there are structures whose functionality feeds into two separate vertices in the circuit graph. These are referred to as *nets* in this thesis. For instance, in Figure 4.3a on page 153, the vertices labeled b and c are examples of nets.

To preserve the overall structure of the circuit, it is important that nets are identified and represented appropriately. Simply reproducing the entire structure of a net each time it occurs in the representation ignores the interconnection information and results in an inaccurate description.

Nets are represented within the identifier as nX, where X is the number of the net. Nets are weighted as are all other vertices, so their numbering and location in the identifier are well-defined. Nets are numbered during the second step of structural identifier generation: the identifier generation stage.

Figure 4.10 on page 176 presents an implementation of a 2-bit full adder. The structural identifier generated by the example algorithm in Section 4.3.1 for that

particular implementation of a 2-bit adder is:

```
(or2(and2(n0, n5), and(n6, n7)), xor2(n0, n5), xor2(n1, n4))
n0:or2(and2(n1, n4), and2(n2, n3)) n1:xor2(n2, n3) n2:X n3:X n4:X
n5:xor2(n6, n7) n6:X n7:X
```

Vertex Identifiers

The identifiers for the three types of circuit graph vertices and the circuit graph itself are described below.

- *Nets*: If the vertex has more than one output, then it is assigned a label "nX", where X is the current net number. Each net is assigned a unique, strictly increasing net number starting with 0.

- *Input Identifiers*: The identifier of a single-output input is simply "X". That identifier carries all necessary information about the input vertex v, including that v is an input and that v has only one output. If an input has more than a single output, then the vertex is treated as a net and will be assigned the appropriate net number.

- *Internal Vertex and Output Identifiers*: The first part of an identifier for a vertex is the vertex functionality label. If the vertex has any inputs, then the functionality label is followed by a left parenthesis, then by the identifiers from each of its input vertices (in order of weight), and finally by a right parenthesis.

- *Circuit Graph Identifiers*: The entire circuit graph may be described by determining the identifier for a null-functionality vertex that has all outputs of the

circuit graph as inputs. This identifier is followed by the description of each of the nets in the following format: "nX: [identifier of nX]".

4.3.1 Example Algorithm

The algorithm in Figure 4.9 is an example of a simple algorithm that could be used to generate structural identifiers for weighted circuit graphs. The identifier generated is in the form of a prefix-order logic formula.

4.3.2 Structural Identifier Example

This example demonstrates the application of vertex weighting and identifier generation together on an actual circuit, the 2-bit adder shown in Figure 4.10. The circuit, transformed into circuit graph form, is presented in Figure 4.11a. The circuit graph is displayed with the vertices clearly separated by level to clarify the vertex weighting procedure.

Level 0 The weighting of the Level 0 vertices is complicated by the fact that none of the vertices may be immediately sorted. These five inputs (A0, B0, A1, B1, C0) are all weighted 0, and the weights 1, 2, 3, and 4 are saved to be assigned during the second pass. Figure 4.11b shows the result of Level 0 weighting.

Level 1 At Level 1, the vertices may be sorted by considering functionality and output information. There are two XOR2 gates and two AND2 gates, so the vertices of each pair is assigned the same weight (5 and 7) appropriately. Figure 4.12a shows

Example of a Canonical Identifier Generation Algorithm for Weighted Circuit Graphs

0. $nNets \leftarrow 0$
1. **foreach** output vertex i of G:
   2. $G.output[i].ident \leftarrow$ **getVertexIdent**($G.output[i]$)
3. End **foreach**.
4. $G.ident \leftarrow$"($G.output$[0].$ident$, $G.output$[1].$ident$, ... , $G.output[n].ident$) $n0 : n0.modIdent\ n1 : n1.modIdent$ ... $nx : nx.modIdent$"

subroutine getVertexIdent(vertex v):
0. **if** vertex $v.ident \neq NULL$ **then** return $v.ident$.
1. **if** vertex v has no inputs **then**
   2. $v.ident \leftarrow$ "X"
3. **else**
   4. $v.ident \leftarrow v.function +$ "("
   5. sort inputs of v in decreasing order by weight
   6. $v.ident \leftarrow v.ident +$ **getVertexIdent**($v.input$[0]) + ","
   7. **foreach** input i of v:
      8. $v.ident \leftarrow v.ident +$ **getVertexIdent**($n.input[i]$)
   9. End **foreach** input i.
   10. $v.ident \leftarrow v.ident +$ ")"
11. End **else**.
12. **if** vertex v has one output **then**
   13. return $v.ident$.
14. **else**
   15. $v.modIdent \leftarrow v.ident$
   16. $v.ident \leftarrow$ "n" + $nNets$
   17. $nNets \leftarrow nNets + 1$
18. End **else**.

Figure 4.9: Example of a Canonical Identifier Generation Algorithm for Weighted Circuit Graphs.

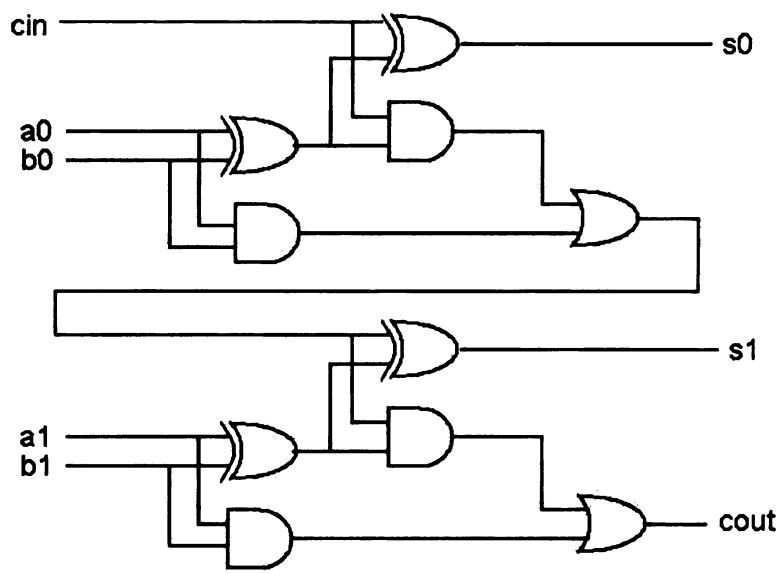


Figure 4.10: Full Equivalence Class Example: 2-bit adder

the result of this weighting.

Level 2 Only vertex functionality is necessary to order the vertices on Level 2 for weighting. AND2 has higher precedence than XOR2 (Table 4.1).

Level 3 Level 3 has only one vertex, so no ordering is necessary before the weight of 11 is assigned to the OR2 gate.

Level 4 The sorting of the vertices of Level 4 is accomplished by simple functionality precedence.

Level 5 The single vertex at Level 5 is assigned the highest weight, and the complete weighting after the first pass through the circuit graph is given in Figure 4.12b.

Second Pass It is necessary to make a second pass through the circuit graph to resolve the conflicts between any vertices that had been assigned the same vertex. After the first pass, all vertices have a weight, so that weight may be used to determine an ordering for the conflicting vertices.

All vertices down to Level 1 have a unique weight assigned to them, so the second pass begins vertex weighting at Level 1.

Level 1 The vertices on Level 1 consists of two pairs of vertices with equal weights. Within the AND2 pair, the vertex on the left has a higher output weight than the vertex on the right, so they are assigned weights 8 and 7, respectively. Similarly with the XOR2 pair, the output of the vertex on the left has a higher weight than the output of the vertex on the right, so they are assigned weights 6 and 5, respectively (Figure 4.13a).

Level 0 The vertex C0 has the highest sorted output weight lists, so is assigned the highest weight on the level, 4.

It is clear that the vertices A0 and B0 have lower sorted output weights ([5,7]) than the vertices A1 and B1 ([6,8]), so A1 and B1 may both assigned the weight value 2. However, A1 and B1 are identical within the structure of the circuit, so they are arbitrarily ordered, with A1 assigned the value 3, and B1 assigned the value 2. A0 and B0 are also identical, so they are also arbitrarily ordered. The final weighting of the circuit graph is presented in Figure 4.13b.

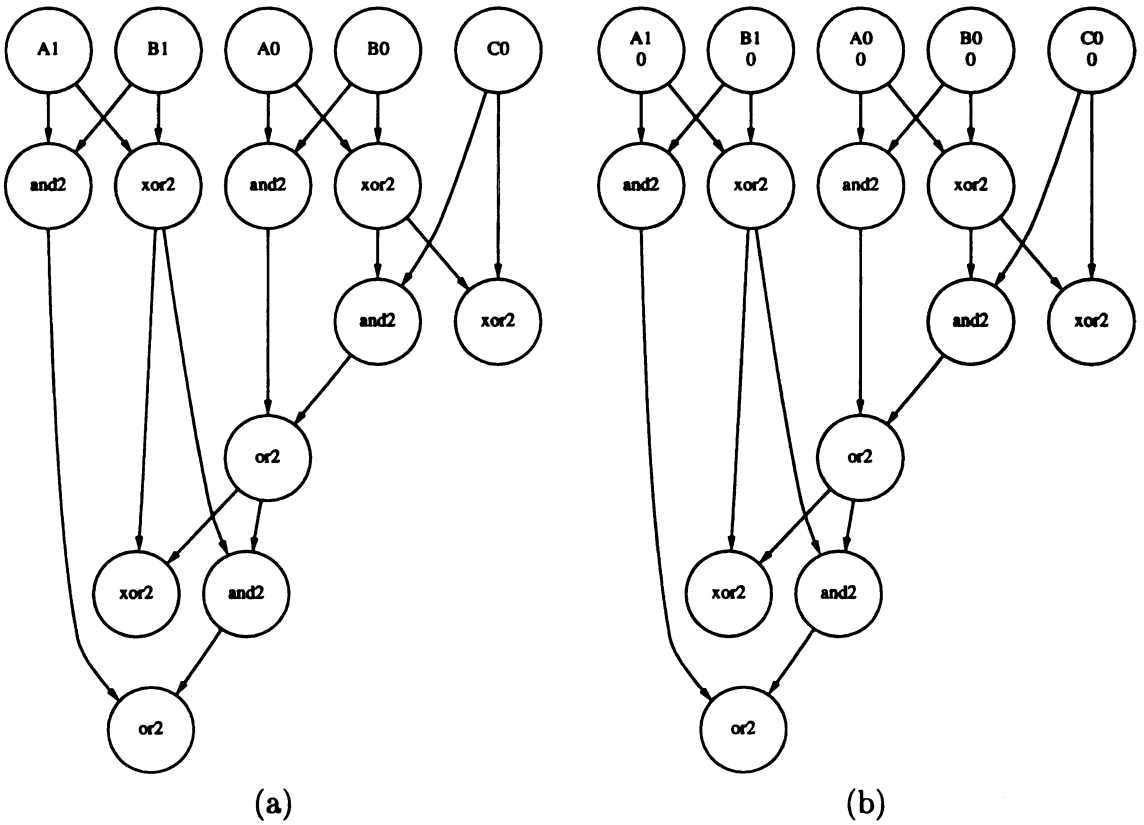


Figure 4.11: Vertex weighting for the circuit graph of the 2-bit full adder in Figure 4.10.

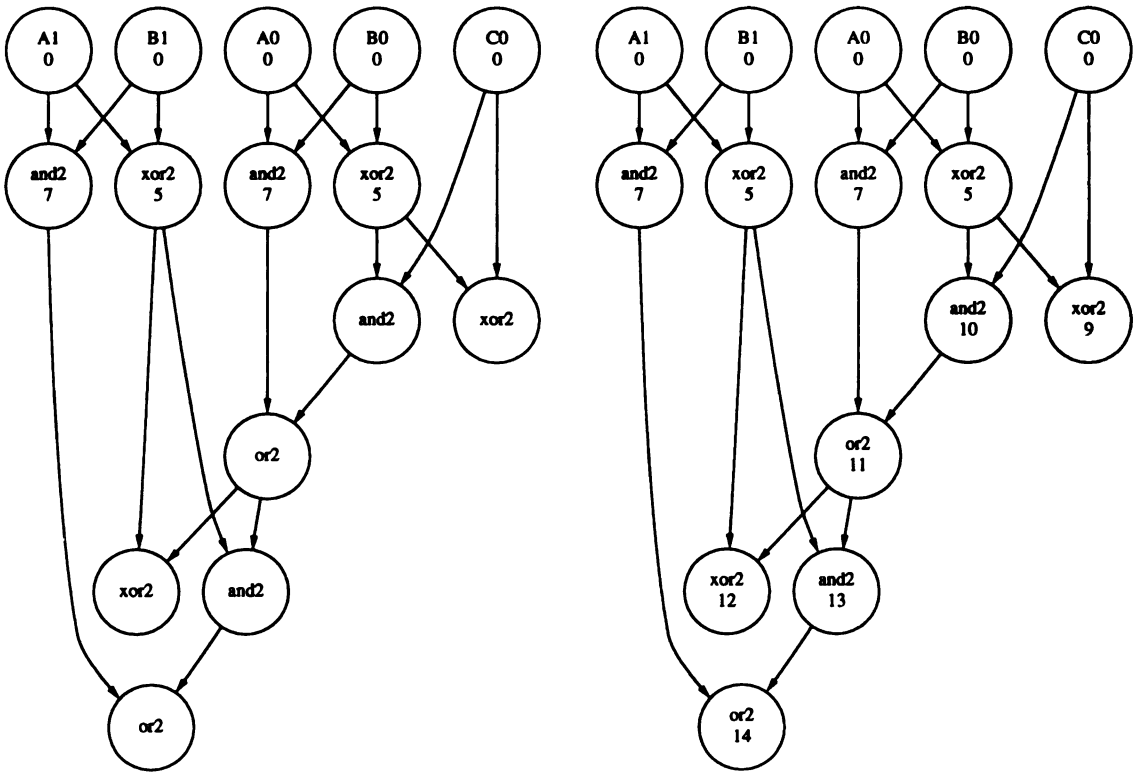


Figure 4.12: Vertex weighting for the circuit graph of the 2-bit full adder in Figure 4.10.

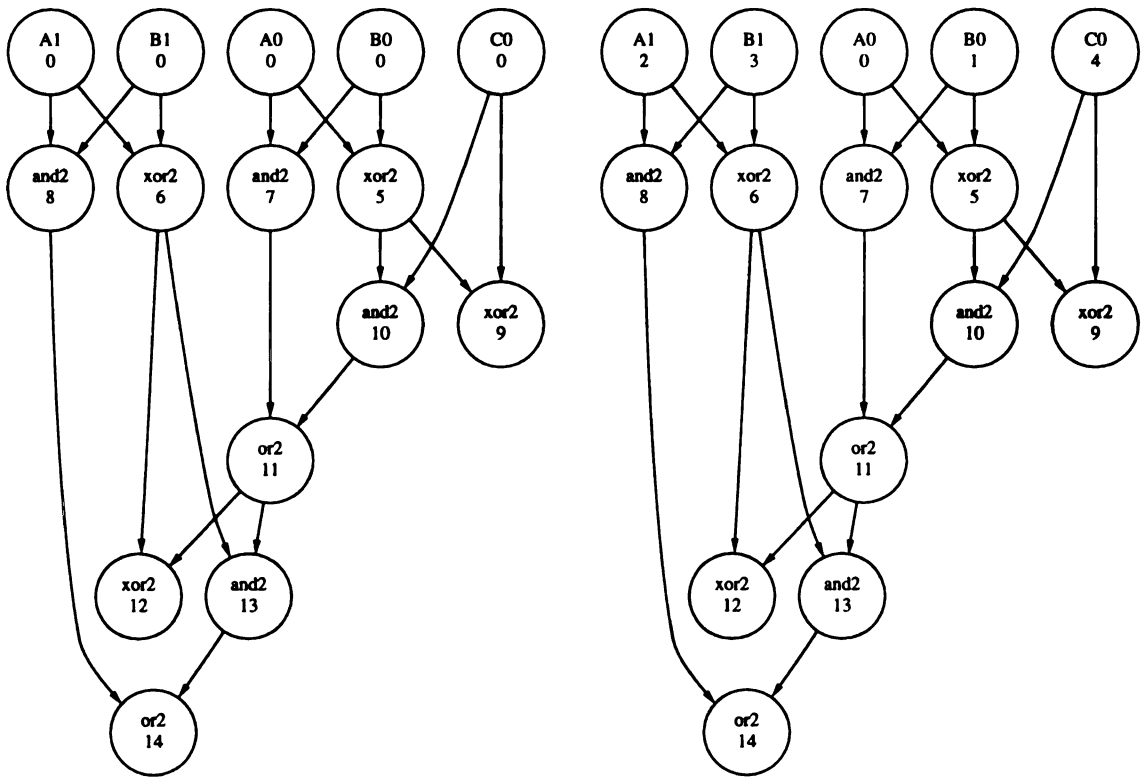


Figure 4.13: Vertex weighting for the circuit graph of the 2-bit full adder in Figure 4.10.

Identifier Generation

Identifier generation for this example is performed by the example algorithm in Figure 4.9. The results of this generation are displayed in Table 4.2.

Outputs The 2-bit full adder in Figure 4.10 has three outputs, represented by the vertices weighted 14, 12, and 9. Determination of the identifier begins with these vertices, in descending order of weight. These vertices will be referred to by their weight to allow them to be indicated clearly.

Vertex 14 has two input vertices, so those are sorted by weight, giving the result of [13, 8]. The identifier for Vertex 13 must be generated first. The recursive procedure considers the inputs of Vertex 13 and orders those [11, 6]. Vertex 11 is a net, so it is assigned the first net number, 0, and given the identifier "n0".

The module identifier for net n0 must be generated, so the traversal continues through the circuit graph. Vertex 10 is considered next, and the identifier "and2(n1, n2)" is generated for Vertex 10. Vertex 5 was given the net label "n1", and Vertex 4 was given the net label "n2" because that is the order in which they were encountered. The identifier eventually created for net n0 is "or2(and2(n1, n4), and2(n1, n2))".

During the consideration of the output Vertex 14, Vertices 11, 5, B0, A0, and C0 are all identified as nets, and assigned identifiers n0, n1, n2, n3, and n4, respectively.

This depth-first procedure continues until each vertex has been assigned an identifier. Each of the outputs is processed in this way. If a vertex is encountered that already has an identifier assigned, the identifier does not need to be recalculated and is simply returned. The results of the vertex identifier generation are presented in

Weight	Vertex	Level	Inputs	Identifier	Net Label
0	A0	0		X	n3
1	B0	0		X	n2
2	B1	0		X	n7
3	A1	0		X	n6
4	C0	0		X	n4
5	xor2	1	1, 0	xor2(n2, n3)	n1
6	xor2	1	3, 2	xor2(n6, n7)	n5
7	and2	1	1, 0	and2(n2, n3)	
8	and2	1	3, 2	and2(n6, n7)	
9	xor2	2	5, 4	xor2(n1, n4)	
10	and2	2	5, 4	and2(n1, n4)	
11	or2	3	10, 7	or2(and2(n1, n4), and2(n2, n3))	n0
12	xor2	4	11, 6	xor2(n0, n5)	
13	and2	4	11, 6	and2(n0, n5)	
14	or2	5	13, 8	or2(and2(n0, n5), and2(n6, n7))	
	top	6	14, 12, 9	(or2(and2(n0, n5), and2(n6, n7)),	
				xor2(n0, n5), xor2(n1, n4))	

Table 4.2: Vertex Identifiers for Circuit Graph in Figure 4.10

Table 4.2.

4.4 Subcircuit Equivalence Classes

The structural identifier can be used to determine an equivalence relation to partition the subcircuits generated by candidate subcircuit enumeration into structural equivalence classes. There are several reasons why finding these equivalence classes is desirable.

Equivalence classes allow syntactic matching to be performed transparently as part of the candidate subcircuit enumeration process. Each time a subcircuit is identified as belonging to a structural equivalence class, it is identified to be syntactically, and therefore semantically, equivalent to every other subcircuit in that equivalence class.

A subcircuit inherits all matching information from its equivalence class. Only one member of each equivalence class must be tested for functional equivalence to library modules. This saves time during module identification. After one instance has been tested, whenever another member of that equivalence class is located in the circuit, its equivalence to a functional module is already known.

The set of structural equivalence classes generated by the structural identifiers created using the ordering presented in this chapter is not necessarily minimal. There may be more than one identifier that represents a single circuit structure. However, all of the subcircuits within an equivalence class will be identical, and the number of equivalence classes that hold a single circuit structure is generally small (one or two), so equivalence classes provide a useful technique for improving the performance of module identification.

4.4.1 Local Information

To apply structural identifiers to the subcircuits generated by candidate subcircuit enumeration, a few preliminary steps are necessary to prepare the subcircuit for identifier generation. The information used to build the identifier is derived entirely from the structure of the circuit being examined, so the information in a subcircuit must be local with respect to that subcircuit, not the entire circuit graph.

If the circuit to be explored, H, is a subcircuit of another circuit G, only the information local to the subcircuit is considered. For instance, if a gate has two outputs in G, but only one of those outputs appears in H, then the gate is considered

to be a single output gate for the purpose of vertex weighting and identifier generation

Vertex level is an example of local circuit information. Although a vertex may be Level 5 in the original circuit, it may only be Level 4 in a subcircuit that does not include the inputs to the original circuit. It is necessary to determine the local level for each vertex, to ensure that two structurally identical subcircuits are represented identically as circuit graphs.

In addition, the inputs and outputs of the circuit must be specified as such. Inputs, outputs, and levels can be determined by iterating through the constituent vertices and declaring vertices with no local inputs as inputs and those with no local outputs as outputs. Level can be determined by labeling the local inputs as Level 0, the levels of their outputs as 1, and so on, labeling each vertex with the highest level at which it is encountered during this traversal.

All circuits considered for identifier generation are expected to contain only fully-specified vertices. That is, all inputs of internal gates must be present. The set of outputs considered during vertex weighting contains only those outputs that are also within the subgraph.

4.4.2 Correctness of Structural Equivalence Classes

Two graphs $G1$ and $G2$ with adjacency matrices A1 and A2 are *isomorphic* if there exists some permutation P for which $P(A1) = A2$.

Two circuits $C1$ and $C2$ with augmented adjacency matrices (Definition 4.1) A1 and A2 are *structurally identical* (isomorphic) if there exists some permutation P for

which $P(A1) = A2$.

For clarity, consider the structural identifier of a circuit to be an augmented adjacency matrix that describes the graph. The unique weights assigned to each vertex during vertex weighting define a specific permutation of the vertex ordering for the augmented adjacency matrix. Any legal permutation of an adjacency matrix still represents the same graph; a single adjacency matrix can only describe a single graph structure, by definition.

Non-isomorphic Circuits Must Have Different Identifiers

Theorem 4.2 *If two augmented adjacency matrices A1 and A2 are identical then the circuits C1 and C1 that they represent are structurally identical.*

Proof: Assume that two circuits C1 and C2 that are not structurally identical exist for which the corresponding augmented adjacency matrices A1 and A2 are identical. As C1 and C2 are not structurally identical, there exists no permutation P of A2 such that P(A1) = A2. Thus, A1 cannot be equal to A2, which contradicts the assumption that A1 and A2 are identical. Hence, by contradiction, if two augmented adjacency matrices are identical then the circuits that they represent are structurally identical.

For the purpose of defining subcircuit equivalence classes, it is vital that each equivalence class contain only those graphs whose structure is isomorphic. Theorem 4.2 states that if the identifiers (adjacency matrices) are identical, then the graphs must be identical. Therefore, two non-isomorphic graphs cannot be placed in the same equivalence class (have the same identifier).

It is not possible for two non-isomorphic graphs to be represented by the same adjacency matrix, so they may not be placed in the same equivalence class. This is important, because it is necessary that no subcircuit be placed into an incorrect equivalence class. That situation would result in incorrect identification of that subcircuit.

Each Subcircuit Structure is Generally Represented by Exactly One Identifier

Although not strictly necessary for the reverse engineering of digital circuits, it is interesting to note that the vertex weighting assigned by the weighting algorithm usually identifies a particular permutation of the vertices based on the structure of the graph, allowing fast isomorphism testing.

The weighting proposed in this chapter restricts the number of vertex permutations represent isomorphic graphs, so the process provides a significant improvement in the context of module identification. The number of vertex permutations that may be defined for a subcircuit is less than or equal to the number of arbitrary decisions that are made. Although arbitrary decisions may arise frequently in general directed acyclic graphs, they are uncommon in this domain due to vertex functionality labels and the irregular nature of subcircuits.

Original Circuit	Gates	Candidate Subcircuits			Contained Subcircuits		
		number	classes	time	number	classes	time
1-bit Adder	8	18	15	0.01	6	6	<0.01
2-bit Adder	15	114	79	0.04	24	17	0.01
3-bit Adder	22	522	339	0.20	66	41	0.04
z4ml3	39	185,196	164,172	177.11	281	275	0.24
C17	24	3,951	2,937	1.4	1199	1021	0.47
majority	24	12,171	12,171	5.25	993	463	0.61
b1	25	19,980	10,290	8.90	143	105	0.08
cm138a	33	726,032	124,174	516.56	26,652	6402	25.66
cm152a	35	59,962	15,692	44.90	6,484	6158	6.49

Table 4.3: Structural Equivalence Classes Applied to Subcircuits and Contained Subcircuits

4.5 Results

Experiments were run on a Sun Sparc Ultra-2 server with 1024MB of memory. Results are shown in Table 4.3 for several small circuits from the LogicSynth93 benchmark suite [82] (see Chapter 5 for discussion application on larger circuits). Times are given in CPU seconds. Refer to Table 3.1 on page 136 to see the execution effort of performing subcircuit enumeration without structural equivalence classes.

These results demonstrate the reduction in the number of subcircuits that must be tested for semantic equivalence to known modules. By applying a solution to the Subcircuit Identification problem to only one instance of each structural equivalence class, the overall performance of the Module Identification can be improved significantly, depending on the degree of regularity present in the circuit.

4.5.1 Implementation

To investigate the application of the structural equivalence classes within the Module Identification Problem, the two steps of the structural identifier generation, vertex weighting and identifier generation, have been implemented in the C programming language. The implementation of the vertex weighting procedure follows the algorithm presented in Section 4.2.4, and the identifier generation implements the example algorithm in 4.3.

The vertex weighting and structural identifier generation techniques are applied to the subcircuits generated by the code that implements the candidate subcircuit enumeration technique described in Chapter 3. This allows the subcircuits to be placed into equivalence classes, which are stored in a hash table with the structural identifier as the key value.

Correctness

Vertex Weighting A case analysis of the vertex weighting technique has been performed in Section 4.2.3. The implementation of the algorithm performs the sorting based on the precedence criteria described in Section 4.2.2 by using case statements to implement each individual criterion. Vertex weighting, based on the technique presented in Section 4.2 was performed by hand for many small circuits and these results were determined to be equivalent to the results of executing the implementation on the same circuits.

Structural Identifier Generation Again, many small experiments were performed by hand to ensure that correct structural identifiers were generated for the subcircuits.

A testing phase was conducted after the implementation of the two algorithms. Once a subcircuit had been placed into an equivalence class, it was compared via a backtracking breadth-first traversal, against all other members of the equivalence class to ensure that they are all structurally equivalent. No subcircuits were incorrectly placed.

Summary Although formal methods were not applied to ensure equivalence between the implementations and the algorithms performing the tasks necessary for subcircuit equivalence classes, comparison of results with hand-determined results and testing and verification within the implementations offer a great level of confidence in the correctness of the implementations. These implementations provide validation of the theory of structural equivalence classes.

4.6 Summary

This chapter has presented a technique for developing a structural identifier for a digital circuit. By employing a strict precedence framework, the vertices within the circuit graph that represents the circuit can be sorted so that each may be assigned a unique weight. With the exception of vertices that perform identical functionality with respect to the global structure of the circuit, this weighting will usually generate

identical weightings for structurally isomorphic circuits.

This weighting may then be applied within an identifier generation method to develop a structural identifier. An identifier represents exactly one circuit structure. Within the context of the Module Identification Problem, these structural identifiers allow the candidate subcircuits generated by the technique in Chapter 3 to be partitioned into structural equivalence classes so semantic matching may be performed on only one instance of each equivalence class. This process simultaneously performs syntactic matching and allows subcircuits to inherit information from their equivalence class.

Although structural identifiers can become quite large, their application within Module Identification is possible because the subcircuits to which they are applied are generally small. Both vertex weighting and identifier generation are easily applied to these subcircuits.

In addition, structural equivalence classes are equivalent to the *template libraries* used within regularity extraction, and the members of the classes describe instances of that template within the target circuit. Thus, by developing these equivalence classes, the first two steps of regularity extraction (Section 2.4.1) are solved.

Chapter 5

Practical Application of Techniques

It is apparent from considering the results presented in Sections 3.3 and 4.5 that applying these techniques unmodified to the circuit to be reverse engineered is not a feasible approach. Circuits of interest may be millions of gates in size, and considering that the number of subcircuits can potentially suffer from exponential growth with each additional vertex, circuits of this size cannot be handled by the techniques as they have been presented thus far. However, this does not preclude the use of these techniques in practical application. The techniques and guidelines presented in this chapter make it possible to effectively apply subcircuit enumeration within Module Identification of any size of circuit.

Details of a parallel implementation of the techniques are presented in Section 5.1. Section 5.2 discusses two methods of reducing the circuit graph order: preliminary partitioning and preliminary syntactic matching. A technique for reducing the theoretical complexity and execution time of subcircuit enumeration is presented in Section 5.3. Hierarchical module identification is presented in Section 5.4, and heuris-

tics for reducing the applications of Subcircuit Identification are discussed in Section 5.5. Section 5.6 summarizes the chapter.

5.1 Parallel Implementation

The solution to the subcircuit enumeration problem is "embarrassingly parallel". Each vertex of the circuit graph may be expanded individually, and this expansion proceeds without interaction with any other expansion processes. By developing a parallel implementation, the execution time required by subcircuit enumeration technique can be vastly improved.

The computation effort can be divided into many smaller parts by expanding each vertex of the circuit graph on individual processors to enumerate all candidate subcircuits derived from that vertex. After each subcircuit is created and labeled with a structural identifier, it can be returned to the server to be placed into the proper subcircuit equivalence class for subcircuit identification.

The results of performing candidate subcircuit enumeration with a parallel implementation of the Focused Enumeration Algorithm (Figure 3.15 on page 116) on circuits in the LogicSynth Benchmark Suite [82] are presented in Figure 5.1. The algorithm was implemented in C and uses the parallel capabilities of Parallel Virtual Machine (PVM) 3.3 [73]. The experiments were run with a Sun Ultra 2 Model 2300 (dual 300mhz processors) with 1024MB of memory. The client machines included Sun Ultra 10s and Sun Ultra 5s.

The CPU execution times are in seconds, and represent the maximum time nec-

Original Circuit	Gates	Candidate Subcircuits			Contained Subcircuits		
		number	single time	parallel max time	number	single time	parallel max time
1-bit Adder	8	18	<0.01	<0.01	6	<0.01	
2-bit Adder	15	114	<0.01	0.01	24	<0.01	0.01
3-bit Adder	22	522	0.04	0.03	66	<0.01	0.01
C17	24	3,951	0.16	0.08	1,199	0.05	0.02
majority	24	12,171	0.45	0.21	993	0.05	0.01
b1	25	19,980	0.75	0.57	143	0.01	0.01
cm138a	33	726,032	27.64	11.14	26,652	1.05	0.41
cm152a	35	59,962	2.27	1.78	6,158	0.24	0.11
z4ml3	40	185,196	14.64	6.89	281	0.07	0.05
x2	54	104,178,928	4282.80	2118.66	98,322	3.93	2.15

Table 5.1: Focused Enumeration Results of Parallel Implementation

essary to enumerate all of the subcircuits from any vertex. The execution time for the smaller circuits suffered slightly due to the message passing overhead, but in the larger circuits, execution time was significantly improved.

Most of the processing time for enumeration of subcircuits is confined to the expansion of the vertices with the highest indices; these vertices are the ones which have the most possible vertices to add because of Rule 1 (page 110). By performing their expansion in parallel, overall execution time can be significantly reduced.

5.2 Circuit Graph Order Reduction

Candidate Subcircuit Enumeration performs an exhaustive generation of the interesting subcircuits within the circuit. Therefore, as the order of the circuit graph grows, the number of potentially interesting subcircuits can grow exponentially. To allow

subcircuit enumeration to be applied in practice, the order of the circuit graphs to be considered must be kept small (< 100 gates).

The following two techniques provide excellent methods for reducing the order of the circuit graph without interfering with the effectiveness of the Module Identification effort.

5.2.1 Preliminary Partitioning

Circuit partitioning is absolutely vital to applying candidate subcircuit enumeration to real circuits. Partitioning allows divide-and-conquer approaches to be applied to otherwise intractably large circuits, and allows many techniques to be applied more swiftly to any circuit. Partitioning has been used to achieve higher quality results in smaller time for standard cell placement, or to determine the appropriate partitions to be designed as custom ASICs.

A considerable amount of research has focused on the development of algorithms to perform effective circuit partitioning. Most current algorithms are based on one of three original circuit partitioning algorithms: Kernigan and Lin [67], Fiduccia and Mattheyses [45], or Krishnamurthy [70] and have resulted in many useful partitioning techniques that may be applied to reduce the complexity of module identification. Several surveys are available [3, 62]. Current research in partitioning techniques for VLSI focuses on the partitioning of hypergraphs [26, 27, 66], so to take advantage of current techniques it would be necessary to describe the circuit as a hypergraph for the purpose of preliminary partitioning.

Partitioning techniques for digital circuits focus on decomposing a complex system into smaller parts, while minimizing the communication between partitions. The result of this particular optimization is that functional modules, which are generally more strongly connected within themselves than to other modules, generally are completely contained within a partition.

The advantage of partitioning the circuit previous to performing Module Identification is that it reduces the order of the circuit graph, thus greatly reducing the number of candidate subcircuits that can exist within the circuit graph, yet it is not likely to make partitions that would render functional modules unidentifiable.

5.2.2 Preliminary Syntactic Matching

It is possible that stock implementations of some basic modules exist within the circuit. Syntactic matching techniques are very fast, so examining the circuit for likely implementations of modules that may exist in the circuit is a reasonable first step. Vertices that have been identified as being a part of a functional module need not be considered for inclusion in other modules, thus reducing the number of remaining subcircuits.

This technique may pose a difficulty if the circuit has been optimized so that the functionality of a cluster of gates is shared by two modules. If the cluster has been identified as being part of one of them, the second will not be located because those vertices are not free to be part of a subcircuit. This situation could be detected by an unreasonably high percentage of unmatched gates after module identification, and

would require intelligent conflict resolution techniques, most likely the consideration of a human engineer.

5.3 Subgraph Order Limiting

Basic functional modules are rarely large; the largest modules are generally composed of smaller modules. It is reasonable, therefore, to limit the order of enumerated subgraphs to allow faster exploration of the circuit graph. Effort spent building subcircuits with hundreds of gates is essentially wasted.

Allowing enumeration to include excessively large subcircuits precludes the application of this technique to large circuits because the number of potential subcircuits may grow exponentially with the size of the circuit. A better solution is to restrict the number of vertices that may be part of a subcircuit.

When the module library is designed, an order limiting guideline may be determined for each, based on common implementations and designer input. This is a preprocessing step that can make use of external knowledge. At runtime, the order limit for the module identification may be set to be the highest suggested order limit of the library modules of interest.

It is certainly possible that a module could have been designed with an excessive number of gates to achieve obfuscation, redundancy, or some other design goal. In this case, the module may not be identified as a functional module if the order limit is imposed. The system or an engineer may make a decision to raise the order limit after an exploration of the circuit based on the number of unmatched logic devices

remaining.

This heuristic is applied during steps 1 and 3 of the **subgraphExpand** subroutine of the subcircuit enumeration algorithm described in Section 3.2.3. Before a vertex is added to a subgraph, the subgraph is tested to determine whether the subgraph has already reached the order or depth limit. These values are calculated as vertices are added and stored within the subgraph, so only a simple equivalence test is necessary. The algorithm with order limiting enforced is presented in Figure 5.1.

Order Limited Focused Enumeration Algorithm

0. Transform circuit into circuit G with unique vertex indices.
1. **foreach** $v \in V(G)$:
   2. Create a subgraph H_0 with $V(H_0) \leftarrow v$.
   3. Determine $\mathcal{F}^R(H_0)$.
   4. expandSubgraph(H_0).
5. End **foreach**.

subroutine expandSubgraph(graph H_i)
0. **foreach** vertex $v_{i+1} \in \mathcal{F}^R(H_i)$ (Rule 1):
   1. Create a subgraph H_{i+1} such that $V(H_{i+1} \leftarrow V(H_i) + v_{i+1}$.
      If order of H_{i+1} exceeds order limit, discard H_{i+1} and return.
   2. Determine $\mathcal{F}^R(H_{i+1})$.
   3. Add vertices necessary to make H_{i+1} into a subcircuit (Rule 2).
      If order of H_{i+1} exceeds order limit at any time, discard H_{i+1} and return.
   4. Output H_{i+1}.
   5. expandSubgraph(H').
6. End **foreach**.

Figure 5.1: Algorithm for Order Limited Focused Enumeration.

Theoretical Complexity Improvement By enforcing an order limit on the subcircuits that may be produced, the number of possible subcircuits can be significantly

Original Circuit	Gates	Candidate Subcircuits		Order Limit	Candidate Subcircuits	
		number	time		number	time
3-bit Adder	22	522	0.04	10	28	0.01
C17	24	3,951	0.16	10	509	0.02
majority	24	12,171	0.45	10	1492	0.10
b1	25	19,980	0.75	10	4078	0.32
cm138a	33	726,032	27.64	10	2098	0.17
cm152a	35	59,962	2.27	10	1015	0.15
z4ml3	40	185,196	14.64	10	823	0.07
decod	48	N/A	-	10	7,450	0.97
decod	48	N/A	-	20	30,584,750	1230.68
x2	54	104,178,928	4282.80	10	3826	0.71
rd53	89	N/A	-	10	55,354	14.44
4b-alu	79	N/A	-	10	4,722	0.97
4b-alu	79	N/A	-	20	12,261,533	1926.01
cmb	86	N/A	-	20	3,616,868	431.79

Table 5.2: Order Limited Focused Enumeration Results

reduced. In fact, the complexity of the subcircuit enumeration algorithm is no longer exponential, because the size of the subgraphs is a constant value independent of the number of vertices in the circuit graph. The complexity of the algorithm in Figure 5.1 is $\mathcal{O}(2^c)$, where c represents the order limit imposed on subgraph expansion.

The results of executing an implementation of the limited order focused enumeration algorithm (Figure 5.1) on circuits in the LogicSynth Benchmark Suite [82] are presented in Figure 5.2. The algorithm was implemented in C. The experiments were run with a Sun Ultra 2 Model 2300 (dual 300mhz processors) with 1024MB of memory.

5.4 Hierarchical Module Identification

The execution effort to enumerate subcircuits can be greatly reduced by restricting the order of the subcircuits to be enumerated. Bottom-up module identification can provide a very effective and fast solution to Candidate Subcircuit Enumeration within the Module Identification Problem.

5.4.1 Module Replacement

To dynamically reduce the order of the circuit graph, as subcircuits are identified as equivalent to a library module, they may be replaced by a single vertex that encompasses all of the functionality contained in the subcircuit. This module replacement technique would allow hierarchical module identification. For instance, two appropriately connected vertices representing 1-bit full adders could be subsumed into a single vertex representing a 2-bit full adder. Replacing a cluster of vertices with a single vertex representing that cluster is a common technique in clustering [54].

It would be necessary to develop a method of differentiating between the outputs of a vertex, because they will not all represent the same function, as do multiple outputs from a logic gate. The outputs of the modules would need to be represented by vertices that are implicitly connected to the module vertex.

The difficulty with this technique is the possibility of premature subsumption of vertices that should be shared with another library module. When a module is replaced by a single vertex representing the functionality of the entire module, then its component vertices are no longer available to be members of another module. This

means that no shared functionality is permitted with other vertices. For this reason, it is best to replace only modules that are represented by contained subcircuits, or to perform fitness evaluation (discussed in Section 5.5 before replacement.

5.4.2 Primitive Functional Modules

The execution of subcircuit enumeration is greatly affected by the order of the subcircuits to be enumerated, which is dependent on the size of the modules to be located. To perform module identification hierarchically, a set of primitive modules should be developed. These modules would perform common functionalities that could be used to build larger functional modules. These primitive modules could reasonably be implemented by fewer than 10 gates and have simple functionality to enhance subcircuit identification performance.

By locating these primitive functional modules, gate clusters could be replaced by a single vertex representing the primitive functionality they perform. Each identification and replacement of a primitive module reduces the complexity of the circuit graph, and thus the performance of the module identification process.

This technique would suffer from the same problem that bottom-up clustering techniques encounter: covering the circuit correctly, so consideration would need to be given to that problem during the development and application of primitive modules. The identification of a comprehensive and effective set of functional primitives provides an interesting focus for future research into solutions to the Module Identification Problem, and would be an interesting theoretical study as well.

5.5 Subgraph Fitness Evaluation

It is possible to evaluate how likely a subcircuit is to be a functional module by taking the following information [32, 53] into consideration:

- the gates comprising a module are likely to be physically close together
  - Euclidean distance between gates
  - number of edges between gates
- modules are generally contained completely within a power/ground bounding box
- all components of a module are generally driven by the same branch of the clock tree
- two gates within a module are likely to be connected less strongly than two gates outside of a module

By using this additional information, a fitness value can be assigned to each subcircuit. If it is unlikely that the subcircuit will be a module or a part of a module, then the expansion of that subcircuit may halt. This prevents the enumeration of all of the descendent subcircuits that are also unlikely to represent a module. This approach can provide a significant reduction of effort, because only likely subcircuits will be tested for equivalence with functional modules.

5.6 Summary

Despite the inherently difficult problem of enumerating all of the subcircuits of a circuit so that their equivalence to a known module may be considered, the techniques presented in this chapter provide sufficient reduction and focus of the enumeration effort to allow practical use of the theoretical approaches presented in Chapters 3 and 4.

The benefit of using semantic matching and subcircuit enumeration to solve the module identification problem is that all modules in the library may be located, regardless of their implementation. This provides the most comprehensive covering of an unknown circuit and thus solves the Module Identification Problem.

Chapter 6

Contributions and Future Directions

Reverse engineering performs the reverse of the synthesis process to obtain a high-level description of a digital circuit from a low-level specification. For instance, a circuit may be reverse engineered from the transistor level to the gate level, or from the gate level to the register transfer level. In each case, the details of the specification are encapsulated to produce a more abstract, but still accurate, description of the circuit.

Reverse engineering is necessary for many reasons, including obtaining information to maintain a competitive edge, or to ensure that one's intellectual property has not been stolen. Reverse engineering is frequently the first step in the redesign or reengineering process, which is very important for replacement of parts to forestall obsolescence, or to allow reimplementation exploiting improved technology, or to allow fault repair and replacement.

One of the major focuses of reverse engineering is to raise the level of abstraction

from a lower- to higher-level specification. We have formulated this problem as follows:

Module Identification Problem [37, 38, 114]. Given a gate-level logic description (*netlist*) of a target circuit, efficiently identify all gate clusters (*subcircuits*) that perform the function of a known library module.

- **Candidate Subcircuit Enumeration Problem.** Identification of gate clusters (*candidate subcircuits*) within the netlist that may comprise a functional module.

- **Subcircuit Identification Problem.** Proving functional equivalence between a candidate subcircuit and a known standard library module.

Most existing techniques to perform the transformation from lower to higher levels rely on syntactic matching techniques to locate functional modules by identifying clusters of gates with identical structure to a known modules. Syntactic matching is a very rigid matching technique because it will only identify subcircuits as equivalent whose implementation is identical. Within the Module Identification Problem, syntactic matching requires that an extensive library of module implementations is available. The computation and memory cost of applying syntactic techniques is frequently prohibitive.

The approach to Module Identification presented in this thesis applies semantic matching instead, a more general matching technique that determines equivalence of subcircuits based on functionality, regardless of implementation. To locate all gate clusters that may be equivalent to a known module, it is necessary to enumerate all subcircuits of the circuit to be reverse engineered. The enumeration of these

candidate subcircuits for the purpose of module identification is the focus of the work in this thesis. The enumeration of all subcircuit of a circuit is a complex process. By ensuring that the enumeration is done efficiently, with no duplication of effort, the task is possible. By focusing the enumeration effort on only the gate clusters that represent functional segments of the circuit, the enumeration effort is further reduced.

Semantic matching can be computationally expensive, so it is beneficial to restrict the number of times that it must be applied. Subcircuits that are syntactically equivalent are necessarily semantically equivalent. Therefore, syntactically equivalent subcircuits may be grouped into structural equivalence classes, so that semantic matching must only be applied to a single instance of each subcircuit structure. To partition the enumerated subcircuits into structural equivalence classes, a structural identifier can be created for each subcircuit. This identifier provides the equivalence relation to identify the correct equivalence class. To allow the creation of an effective identifier, the vertices in the subcircuit are ordered based on structural invariants.

Overall, this work has provided several steps within the reverse engineering process presented in Figure 1.5. Focused subcircuit enumeration allows the application of semantic techniques to locate modules and raise the level of abstraction. Structural equivalence classes ensure that each subcircuit structure must only be considered once by semantic equivalence identification. Thus, the necessary applications of semantic matching to perform subcircuit identification are significant reduced without lack of efficacy.

6.1 Contributions

6.1.1 Module Identification Problem

We have developed a well-defined formulation of a specific step in the reverse engineering process: the location of modules to facilitate the transformation from a gate-level specification to a module-level specification (Section 1.3). This formulation defines two subproblems: Candidate Subcircuit Enumeration and Subcircuit Identification. The work presented in this thesis focuses on developing techniques to solve the Candidate Subcircuit Enumeration problem and reduce the number of necessary applications of the solution to the Subcircuit Identification Problem.

6.1.2 Unique Subgraph Enumeration

We have represented circuits by a *circuit graph*, a directed graph in which vertices represent gates or flip-flops and arcs represent wires and interconnects (Section 3.1). A naïve algorithm for subgraph enumeration was presented in Figure 3.5. This algorithm produces many identical subgraphs and is therefore very inefficient, but it provides a framework for developing a more focused algorithm. One of the focuses of the work described in Chapter 3 is on refinement of the algorithm to avoid duplication of subcircuits.

The duplicate subgraphs created by the naïve algorithm were eliminated by developing a rule to ensure that each subgraph was created exactly once (Section 3.2.1). A subset of the subgraph neighborhood, the *reachable frontier* (Definition 3.10) is defined. Instead of allowing any neighboring vertex to be added to the subgraph

during subgraph expansion, only vertices on the reachable frontier may be added. The reachable frontier is dependent on the index of the subgraph and the vertices that have already been added, and thus enforces an ordering in which vertices may be added to create subgraphs. The unique subgraph enumeration algorithm is presented in Figure 3.11 and the proof that this algorithm enumerates each subgraph exactly once is presented in Section 3.2.3.

6.1.3 Restricted Subgraph Types

The number of subgraphs in any directed graph can be prohibitively large. Even efficient unique enumeration of subgraphs is not an acceptable solution to the Candidate Subcircuit Enumeration Problem. Because candidate subcircuit enumeration is applied within a specific domain, in which subgraphs represent gate clusters in a digital circuit, enumeration may be focused on subsets within the class of subgraphs: subcircuits and contained subcircuits.

Subcircuits

A significant restriction on the number of subgraphs that must be enumerated can be obtained by focusing the enumeration only on subgraphs that may possibly be equivalent to a known module. A subgraph representing a non-functional cluster of gates is not interesting with regard to the Module Identification Problem. Only functional gate clusters, *subcircuits* (Definition 3.6), need to be enumerated.

A subcircuit is a subgraph in which none of the vertices include a proper subset of the vertices representing its input gates. It is important to consider these subgraphs

because only they have functional meaning. If only one input of a 2-input AND gate is known, then the functionality of the AND gate is ambiguous. Therefore, the functionality of the circuit represented by any subgraph that contains one of these ambiguous vertices is also ambiguous.

We have modified the unique subgraph enumeration algorithm to focus on enumerating only subcircuits; this algorithm is presented in Figure 3.15. As soon as a vertex is added to a subgraph, all of the vertices that are necessary to make the subgraph a subcircuit are added (provided that they are on the reachable frontier). Subgraphs that cannot have functional meaning are not investigated or enumerated. This provides a significant reduction in computational effort, as evidenced by the results in Section 3.3.

Contained Subcircuits

When circuits are designed using standard cell libraries, each cell or module is placed into the circuit and connected together with glue logic. The logic gates that comprise these modules are connected exclusively to each other, without sharing their functionality with other modules. The modules are represented within a circuit graph by a *contained subcircuit* (Definition 3.8).

With regard to the Module Identification Problem, a standard cell or module is represented by a contained subcircuit. A contained subcircuit is a subcircuit in which none of the vertices have only a partial set of their outputs also in the subcircuit. If a vertex has an output arc that leaves the subcircuit, then all of its output arcs must also leave the subcircuit, which defines the vertex as an output of the subcircuit. If

all of the output arcs lead to vertices within the subcircuit, then the vertex represents an internal gate. If all of the output arcs lead to vertices outside of the subcircuit, then the vertex represents an output of the subcircuit.

The number of contained subcircuits is significantly smaller than the number of subcircuits, so the enumeration of this set can be performed more quickly than the entire set of subcircuits. Although contained subcircuit enumeration is not sufficient to locate all modules in a circuit, it provides a useful preliminary matching technique for reducing the order of the entire circuit graph, thus reducing the execution effort necessary to locate all remaining modules by applying subcircuit enumeration.

6.1.4 Circuit Structural Identifier

Another objective of the work presented in this thesis is to reduce the number of times that the solution to the Subcircuit Identification subproblem must be applied. Many of the subcircuits enumerated by Candidate Subcircuit Enumeration will be structurally identical, and are therefore semantically identical. It is only necessary to determine equivalence of a single instance of each set of structures. These are called *structural equivalence classes*, presented in Chapter 4.

We have developed a method for determining an effective structural identifier for the structure of a circuit, which defines an equivalence relation to partition the subcircuits into a set of structural equivalence classes. To allow the creation of these identifiers, a well-defined vertex ordering based on structural invariants of the vertices within the circuit graph was developed. The set of equivalence classes is not minimal,

because more than one identifier may represent a single structure, but it is an effective reduction technique for module identification.

This ordering is developed by sorting the vertices at each level of the circuit graph using information about the vertices themselves, their input vertices, and their output vertices, then applying a unique weight that represents the vertex ordering. If that information is not enough to induce a total ordering on the vertices, then a second pass through the circuit graph is performed, taking advantage of the information developed during the first pass. After the vertices have been sorted and weighted by the algorithm presented in Figure 4.2.4, a total ordering has been defined for the set of vertices in the circuit graph.

An example of a structural identifier generation technique is provided that creates a prefix logic formula that uniquely describes the circuit structure. Interconnection information must be preserved correctly, so *nets* are used to represent networks that appear more than once within the structural identifier. These identifiers can be used to determine appropriate structural equivalence classes for the subcircuits, thus reducing the required applications of subcircuit identification.

6.1.5 Heuristics for Practical Application of Candidate Enumeration

The enumeration and structural classification of candidate subcircuits provides significant improvement to the application of module identification, but to handle realistic circuits containing thousands or millions of gates, several heuristic techniques must

be applied.

The limiting element of candidate subcircuit enumeration is the order of the circuit graph that represents the target circuit. In Section 5.2, we described several techniques for reducing target circuit order, such as preliminary partitioning, preliminary syntactic matching coupled with module replacement, and hierarchical module replacement throughout the module identification process. Another factor that affects the complexity of the enumeration process is the order of the subcircuits that must be enumerated. We described and justify an order limiting heuristic in Section 5.3 that is based on a reasonable order derived from the size of the library modules.

The process of candidate subcircuit enumeration is embarrassingly parallel because expansion of a vertex into its descendent candidate subcircuits is a discrete operation for each vertex in the circuit graph, so employing a parallel implementation of the enumeration algorithm reduces overall execution time for larger circuit graphs (Section 5.1) without duplication of effort.

6.2 Future Directions

6.2.1 Candidate Subcircuit Enumeration

Future work within the Candidate Subcircuit Enumeration solution includes a possible improvement to the implementation of the focused enumeration algorithm.

Rules 1 and 2, when producing subcircuits or contained subcircuits, respectively, occasionally must discard a subgraph because Rule 1 prohibits the addition of a ver-

tex or vertices necessary to make constituent vertices fully specified or contained. Although the number of subgraphs discarded is small, and the computation improvement provided by Rules 2 and 3 far outweighs this cost, an ideal solution would proceed without any discarded subcircuits. This may be possible by implementing a look-ahead algorithm to investigate the validity of a creation path before the actual subgraph duplication and expansion is performed.

6.2.2 Module Identification Problem

The techniques described in this thesis have been designed to be applied within the solution to the Module Identification Problem. There are several improvements that may be developed that would allow module identification to be applied to circuits of any size. The limitation at this time is the order of the circuit graphs that may undergo candidate subcircuit enumeration because the number of subcircuits increases rapidly with the order of the graphs. By implementing the following techniques described in Chapter 5, circuits of any size may be reverse engineered by the application of Module Identification.

- *Preliminary partitioning* (Section 5.2.1): The application of a partitioning technique to effectively reduce the order of the circuit graphs to be explored may be applied. If effective partitioning techniques are applied, each partition may be considered individually without interfering with module identification.

- *Module replacement* (Section 5.4.1): Subcircuits that have been found to represent functional modules in the circuit may be replaced by a single vertex that

represents the functionality of the subcircuit. This reduces the order of the overall circuit graph, and thus decreases the number of candidate subcircuits in the circuit.

- *Hierarchical module identification* (Section 5.4): The efficiency of enumerating candidate subcircuits is strongly dependent on the size of the subcircuits that must be enumerated. By locating smaller modules first, and replacing them with a single vertex, and then considering larger modules that may contain those modules, module identification can be performed hierarchically. This allows the order limit of the enumerated subcircuits to be kept small, thus allowing faster identification of modules.

- *Subgraph fitness evaluation* (Section 5.5): Development of an effective fitness function to determine the likelihood that a subcircuit could comprise a functional module is a future focus for improving the module identification process.

6.2.3 System Integration

To allow these techniques to be applied easily to real circuits, many techniques would need to be integrated to create an effective reverse engineering tool. The following systems, presented in Figure 1.5 on page 25 need to be designed and implemented around the solution to Candidate Subcircuit Enumeration presented in this thesis.

- *Netlist partitioning*: There are existing partitioning systems that can be applied at the beginning of the reverse engineering process to allow a divide-and-conquer approach to module identification.

- *Combinational logic extraction*: The current solutions available for semantic matching are unable to perform equivalence comparisons for sequential circuits. To allow module identification to be performed, the combinational logic should be extracted from the circuit.

- *Subcircuit enumeration*: Enumeration of candidate subcircuits that may be equivalent to a higher-level functional module.

- *Structural equivalence classes*: Syntactically equivalent subcircuits are also semantically equivalent. This step places the candidate subcircuits into structural equivalence classes to eliminate extraneous applications of semantic matching to syntactically equivalent subcircuits.

- *Semantic matching*: Functional equivalence comparison of an instance of a subcircuit equivalence class with known functional modules.

- *Circuit covering*: If desired, an engineer can view the high-level modules and determine which ones comprise the best covering of the circuit. In general, modules are likely to share logic, so this step is not necessary. Modules may overlap in the module-level representation of the circuit and still present a useful and accurate view of the circuit.

6.3 Conclusion

The work presented in this thesis provides techniques to solve the Candidate Subcircuit Enumeration Problem within the Module Identification Problem. Efficient

enumeration of only those subgraphs of the circuit graph that may be equivalent to a known module, candidate subcircuits, is the focus of this work. To provide further improvement, the candidate subcircuits are placed into structural equivalence classes by creation of a structural identifier. One instance of each class must be tested for equivalence to known modules. All other members of the class inherit matching results.

These techniques, when applied in conjunction with a semantic matching solution to the Subcircuit Identification Problem, provide the basis of a tool to perform the reverse engineering of digital circuits.

Bibliography

Bibliography

[1] Alfred V. Aho and Neil J. A. Sloane. Some doubly exponential sequences. *Fibonacci Quarterly*, 11:429–437, 1973.

[2] Sherri Al-Ashari. System verification from the ground up. *Integrated System Design*, January 1999.

[3] Charles J. Alpert and Andrew B. Kahng. Recent directions in netlist partitioning: A survey. *Integration: The VLSI Journal (1995)*, 19:1–81, 1995.

[4] Charles J. Alpert and Andrew B. Kahng. A general framework for vertex orderings with applications to netlist clustering. *IEEE Transactions of VLSI Systems*, 4(2):240 – 246, 1996.

[5] Srinivasa R. Arikati and Ravi Varadarajan. A signature based approach to regularity extraction. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 542 – 545, 1997.

[6] Peter J. Ashenden. *The Designers Guide to VHDL.* Morgan Kauffman Publishers, 1995.

[7] Mikhail J. Atallah. *Algorithms and Theory of Computation.* CRC Press LLC, 1998.

[8] E. Augustus. VLSI circuit layer determination by reflectance for use in reverse engineering. Master's thesis, Air Force Institute of Technology, 1990.

[9] Pradeep Batra and David Cooke. Hcompare: A hierarchical netlist comparison program. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 299 – 304, 1992.

[10] Carter Bays. A non-recursive technique for recreating a digraph from its K-formula representation. *The Computer Journal*, 19(4):326 – 329, 1976.

[11] C. Benecke, Adalbert Kerber, and Reinhard Laue. Canonical numbering of 3D-molecules. In *Proceedings of the International Electronic Conference on Computational Chemistry*, November 1995.

[12] J. D. Benstock, D. J. Berndt, and K. K. Agarwal. Graph embedding in synchem2, an expert system for organic synthesis discovery. *Discrete Applied Mathematics*, 19:45 – 63, 1988.

[13] Alfs T. Berztiss. *Data Structures: Theory and Practice.* Academic Press, Inc., second edition, 1972.

[14] Alfs T. Berztiss. A backtrack procedure for isomorphism of directed graphs. *Journal of the ACM*, 20(3):365 – 372, July 1973.

[15] J. Bhasker. *A Verilog HDL Primer.* Star Galaxy Publishers, second edition, 1999.

[16] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. Program assignment and the concept assignment problem. *Communications of the ACM*, 37(5):72 – 82, May 1994.

[17] Michael Boehner. LOGEX – an automatic logic extractor from transistor to gate level for CMOS technology. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 517 – 522, June 1988.

[18] Daniel R. Brasen and Gabrièle Saucier. Using cone structures for circuit partitioning into fpga packages. In *Proceedings of the Computer Aided Design of Integrated Circuits and Systems*, pages 592 – 600, July 1998.

[19] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677 – 691, August 1986.

[20] Randal E. Bryant. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 236 – 243, 1995.

[21] Anthony Bumbalough. Electronic parts obsolescence initiative workshop. Technical report, Department of the Air Force, Air Force Research Laboratory, Wright-Patterson Air Force Base, Dayton, OH., April 2000.

[22] H. Bunke and Messmer. Recent advances in graph matching. *International Journal of Pattern Recognition and Artificial Intelligence*, 11(1):169 – 203, 1997.

[23] Jerry R. Burch and Vigyan Singhal. Robust latch mapping for combinational equivalence checking. In *Proceedings of theIEEE International Conference on Computer-Aided Design*, pages 563 – 569, 1998.

[24] Eric J. Byrne. A conceptual foundation for software re-engineering. In *Proceedings of the Conference on Software Maintenance*, pages 226 – 235, November 1992.

[25] Andrew E. Caldwell, Hyun-Jin Choi, Andrew B. Kahng, Stefanus Mantik, Miodrag Potkonjak, Gang Qu, and Jennifer L. Wong. Effective iterative techniques for fingerprinting design IP. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 843 – 848, June 1999.

[26] Andrew E. Caldwell, Andrew B. Kahng, Andrew A. Kennings, and Igor L. Markov. Hypergraph partitioning for VLSI cad: Methodology for heuristic development, experimentation and reporting. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 349 – 354, June 1999.

[27] Andrew E. Caldwell, Andrew B. Kahng, and Igor L. Markov. Hypergraph partitioning with fixed vertices. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 355 – 359, June 1999.

[28] Gary Chartrand and Ortrud R. Oellermann. *Applied and Algorithmic Graph Theory.* McGraw-Hill, 1993.

[29] J. K. Cheng and T. S. Huang. A subgraph isomorphism algorithm using resolution. *Pattern Recognition*, 13:371 – 379, 1981.

[30] Elliot J. Chikofsky and James H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13 – 17, January 1990.

[31] Chipworks. Circuit analysis. Available on the world wide web at http://www.chipworks.com/.

[32] Gregory H. Chisholm, Steven T. Eckmann, Christopher M. Lain, and Robert L. Veroff. Understanding integrated circuits. *IEEE Design & Test of Computers*, 16(2):26 – 37, April - June 1999.

[33] Amit Chowdhary, Sudhakar Kale, Naresh Sehgal, and Rajesh Gupta. A general approach for regularity extraction in datapath circuits. In *Proceedings of the 1998 International Conference on Computer Aided Design*, pages 332 – 340, November 1998.

[34] Nicos Christofides. *Graph Theory: An Algorithmic Approach.* Academic Press Inc., 1975.

[35] D. G. Corneil and C. C. Gotlieb. An efficient algorithm for graph isomorphism. *Journal of the ACM*, 17(1):51 – 64, January 1970.

[36] Travis E. Doom. *Design Recovery for Combinational Logic Exploiting Boolean Relationships.* PhD thesis, Michigan State University, East Lansing, MI, May 1998.

[37] Travis E. Doom, Jennifer L. White, Gregory Chisholm, and Anthony S. Wojcik. Identification of functional components in combinational circuits. Technical Report ANL/DIS/TM-47, Argonne National Laboratory, January 1998.

[38] Travis E. Doom, Jennifer L. White, Anthony S. Wojcik, and Gregory H. Chisholm. Identifying high-level components in combinational circuits. In *Proceedings of the 1998 Great Lakes Symposium on VLSI*, pages 313–318, February 1998.

[39] M. A. Dukes. A generalized extraction system for VHDL. In *Proceedings of the Conference on Advances in Modeling and Simulation*, pages 165 – 171, 1994.

[40] Paul J. Durand, Rohit Pasari, Johnnie W. Baker, and Chun che Tsai. An efficient algorithm for similarity analysis of molecules. *Internet Journal of Chemistry*, 2, June 1999.

[41] Carl Ebeling. GeminiII: A second generation layout validation tool. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 322–325, 1988.

[42] Carl Ebeling and Ofer Zajicek. Validating VLSI circuit layout by wirelist comparison. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 172 – 173, September 1983.

[43] Steven T. Eckmann and Gregory H. Chisholm. Assigning functional meaning to digital circuits. Technical Report ANL/DIS/TM-43, Argonne National Laboratory, July 1997.

[44] Adrian Evans, Allan Silburt, Gary Vrckovnik, Thane Brown, Mario Dufresne, Geoffrey Hall, Tung Ho, and Ying Liu. Functional verification of large ASICs. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 650 – 655, June 1998.

[45] C. Fiduccia and R. Mattheyses. A linear time heuristic for improving network partitions. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 175 – 181, 1982.

[46] The Center for Software Maintenance. The maintainer's assistant: A tool for reverse engineering and re-engineering code using formal methods. Available on the world wide web at http://www.dur.ac.uk/CSM/projects/ma/.

[47] E. Fretheim. Reverse engineering VLSI using pattern recognition techniques. Master's thesis, Air Force Institute of Technology, 1988.

[48] Daniel D. Gajski. *Principles of Digital Design.* Prentice-Hall, Inc., 1997.

[49] Gerald C. Gannod and Betty H.C. Cheng. Strongest postcondition semantics as the formal basis for reverse engineering. *The Journal of Automated Software Engineering*, 3(1), 1996.

[50] Michael R. Garey and David S. Johnson. *Computers and Intractability.* W. H. Freeman and Company, 1979.

[51] Thomas Grüner, Adalbert Kerber, Reinhard Laue, and Andri Ruckdeschel. MOLGEN. Available on the web at http://www.mathe2.uni-bayreuth.de/molgen/.

[52] Lars Hagen and Andrew B. Kahng. A new approach to effective circuit clustering. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 422 – 427, 1992.

[53] Mark C. Hansen, Hakan Yalcin, and John P. Hayes. Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering. *IEEE Design & Test of Computers*, 16(3):72–80, July - September 1999.

[54] Scott Hauck and Gaetano Borriello. An evaluation of bipartitioning techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(8):849 – 866, 1997.

[55] R. Hayden. Analysis system for reverse engineering VLSI circuits. Master's thesis, Air Force Institute of Technology, 1989.

[56] M. G. Hinchey and J. P. Bowen. *High-Integrity System Specification and Design.* Springer-Verlag, 1999.

[57] Christoph M. Hoffman. *Group Theoretic Algorithms and Graph Isomorphism.* Springer-Verlag, 1982.

[58] Kai-Ti Huang and David Overhauser. A novel graph algorithm for circuit recognition. In *Proceedings of the International Symposium on Circuits and Systems*, pages 1695 – 1698, 1995.

[59] Sung-Woo Hur and John Lillis. Relaxation and clustering in a local search framework: Application to linear placement. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 360 – 366, June 1999.

[60] Masayasu Ito and Yasuhiro Nikaido. Recognition of pattern defects of printed circuit board using topological information. In *Proceedings of the 11th International Electronics Manufacturing Technology Symposium*, pages 202 – 206, 1991.

[61] Jawahar Jain, Amit Narayan, M. Fujita, and A. Sangiovanni-Vincentelli. A survey of techniques of formal verification of combinational circuits. In *Proceedings of the IEEE International Conference on Computer Design*, pages 445 – 454, October 1997.

[62] Frank W. Johannes. Partitioning of VLSI circuits and systems. In *Proceedings of the 33rd ACM/IEEE Design Automation Conference*, pages 83 – 87, June 1996.

[63] Andrew B. Kahng, J. Lach, W.H. Mangione-Smith, Stefanus Mantik, Igor L. Markov, Miodrag Potkonjak, Paul Tucker, Huijuan Wang, and Gregory Wolfe. Watermarking techniques for intellectual property protection. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 776 – 781, June 1998.

[64] Andrew B. Kahng, Stefanus Mantik, Igor L. Markov, Miodrag Potkonjak, Paul Tucker, Huijuan Wang, and Gregory Wolfe. Robust IP watermarking methodologies for physical design. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 782 – 787, June 1998.

[65] Andrew B. Kahng and Rahul Sharma. Studies of clustering objectives and heuristics for improved standard cell placement. http://nexus6.cs.ucla.edu/ abk/papers/report/clu10dist.pdf, January 1997.

[66] George Karypis and Vipin Kumar. Multi-way k-way hypergraph partitioning. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 343 – 348, June 1999.

[67] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning of electrical circuits. *Bell System Technical Journal*, 49(2):291 – 307, January 1970.

[68] Kurt Keutzer. DAGON: Technology binding and local optimization by DAG matching. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 341 – 347, 1987.

[69] L. Krider. A flow analysis algorithm. *Journal of the ACM*, 11(4):429 – 436, October 1964.

[70] B. Krishnamurthy. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Transactions of Computers*, pages 438 – 446, 1984.

[71] Yuji Kukimoto, Robert K. Brayton, and Prashant Sawkar. Delay-optimal technology mapping by DAG covering. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 348 – 351, June 1998.

[72] Sandip Kundu. GateMaker: A transistor to gate level model extractor for simulation, automatic test pattern generation and verification. In *Proceedings of the IEEE International Test Conference*, pages 372–381, 1998.

[73] Oak Ridge National Laboratory. PVM: Parallel virtual machine. http://www.csm.ornl.gov/pvm/, 1993.

[74] Yung-Te Lai, Sarma Sastry, and Massoud Pedram. Boolean matching using binary decision diagrams with applications to logic synthesis and verification. In *Proceedings of the IEEE International Conference on Computer Design*, pages 452–458, October 1992.

[75] Richard H. Lathrop, Robert J. Hall, and Robert S. Kirk. Functional abstraction from structure in VLSI simulation models. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 822 – 828, 1987.

[76] F. Luellau, T. Hoepken, and E. Barke. A technology independent block extraction algorithm. In *Proceedings of the 21st ACM/IEEE Design Automation Conference*, pages 610 – 615, 1984.

[77] Vijay K. Madisetti. Reengineering digital systems. *IEEE Design & Test of Computers*, 16(2):15 – 16, April - June 1999.

[78] Vijay K. Madisetti, Yong-Kyu Jung, Moinul H. Khan, Jeongwook Kim, and Theodore Finnessy. Reengineering legacy embedded systems. *IEEE Design & Test of Computers*, 16(2):38 – 47, April - June 1999.

[79] F. Mailhot and Giovanni De Micheli. Algorithms for technology mapping based on binary decision diagrams and on boolean operations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(5):599 – 620, 1993.

[80] Lawrence Markosian, Philip Newcomb, Russell Brand, Scott Burson, and Ted Kitzmiller. Using an enabling technology. *Communications of the ACM*, 37(5):58 – 70, May 1994.

[81] Yusuke Matsunaga. On accelerating pattern matching for technology mapping. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 118 – 123, 1998.

[82] Ken McElvain. Lgsynth93 benchmark set. Available on the World Wide Web at http://www.cbl.ncsu.edu/CBL_Docs/lgs93.html, 1993.

[83] B.T. Messmer and H. Bunke. Subgraph isomorphism in polynomial time. Technical Report IAM 95-003, Institut für Informatik, 1995.

[84] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, Inc., 1994.

[85] Amit Narayan, Adrian J. Isles, Jawahar Jain, and Robert K. Brayton. Reachability analysis using partitioned ROBDDs. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 388 – 393, 1997.

[86] Amit Narayan, Jawahar Jain, M. Fujita, and A. Sangiovanni-Vincentelli. Partitioned ROBDDs - a compact, canonical and efficiently manipulable representation for boolean functions. In *Proceedings of theIEEE International Conference on Computer-Aided Design*, pages 547 – 544, 1996.

[87] Raymond X.T. Nijssen and Jochen A.G. Jess. Two-dimensional datapath regularity extraction. In *Proceedings of the 5th ACM/IEEE Physical Design Workshop*, April 1996.

[88] Raymond X.T. Nijssen and C.A.J. van Eijk. Regular layout generation of logically optimized datapaths. In *Proceedings of the 1997 ACM/IEEE International Symposium on Physical Design*, pages 42 – 47, April 1997.

[89] Jim Q. Ning, Andre Engberts, and Wojtek Kozaczynski. Recovering reusable components from legacy systems by program segmentation. In *The Working Conference on Reverse Engineering*, 1993.

[90] Jim Q. Ning, Andre Engberts, and Wojtek Kozaczynski. Automated support for legacy code understanding. *Communications of the ACM*, 37(5):50 – 57, May 1994.

[91] Miles Ohlrich, Carl Ebeling, Eka Ginting, and Lisa Sather. *SubGemini*: Identifying subcircuits using a fast subgraph isomorphism algorithm. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 31–37, 1993.

[92] Masahiko Ohmura, Hiroto Yasuura, and Keikichi Tamaru. Extraction of functional information from combinational circuits. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 176–179, November 1990.

[93] George Pelz and Uli Roettcher. Circuit comparison by hierarchical pattern matching. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 290 – 293, 1991.

[94] Marek A. Perkowski, Malgorzata Chrzanowska-Jeske, Alan Coppola, and Edmund Pierzchala. An exact algorithm for the technology fitting problem in the application specific state machine device. In *Proceedings of the International Symposium on Circuits and Systems*, pages 1977 – 1980, 1992.

[95] D. Sreenivasa Rao and Fadi J. Kurdahi. Partitioning by regularity extraction. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 235 – 238, 1992.

[96] D. Srinivasa Rao. On clustering for maximal regularity extraction. *IEEE Transactions on Computer Aided Design*, 12(8):1198 – 1208, 1993.

[97] M. G. Rekoff, Jr. On reverse engineering. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-15(2):244 – 252, March/April 1985.

[98] Kenneth Rose and David Miller. Constrained clustering for data assignment problems with examples of module placement. pages 1937 – 1940, 1992.

[99] Patrick Schaumont, Radim Cmar, Serge Vernalde, Marc Engels, and Ivo Bolsens. Hardware reuse at the behavioral level. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 784 – 789, June 1998.

[100] Yehuda Shiran. YNCC: A new algorithm for device-level comparison between two functionally isomorphic VLSI circuits. In *Proceedings of theIEEE International Conference on Computer-Aided Design*, pages 298–301, 1986.

[101] T. M. Sobh and J.C. Owen. A sensing strategy for the reverse engineering of machined parts. *Journal of Intelligent and Robotic Systems*, pages 1 – 18, August 1995.

[102] Fabio Somenzi. CUDD: Colorado university decision diagram package. http://www.bessie.colorado.edu/ fabio/CUDD/, 1997.

[103] Sponsored by Argonne National Laboratory. *Record of the Argonne/DoD Workshop on Reverse Engineering of Digital Systems*, Del Mar, California, January 1998.

[104] Ruth V. Spriggs. Identification of β-sheet motifs in three-dimensional protein structures, using a subgraph isomorphism algorithm: an update of a 1992 study. Master's thesis, University of Sheffield, 1999.

[105] Ronald C. Stogdill. Dealing with obsolete parts. *IEEE Design & Test of Computers*, 16(2):17 – 25, April - June 1999.

[106] Makoto Takashima, Atsuhiko Ikeuchi, Shoichi Kojima, Toshikazu Tanaka, Tamaki Saitou, and Jun ichi Sakata. A circuit comparison system with rule-based functional isomorphism checking. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 512 – 516, 1988.

[107] Armistead Techologies. Armistead technologies reverse engineering. Available on the world wide web at http://www.armtec.net.

[108] W.B. Thompson, S.R. Stark J.C. Owen, H.J. de St. Germain, and T.C. Henderson. Feature-based reverse engineering of mechanical parts. *IEEE Transactions on Robotics and Automation*, February 1999.

[109] Donald G. Truhlar, W. Jeffrey Howe, Anthony J. Hopfinger, Jeff Blaney, and Richard A. Dammkoehler, editors. *Rational Drug Design*, volume 108 of *The IMA Volumes in Mathematics and its Applications*. Springer-Verlag, 1999.

[110] J. R. Ullman. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, January 1976.

[111] John F. Wakerly. *Digital Design: Principles and Practices*. Prentice-Hall, Inc., third edition, 2000.

[112] Richard C. Waters and Elliot Chikofsky. Reverse engineering: Progress along many dimensions. *Communications of the ACM*, 37(5):23 – 24, May 1994.

[113] Jennifer L. White and Anthony S. Wojcik. A technique for unique subgraph enumeration. Technical Report MSU-CSE-99-35, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, October 99.

[114] Jennifer L. White, Anthony S. Wojcik, Moon-Jung Chung, and Travis Doom. Candidate subcircuits for functional module identification in logic circuits. In *Proceedings of the 10th Great Lakes Symposium on VLSI*, March 2000.

[115] Qinghong Wu, C. Y. Roger Chen, and John M. Acken. Efficient boolean matching algorithm for cell libraries. In *Proceedings of the IEEE International Conference on Computer Design*, pages 36 – 39, October 1994.