

LEVEL OF ESSENTIALNESS OF A NODE IN FLOWCHARTS
AND ITS APPLICATION TO PROGRAM TESTING

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by

Osman Kandara

B.S., Marmara University, Turkey, 1991

M.S. in Systems Science, Louisiana State University, 1996

December 2003

Acknowledgements

Thanks to the God for everything He has been providing.

Table of Contents

Acknowledgements	ii
List of Tables	v
List of Figures	vi
Abstract	viii
Chapter 1 Introduction	1
1.1 Definition of Software Engineering	1
1.2 Needs for Software Testing	1
1.3 Testing Methods	2
1.3.1 Black Box Testing	2
1.3.2 White Box Testing	4
1.4 Some Previous Works in Software Testing	5
1.5 Motivation	8
1.6 Contribution of the Dissertation	9
1.7 Outline of the Dissertation	10
Chapter 2 Basic Definitions and Conventions	11
2.1 Semi-structured Flowchart	11
2.2 Domination Relationship	12
2.3 Post-domination Relationship	12
Chapter 3 Concept of Level of Essentialness	13
3.1 Essential-for Relationship	13
3.2 Degree of Essentialness	14
3.3 Level of Essentialness	15
3.3.1 Formal Approach	15
3.3.2 Intuitive Approach	17
Chapter 4 Properties of the Concept	20
4.1 Covering Relationship	20
4.2 Successively Merging E_i and E_{i-1}	21
4.2.1 Example	21
4.3 Pruning Merged Sets	21
4.4 Example for Pruning Process	21
4.4.1 Step 1	22
4.4.2 Step 2	22
4.4.3 Step 3	22
4.4.4 Step 4	24

4.4.5	Step 5	24
4.4.6	Step 6	24
4.4.7	Step 7	24
4.5	Properties of Pruned Merged Sets	24
Chapter 5	Application to Program Testing	30
5.1	Covering a Subset of Nodes	30
5.2	Finding W'	31
5.2.1	Determining the Smallest Sufficient Set H_i for W	31
5.2.2	Pruning the SP Set to Obtain W'	32
5.3	An Example Application	32
5.3.1	Example for Finding W	32
5.4	Finding W' : $W' \subseteq W$	37
5.4.1	Example	38
5.5	Samples	38
Chapter 6	Algorithms	42
6.1	Algorithm DAL	42
6.1.1	Example	44
6.2	Algorithm PML	46
Chapter 7	Summary	53
7.1	Review	53
7.2	Future Work	54
References		55
Appendix A: Glossary		57
Appendix B: C Program to Compute Domination Tree		60
Appendix C: C Program to Compute CDT and CPT		69
Appendix D: C Program to Implement the PML Algorithm		76
Vita		83

List of Tables

3.1	$\mathcal{E}_{x,y}$ for the flowchart in Figure 1.1	14
4.1	The pruned merged sets in Table 4.7	22
4.2	Steps to prune the merged set in Table 4.7 to compute the pruned sets in Table 4.1	25
4.3	Merged levels of sets of nodes for the flowchart in Figure 4.3	25
4.4	Pruned merged sets of nodes in Table 4.3. The number of H sets and the number of G sets are the same	25
4.5	Merged levels of sets of nodes for the flowchart in Figure 4.2	26
4.6	Nodes and their level of essentialness for the flowchart in Figure 4.1	28
4.7	Levels in Table 4.6 after merging	28
5.1	Computing the SP set for $W = \{6, 24, 26, 38, 68\}$ for the flowchart in Figure 5.1	33
5.2	Refining the SP set H_8 for $W = \{6, 24, 26, 38, 68\}$ for the flowchart in Figure 5.1	33
5.3	Various sets W and their corresponding W' for the flowchart in Figure 4.1	38
5.4	Nodes and their level of essentialness for the flowchart in Figure 5.1	39
5.5	Levels in Table 5.4 after merging.	39
5.6	Pruned merged sets of nodes in Table 5.5.	40
5.7	Mutually essential nodes and their representatives in Table 5.6.	40
6.1	Nodes and their degrees of essentialness for the flowchart in Figure 4.1	46

List of Figures

1.1	A structured flowchart with nested while–do loops. Here, 0 is the <i>start</i> node and 8 is the <i>stop</i> node.	7
3.1	A structured flowchart with nested while–do’s and nested if’s. . . .	13
3.2	The DT of the flowchart in Figure 1.1 where the minimally essential nodes are shaded.	17
3.3	The PT of the flowchart in Figure 1.1 where the minimally essential nodes are shaded.	17
3.4	The flowchart in Figure 3.1 where the maximally essential nodes and their incoming and outgoing arcs are dotted.	19
3.5	The flowchart forest with dummy start and stop nodes after the maximally essential nodes and their arcs are removed from the flowchart in Figure 3.1.	19
4.1	A semi–structured flowchart with break and continue.	23
4.2	A flowchart disproving Lemma 4.1 for G_i	29
4.3	A semi–structured flowchart. Node $6 \in H_5$ covers nodes 4 and 7 in H_4 and node $5 \in H_5$ covers node $4 \in H_4$	29
5.1	Flowchart for the program <i>loancalc.c</i> . Here, a node x represents all the statements denoted by $[x]$ in the program.	41
6.1	An efficient algorithm to compute the degrees and levels of essentialness for all nodes.	43
6.2	DT for the flowchart in Figure 4.1. Here, $(x, \#n)$ denotes that the size of the subtree rooted at node x is n . Nodes 1, 3, 15, and 19 are the while–do nodes.	45
6.3	PT for the flowchart in Figure 4.1. Here, $(x, \#n)$ denotes that the size of the subtree rooted at node x is n . Nodes 1, 3, 15, and 19 are the while–do nodes.	45
6.4	An efficient algorithm to compute all pruned merged sets of nodes. .	46
6.5	CDT for DT in Figure 6.2, where the mutually essential nodes are combined into single nodes shown as shaded.	48

6.6	CPT for PT in Figure 6.3, where the mutually essential nodes are combined into single nodes shown as shaded.	48
6.7	CDT - Computing $H_{min=7} = \{2, 5, 8, 9, 11, 13, 16, 21, 22, 24, 26, 27\}$ shown as shaded and updating CDT. Here, H_7 nodes are the common terminal nodes in both CDT and CPT in Figure 6.5 and Figure 6.6 respectively. Nodes in H_7 and their edges incident to them shown as dashed will be removed from CDT to update it.	49
6.8	CPT - Computing $H_{min=7} = \{2, 5, 8, 9, 11, 13, 16, 21, 22, 24, 26, 27\}$ shown as shaded and updating CPT. Here, H_7 nodes are the common terminal nodes in both CDT and CPT in Figure 6.5 and Figure 6.6 respectively. Nodes in H_7 and their edges incident to them shown as dashed will be removed from CPT to update it.	49
6.9	CDT - Computing $H_6 = \{6, 15, 18, 20\}$. Here, we take the smallest node number as the representative for the mutually essential nodes in the combined node (15,17) shown as shaded	50
6.10	CPT - Computing $H_6 = \{6, 15, 18, 20\}$. Here, we take the smallest node number as the representative for the mutually essential nodes in the combined node (15, 17) shown as shaded	50
6.11	CDT - Computing $H_5 = \{14, 19\}$	51
6.12	CPT - Computing $H_5 = \{14, 19\}$	51
6.13	CDT - Computing $H_4 = \{12\}$	51
6.14	CPT - Computing $H_4 = \{12\}$	51
6.15	CDT - Computing $H_3 = \{10\}$	52
6.16	CPT - Computing $H_3 = \{10\}$	52
6.17	CDT - Computing $H_2 = \{4\}$ and $H_1 = \{0\}$	52
6.18	CPT - Computing $H_2 = \{4\}$ and $H_1 = \{0\}$	52

Abstract

Program testing is important to develop bug free software. A common form of program testing involves selecting test cases which execute (cover) a given set W of statements in the program. In regression testing, W typically forms a small subset of the program. It is often possible to find an alternate small set W' so that execution of W' implies execution of W .

We develop concepts and algorithms for finding W' as small as possible with the condition that the statements in W' are "close" to those in W in terms of program structure. These concepts generalize the notion of the essential set, which was introduced by Bertolino for the special case $W =$ set of all program statements.

We define the *essential-for* relationship between two nodes x and y and *degree of essentialness* for a node x in a program flowchart. The sets $E_i =$ all nodes whose degrees of essentialness is the i^{th} largest value, i.e., i^{th} level essential nodes form a partition. We group them in a certain way to form the sets G_j so that each G_j "covers" G_1, G_2, \dots, G_{j-1} . The sets G_j are then pruned by using a suitable notion of "equivalence" to form the sets H_i , which have two important properties: H_i covers H_{i-1} and $|H_i| \geq |H_{i-1}|$. The sets H_i are then used to construct our desired set W' .

We give efficient algorithms to compute the sets H_i and the set W' and illustrate our method with example programs.

Chapter 1

Introduction

1.1 Definition of Software Engineering

There are various definitions of the term software engineering. According to the definition given at the first NATO conference [4], software engineering is the establishment and use of sound engineering principles in order to obtain economical software that is reliable and works on real machines.

IEEE Standard Glossary of Software Engineering Terminology [3] defines the term as the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

Although there are many more definitions with rather different words, the main objective of the software engineering discipline that we could derive from these and other definitions is to find practical solutions to the many problems that software development projects are likely to face particularly when developing large scaled software [2].

One of the problems that the software engineering discipline has to address is the problem of assuring the quality of the developed software. The first requirement for quality software is to be bug-free. Therefore, testing to ensure bug-free software is an important issue for quality software.

1.2 Needs for Software Testing

Today, almost all systems from health care to military, from education to transportation, from financial institutions to household products and so on are using some type of software. Clearly, their proper operations heavily depend on the soft-

ware they are utilizing. Therefore, testing their software against possible bugs is of a critical importance for their proper operations.

Following reported news illustrates the importance of software testing and shows what the consequence we would have to pay, otherwise [5].

In January of 2001 newspapers reported that a major European railroad was hit by the aftereffects of the Y2K bug. The company found that many of their newer trains would not run due to their inability to recognize the date '31/12/2000'; the trains were started by altering the control system's date settings.

Software bugs are a fact of life. On average, even well-written programs have one to three bugs for every 100 statements [7]. Software could have bugs for several reasons. Misscommunication or no communication between the developer and the client, software complexity to comprehend, programming errors, changing requirements, time pressure, egos, poorly documented code, and software development tools are just few of those reasons [5].

Although testing is an important process embedded in all software life cycles, we are only interested in testing the developed software. Several testing methods have been developed. In the following section, we first categorize the testing methods and then give the literature review for various testing methods.

1.3 Testing Methods

There are different types of testing methods. The *black box* and *white box* tests represent the two broad categories of test types.

1.3.1 Black Box Testing

Black box testing is also referred to as *closed box*, *functional*, or *behavioral* testing. Black back testing compares the tested program behavior against its requirements specification. Source code of the developed software is not needed. Its main objec-

tive is to determine if the program does what it is supposed to do. How it does doesn't matter.

Functionality testing, volume tests, stress tests, recovery testing, benchmarks, field and laboratory tests are among the most important black box testing strategies. Programmers have the tendency not to break their own programs. Therefore, black box testing should be performed by a third party who doesn't know much about the program internals. [6].

Some of the advantages of black box testing are as follows:

- Tester needs no knowledge of the implemented programming language.
- Tester and programmer are independent of each other.
- Tests are done from a user's point view.
- It will help to expose any ambiguities or inconsistency in the specifications.
- Test cases can be designed as soon as the specifications are complete.

Despite its advantages, black box testing has some disadvantages, too. Some of them are listed as follows:

- Real-life systems may have too many different kinds of inputs, resulting in a combinatorial explosion of test cases. Therefore, only a small number of possible inputs can actually be tested [7].
- Without clear specifications, test cases are hard to design [8].
- It is impossible to know which portions of the code have been executed. Therefore, when a problem is discovered, extensive engineering time and resources are required to locate the root cause of the problem. Sometime, it may not be possible to do so.

- Some program paths may not be executed, therefore not tested at all.
- The correct operation of the program may not be a measurable output. Therefore, the output may not be so obvious to determine if it is the desired one [7].

Common methods to generate black box test cases include equivalence partitioning, boundary value analysis, and cause effect graphing techniques [9].

1.3.2 White Box Testing

White box testing is also referred to as *structural*, *open box*, or *glass box* testing. As a contradiction to black box testing, white box testing compares the tested program behavior against the apparent intention of the source code. It examines how the program works, considering the structural and logical details of the source code. Therefore, source code is needed to perform white box testing.

White box testing strategies include:

- Basis path testing [11].
- Control structure testings.
 - Conditions testing.
 - Data flow testing.
 - Loop testing.

White box testing is also called *path* testing or *coverage-based* testing since we choose test cases that cause paths to be taken through the structure of the program or cause some nodes or branches to be covered [10].

White box testing can be performed as either static or dynamic analysis where the static analysis don't require the execution of the code [12]. Note that static analysis is more than testing. It is also related to code inspection.

Some of the advantages of white box testing are as follows:

- It doesn't heavily rely on the requirement specifications. Requirement specifications sometime don't exist or are not complete.
- It can identify unreachable code and unused variables (through static analysis).
- It forces the test developer to reason carefully about the implementation of the source code [8].
- Portions of the code that are executed (or not executed) can be tracked.

Some of the disadvantages of white box testing include:

- It is expensive. Covering all paths may take forever.
- Tester needs strong knowledge of the programming language used to develop the software.
- It is common to use third party products to develop software. Therefore, source code may not be available for all of the software.
- In most cases, it requires modifications in the original program, changing values to get alternative paths. Code instrumentation introduces overhead.

1.4 Some Previous Works in Software Testing

Literature on software testing has appeared since the beginning of computer. However, John Goodenough and Susan Gerhart are the first ones who published a paper [13] defining the field of software testing and instigating much of today's research agenda [14].

Glenford Myers's seminal book [15] was the only testing book of notes for several years. Brian Marick's *The Craft of Software Testing* [17] has been a popular book as an introduction to the subject of software testing.

Boris Beizer in his book *Software Testing Techniques* [18] first classifies the different types of bugs as functional, structural, data, coding, system, design, and test bugs. Then, the author introduces the concept of a *transaction flow* representation which is a way to model the system's behavior and used for functional testing. Moreover, Beizer's book covers path and data-flow testing followed by an overview of domain testing. Data-flow testing is utilized to identify *data anomalies*. Domain testing attempts to test whether inputs are fulfilling some prior specification. Beizer also covers logic-based testing, which uses Boolean algebra. These are not the only books, of course. Today on the market there are more than 100 software engineering books talking about program testing [14].

Besides books, numerous number of studies have also been made looking for new methodologies or applications for better testing. For example, Philip Stocks in his PhD thesis [19] examines the application of formal methods to software testing. According to Stocks, formal specifications offer the bases for rigorous testing practices. The most immediate use of formal specifications in software testing is as sources of black-box test suites.

After a program is modified (to fix bugs), we must ensure that both modified and unmodified parts of the program work correctly. Unfortunately, complete regression testing cannot always be accommodated during frequent modifications since it is often time consuming. Hiralal Agrawal et al., in [20] introduced efficient incremental regression testing methods.

Mark Weiser first introduced the concept of program slicing in [21], then gave an application of it to debug programs. [23]. He noticed that not all statements of code

contribute to the value of an variable computed at some point. A program slice consists of the parts of a program that (potentially) affect the values computed at some point of interest, referred to as a slicing criterion. To test the variable at that point, not all statements but the corresponding slice needs to be executed. The task of computing program slices is called program slicing.

David Binkley explores the use of program slicing and dependency graph to reduce the cost of regression testing in [16]. He provides an algorithm. Both the number of test cases and the size of the program that must be rerun are reduced.

Thomas McCabe introduced a structured testing methodology for software testing also known as *basis path testing*. It uses the measure of *McCabe Cyclomatic complexity* to determine the number of *independent paths* that guarantees coverage of all statements in the program [11].

Stephen Edwards attempts to combine the black box testing with white box strategies in [25]. Their approach first generates a flowchart from a component's specification then applies the white box strategies. Jeffrey Voas and Keith Miller use fault-injection methods to predict where actual bugs are more likely to hide.

In the above sections, we studied the previous work in program testing. In the following section we will explain the motivation.

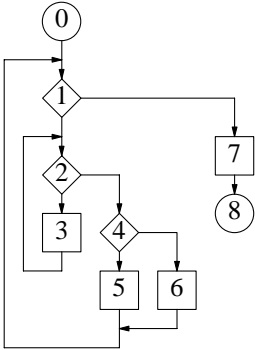


FIGURE 1.1. A structured flowchart with nested while-do loops. Here, 0 is the *start* node and 8 is the *stop* node.

1.5 Motivation

Program testing is essential to ensure bug-free software. Almost all program testing methods [2, 6, 7, 10, 11, 14, 17, 18] assume that every statement in a program (i.e., a node in the program's flowchart F) is equally important in program testing and test planning. This is, however, rarely the case.

For example, in Figure 1.1 nodes $\{0, 1, 7, 8\}$ are the most essential nodes since they appear in all complete paths from the *start* node to the *stop* node. Removal of any one of them (except 0 and 8) disconnects the flowchart. On the other hand, nodes $\{3, 5, 6\}$ are the least essential since if we remove any one of them, there is still a complete path through each of the remaining nodes.

Very little work has been done to classify the nodes in a flowchart in terms of their essentialness in program testing. Bertolino and Marre [1] define the notion of an essential set of nodes. They do not, however, classify a node being essential or not essential. For a structured flowchart, the (locally) deepest level nodes make up the essential set. In Figure 1.1, $S = \{3, 5, 6\}$ is the essential set of nodes.

An important property of an essential set S is that if a set of paths cover S , then they cover every node of the flowchart. Our work refines the results of [1] in two ways: (1) we define the essentialness of a node x for another node y . (2) we classify the nodes in a hierarchy via level of essentialness and show its application in program test coverage.

EXAMPLE 1.1. The nodes in Figure 1.1 are classified as follows:

- $E_1 = \{0, 1, 7, 8\}$. (The first level – maximally essential – nodes)
- $E_2 = \{2, 4\}$. (The second level nodes)
- $E_3 = \{3, 5, 6\}$. (The third level – minimally essential – nodes; this is the same as S above)

Notice that a set of paths that cover all nodes in E_3 will also cover all nodes in E_2 and E_1 . Similarly, covering E_2 nodes will also cover all E_1 nodes. According to [1], we cannot say whether node 1 in itself is essential or not essential.

1.6 Contribution of the Dissertation

In this dissertation, a new concept of level of essentialness for nodes in a flowchart is developed and its application in program test coverage is explored. The contributions of this dissertation are summarized as follows:

- define the essentialness of a node x for a node y in a flowchart by utilizing the notions of domination and post–domination relationships between x and y .
- define the degree of essentialness for a node x .
- classify the nodes in the flowchart in a hierarchy via level of essentialness which generalizes the notion of the essential set introduced by Bertolino and Marre.
- group the levels in a certain way and pruned the grouped levels by the removing equivalent nodes from them to obtain the pruned sets, which have two important properties, namely covering and none–increasing property.
- utilize the pruned sets to obtain a smaller size set W' of nodes to cover a desired set W .
- provide an alternate method to obtain W' for W where W' is a subset of W by utilizing domination and post–domination trees.
- develop algorithms to compute the degree of essentialness for all nodes and compute the pruned merged sets of nodes in a flowchart.

1.7 Outline of the Dissertation

essentialness for a nodes is proposed and its application is explored. Chapter 2 presents the basic definitions about semi-structured flowchart, dominations relationship, and post-dominations relationship. Chapter 3 defines the essential for relationship between two nodes x and y and the degree of essentialness for a node x and introduces the concept of the level of essentialness. Chapter 4 discusses the properties of the concept. Chapter 5 illustrates an application of the concept in program test coverage. Chapter 6 illustrates algorithms to compute the degree of essentialness for all nodes and the pruned merged sets in a flowchart. Chapter 7 concludes the dissertation along with the future research.

Chapter 2

Basic Definitions and Conventions

Every program P can be represented as a flowchart F . In our research, we assume that P terminates for every input. In our research, we restrict ourself to semi-structured flowcharts.

In the following sections, we first define the term *semi-structured* flowchart. Then, we review the definitions of domination and post-dominance relationships. We utilize these relationships to develop our concepts and algorithms.

2.1 Semi-structured Flowchart

DEFINITION 2.1. A flowchart F is called *semi-structured* if F is structured except for the use of *breaks*, *continues*, and *returns* (as in C-programs for example). Arbitrary *gotos* are not allowed.

DEFINITION 2.2. Nodes x and y of the same type are called *sequential* (and hence can be *collapsed* into a single node) if (x, y) is the only arc from x and also it is the only arc to y .

Note that if nodes x_i and x_{i+1} are sequential for $i = 0, 1, \dots, n - 1$, then we can collapse all the nodes $\{x_0, x_1, x_2, \dots, x_{n-1}, x_n\}$ into a single node. Throughout the paper, all flowcharts are assumed to be semi-structured with no sequential nodes.

DEFINITION 2.3. A *path* from node a to node b is a sequence of one or more arcs of the form $\pi_{a,b}^F = \langle (x_0, x_1), (x_1, x_2), (x_2, x_3), \dots, (x_{n-1}, x_n) \rangle$, where $x_0 = a$ and $x_n = b$; we sometimes use the compact notation $\pi_{a,b}^F = \langle x_0, x_1, x_2, \dots, x_{n-1}, x_n \rangle$ and write $x_i \in \pi_{a,b}^F$ to indicate that $\pi_{a,b}^F$ goes through x_i .

A path $\pi_{start,stop}^F$ from the start node in F to its end node is called a *complete path*.

2.2 Domination Relationship

DEFINITION 2.4. A node x is said to *dominate* a node y if $x \in \pi_{start,y}^F$ for all $\pi_{start,y}^F$. We write $\mathcal{D}_{x,y} = 1$ to indicate that x dominates y ; otherwise, $\mathcal{D}_{x,y} = 0$. \mathcal{D} is called the *domination* relationship. A node x *immediately dominates* a node y if only if $\mathcal{D}_{x,y} = 1$ and there is no node z such that $\mathcal{D}_{x,z} = \mathcal{D}_{z,y} = 1$; we denote this by $\mathcal{D}_{x,y}^{im} = 1$.

2.3 Post–domination Relationship

DEFINITION 2.5. A node x is said to *post–dominate* a node y if $x \in \pi_{y,stop}^F$ for all $\pi_{y,stop}^F$. We write $\mathcal{P}_{x,y} = 1$ to indicate that x post–dominates y ; otherwise, $\mathcal{P}_{x,y} = 0$. \mathcal{P} is called *post–domination* relationship. A node x *immediately post–dominates* a node y if only if $\mathcal{P}_{x,y} = 1$ and there is no node z such that $\mathcal{P}_{x,z} = \mathcal{P}_{z,y} = 1$. We write $\mathcal{P}_{x,y}^{im} = 1$ to indicate that x immediately post–dominates y .

Note that $\mathcal{D}_{start,stop} = \mathcal{P}_{stop,start} = 1$. Both \mathcal{D} and \mathcal{P} are *reflexive* and *transitive*, but *not symmetric*. In view of this, we can represent these relationships as trees called *domination tree* $DT(F)$ and *post–domination tree* $PT(F)$. The root of $DT(F)$ is the *start* node, and the root of $PT(F)$ is the *stop* node. We write $\pi_{a,b}^D$ for the unique path from node a to node b in $DT(F)$ (and similarly $\pi_{a,b}^P$ in $PT(F)$). A fast algorithm to compute $DT(F)$ and $PT(F)$ in a general flowchart (which may not be semi–structured) is given by Tarjan and Lengauer[22].

REMARK 2.1. Note that if $\pi = \langle x_0, x_1, x_2, \dots, x_{k-1}, x_k \rangle$ is a path in $DT(F)$ such that $\mathcal{P}_{x_k,x_0} = 1$, then $\langle x_k, x_{k-1}, x_{k-2}, \dots, x_1, x_0 \rangle$ will be a path in $PT(F)$.

Chapter 3

Concept of Level of Essentialness

3.1 Essential-for Relationship

DEFINITION 3.1. A node x is *essential* for a node y if $\mathcal{D}_{x,y} = 1$ or $\mathcal{P}_{x,y} = 1$ (or both). Put another way, a node x is *essential* for a node y if each $\pi_{start,stop}^F$ that goes through y also goes through x . We write $\mathcal{E}_{x,y} = 1$ if x is essential for y , otherwise $\mathcal{E}_{x,y} = 0$. Clearly, $\mathcal{E}_{x,x} = 1$. We call $\mathcal{E}_{x,y}$ *essential-for relationship*.

In the flowchart in Figure 1.1, $\mathcal{E}_{2,4} = 1$ because $\mathcal{D}_{2,4} = 1$. Similarly, $\mathcal{E}_{4,5} = 1$ and $\mathcal{E}_{4,6} = 1$. Also, $\mathcal{E}_{4,2} = 1$ since $\mathcal{P}_{4,2} = 1$.

REMARK 3.1. Note that although $\mathcal{E}_{2,4} = \mathcal{E}_{4,2} = 1$, the $\mathcal{E}_{x,y}$ relationship is in general neither symmetric nor asymmetric. For example, $\mathcal{E}_{1,2} = 1$ while $\mathcal{E}_{2,1} = 0$. On the other hand, $\mathcal{E}_{x,y}$ relationship is transitive.

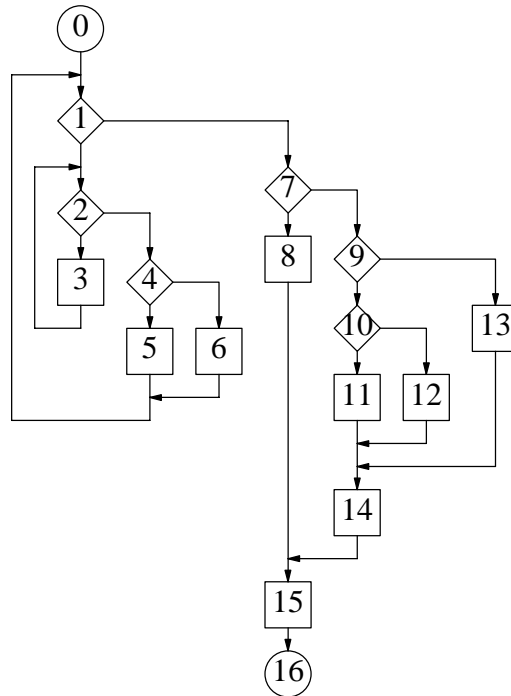


FIGURE 3.1. A structured flowchart with nested while-do's and nested if's.

TABLE 3.1. $\mathcal{E}_{x,y}$ for the flowchart in Figure 1.1

$\mathcal{E}_{x,y}$	0	1	2	3	4	5	6	7	8	δ_x	α_x
0	1	1	1	1	1	1	1	1	1	9	1
1	1	1	1	1	1	1	1	1	1	9	1
2	0	0	1	1	1	1	1	0	0	5	2
3	0	0	0	1	0	0	0	0	0	1	3
4	0	0	1	1	1	1	1	0	0	5	2
5	0	0	0	0	0	1	0	0	0	1	3
6	0	0	0	0	0	0	1	0	0	1	3
7	1	1	1	1	1	1	1	1	1	9	1
8	1	1	1	1	1	1	1	1	1	9	1

DEFINITION 3.2. Nodes x and y are called *mutually essential* if $\mathcal{E}_{x,y} = \mathcal{E}_{y,x} = 1$. If $\mathcal{E}_{x,y} = \mathcal{E}_{y,x} = 0$, then they are called *mutually nonessential*. Node x is called *one-way essential* to node y if $\mathcal{E}_{x,y} = 1$ and $\mathcal{E}_{y,x} = 0$.

In the flowchart in Figure 1.1, nodes 2 and 4 are mutually essential. Similarly, nodes 0, 1, 7, and 8 are also mutually essential.

On the other hand, nodes 3, 5 and 6 are mutually nonessential. Node 1 is one-way essential to node 4.

3.2 Degree of Essentialness

DEFINITION 3.3. Let δ_x denote the number of nodes y such that x is essential for. δ_x is called the *degree of essentialness* for x .

In the flowchart in Figure 1.1, nodes 3, 5, and 6 are not essential to any other node but themselves. Nodes 0, 1, 7, and 8 are essential for all nodes. The essential-for relationship for the flowchart in Figure 1.1 is given in Table 3.1 along with δ_x values.

THEOREM 3.1. Let $\mathcal{S}_x^D = \{y : \mathcal{D}_{x,y} = 1\}$ and $\mathcal{S}_x^P = \{y : \mathcal{P}_{x,y} = 1\}$. $\delta_x = |\mathcal{S}_x^D \cup \mathcal{S}_x^P| = |\mathcal{S}_x^D| + |\mathcal{S}_x^P| - |\mathcal{S}_x^D \cap \mathcal{S}_x^P|$.

PROOF. By definition, δ_x is the number of all nodes y such that $\mathcal{E}_{x,y} = 1$. Note that $\mathcal{E}_{x,y} = 1$ if $\mathcal{D}_{x,y} = 1$ or $\mathcal{P}_{x,y} = 1$. Therefore, δ_x comes from the number of

nodes x either dominates or post-dominates. Note that some nodes such as node x itself and all the nodes within the while-do loop without break are dominated and post-dominated by node x at the same time. Therefore, when we count the number of all those nodes y , we have to make sure that we count each node y once. Doing so, δ_x is the number of nodes x dominates plus the number of nodes x post-dominates minus the number of nodes x dominates and post-dominates. So, Theorem is proved.

Note that a while-do loop without break has an interesting property. The head of the loop dominates and post dominates all the nodes within its loop body. Therefore, $\delta_x = |\mathcal{S}_x^D| + |\mathcal{S}_x^P| - |\mathcal{S}_x^D \cap \mathcal{S}_x^P|$, where x is the head of a while-do loop. In all other cases, node x doesn't dominate and post-dominate any other node but itself at the same time. So, $\delta_x = |\mathcal{S}_x^D| + |\mathcal{S}_x^P| - 1$ for all other cases.

REMARK 3.2. If nodes x and y are mutually essential, then $\mathcal{D}_{x,y} = \mathcal{P}_{y,x} = 1$ and $\delta_x = \delta_y$. Similarly, if nodes x and y are mutually nonessential, then $\mathcal{D}_{x,y} = \mathcal{P}_{y,x} = 0$. However, we cannot say anything about their degrees' relationship. If node x is one-way essential to node y , then $\delta_x > \delta_y$.

3.3 Level of Essentialness

node x is related to δ_x . We can approach the concept in two ways. In the following, we first give the formal approach. It utilizes the $\mathcal{E}_{x,y}$ table. Then, we explain the intuitive approach we initially started with. The intuitive approach works only for structured cases. Therefore, we need a formal one to generalize the concept for the semi-structured flowcharts, too.

3.3.1 Formal Approach

DEFINITION 3.4. A node x is *maximally* essential if it is essential for all the nodes. Let E_{max} denote the set of all maximally essential nodes. A node y is

minimally essential if it is not essential to any other node. Let E_{min} denote the set of all minimally essential nodes.

For the flowchart in Figure 1.1, $E_{max} = \{0, 1, 7, 8\}$ and $E_{min} = \{3, 5, 6\}$. Note that always $start \in E_{max}$ and $stop \in E_{max}$.

The concept of essentialness is defined by Bertolino and Marre from a different perspective[1]. A set made out of all the minimally essential nodes is called the essential set of nodes in [1] based on the fact that a set of test cases that covers all the nodes in the essential set will also cover all the other nodes. In other words, covering the nodes in the essential set is essential to cover all the other nodes. As defined in the paper, every node $y \in E_{min}$ appears as a leaf node in both $DT(F)$ and $PT(F)$ of F . Obviously, if a node x in F appears as a leaf node in $DT(F)$, that means it doesn't dominate any other node but itself. Similarly, if a node x in F appears as a leaf node in $PT(F)$, that means it doesn't post-dominate any other node but itself. Therefore, if a node x in F appears as a leaf node in both $DT(F)$ and $PT(F)$, by definition it is not essential to any other node but itself. That is, it is minimally essential. Figure 3.2 and Figure 3.3 show the DT and PT of the flowchart in Figure 1.1 respectively, where the minimally essential nodes appearing as the leaf nodes in both trees are shaded.

Note that every $x \in E_{max}$ belongs to $\pi_{start,stop}^D$ and $\pi_{stop,start}^P$.

DEFINITION 3.5. If node x has the i^{th} largest δ_x , then the essentialness level of x is i . Let E_i denote the set of all nodes with the essentialness level i and α_x denote the level of essentialness for node x .

Note that E_1 and E_{max} are the same set. In other words, $\delta_x = 1$ for all $x \in E_{max}$. Similarly, E_k and E_{min} are the same set, where k is the least order possible. In other words, $\delta_y = 1$ and $\alpha_y = k$ for all $y \in E_k$. Table 3.1 shows δ_x and α_x for all nodes x in Figure 1.1.

REMARK 3.3. If nodes x and y are mutually essential, then x and y belong to the same E_i . If a node x is one-way essential to a node y , then $x \in E_i$ and $y \in E_j$, where $i < j$. See Remark 3.2.

REMARK 3.4. Note that every pair of x and $y \in E_{max}$ are mutually essential. Similarly, every pair of x and $y \in E_{min}$ are mutually nonessential.

LEMMA 3.1. If $x, y \in E_i$, then x and y are either mutually essential or mutually nonessential.

PROOF. Let us assume that x is one-way essential to y . If x and y are in the same E_i , then from Remark 3.2 our assumption cannot be true. Therefore, by contradiction Lemma is proved.

THEOREM 3.2. If node $x \in E_{max}$, then x appears in all complete paths.

PROOF. Follows directly from the definitions of $\mathcal{E}_{x,y}$ and E_{max} .

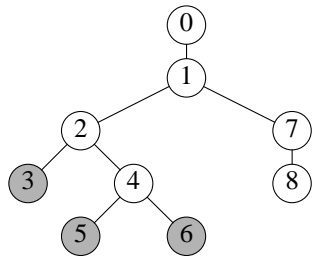


FIGURE 3.2. The DT of the flowchart in Figure 1.1 where the minimally essential nodes are shaded.

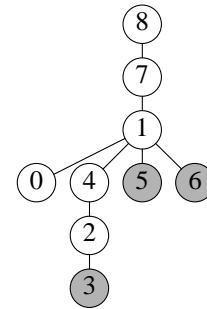


FIGURE 3.3. The PT of the flowchart in Figure 1.1 where the minimally essential nodes are shaded.

Now, we illustrate the intuitive approach that we started with.

3.3.2 Intuitive Approach

The nodes 0, 1, 7, 15, and 16 in the structured flowchart in Figure 3.1 are the 1st level nodes. Clearly, they are the maximally essential nodes because they appear in all complete paths. When the maximally essential nodes and their incoming

and outgoing arcs, which are shown in Figure 3.4 as dotted, are removed, we end up with several connected components (in this case 3). In order to consider each component as a flowchart, we add to it a dummy start (S) and a dummy stop (E) node, where all the nodes with no outgoing arcs converge into the dummy stop node. Note that each component in general has a single entry point that is the smallest node number in the component and may have multiple nodes with no outgoing arcs which will converge into the single dummy stop node. Figure 3.5 shows the flowchart forest after the maximally essential nodes and their arcs are removed from the flowchart in Figure 3.1.

Considering each flowchart in the forest in Figure 3.5, nodes 2, 4, 9, and 14 have the 2^{nd} level of essentialness. Note that nodes 2 and 4 are the maximally essential nodes of the first little flowchart. Similarly, nodes 9 and 14 are the maximally essential nodes of the third little flowchart. Note also that node 8 has the least degree of essentialness, say *min* degree since it is the only node in the second little flowchart. Obviously, node 8 is one of the 7 minimally essential nodes in the flowchart in Figure 3.1.

We now remove the second little flowchart completely from the forest since it has a single node in it. Then, applying the same idea to each of the remaining little flowcharts in the forest, we remove the nodes with the 2^{nd} level of essentialness and their incoming and outgoing arcs from the little flowcharts they belong to identify the nodes with the 3^{rd} level.

Obviously, node 10 is the only node with the 3^{rd} level of essentialness. Again, nodes 3, 5, 6, and 13 are minimally essential in addition to node 8 from the previous step.

Finally, nodes 11 and 12 become the last nodes we should consider when node 10 and its arcs are removed. Nodes 11 and 12 are also minimally essential nodes.

For the flowchart in Figure 3.1, $E_1 = \{0, 1, 7, 15, 16\}$, $E_2 = \{2, 4, 9, 14\}$, $E_3 = \{10\}$, and $E_4 = \{3, 5, 6, 8, 11, 12, 13\}$.

Note that i^{th} level essential node of the component flowchart are called the $(i + 1)^{th}$ level of the original flowchart.

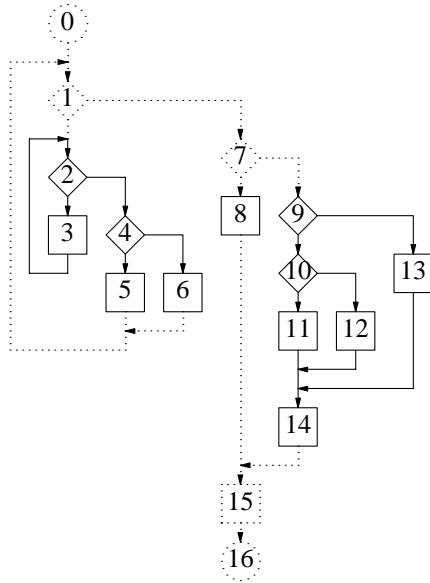


FIGURE 3.4. The flowchart in Figure 3.1 where the maximally essential nodes and their incoming and outgoing arcs are dotted.

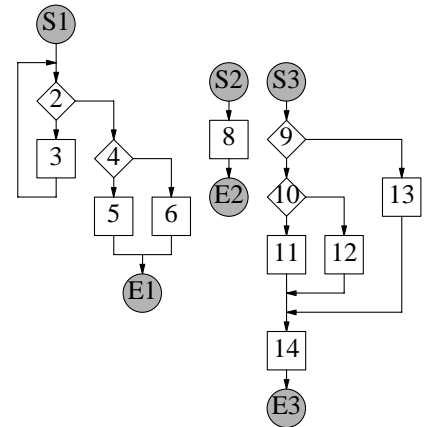


FIGURE 3.5. The flowchart forest with dummy start and stop nodes after the maximally essential nodes and their arcs are removed from the flowchart in Figure 3.1.

Chapter 4

Properties of the Concept

4.1 Covering Relationship

Let F be a flowchart for a terminating program P for all inputs. Then, every test case τ running in P has a corresponding complete path $\pi(\tau)$ in F .

DEFINITION 4.1. A test case τ *covers* a node x if $x \in \pi(\tau)$. A node x *covers* a node y if all τ that cover x also cover y . A node x covers itself.

REMARK 4.1. A node x covers a node y if $\mathcal{E}_{y,x} = 1$. For example, node 5 covers node 4 in Figure 1.1 since $\mathcal{E}_{4,5} = 1$.

Note that in a real-life situation, not every path may be feasible. Therefore, the statement in Remark 4.1 is the best approximation we could come up. In view of this, mutually essential nodes cover each other. On the other hand, mutually nonessential nodes do not cover each other. If node x is one-way essential to node y , then only y covers x , but not vice versa.

DEFINITION 4.2. A set of nodes S *covers* a set of nodes W if each node $y \in W$ is covered by a node $x \in S$.

Considering the sets of nodes $E_1 = \{0, 1, 7, 8\}$, $E_2 = \{2, 4\}$, and $E_3 = \{3, 5, 6\}$ (as in Example 1.1) for the flowchart in Figure 1.1, E_3 covers both E_1 and E_2 , and E_2 covers E_1 .

REMARK 4.2. Covering relationship is transitive. Remark is obvious from Remark 3.1 and Remark 4.1.

Note that If E_i covers E_j , then $i \geq j$. This is obvious directly from Remark 3.3, Remark 4.1, and Remark 4.2. In view of this, E_{min} covers every E_i and E_{max} just covers itself.

4.2 Successively Merging E_i and E_{i-1}

Level of essentialness has interesting properties. We can successively merge E_i with E_{i-1} starting from the right end E_{min-1} until we form a union $G_l = E_{i,j} = \bigcup E_k$ for all $i \leq k \leq j$ such that $E_{i,j}$ covers all sets from E_1 through E_j . We are going to illustrate our method with an example in the following section.

REMARK 4.3. G_i covers G_j , where $1 \leq j \leq i$. It is clear from the method.

4.2.1 Example

Nodes of the flowchart in Figure 4.1 and their levels of essentialness are shown in Table 4.6. Table 4.7 shows the new merged levels G_i in Table 4.6.

The flowchart in Figure 4.1 is a semi-structured flowchart, where node 8 denotes a continue node going back to the head node and node 21 denotes a break node going out to node 23. Similarly, node 17 is a break node going out to node 25.

4.3 Pruning Merged Sets

We can successively prune merged sets G_i starting from the right end G_{min} by removing all nodes y from G_i such that y is covered by a another node $x \in G_i$, i.e., $\mathcal{E}_{y,x} = 1$. If $\mathcal{E}_{y,x} = 1$ and $\mathcal{E}_{x,y} = 0$, then $G_{i-1} = G_i \setminus \{y\}$.

Let H_i denote a pruned set of nodes G_j . Table 4.1 shows the pruned merged sets for Table 4.7 We illustrate the pruning process in the following example to show how such a table can be obtained from the given merged sets.

4.4 Example for Pruning Process

Table 4.2 illustrates the pruning steps for the given merged sets in Table 4.7. In the table, a faded set illustrates a pruned set H_i and a non-faded set illustrate a merged set G_j to be pruned. A faded node in a non-pruned set G_{i-1} is the node that has been moved up from G_i to G_{i-1} during the pruning process of G_i . At the end of the pruning process, the number of pruned sets will be higher than or equal

TABLE 4.1. The pruned merged sets in Table 4.7
 H_i | Pruned sets of nodes

1	{0}
2	{4}
3	{10}
4	{12}
5	{14, 19}
6	{6, 15, 18, 20}
7	{2, 5, 8, 9, 11, 13, 16, 21, 22, 24, 26, 27}

to the number of merged sets depending on whether there exists some nodes in G_j , where $j \leq 1$ that need to move up to the artificially introduced initially empty sets G_{j-1} . In this example, $H_{min=7}$.

4.4.1 Step 1

Since all nodes in $G_{min=6}$ are mutually nonessential, no one can be removed from the set. Thus, $H_{min=7} = G_{min=6}$.

Note that the set $H_{min} =$ Bertolino's essential set.

4.4.2 Step 2

mutually essential. Therefore, they cover each other. As a result of this, one of them (say 17) is removed from G_5 to prune the set, i.e., to compute H_6 . Note that the mutually essential (equivalent) nodes cover the same set of nodes. Therefore, removal of them except one will not effect the set of nodes the final pruned set will cover.

As a convention, we remove the smallest node number from the list.

4.4.3 Step 3

In G_4 , first note that nodes 19 and 23 are mutually essential. Thus, one of them (say node 23) is removed from G_4 . In addition, node 14 covers node 12. Thus, node 12 is removed from G_4 to prune the set, i.e., to compute H_5 .

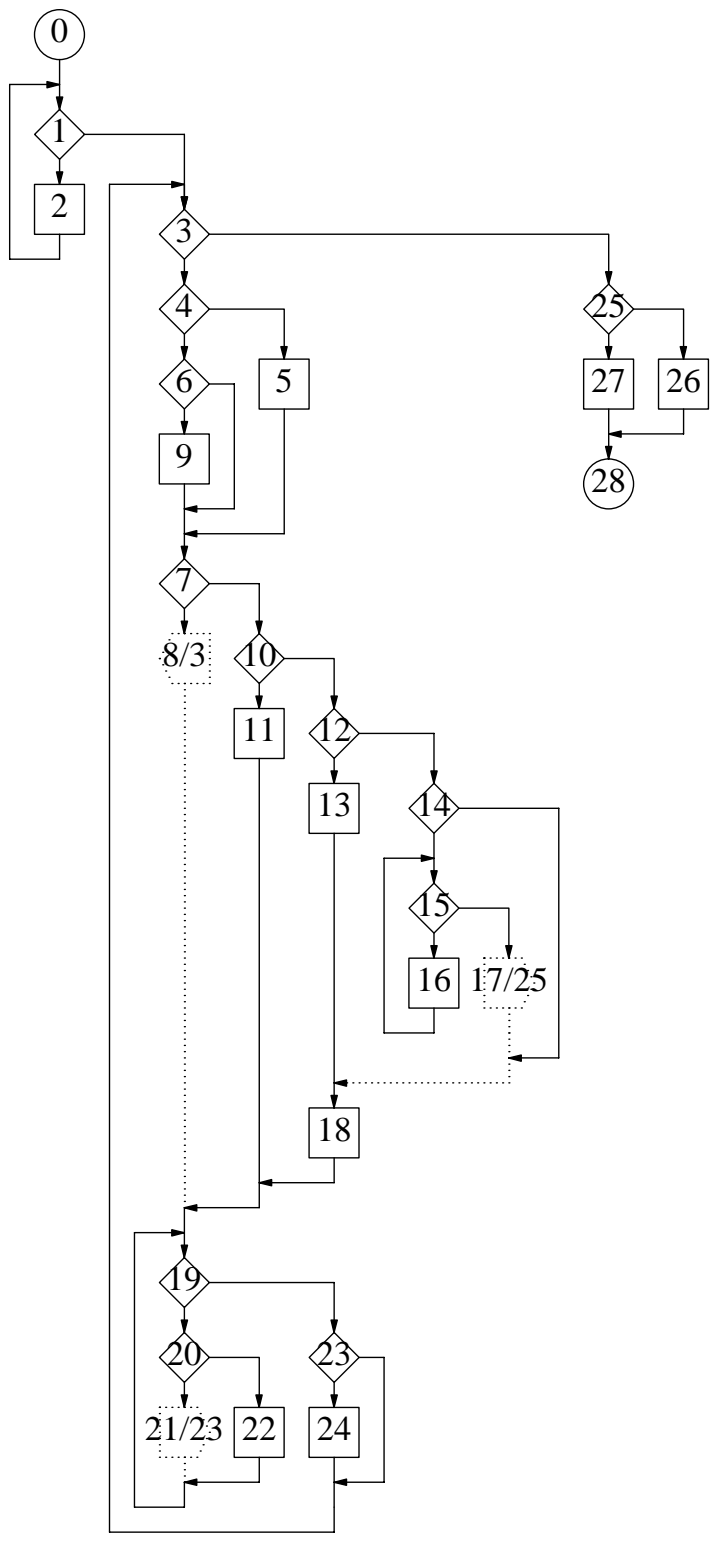


FIGURE 4.1. A semi-structured flowchart with break and continue.

Note that, on the other hand, node 12 does not cover node 14. Therefore, node 12 is moved into G_3 shown as shaded in G_3 .

4.4.4 Step 4

In pruning G_3 , node 12 covers node 10. Nevertheless, node 10 does not cover node 12. Thus, node 10 is removed from G_3 and moved into G_2 as shown shaded.

4.4.5 Step 5

To prune G_2 , we first note that nodes 4 and 7 are mutually essential to each other. Therefore, one of them (say 7) is removed from the set. Then, we also note that node 10 covers node 4. Nevertheless, node 4 does not cover node 10. Thus, node 4 in turn is removed from G_2 and moved into G_1 as shown shaded.

4.4.6 Step 6

In pruning G_1 , we first note that nodes 0, 1, 3, 25, and 28 are mutually essential to each other. Therefore, all of them except one (say 0) is removed from the set. Then, node 4 covers node 0. Nevertheless, node 0 does not cover node 4. Thus, node 0 is removed from G_1 and moved into newly introduced empty set G_0 .

4.4.7 Step 7

Note that H_1 is always a set of a single node $x \in E_{max}$, i.e., *start* node if we chose the smallest node number. The completely pruned merged sets are now shown in Table 4.1.

Note that the number of H sets is at least the number of G sets. Table 4.3 and Table 4.4 for the flowchart in Figure 4.3 show that the number of H sets and the number of G sets can be the same.

4.5 Properties of Pruned Merged Sets

THEOREM 4.1. H_i covers H_j , where $1 \leq j \leq i$.

PROOF. Follows from Remark 4.3 and the pruning process.

TABLE 4.2. Steps to prune the merged set in Table 4.7 to compute the pruned sets in Table 4.1

Step #	1	2	3	4	5	6	7
G_6/H_7	{2, 5, 8, 9, 11, 13, 16, 21, 22, 24, 26, 27}						
G_5/H_6	{6, 15, 17, 18, 20}	{6, 15, 18, 20}					
G_4/H_5		{12, 14, 19, 23}	{14, 19}				
G_3/H_4			{10, 12}	{12}			
G_2/H_3				{4, 7, 10}	{10}		
G_1/H_2					{0, 1, 3, 4, 25, 28}	{4}	
G_0/H_1						{0}	{0}

LEMMA 4.1. A node $x \in H_i$ can cover at most two nodes y and z in H_{i-1} .

PROOF. In order a node $y \in H_{i-1}$ to be covered by a node $x \in H_i$, y has to become a terminal node (see the algorithm in Chapter 5) in both domination DT and post-dominance PT trees by the removal of node x (and possibly some other nodes $x' \in H_i$) from both trees. In particular, if x is a child of y , then all other children $x' \in H_i$ of y are also removed. Note that the removal of x can make at most two new terminal nodes, one in each of DT and PT. In other words, x can cover at most two nodes in H_{i-1} set.

TABLE 4.3. Merged levels of sets of nodes for the flowchart in Figure 4.3

G_i	Merged sets of nodes
1	$E_1 = \{0, 1, 9\}$
2	$E_2 = \{2\}$
3	$E_3 = \{3\}$
4	$E_4 = \{4, 8\}$ $E_5 = \{7\}$
5	$E_9 = \{5, 6\}$

TABLE 4.4. Pruned merged sets of nodes in Table 4.3. The number of H sets and the number of G sets are the same

H_i	Pruned sets of nodes
1	{0}
2	{2}
3	{3, 8}
4	{4, 7}
5	{5, 6}

Note that Lemma 4.1 does not hold for G_i and G_{i-1} . For example, node 11 in G_4 in Table 4.5 for Figure 4.2 covers all the nodes 3, 6, 9, and 12 in G_3 .

LEMMA 4.2. If a node $x \in H_i$ covers two nodes y and z in H_{i-1} , then there also exists a node $w \in H_i$ such that w covers only one of y and z .

PROOF. By the definition of H_{i-1} , nodes y and z do not cover each other. Otherwise, one of them would have been removed from H_{i-1} . Note that in a semi-structured flowchart, if x covers y and z and y and z do not cover each other, then (1) one of them has to dominate x and the other has to post-dominate x . Let $\mathcal{D}_{y,x} = \mathcal{P}_{z,x} = 1$ and (2) there has to be a path from y to the end node that does not go through z . Such a path is possible only with an escape node, i.e., break or continue node that y dominates. For $i = \min$, $w \in H_i$ is the escape node that $y \in H_{i-1}$ immediately dominates. Therefore, w covers y . Note that in order a node w to cover a node y , y has to either immediately dominate or immediately post-dominate w . Therefore, y is the only node that w covers since w 's immediate post-dominator cannot be in H_{i-1} . For $i = \min - 1$, if there are two nodes y' and z' in H_{i-1} that are covered by the same node $x' \in H_i$, then $w' \in H_i$ is the node that immediately dominates the escape node $w \in H_{i+1}$. In other words, node $y \in H_i$ becomes node w' (and z becomes x'). Note that y' immediately dominates w' . Hence, w' covers y' . Note also that y' is the only node that is covered by w' since the immediate post-dominator of w' cannot be in the same set with w' . This analogy can easily be generalized for all pairs of sets H_i and H_{i-1} .

TABLE 4.5. Merged levels of sets of nodes for the flowchart in Figure 4.2

G_i	Merged sets of nodes
1	$E_1 = \{0, 1, 19\}$
2	$E_2 = \{2, 4, 17\}$
3	$E_3 = \{3, 6, 9, 12\}$
4	$E_4 = \{4, 5, 7, 8, 10, 11, 13, 15, 16, 18\}$

THEOREM 4.2. $|H_i| \geq |H_j|$, where $1 \leq j \leq i$.

PROOF. Obvious from Theorem 4.2, Definition 4.2, Lemma 4.1, and Lemma 4.2.

Covering and non-increasing properties that we stated in Theorem 4.1 and Theorem 4.2 are important properties we have observed. These properties will later on be utilized to obtain a smaller size set W' to cover a set W of nodes.

TABLE 4.6. Nodes and their level of essentialness for the flowchart in Figure 4.1

δ_x	Nodes and their levels of essentialness
29	$E_1 = \{0, 1, 3, 25, 28\}$
21	$E_2 = \{4, 7\}$
15	$E_3 = \{10\}$
9	$E_4 = \{19, 23\}$
7	$E_5 = \{12\}$
4	$E_6 = \{14\}$
3	$E_7 = \{15, 17, 20\}$
2	$E_8 = \{6, 18\}$
1	$E_9 = \{2, 5, 8, 9, 11, 13, 16, 21, 22, 24, 26, 27\}$

TABLE 4.7. Levels in Table 4.6 after merging

G_i	Levels after merging
1	$E_1 = \{0, 1, 3, 25, 28\}$
2	$E_2 = \{4, 7\}$
3	$E_3 = \{10\}$
4	$E_4 = \{19, 23\}$ $E_5 = \{12\}$ $E_6 = \{14\}$
5	$E_7 = \{15, 17, 20\}$ $E_8 = \{6, 18\}$
6	$E_9 = \{2, 5, 8, 9, 11, 13, 16, 21, 22, 24, 26, 27\}$

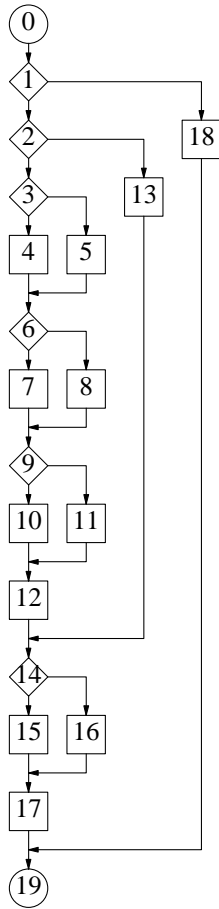


FIGURE 4.2. A flowchart disproving Lemma 4.1 for G_i .

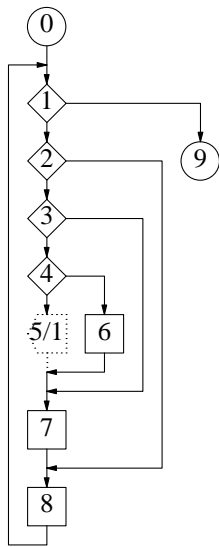


FIGURE 4.3. A semi-structured flowchart. Node $6 \in H_5$ covers nodes 4 and 7 in H_4 and node $5 \in H_5$ covers node 4 in H_4

Chapter 5

Application to Program Testing

In this chapter, we describe an application of pruned merged sets H_i to program test coverage.

A program can be represented by a flowchart. In view of this, a form of testing a statement in the program is to cover the corresponding node for the statement in the flowchart. If we want to cover all the nodes in a flowchart, covering all the nodes in the essential set introduced by Bertolino is sufficient since covering all those nodes will assure covering all the nodes in the flowchart. However, covering all nodes in a flowchart may not be a desired goal in certain program testing strategies. For example, covering all nodes in the flowchart is typically unnecessary in regression testing. Therefore, instead of covering all the nodes in the flowchart, one may want to cover a subset W of those.

In the following sections, we present two methods to find a smaller size W' of nodes to cover the desired set W .

5.1 Covering a Subset of Nodes

DEFINITION 5.1. A set S of nodes is *sufficient* for a set W of nodes if covering S will suffice to cover W . That is, any set of paths that covers S will also cover W .

Let W denote a subset of nodes we want to cover in a flowchart. Obviously, Bertolino's essential set is sufficient for W since it covers all the nodes in the flowchart.

Here, we provide two methods to find a smaller size set W' of nodes that will be sufficient to cover our desired subset W . Note that $W' =$ the essential set, which is the smallest size set we could obtain if $W =$ the set of all nodes in the flowchart.

In the first method, we do not have any restrictions on W' . That is, the set W' in general may include some nodes which are not in W . We utilize our pruned merged sets to obtain such w' .

In the second method, we enforce W' to be a subset of W . The fact that the essential set is a subset of all nodes was the motivation for us to have such restriction to generalize the essential set concept in terms of application.

5.2 Finding W'

We utilize the pruned merged sets H_i to obtain W' . We first determine the smallest pruned set H_i that covers W . Then, we refine the set H_i by removing the redundant nodes from it to compute W' .

5.2.1 Determining the Smallest Sufficient Set H_i for W

To find W' , we first determine the smallest size pruned set H_i such that H_i covers W . Then we further prune it.

Note that each node x in a flowchart either appears in a pruned set H_i or is represented by an equivalent (mutually essential) node $y \in H_i$. This is from the fact that during the pruning process of G_j to compute H_i , all mutually essential nodes except one (with the smallest node number as the representative to the others) are removed from G_j . For the flowchart in Figure 5.1, Table 5.7 shows the nodes that are removed from the merged sets in Table 5.5 to prune the sets and their representatives in the pruned sets in Table 5.6.

DEFINITION 5.1. Let $h_level(x) = j$ such that node x or its representative belongs to H_j and $i = \max\{h_level(x) : \text{for all } x \in W\}$. The set H_i is called the *smallest size sufficient pruned set* (SP) for W .

THEOREM 5.1. For a given set of nodes W , the SP set H_i is the smallest size pruned set that covers W .

PROOF. Theorem 4.1 basically says that a pruned set H_i is sufficient for all pruned sets H_j , where $i \geq j$. Definition 5.1 says that all nodes in W will be in some pruned sets H_j , where $i \geq j$. Thus, proof is evident.

5.2.2 Pruning the SP Set to Obtain W'

DEFINITION 5.2. Let a set S of nodes covers a set W of nodes. Node $x \in S$ is called *redundant* for W if $S - \{x\}$ also covers W .

We further refine the SP set to obtain W' by removing all redundant nodes from the SP set. Therefore, $W' = SP - \{x : \text{all redundant nodes } x \in SP\}$.

5.3 An Example Application

We have developed the following C program to process two types of loan applications, signature loan and mortgage loan. When we were developing the program, efficiency in terms of execution or otherwise was not the main concern. Instead, for the illustration purposes we focused on a certain structure while keeping it a reasonably good looking program.

The program utilizes no arbitrary goto but break and continue. Therefore, it is semi-structured. We give the corresponding flowchart for the program in Figure 5.1. In the flowchart, a node x represents all the statements denoted by $[x]$ in the program. Note that a single node x in the flowchart may represent multiple statements in the program to assure that the flowchart has no sequential nodes as we discussed in Chapter 2.

5.3.1 Example for Finding W

Let $W = \{6, 24, 26, 38, 68\}$ for the program given below. Table 5.1 shows that the SP set for W is H_8 .

Table 5.2 shows the subsets of nodes of W covered by nodes in the SP set. From the table, we notice that nodes 13, 28, and 34 are redundant. We need node 26 to

TABLE 5.1. Computing the SP set for $W = \{6, 24, 26, 38, 68\}$ for the flowchart in Figure 5.1

$W =$	{	6,	24,	26,	38,	68	}
Rep. =	{	6,	16,	26,	37,	0	}
$j =$	{	2,	7,	8,	4,	1	}
SP set=	H_8						

cover itself. Then, we can chose one of the nodes 40, 53, and 55 to ensure covering node 38. Thus, W' would be $\{26, 40\}$.

TABLE 5.2. Refining the SP set H_8 for $W = \{6, 24, 26, 38, 68\}$ for the flowchart in Figure 5.1

$x \in H_8$	Nodes $y \in W$ covered by x
13	{6, 68}
26	{6, 24, 26, 68}
28	{6, 24, 68}
34	{6, 68}
40	{6, 38, 68}
53	{6, 38, 68}
55	{6, 38, 68}

```
//Osman Kandara
//Simplified loan calculator program
//to demonstrate the application of the concept
//Sept 25, 2003
//gcc loancalc.c
#include <stdio.h>

void PrintMenu(void)
{printf("LOAN APPLICATION MENU\n");
 printf("-----\n");
 printf("1- Signature loan\n");
 printf("2- Mortgage loan\n");
 printf("3- Exit\n");
 printf("Enter loan type (1-3):");
}

int main(void)
{char yesNo, applicant[25];
 int option, occupation, term, fromStart,
    mortFloodLevel, creditScore;
 float loanAmnt, loanRate, minLoan, totInt, intrRate,
```

```

        mortDown, mortHazIns, homePrice, mortApValue,
        mortFloodIns, mortIns, monthPay, mortMonthIns;
[1]PrintMenu();
[1]scanf("%d", &option);
[2]while ((option < 1) || (option > 3)) {
[3]    PrintMenu();
[3]    scanf("%d", &option);
    }
[4]fromStart = 1;
[5]while (option != 3) {
[6]    if (option == 1) {
[7]        printf("Loan type: Signature loan\n");
[7]        printf("Enter applicant's name:");
[7]        fflush(stdin);
[7]        fgets(applicant, 24, stdin);
[7]        printf("Minimum loan ammounts\n");
[7]        printf("-----\n");
[7]        printf("Students: $3000.00\n");
[7]        printf("Others : $5000.00\n");
[7]        printf("Enter occupation (1-Student, 2-Others):");
[7]        scanf("%d", &occupation);
[8]        if (occupation == 1)
[9]            minLoan = 3000.0;
[10]        else minLoan = 5000.0;
[11]        printf("Enter loan amount:");
[11]        scanf("%f", &loanAmnt);
[12]        if (loanAmnt < minLoan) {
[13]            printf("Loan amount is too low!\n");
[13]            printf("Continue? (Y-Go with minimum, N-Exit):");
[13]            fflush(stdin);
[13]            scanf("%c", &yesNo);
[14]            if ((yesNo == 'N') || (yesNo == 'n')) {
[15]                printf("Loan application not completed!\n");
                break;
            }
        }
    }
[16]    printf("Enter loan term in months:");
[16]    scanf("%d", &term);
[17]    if (term < 24)
[18]        loanRate = 0.05;
[19]    else loanRate = 0.06;
[20]    if (loanAmnt < 5000.0)
[21]        loanRate = loanRate + 0.05;
[22]    else loanRate = loanRate + 0.001;
[23]    printf("Enter credit score (1-Good, 2-Fair, 3-Bad):");
[23]    scanf("%d", &creditScore);
[24]    if (creditScore == 2)

```

```

[25]         loanRate = loanRate * 1.1;
[26]     else if (creditScore > 2) {
[27]         printf("Loan not approved due to low score!\n");
           break;
           }
[28]     printf("\nSignature Loan Summary\n");
[28]     printf("-----\n");
[28]     printf("Applicant      :%s\n", applicant);
[28]     printf("Loan ammount   :$%8.2f\n", loanAmnt);
[28]     printf("Loan term      :%d months\n", term);
[28]     printf("Loan rate      :%4.2f%%/year\n", (loanRate*100));
[28]     totInt = loanAmnt * (loanRate/12) * term;
[28]     printf("Total interest :$%6.2f/term\n", totInt);
[28]     monthPay = (loanAmnt + totInt)/term;
[28]     printf("Monthly payment:$%6.2f\n", monthPay);
} //signature loan
else {
[29]     if (fromStart) {
[30]         printf("Loan type: Mortgage\n");
[30]         printf("Enter applicant's name:");
[30]         fflush(stdin);
[30]         fgets(applicant, 24, stdin);

           }

[31]     printf("Enter loan term in years (15 or 30):");
[31]     scanf("%d", &term);
[32]     if (term == 15)
[33]         loanRate = 0.055;
[34]     else if (term == 30)
[35]         loanRate = 0.063;
           else {
[36]             printf("Loan term not supported!\n");
[36]             fromStart = 0;
           continue;
           }

[37]     printf("Enter credit score (1-Good, 2-Fair, 3- Bad):");
[37]     scanf("%d", &creditScore);
[38]     if (creditScore == 2)
[39]         loanRate = loanRate * 1.2;
[40]     else if (creditScore > 2) {
[41]         printf("Loan not approved due to low score!\n");
           break;
           }

[42]     printf("Enter home's price:");
[42]     scanf("%f", &homePrice);
[42]     printf("Enter down payment (%%):");
[42]     scanf("%f", &mortDown);

```

```

[42]     mortDown = mortDown/100;
[42]     loanAmnt = homePrice - (homePrice * mortDown);
[42]     printf("Enter appraisal value:");
[42]     scanf("%f", &mortApValue);
[43]     if (loanAmnt > mortApValue) {
[44]         printf("Loan cannot be approved due to low appraisal!\n");
            break;
        }
[45]     mortFloodIns = mortIns = 0.0;
[45]     mortHazIns = 0.65 * homePrice * 0.005;
[46]     if (mortDown < 0.20)
[47]         mortIns = loanAmnt * 0.002;
[48]     printf("Accepted flood zone levels:\n");
[48]     printf("-----\n");
[48]     printf("2-Moderate high, 1-low, 0-Not in flood zone\n");
[48]     printf("Enter flood zone level (0-2):");
[48]     scanf("%d", &mortFloodLevel);
[49]     if (mortFloodLevel == 1)
[50]         mortFloodIns = 0.65 * homePrice * 0.003;
[51]     else if (mortFloodLevel == 2)
[52]         mortFloodIns = 0.65 * homePrice * 0.006;
[53]         else if ((mortFloodLevel > 2) || (mortFloodLevel < 0)) {
[54]             printf("Flood zone not accepted for approval!\n");
                break;
            }
        }
[55]     printf("\nMortgage Loan Summary\n");
[55]     printf("-----\n");
[55]     printf("Applicant           :%s\n", applicant);
[55]     printf("Home price            :$%10.2f\n", homePrice);
[55]     printf("Down payment          :$%10.2f\n", (homePrice * mortDown));
[55]     printf("Loan ammount          :$%10.2f\n", loanAmnt);
[55]     printf("Term                  :%d months\n", (term * 12));
[55]     printf("Mortgage Rate         :%4.2f%%/year\n", (loanRate*100));
[55]     totInt = loanAmnt * loanRate * term;
[55]     printf("Total interest       :$%10.2f/term\n", totInt);
[56]     if (mortFloodLevel == 0)
[57]         printf("Flood ins.           :NA\n");
[58]     else printf("Flood ins.           :$%8.2f/year\n", mortFloodIns);
[59]     printf("Hazard ins.          :$%8.2f/year\n", mortHazIns);
[60]     if (mortDown < 0.20)
[61]         printf("Mortgage ins.        :$%8.2f/month\n", (mortIns/12));
[62]     else printf("Mortgage ins.        :NA\n");
[63]     mortMonthIns = (mortFloodIns + mortHazIns)/ 12 + mortIns;
[63]     printf("Total monthly insurance:$%6.2f\n", mortMonthIns);
[63]     monthPay = (totInt + loanAmnt) / (term * 12) + mortMonthIns;
[63]     printf("Total monthly payment :$%6.2f\n", monthPay);
} //mortgage

```

```

[64]     printf("Enter your choice (1-Go to menu, 2-Exit program):");
[64]     scanf("%d", &option);
[65]     if (option == 1) {
[66]         fromStart = 1;
[66]         PrintMenu();
[66]         scanf("%d", &option);
        }
        else {
[67]         printf("No more application to be processed!\n");
            break;
        }
    } //while
[68]printf("Exiting...\n");
} //main

```

In the previous section, we generalized the essential set by finding a smaller set W' to cover a subset W of nodes. The set W' may include some nodes that are not in W .

5.4 Finding W' : $W' \subseteq W$

In this section, we not only find the smallest set W' to cover W , we also restrict W' to be the subset of W . Note that in the essential set case, W' = the essential set and W' is the smallest subset of W where $W =$ all nodes.

We directly utilize the DT and PT to do the second level generalization. We executes the following steps to compute smallest subset W' of W : (1) we throw away all nodes (and their edges incident to them) of the DT which are not in W and which do not have a descendent in W ; (2) we do the same thing in Step (1) for the PT; (3) we compute W' =the intersection of terminal nodes in the two trees.

Note that if a node x covers a node y , then $\mathcal{D}_{y,x} = 1$ or $\mathcal{P}_{y,x} = 1$. In Step 1, we eliminate all nodes that will not be covered through domination by any node in W . In other words, all nodes in the remaining tree will be covered through domination at least by a node in W . That is, all terminal nodes in the remaining tree will be among nodes in W . We basically achieve the same thing for the PT

in Step 2. That is, all terminal nodes in the remaining PT will be among nodes in W . In Step 3, we compute the smallest subset W' of W . It is already proven by Bertolino that taking the intersection of terminal nodes in both DT and PT gives the smallest subset of nodes to cover all nodes in both trees.

5.4.1 Example

Let us consider the flowchart in Figure 4.1. The DT and PT for the flowchart are given in Figure 6.2 and Figure 6.3.

Let $W = \{3, 7, 12, 15, 19, 25\}$. After the step (1), the terminal nodes in the DT will be 15, 19, and 25. After step (2), the terminal nodes in PT will be 7, 12, 15, and 19. Thus, $W' = \{15, 19\}$ as the common terminal nodes in the both updated DT and updated PT.

5.5 Samples

We have randomly chosen various sets W and computed the W' for each W . Table 5.3 shows each W along with the corresponding smallest size set H_i and computed W' .

TABLE 5.3. Various sets W and their corresponding W' for the flowchart in Figure 4.1

W	H_i	W'
$\{4, 12, 19\}$	H_5	$\{14, 19\}$
$\{15, 24, 28\}$	H_7	$\{16, 24\}$
$\{7, 9, 12, 14, 28\}$	H_7	$\{16\}$
$\{2, 10, 12, 21, 26\}$	H_7	$\{2, 26, 21, 13\}$
$\{0, 1, 10, 14, 16, 17, 18, 19, 21, 24\}$	H_7	$\{16, 21, 24, 13\}$
$\{1, 3, 4, 6, 7, 10, 15, 23, 25, 28\}$	H_6	$\{6, 15, 20\}$

TABLE 5.4. Nodes and their level of essentialness for the flowchart in Figure 5.1

δ_x		Nodes and their levels of essentialness
70	$E_1 =$	$\{0, 1, 2, 4, 5, 68, 69\}$
62	$E_2 =$	$\{6\}$
35	$E_3 =$	$\{29, 31, 32\}$
29	$E_4 =$	$\{37, 38\}$
23	$E_5 =$	$\{42, 43\}$
22	$E_6 =$	$\{7, 8, 11, 12\}$
19	$E_7 =$	$\{45, 46, 48, 49\}$
17	$E_8 =$	$\{64, 65\}$
13	$E_9 =$	$\{16, 17, 20, 23, 24\}$
11	$E_{10} =$	$\{55, 56, 59, 60, 63\}$
4	$E_{11} =$	$\{51\}$
3	$E_{12} =$	$\{13, 14, 34\}$
2	$E_{13} =$	$\{26, 28, 40, 53\}$
1	$E_{14} =$	$\{3, 9, 10, 15, 18, 19, 21, 22, 25, 27, 30, 33, 35, 36, 39, 41, 44, 47, 50, 52, 54, 57, 58, 61, 62, 66, 67\}$

TABLE 5.5. Levels in Table 5.4 after merging.

G_i		Merged sets of nodes
1	$E_1 =$	$\{0, 1, 2, 4, 5, 68, 69\}$
2	$E_2 =$	$\{6\}$
3	$E_3 =$	$\{29, 31, 32\}$
4	$E_4 =$	$\{37, 38\}$
5	$E_5 =$ $E_6 =$	$\{42, 43\}$ $\{7, 8, 11, 12\}$
6	$E_7 =$ $E_8 =$ $E_9 =$	$\{45, 46, 48, 49\}$ $\{64, 65\}$ $\{16, 17, 20, 23, 24\}$
7	$E_{10} =$ $E_{11} =$ $E_{12} =$ $E_{13} =$	$\{55, 56, 59, 60, 63\}$ $\{51\}$ $\{13, 14, 34\}$ $\{26, 28, 40, 53\}$
8	$E_{14} =$	$\{3, 9, 10, 15, 18, 19, 21, 22, 25, 27, 30, 33, 35, 36, 39, 41, 44, 47, 50, 52, 54, 57, 58, 61, 62, 66, 67\}$

TABLE 5.6. Pruned merged sets of nodes in Table 5.5.

H_i	Pruned sets of nodes
1	{0}
2	{6}
3	{29}
4	{37}
5	{42}
6	{7, 45}
7	{16, 51, 64}
8	{13, 26, 28, 34, 40, 53, 55}
9	{3, 9, 10, 15, 18, 19, 21, 22, 25, 27, 30, 33, 35, 36, 39, 41, 44, 47, 50, 52, 54, 57, 58, 61, 62, 66, 67}

TABLE 5.7. Mutually essential nodes and their representatives in Table 5.6.

$x \in H_i$	All mutually essential nodes y represented by $x \in H_i$
0	{1, 2, 4, 5, 68, 69}
7	{8, 11, 12}
13	{14}
16	{17, 20, 23, 24}
29	{31, 32}
37	{38}
42	{43}
45	{46, 48, 49}
55	{56, 59, 60, 63}
64	{65}

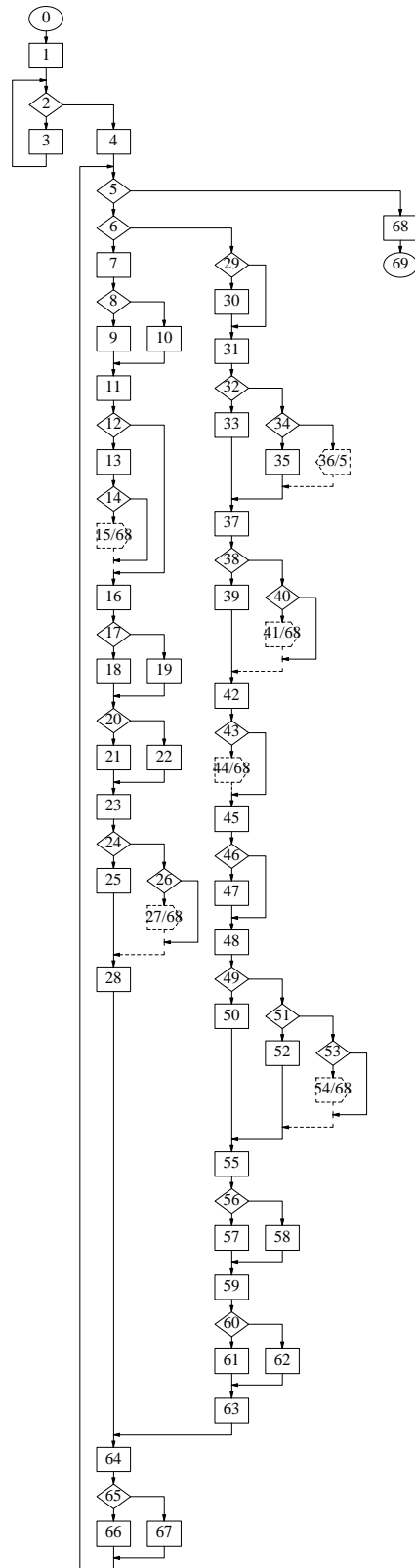


FIGURE 5.1. Flowchart for the program loancalc.c. Here, a node x represents all the statements denoted by $[x]$ in the program.

Chapter 6

Algorithms

We provide two efficient algorithms namely *DegreesAndLevels* (DAL) and *Pruned-MergedLevels* (PML). Algorithm DAL computes the degrees of essentialness δ_x for all nodes x and classify them into various levels of essentialness E_i . PML computes all pruned merged levels H_i .

Both algorithms mainly utilize the domination tree $DT(F)$ and post–domination tree $PT(F)$ of a flowchart F . A linear algorithm to compute the $DT(F)$ is given in [24]. The same algorithm can also be used to compute $PT(F)$ after F is reversed. Note that post–domination tree of a flowchart F is the same with the domination tree of the reversed flowchart F' , which is simply the digraph obtained by reversing the direction of each arc in F . Note that F' is not a flowchart in most cases.

Note that both algorithms are independent of each other. In other words, to compute the pruned merged levels, we don't need to compute the degrees and the levels of essentialness. From the $DT(F)$ and $PT(F)$, we can directly compute all the pruned merged levels H_i .

6.1 Algorithm DAL

We provide an efficient algorithm in Figure 6.1 called DAL to compute the degrees and levels of essentialness for all nodes x in a flowchart F , where the $DT(F)$ and $PT(F)$ are given. A linear algorithm to compute the $DT(F)$ is already developed in [24].

Algorithm DAL utilizes the property we stated in Theorem 3.1. Let $\mathcal{S}_x^D = \{y : \mathcal{D}_{x,y} = 1\}$ and $\mathcal{S}_x^P = \{y : \mathcal{P}_{x,y} = 1\}$ for a node x in F . Then, Theorem 3.1 says that $\delta_x = |\mathcal{S}_x^D| + |\mathcal{S}_x^P| - |\mathcal{S}_x^D \cap \mathcal{S}_x^P|$.

ALGORITHM: DegreesAndLevels (DAL)**Input** : The DT and PT of a flowchart F.**Output** : δ_x for all nodes x and all E_i .

1. Compute the sizes of all subtrees in both DT and PT.
2. [compute the degree of essentialness for all nodes utilizing Theorem 3.1]
 For (each node x) do the following:
 - Let $sizeSubDT(x)$ denote the size of subtree rooted at x in DT.
 - Let $sizeSubPT(x)$ denote the size of subtree rooted at x in PT.
 - If (node x is not *while-do*), then $\delta_x = sizeSubDT(x) + sizeSubPT(x) - 1$.
 - Else do the following:
 - [Get x 's immediate post-dominator]
 - $y =$ the father of x in PT.
 - [Computes the number of all nodes k such that $\mathcal{D}_{x,k} = 1$ and $\mathcal{P}_{x,k} = 0$]
 - $numJustDoms(x) = sizeSubPT(y) - sizeSubPT(x) + sizeSubDT(y) - 1$.
 - [Compute $|\mathcal{S}_x^D \cap \mathcal{S}_x^P|$]
 - $numDomsAndPosts(x) = sizeSubDT(x) - numJustDoms(x)$.
 - [Compute δ_x]
 - $\delta_x = sizeSubDT(x) + sizeSubPT(x) - numDomsAndPosts$.
3. [Compute levels of essentialness]
 For (each node x) do the following:
 - Classify x into E_i , where i is the order of the magnitude of δ_x among all.

FIGURE 6.1. An efficient algorithm to compute the degrees and levels of essentialness for all nodes.

As the theorem says, δ_x comes from both the number of nodes x dominates and the number of nodes x post dominates. Subtracting $|\mathcal{S}_x^D \cap \mathcal{S}_x^P|$ ensures that we count the same node once only.

Node x is called a *while-do* node if it is the head of a while-do loop. Otherwise, x is called *regular*. Note that $|\mathcal{S}_x^D \cap \mathcal{S}_x^P| = 1$ for all regular nodes x since node x is the only node that x both dominates and post dominates. Thus, it is the only node counted twice. Let $sizeSubDT(x)$ and $sizeSubPT(x)$ denote the sizes of subtrees rooted at node x in both DT(F) and PT(F), respectively. Note that $sizeSubDT(x) = |\mathcal{S}_x^D|$ and $sizeSubPT(x) = |\mathcal{S}_x^P|$. Thus, for a regular node x we compute δ_x as follows:

$$\delta_x = \text{sizeSubDT}(x) + \text{sizeSubPT}(x) - 1$$

Note that a while–do node x both dominates and post–dominates all nodes within its loop body in a structured flowchart and possibly some nodes in a semi–structured flowchart. Let $\text{numJustDoms}(x)$ denote the number of all nodes k such that $\mathcal{D}_{x,k} = 1$ and $\mathcal{P}_{x,k} = 0$ and y denote x 's immediate post–dominator. Then, we can compute $\text{numJustDoms}(x)$ as follows:

$$\text{numJustDoms}(x) = \text{sizeSubPT}(y) - \text{sizeSubPT}(x) + \text{sizeSubDT}(y) - 1$$

Let $\text{numDomsAndPosts} = |\mathcal{S}_x^D \cap \mathcal{S}_x^P|$. Then, we can compute $\text{numDomsAndPosts}(x)$ as follows.

$$\text{numDomsAndPosts}(x) = \text{sizeSubDT}(x) - \text{numJustDoms}(x)$$

6.1.1 Example

We use the flowchart in Figure 4.1 to illustrate our algorithms. The DT and PT for the flowchart are given in Figure 6.2 and Figure 6.3 respectively. In both trees, $(x, \#n)$ inside a node x denotes that the size of subtree rooted at that node x is n .

Consider node 23 in the flowchart. It is a regular node. From the DT and PT, we note that $\text{sizeSubDT}(23) = 2$ and $\text{sizeSubPT}(23) = 8$. Thus, the algorithm computes $\delta_{23} = 2 + 8 - 1 = 9$.

Consider node 19. It is a while–do node. Note that the father of node 19 in PT is node 23. Thus, its immediate post–dominator is node 23. The algorithm first computes $\text{numJustDoms}(19) = \text{sizeSubPT}(23) - \text{sizeSubPT}(19) + \text{sizeSubDT}(23) - 1 = 8 - 5 + 2 - 1 = 4$. Then, the algorithm computes $\text{numDomsAndPosts}(19) = \text{sizeSubDT}(19) - \text{numJustDoms}(19) = 6 - 4 = 2$. Finally, $\delta_{19} = \text{sizeSubDT}(19) + \text{sizeSubPT}(19) - \text{numDomsAndPosts}(19) = 6 + 5 - 2 = 9$.

Nodes and their degrees of essentialness are shown in Table 6.1.

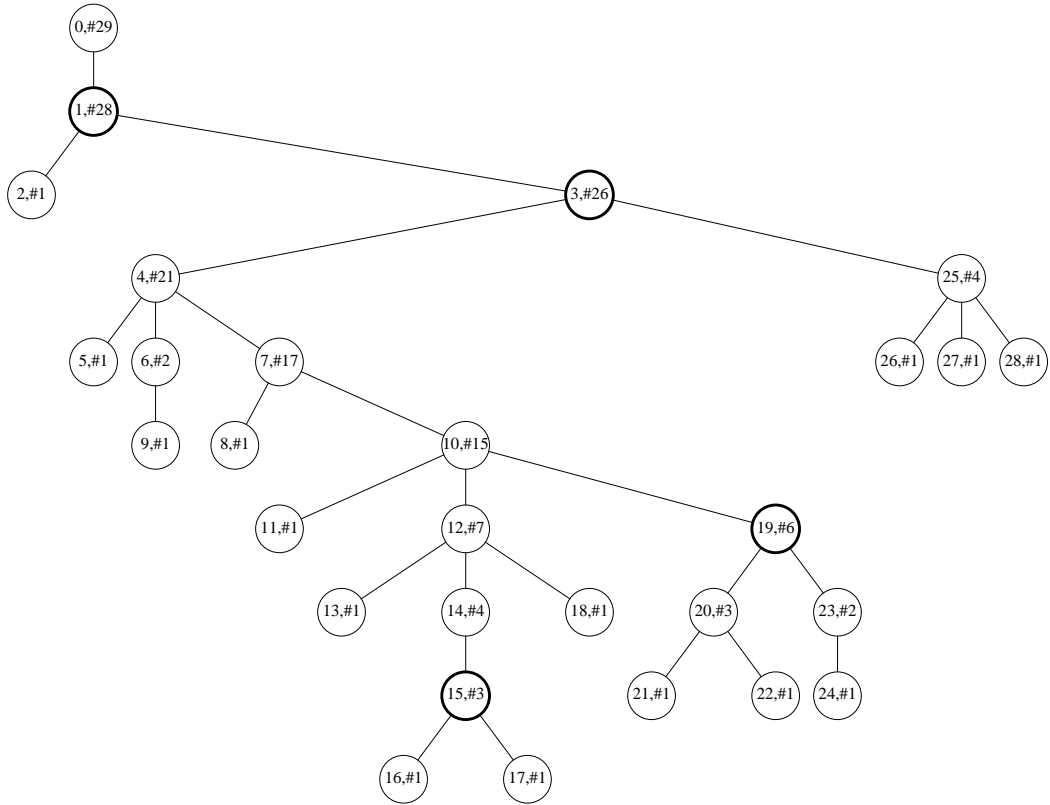


FIGURE 6.2. DT for the flowchart in Figure 4.1. Here, $(x, \#n)$ denotes that the size of the subtree rooted at node x is n . Nodes 1, 3, 15, and 19 are the while-do nodes.

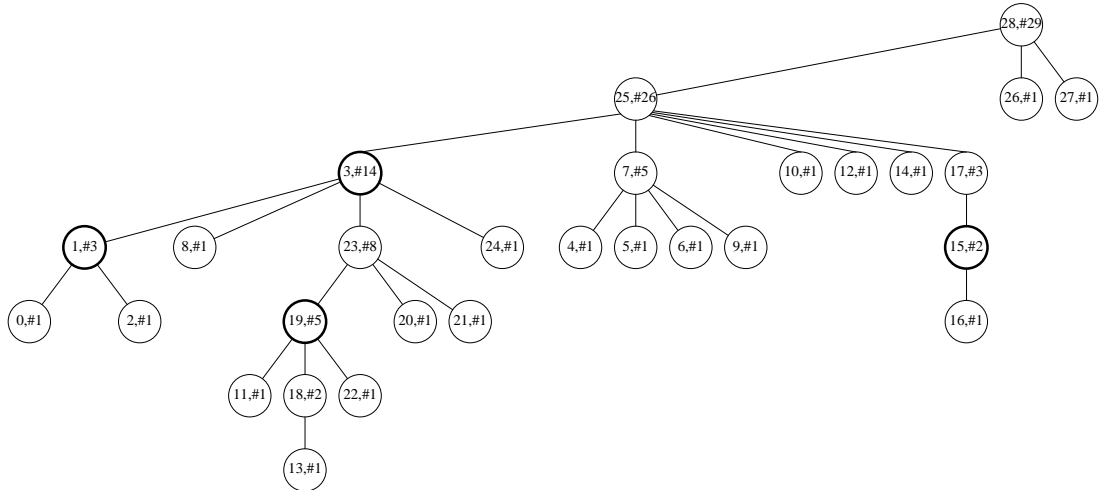


FIGURE 6.3. PT for the flowchart in Figure 4.1. Here, $(x, \#n)$ denotes that the size of the subtree rooted at node x is n . Nodes 1, 3, 15, and 19 are the while-do nodes.

TABLE 6.1. Nodes and their degrees of essentialness for the flowchart in Figure 4.1

x	δ_x	x	δ_x	x	δ_x
0	29	10	15	20	3
1	29	11	1	21	1
2	1	12	7	22	1
3	29	13	1	23	9
4	21	14	4	24	1
5	1	15	3	25	29
6	2	16	1	26	1
7	21	17	3	27	1
8	1	18	2	28	29
9	1	19	9		

6.2 Algorithm PML

We develop an efficient algorithm in Figure 6.4 called PML to compute all pruned merged levels of nodes in a flowchart F . PML is independent of our previous algorithm DAL. In other words, PML compute the pruned merged levels of essentialness of all nodes without even computing the degrees and levels of essentialness of nodes. It also utilizes the $DT(F)$ and $PT(F)$.

ALGORITHM: PrunedMergedLevels (PML)

Input : The DT and PT of a flowchart F .

Output : All pruned merged sets $H_i, 1 \leq i \leq min$.

1. [Compute CDT and CPT]

Combine all mutually essential nodes into single nodes in both DT and PT.

2. [Compute all pruned merged sets H_i starting from the right end]

$i = min$.

While (CDT is not empty) do the following:

$H_i = \{\}$.

[Compute all common terminal nodes in both CDT and CPT]

For (all terminal nodes x in CDT) do the following:

If (x is a terminal node in CPT), then $H_i = H_i \cup \{x\}$.

[Update CDT and CPT]

Remove all nodes $x \in H_i$ (and their edges incident to them) from CDT and CPT.

$i = i - 1$.

FIGURE 6.4. An efficient algorithm to compute all pruned merged sets of nodes.

First, PML computes the compact DT (CDT) and compact PT (CPT), where all mutually essential nodes in DT and PT are combined into single nodes. Figure 6.5 and Figure 6.6 show the CDT and CPT after the mutually essentialness nodes in DT in Figure 6.2 and in PT Figure 6.3 are combined, respectively. Note that nodes x and y are mutually essential if $\mathcal{D}_{x,y} = \mathcal{P}_{y,x} = 1$.

Our algorithm computes all pruned merged sets starting from the right end H_{min} . H_{min} is the set of all common terminal nodes in both CDT and CPT. Note that H_{min} and Bertolino's essential set are the same.

To compute H_{min-1} , first all nodes in H_{min} and their edges incident to them are removed from both CDT and CPT. Then, H_{min-1} becomes the set of all common terminal nodes in both updated CDT and CPT. The same analogy is applied to compute all upper level pruned merged sets. Following figures illustrate each step toward computing all the sets. We utilize the flowchart in Figure 4.1 to illustrate our algorithm.

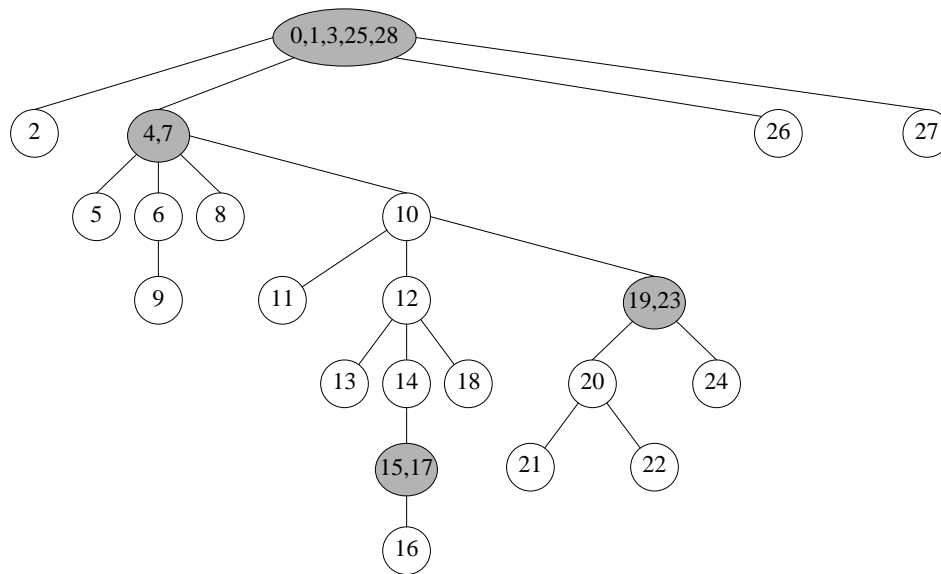


FIGURE 6.5. CDT for DT in Figure 6.2, where the mutually essential nodes are combined into single nodes shown as shaded.

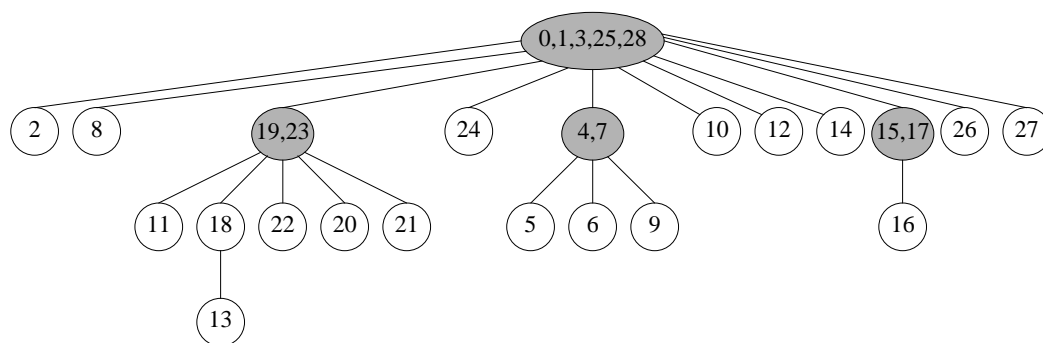


FIGURE 6.6. CPT for PT in Figure 6.3, where the mutually essential nodes are combined into single nodes shown as shaded.

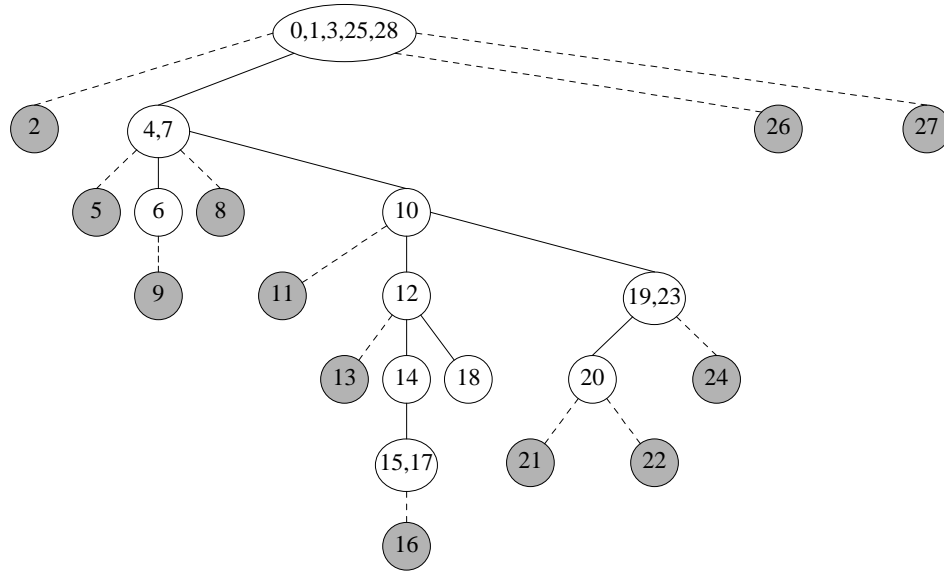


FIGURE 6.7. CDT - Computing $H_{min=7} = \{2, 5, 8, 9, 11, 13, 16, 21, 22, 24, 26, 27\}$ shown as shaded and updating CDT. Here, H_7 nodes are the common terminal nodes in both CDT and CPT in Figure 6.5 and Figure 6.6 respectively. Nodes in H_7 and their edges incident to them shown as dashed will be removed from CDT to update it.

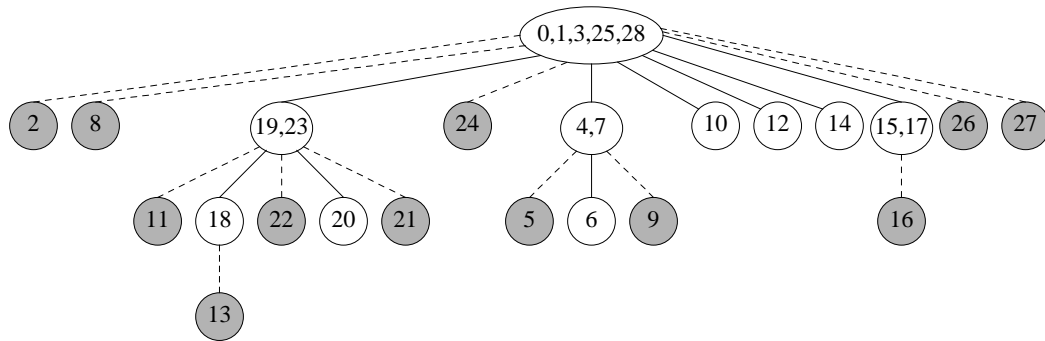


FIGURE 6.8. CPT - Computing $H_{min=7} = \{2, 5, 8, 9, 11, 13, 16, 21, 22, 24, 26, 27\}$ shown as shaded and updating CPT. Here, H_7 nodes are the common terminal nodes in both CDT and CPT in Figure 6.5 and Figure 6.6 respectively. Nodes in H_7 and their edges incident to them shown as dashed will be removed from CPT to update it.

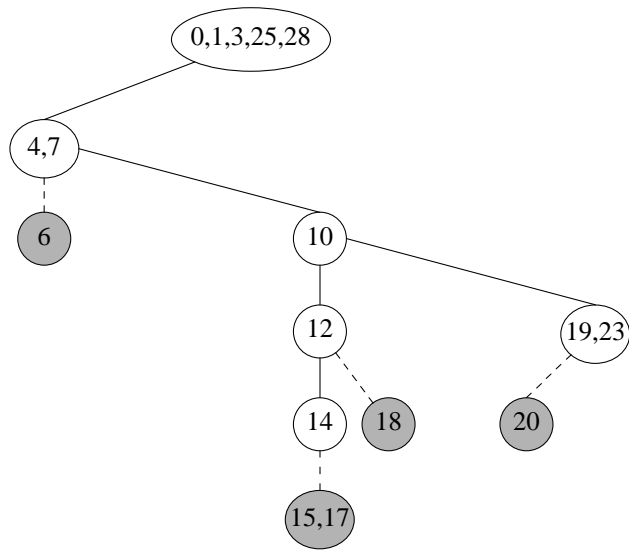


FIGURE 6.9. CDT - Computing $H_6 = \{6, 15, 18, 20\}$. Here, we take the smallest node number as the representative for the mutually essential nodes in the combined node (15,17) shown as shaded

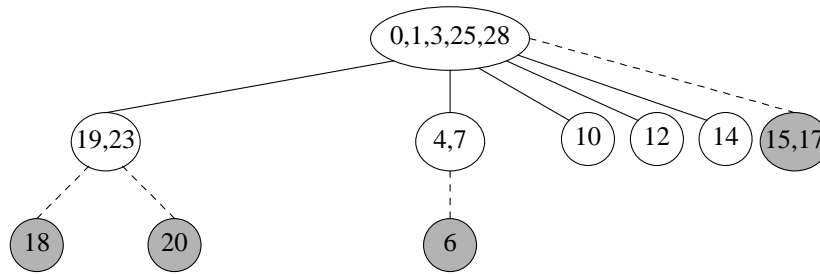


FIGURE 6.10. CPT - Computing $H_6 = \{6, 15, 18, 20\}$. Here, we take the smallest node number as the representative for the mutually essential nodes in the combined node (15, 17) shown as shaded

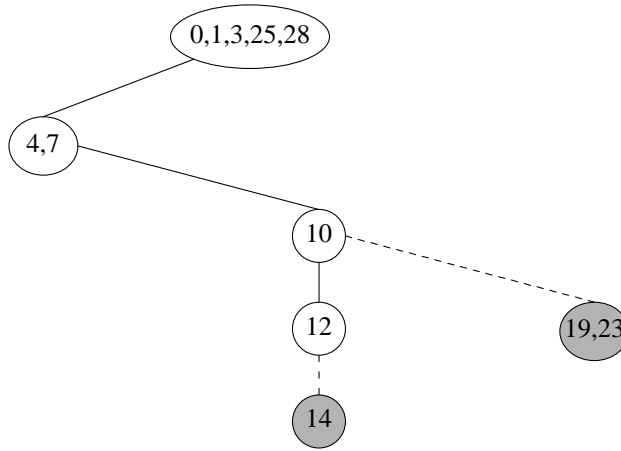


FIGURE 6.11. CDT - Computing $H_5 = \{14, 19\}$

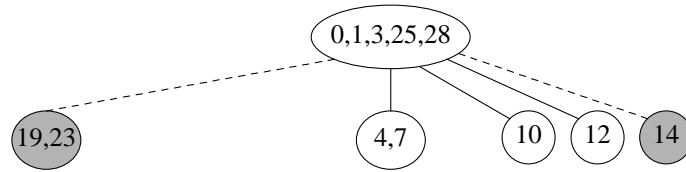


FIGURE 6.12. CPT - Computing $H_5 = \{14, 19\}$

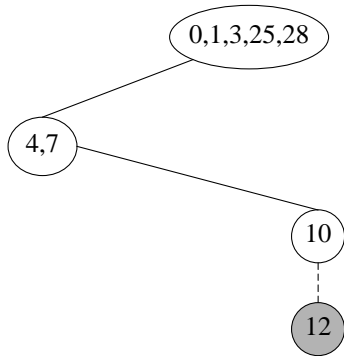


FIGURE 6.13. CDT - Computing $H_4 = \{12\}$

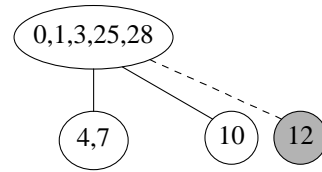


FIGURE 6.14. CPT - Computing $H_4 = \{12\}$

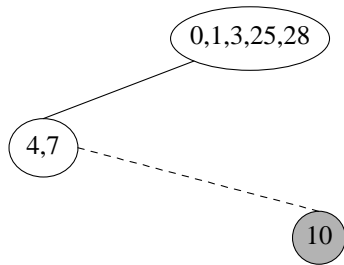


FIGURE 6.15. CDT - Computing $H_3 = \{10\}$

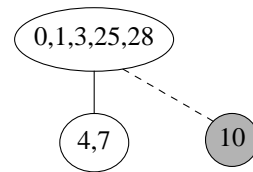


FIGURE 6.16. CPT - Computing $H_3 = \{10\}$

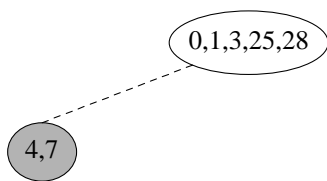


FIGURE 6.17. CDT - Computing $H_2 = \{4\}$ and $H_1 = \{0\}$.

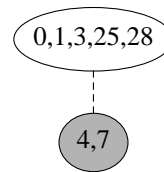


FIGURE 6.18. CPT - Computing $H_2 = \{4\}$ and $H_1 = \{0\}$.

Chapter 7

Summary

7.1 Review

In this dissertation, a new concept of level of essentialness has been developed and its application to program test coverage has been introduced.

We first defined a node x being essential for a node y in a flowchart. Then, we introduced a notion of degree of essentialness for a node x . We classify nodes in a hierarchy via various levels of essentialness based on their degrees. As result of this, we generalized the crisp notion of the essential set of nodes. The nodes in the essential set are at the lowest (minimally essential) level of essentialness.

We develop two efficient algorithms to compute degrees and levels of essentialness for all nodes in a flowchart and to compute all pruned merged levels of nodes. Both algorithms are independent of each other.

We systematically studied the concepts of domination and post-domination relationships among nodes in a flowchart and exploited some of their properties. Those properties were further utilized in our proposed concepts and algorithms.

We discovered some important properties of our proposed concepts. Our merged levels of nodes have the covering property. That is, a set of test cases that covers all nodes in the i^{th} merged level will ensure to cover all nodes in the above levels. We further pruned the merged levels of nodes. Our pruned merged levels have a non-increasing size property in addition to the covering property.

We introduced an application to program test coverage. A form of testing a statement in a program is to find a test case that will execute the statement in the program (or cover the corresponding node in its flowchart). Covering all nodes in

the essential set is sufficient to cover all nodes in the flowchart. In certain program strategies, covering all nodes may not be an ultimate goal. For example, covering all nodes in regression testing is not necessary. We proposed two methods to find a smaller set W' of nodes to cover a set W of desired nodes such that a set of test cases that cover the nodes in W' will also cover all nodes in the desired set W (we say W' cover W).

We developed a loan calculation program to illustrate our method.

7.2 Future Work

We focused on semi-structured flowcharts only. Thus, generalize our concepts for flowcharts with arbitrary gotos.

Note that certain statements in a program may not be executable at all. Therefore, it may not be possible to come up with a set of test cases that will cover all nodes in W' . Let W_1 denote the set of nodes in W' that cannot be covered. Thus, explore the possibilities to successively utilize our concepts to find W'_1 , W'_2 and so on.

Study the characteristics of test cases (paths) for the levels of essentialness, particularly for the pruned merged sets so that test cases could be obtained more efficiently/effectively for the smallest set H_i , which covers the set W' .

References

- [1] Bertolino, A. and Marre, M. Automatic generation of path covers on the control flow analysis of computer programs. *IEEE Transactions on Software Engineering*, Vol. 20, No. 12, pages 885-899, 1994.
- [2] Vliet, H. V. *Software Engineering: Principles and Practice*. Wiley, 2000.
- [3] *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Std 610.12, 1990.
- [4] Naur P. and Randell B., editors. *Software Engineering, Report on a Conference*. NATO Scientific Affairs Division, Garmisch, 1968.
- [5] *Software QA/Test Resource Center*. <http://www.softwareqatest.com/qatfaq1.html>, November 22, 2002.
- [6] Survey of glass and black box testing techniques. <http://www.issco.unige.ch/ewg95/node82.html>, November 22, 2002.
- [7] Cole, O. White box testing. *Dr. Dobb's Journal*, March, 2000.
- [8] Strengths of black box testing. <http://www.cse.fau.edu/maria/COURSES/CEN4010-SE/C13/black1.html>, November 22, 2002.
- [9] Sobey, A. J. Black box testing. http://louisa.levels.unisa.edu.au/se1/testing-notes/test01_5.htm, November 22, 2002.
- [10] Cornett, S. Code coverage analysis. <http://www.bullseye.com/webCoverage.html>, August 31, 2002.
- [11] Watson, A. H. and McCabe, T. J. Structured testing: A testing methodology using the cyclomatic complexity metric. http://www.mccabe.com/nist/nist_pub.php, August, 1996.
- [12] Mary, J. H. Testing: A roadmap. In *Future of Software Engineering, 22nd International Conference on Software Engineering*, June 2000.
- [13] Goodenough, J. B. and Gerhart, S. L. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, Vol. 2, pages 156-173, June, 1975.
- [14] Whittaker, J. A. What is software testing? And why is it so hard? *IEEE Software*, pages 70-79, January/February, 2000.
- [15] Myers, G. J. *The Art of Software Testing*. John Wiley and Sons, Newyork, 1976.

- [16] Binkley, D. Using semantic differencing to reduce the cost of regression testing. Loyola College, Maryland, 1992.
- [17] Marick, B. *The Craft of Software Testing*. Prentice Hall, 1995.
- [18] Beizer, B. *Software Testing Techniques (2E)*. The Coriolis Group, Newyork, 1990.
- [19] Stocks, P. *Applying Fromal Methods to Software Testing*. PhD Thesis, St. Lucia, Autralia, 1993.
- [20] Agrawal, H., Horgan, J. R., Krause, E. W. and London, S. A. Incremental regression testing. *Proceedings of the 1993 IEEE Conference on Software Maintenance*, Montreal, Canada, September 27-30, 1993.
- [21] Weiser, M. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, University of Michigan, Ann Arbor, 1979.
- [22] Lengauer, T. and Tarjan R. E., A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 1, No 1, July, 1979.
- [23] Weiser, M. Programmers use slices when debugging. *Communications of the ACM*, Julay 25, 1982.
- [24] Alstrup, S., Harel, D., Lauridsen, P. W. and Thorup, M. Dominators in linear time. *SIAM Journal on Computing*, Vol. 28, No. 6, 1999.
- [25] Edwards, S. H. Black box testing using flowgraphs: An experimental assesment of effectiveness and automation potential. *Software Testing Verification and Reliability*, Vol. 10, No. 4, pages 249-262, December, 2000.
- [26] Voas, J. M. and Miller, K. W. Software testability: The new verification. *IEEE Software*, Vol. 12, No. 3, May, 1995.

Appendix A:

Glossary

Semi-structured flowchart: A structured flowchart where breaks, continues, and returns are also allowed. No arbitrary gotos are allowed.

Sequential nodes: Nodes x and y of the same type are called sequential if (x, y) is the only arc from x and also it is the only arc to y .

Path: A path from node a to node b in a flowchart is a sequence of one or more arcs of the form $\pi_{a,b}^F = \langle (x_0, x_1), (x_1, x_2), (x_2, x_3), \dots, (x_{n-1}, x_n) \rangle$, where $x_0 = a$ and $x_n = b$. We sometimes use the compact notation $\pi_{a,b}^F = \langle x_0, x_1, x_2, \dots, x_{n-1}, x_n \rangle$ and write $x_i \in \pi_{a,b}^F$ to indicate that $\pi_{a,b}^F$ goes through x_i .

Complete path: A path $\pi_{start,stop}^F$ from the start node in F to its end node is called a complete path.

Domination: A node x is said to dominate a node y if $x \in \pi_{start,y}^F$ for all $\pi_{start,y}^F$. We write $\mathcal{D}_{x,y} = 1$ to indicate that x dominates y ; otherwise, $\mathcal{D}_{x,y} = 0$.

Immediate domination: A node x immediately dominates a node y if only if $\mathcal{D}_{x,y} = 1$ and there is no node z such that $\mathcal{D}_{x,z} = \mathcal{D}_{z,y} = 1$.

Domination tree (DT): A tree that represents the domination relationships among all nodes in a flowchart.

Post-domination: A node x is said to post-dominate a node y if $x \in \pi_{y,stop}^F$ for all $\pi_{y,stop}^F$. We write $\mathcal{P}_{x,y} = 1$ to indicate that x post-dominates y ; otherwise, $\mathcal{P}_{x,y} = 0$.

Immediate post-domination: A node x immediately post-dominates a node y if only if $\mathcal{P}_{x,y} = 1$ and there is no node z such that $\mathcal{P}_{x,z} = \mathcal{P}_{z,y} = 1$.

Post–domination tree (PT): A tree that represents the post–domination relationships among all nodes in a flowchart.

Essential for relationship: A node x is essential for a node y if $\mathcal{D}_{x,y} = 1$ or $\mathcal{P}_{x,y} = 1$ (or both). We write $\mathcal{E}_{x,y} = 1$ if x is essential for y , otherwise $\mathcal{E}_{x,y} = 0$.

Mutually essential: Nodes x and y are called mutually essential if $\mathcal{E}_{x,y} = \mathcal{E}_{y,x} = 1$.

Mutually nonessential: Nodes x and y are called mutually nonessential if $\mathcal{E}_{x,y} = \mathcal{E}_{y,x} = 0$.

One–way essential: Node x is called *one–way essential* to node y if $\mathcal{E}_{x,y} = 1$ and $\mathcal{E}_{y,x} = 0$.

Degree of essentialness: Let δ_x denote the number of nodes y such that x is essential for. δ_x is called the degree of essentialness for x .

Maximally essential: A node x is maximally essential if it is essential for all the nodes. We write E_{max} to denote the set of all maximally essential nodes.

Minimally essential: A node y is minimally essential if it is not essential to any other node. We write E_{min} to denote the set of all minimally essential nodes.

Level of essentialness: If node x has the i^{th} largest δ_x , then the essentialness level of x is i . We write E_i to denote the set of all nodes with the essentialness level i and α_x denote the level of essentialness for node x .

Covering relationship: A test case τ *covers* a node x if $x \in \pi(\tau)$. A node x *covers* a node y if all τ that cover x also cover y . A node x covers itself. A set of nodes S *covers* a set of nodes W if each node $y \in W$ is covered by a node $x \in S$.

Sufficient set: A set of nodes S is *sufficient* for a set of nodes W if covering S will suffice to cover W . That is, any set of paths that covers S will also cover W .

Master pruned set: Let W denote a set of nodes and $h_level(x) = j$ such that node x or its representative belongs to H_j and $h_max(W) = \max\{h_level(x_i) : \text{for all } x_i \in W\}$. $H_{h_max(W)}$ is called *master pruned set* for W .

While-do node: Node x is called a while-do node if it is the head of a while-do loop.

Regular node: A node x is called a regular node if it is not a while-do node.

Cyclomatic complexity: The number of if statements in a program plus one is called the cyclomatic complexity of the program.

Appendix B:

C Program to Compute Domination Tree

```
//Osman Kandara
//10/9/2001
//g++ domtree.c
//Computes DT for a given flow chart.
//Input file: A semi-structured flowchart.

#include <stdio.h>
#include <stdlib.h>

typedef struct {
    char nodeType[9]; //start, action, decision, end
    int nodeNum, sourceLinenum, nextNodes[2];
    //1 for action, 2 for decision
} FlowGraphNode;
typedef struct {
    int numPreds, *preds;
} Predecessor;
typedef struct {
    int numDoms, *doms;
} Dominator;
typedef struct DomTree {
    int nodeNum;
    struct DomTree *rightBro,
        *leftMostChild;
} DomTreeNode;

void SkipToEndOfLine(FILE *inFile)
{while (getc(inFile) !='\n');
}

FlowGraphNode *ReadFlowChartData(FILE *inFile, int *numNodes)
{int i;
    FlowGraphNode *nodes;
    fscanf(inFile, "%d", numNodes); SkipToEndOfLine(inFile);
    nodes = (FlowGraphNode *)malloc((*numNodes)*sizeof(FlowGraphNode));
    for (i=0; i<*numNodes; i++) {
        fscanf(inFile, "%d %d %s", &nodes[i].nodeNum,
            &nodes[i].sourceLinenum,
            nodes[i].nodeType);
        if (strcmp(nodes[i].nodeType, "decision") == 0)
            fscanf(inFile, "%d %d", &nodes[i].nextNodes[0],
```

```

                                &nodes[i].nextNodes[1]);
    else fscanf(inFile, "%d", &nodes[i].nextNodes[0]);
    SkipToEndOfLine(inFile);
}
return(nodes);
}

void PrintFlowChartData(FlowGraphNode nodes[], int numNodes)
{int i;
 printf("\nFlow-chart Data\n-----\n");
 for (i=0; i<numNodes; i++) {
     printf("%d %d %s ", nodes[i].nodeNum, nodes[i].sourceLinenum,
            nodes[i].nodeType);
     if (strcmp(nodes[i].nodeType, "decision") == 0)
         printf("%d %d", nodes[i].nextNodes[0], nodes[i].nextNodes[1]);
     else printf("%d", nodes[i].nextNodes[0]);
     printf("\n");}
}

Predecessor *ComputePreds(FlowGraphNode nodes[], int numNodes)
{int i, j, tempPreds[numNodes][numNodes]; //local dynamic array
 int c, a[5];
 Predecessor *preds;
 preds = (Predecessor *)calloc(numNodes, sizeof(Predecessor));
 preds[0].numPreds = 0; //start node has NO preds
 for (i=0; i<numNodes-1; i++) {
     tempPreds[nodes[i].nextNodes[0]][preds[nodes[i].
     nextNodes[0]]. numPreds++] = nodes[i].nodeNum;
     if (strcmp(nodes[i].nodeType, "decision") == 0)
         tempPreds[nodes[i].nextNodes[1]][preds[nodes[i].
         nextNodes[1]].numPreds++] = nodes[i].nodeNum;
 }
 for (i=1; i<numNodes; i++) { //copy tempPreds[][] to preds[][]
     preds[i].preds = (int *)malloc(preds[i].numPreds*sizeof(int));
     for (j=0; j<preds[i].numPreds; j++)
         preds[i].preds[j] = tempPreds[i][j];
 }
 return (preds);
}

void PrintPreds(Predecessor preds[], int numNodes)
{int i, j;
 printf("\nPredecessors\n-----\n");
 for (i=0; i<numNodes; i++) {
     printf("preds[%d]= ", i);
     for (j=0; j<preds[i].numPreds; j++)
         printf("%d ", preds[i].preds[j]);
}
}

```

```

        printf("\n");}
}

Dominator Intersec(Dominator d1, Dominator d2 )
{int i, j, k, min;
  Dominator intersec;
  min = d1.numDoms < d2.numDoms ? d1.numDoms : d2.numDoms;
  intersec.doms = (int *)calloc(min+1, sizeof(int));
  //min + 1= all intersections + node at the end
  intersec.numDoms = 0;
  for (i=0; i<d1.numDoms; i++)
    for (j=0; j<d2.numDoms; j++)
      if (d1.doms[i] == d2.doms[j]) {
        intersec.doms[intersec.numDoms++] = d1.doms[i];
        break;}
  return(intersec);
}

int IsDomsDiff(Dominator d1, Dominator d2)
{int i, dif = 0;
  if (d1.numDoms != d2.numDoms) {dif = 1; return (dif);}
  for (i=0; i<d1.numDoms; i++)
    if (d1.doms[i] != d2.doms[i] ) {
      dif = 1;
      break;}
  return (dif);
}

Dominator CopyDom(Dominator d, char option)
{int i;
  Dominator copy;
  copy.numDoms = d.numDoms;
  copy.doms = (int *)calloc(copy.numDoms+1, sizeof(int));
  for (i=0; i<d.numDoms; i++) copy.doms[i] = d.doms[i];
  if (option == 'f' ) free (d.doms);
  return (copy);
}

void PrintDoms(Dominator doms[], int numNodes)
{int i, j;
  printf("\nDominators\n-----\n");
  for (i=0; i<numNodes; i++) {
    printf("doms[%d]= ", i);
    for (j=0; j<doms[i].numDoms; j++) printf("%d ", doms[i].doms[j]);
    printf("\n");}
}

```

```

Dominator *ComputeDoms(FlowGraphNode nodes[], Predecessor preds[],
                        int numNodes)
{int i, j, k, p, node, dif;
 Dominator temp, intersec, old, *tempDoms;
 tempDoms = (Dominator *)calloc(numNodes, sizeof(Dominator));
 tempDoms[0].numDoms = 1;
 tempDoms[0].doms = (int *)calloc(1, sizeof(int));
 tempDoms[0].doms[0] = nodes[0].nodeNum;
 for (i=1; i<numNodes; i++) { //initialize to all nodes
     tempDoms[i].numDoms = numNodes;
     tempDoms[i].doms = (int *)calloc(numNodes, sizeof(int));
     for (j=0; j<numNodes; j++) tempDoms[i].doms[j] = nodes[j].nodeNum;
 }
 do {
     for (node=1; node<numNodes; node++) {
         intersec = CopyDom(tempDoms[node], 'u'); //u= unfree .*doms
         for (p=0; p<preds[node].numPreds; p++) {
             temp = Intersec(intersec, tempDoms[preds[node].preds[p]]);
             free (intersec.doms);
             intersec = CopyDom(temp, 'f'); //f= free .*doms
         }
         old = CopyDom(tempDoms[node], 'f');
         tempDoms[node] = CopyDom(intersec, 'f');
         tempDoms[node].doms[tempDoms[node].numDoms] = node;
         tempDoms[node].numDoms++;
         dif = IsDomsDiff(old, tempDoms[node]);
         free (old.doms);
     }
 } while (dif);
 return (tempDoms);
}

```

```

void PrintChilds(DomTreeNode *currNode)
{while(currNode) {
    printf("%d - ", currNode->nodeNum);
    currNode=currNode->rightBro;
}
printf("\n");
}

```

```

void PrintDomTree(DomTreeNode *domTree)
{DomTreeNode *currNode;
 currNode=domTree;
 while (currNode) {
     printf("parent %d: ", currNode->nodeNum);
     PrintChilds(currNode->leftMostChild);
     PrintDomTree(currNode->leftMostChild);
 }
}

```

```

        currNode = currNode->rightBro;}
    }

DomTreeNode *BuildDomTree(Dominator doms[], int numNodes)
{int i, j, found;
  DomTreeNode *domTree, *currNode, *prevNode, *parentNode, *newNode;
  domTree = (DomTreeNode *)calloc(1, sizeof(DomTreeNode));
  domTree->nodeNum = doms[0].doms[0];
  for (i=1; i<numNodes; i++) {
    parentNode = domTree;
    currNode = domTree->leftMostChild;
    prevNode = NULL;
    for (j=1; j<doms[i].numDoms; j++) {
      found = 0;
      while (currNode && !found) {
        if (currNode->nodeNum == doms[i].doms[j])
          found = 1;
        else {
          prevNode = currNode;
          currNode = currNode->rightBro;}}
      if (!found) {
        newNode = (DomTreeNode *)calloc(1, sizeof(DomTreeNode));
        newNode->nodeNum = doms[i].doms[j];
        if (!parentNode->leftMostChild)
          parentNode->leftMostChild = newNode;
        else prevNode->rightBro = newNode;
        parentNode = newNode;}
      else {
        parentNode = currNode;
        currNode = parentNode->leftMostChild;
        prevNode = NULL;}
    } //for_j
  }
  return(domTree);
}

main()
{int numNodes;
  FlowGraphNode *nodes;
  Predecessor *preds;
  Dominator *doms;
  DomTreeNode *domTree;
  FILE *inFile;
  inFile = fopen("fchart.data", "r");
  nodes = ReadFlowChartData(inFile, &numNodes);
  PrintFlowChartData(nodes, numNodes);
  preds = ComputePreds(nodes, numNodes);
}

```

```

PrintPreds(preds, numNodes);
doms = ComputeDoms(nodes, preds, numNodes);
PrintDoms(doms, numNodes);
domTree = BuildDomTree(doms, numNodes);
printf("\nDominator tree\n-----\n");
PrintDomTree(domTree);
}

```

Input File

The following sample input data file is for the flowchart in Figure 4.1.

```

29 Figure_4.1          //numNodes and figure-name; first node is the start
0 1 start            1 0 //nodes can be in any order but
1 1 decision        2 3 1 i //but they have to be 0, 1, 2, ...
2 1 action          1 1 i
3 1 decision        4 25 1 i
4 1 decision        5 6 1 i
5 1 action          7 1 i
6 1 decision        7 9 1 i
7 1 decision        8 10 1 i
8 1 continue        3 1 i
9 1 action          7 1 i
10 1 decision       11 12 1 i
11 1 action         19 1 i
12 1 decision       13 14 1 i
13 1 action         18 1 i
14 1 decision       15 18 1 i
15 1 decision       16 17 1 i
16 1 action         15 1 i
17 1 break          18 1 i
18 1 action         19 1 i
19 1 decision       20 23 1 i
20 1 decision       21 22 1 i
21 1 break          19 1 i
22 1 action         19 1 i
23 1 decision       3 24 1 i
24 1 action         3 1 i
25 1 decision       26 27 1 i
26 1 action         28 1 i
27 1 action         28 1 i
28 1 end            -1

```

Output

Flow-chart Data

```
-----  
0 1 start 1  
1 1 decision 2 3  
2 1 action 1  
3 1 decision 4 25  
4 1 decision 5 6  
5 1 action 7  
6 1 decision 7 9  
7 1 decision 8 10  
8 1 continue 3  
9 1 action 7  
10 1 decision 11 12  
11 1 action 19  
12 1 decision 13 14  
13 1 action 18  
14 1 decision 15 18  
15 1 decision 16 17  
16 1 action 15  
17 1 break 18  
18 1 action 19  
19 1 decision 20 23  
20 1 decision 21 22  
21 1 break 19  
22 1 action 19  
23 1 decision 3 24  
24 1 action 3  
25 1 decision 26 27  
26 1 action 28  
27 1 action 28  
28 1 end -1
```

Predecessors

```
-----  
preds[0]=  
preds[1]= 0 2  
preds[2]= 1  
preds[3]= 1 8 23 24  
preds[4]= 3  
preds[5]= 4  
preds[6]= 4  
preds[7]= 5 6 9  
preds[8]= 7  
preds[9]= 6  
preds[10]= 7  
preds[11]= 10  
preds[12]= 10  
preds[13]= 12
```

```
preds[14]= 12
preds[15]= 14 16
preds[16]= 15
preds[17]= 15
preds[18]= 13 14 17
preds[19]= 11 18 21 22
preds[20]= 19
preds[21]= 20
preds[22]= 20
preds[23]= 19
preds[24]= 23
preds[25]= 3
preds[26]= 25
preds[27]= 25
preds[28]= 26 27
```

Dominators

```
doms[0]= 0
doms[1]= 0 1
doms[2]= 0 1 2
doms[3]= 0 1 3
doms[4]= 0 1 3 4
doms[5]= 0 1 3 4 5
doms[6]= 0 1 3 4 6
doms[7]= 0 1 3 4 7
doms[8]= 0 1 3 4 7 8
doms[9]= 0 1 3 4 6 9
doms[10]= 0 1 3 4 7 10
doms[11]= 0 1 3 4 7 10 11
doms[12]= 0 1 3 4 7 10 12
doms[13]= 0 1 3 4 7 10 12 13
doms[14]= 0 1 3 4 7 10 12 14
doms[15]= 0 1 3 4 7 10 12 14 15
doms[16]= 0 1 3 4 7 10 12 14 15 16
doms[17]= 0 1 3 4 7 10 12 14 15 17
doms[18]= 0 1 3 4 7 10 12 18
doms[19]= 0 1 3 4 7 10 19
doms[20]= 0 1 3 4 7 10 19 20
doms[21]= 0 1 3 4 7 10 19 20 21
doms[22]= 0 1 3 4 7 10 19 20 22
doms[23]= 0 1 3 4 7 10 19 23
doms[24]= 0 1 3 4 7 10 19 23 24
doms[25]= 0 1 3 25
doms[26]= 0 1 3 25 26
doms[27]= 0 1 3 25 27
doms[28]= 0 1 3 25 28
```

Dominator tree

parent 0: 1 -
parent 1: 2 - 3 -
parent 2:
parent 3: 4 - 25 -
parent 4: 5 - 6 - 7 -
parent 5:
parent 6: 9 -
parent 9:
parent 7: 8 - 10 -
parent 8:
parent 10: 11 - 12 - 19 -
parent 11:
parent 12: 13 - 14 - 18 -
parent 13:
parent 14: 15 -
parent 15: 16 - 17 -
parent 16:
parent 17:
parent 18:
parent 19: 20 - 23 -
parent 20: 21 - 22 -
parent 21:
parent 22:
parent 23: 24 -
parent 24:
parent 25: 26 - 27 - 28 -
parent 26:
parent 27:
parent 28:

Appendix C:

C Program to Compute CDT and CPT

```
//Osman Kandara
//3/29/2003
//g++ ct.c
//Computes CDT and CPT and mutually essential nodes.
//Input files: DT and PT of a flowchart.
#include <stdio.h>
#include <stdlib.h>

int **mutEss;

void SkipToEndOfLine(FILE *fpIn)
{while (getc(fpIn) != '\n');
}

int ReadData(FILE *fpIn, int **t)
{int node, numChild, child,
    i, j, numNodes;
  fscanf(fpIn, "%d", &numNodes);
  SkipToEndOfLine(fpIn);
  *t = (int *)malloc(sizeof(int) * numNodes);
  for (i=0; i<numNodes; i++) {
    fscanf(fpIn, "%d%d", &node, &numChild);
    for (j=0; j<numChild; j++) {
      fscanf(fpIn, "%d", &child);
      (*t)[child] = node;
    }
    SkipToEndOfLine(fpIn);
  }
  return(numNodes);
}

void PrintT(int numNode, int t[])
{int i;
  for (i=0; i<numNode; i++)
    if (t[i] != -1)
      printf("%2d's father is %2d\n", i, t[i]);
  printf("\n");
}

void CombineMutuals(int numNodes, int dt[], int pt[])
{int i, j, father;
```

```

for (i=1; i<numNodes; i++) {
    father = dt[i];
    if (pt[father] == i) {
        //update dt
        for (j=1; j<numNodes; j++)
            if (j != i && dt[j] == i)
                dt[j] = father;

        //update pt
        for (j=0; j<numNodes-1; j++)
            if (j != father && pt[j] == i)
                pt[j] = father;
        pt[father] = pt[i];
        pt[i] = -1;
        dt[i] = -1;
        mutEss[father][++mutEss[father][0]] = i;
    }
}
}

```

```

PrintMutEss(int numNodes)
{int i, j;
    printf("Mutually Essential Nodes:\n");
    printf("=====\n");
    for (i=0; i<numNodes; i++)
        if (mutEss[i][0] > 0) {
            printf("%d => ", i);
            for (j=1; j<=mutEss[i][0]; j++)
                printf("%d ", mutEss[i][j]);
            printf("\n");
        }
}

```

```

main()
{int numNodes, *dt, *pt, i;
    FILE *fpIn;
    fpIn = fopen("dt.data", "r");
    numNodes = ReadData(fpIn, &dt);
    fclose(fpIn);
    printf("Domination Tree (DT):\n");
    printf("=====\n");
    PrintT(numNodes, dt);
    fpIn = fopen("pt.data", "r");
    numNodes = ReadData(fpIn, &pt);
    fclose(fpIn);
    printf("Post-Domination Tree (PT):\n");
    printf("=====\n");
}

```

```

PrintT(numNodes, pt);
mutEss = (int **)malloc(sizeof(int *) * numNodes);
for (i=0; i<numNodes; i++)
    mutEss[i] = (int *)malloc(sizeof(int) * numNodes);
CombineMutuals(numNodes, dt, pt);
printf("Combined Domination Tree (CDT):\n");
printf("=====\n");
PrintT(numNodes, dt);
printf("Combined Post-Domination Tree (CPT):\n");
printf("=====\n");
PrintT(numNodes, pt);
PrintMutEss(numNodes);
return(0);
}

```

Input Files

The following data are for the flowchart in Figure 4.1.

```

--- dt.data ---
29      // #of nodes
0 1 1   // father Node, #of children, and children
1 2 2 3
2 0
3 2 4 25
4 3 5 6 7
5 0
6 1 9
7 2 8 10
8 0
9 0
10 3 11 12 19
11 0
12 3 13 14 18
13 0
14 1 15
15 2 16 17
16 0
17 0
18 0
19 2 20 23
20 2 21 22
21 0
22 0
23 1 24

```

```

24 0
25 3 26 27 28
26 0
27 0
28 0

--- pt.data ---
29 // #of nodes
0 0 // father node, #children, and children nodes
1 2 0 2
2 0
3 4 1 8 23 24
4 0
5 0
6 0
7 4 4 5 6 9
8 0
9 0
10 0
11 0
12 0
13 0
14 0
15 1 16
16 0
17 1 15
18 1 13
19 3 11 18 22
20 0
21 0
22 0
23 3 19 20 21
24 0
25 6 3 7 10 12 14 17
26 0
27 0
28 3 25 26 27

```

Output

Domination Tree (DT):

=====

```

0's father is 0
1's father is 0
2's father is 1
3's father is 1

```

4's father is 3
5's father is 4
6's father is 4
7's father is 4
8's father is 7
9's father is 6
10's father is 7
11's father is 10
12's father is 10
13's father is 12
14's father is 12
15's father is 14
16's father is 15
17's father is 15
18's father is 12
19's father is 10
20's father is 19
21's father is 20
22's father is 20
23's father is 19
24's father is 23
25's father is 3
26's father is 25
27's father is 25
28's father is 25

Post-Domination Tree (PT):

=====

0's father is 1
1's father is 3
2's father is 1
3's father is 25
4's father is 7
5's father is 7
6's father is 7
7's father is 25
8's father is 3
9's father is 7
10's father is 25
11's father is 19
12's father is 25
13's father is 18
14's father is 25
15's father is 17
16's father is 15
17's father is 25
18's father is 19

19's father is 23
20's father is 23
21's father is 23
22's father is 19
23's father is 3
24's father is 3
25's father is 28
26's father is 28
27's father is 28
28's father is 0

Combined Domination Tree (CDT):

=====

0's father is 0
2's father is 0
4's father is 0
5's father is 4
6's father is 4
8's father is 4
9's father is 6
10's father is 4
11's father is 10
12's father is 10
13's father is 12
14's father is 12
15's father is 14
16's father is 15
18's father is 12
19's father is 10
20's father is 19
21's father is 20
22's father is 20
24's father is 19
26's father is 0
27's father is 0

Combined Post-Domination Tree (CPT):

=====

0's father is 0
2's father is 0
4's father is 0
5's father is 4
6's father is 4
8's father is 0
9's father is 4
10's father is 0
11's father is 19

12's father is 0
13's father is 18
14's father is 0
15's father is 0
16's father is 15
18's father is 19
19's father is 0
20's father is 19
21's father is 19
22's father is 19
24's father is 0
26's father is 0
27's father is 0

Mutually Essential Nodes:

=====

0 => 1 3 25 28

4 => 7

15 => 17

19 => 23

Appendix D:

C Program to Implement the PML Algorithm

```
//Osman Kandara
//4/6/2003
//g++ pml.c
//Computes pruned merged sets.
//Input files: CDT and CPT of a flowchart.

#include <stdio.h>
#include <stdlib.h>
typedef struct {
    int father;
    int numChild;
    int mut;
} Node;

void SkipToEndOfLine(FILE *fpIn)
{while (getc(fpIn) != '\n');
}

void ReadData(FILE *fpIn, int numNodes, Node nodes[])
{int node, child, numChild, i,j;
  for (i=0; i<numNodes; i++) {
    fscanf(fpIn, "%d %d", &node, &numChild);
    nodes[node].numChild = numChild;
    if (nodes[node].numChild == 0)
        fscanf(fpIn, "%d", &nodes[node].mut);
    else {
        nodes[node].mut = -1;
        for (j=0; j<nodes[node].numChild; j++) {
            fscanf(fpIn, "%d", &child);
            nodes[child].father = node;
        }
    }
    SkipToEndOfLine(fpIn);
}
}

void PruneSet(int numNodes, Node cdt[], Node cpt[])
{int i, k, level=-1, terms[numNodes][numNodes];
  //terms[i][0] holds #of nodes to be deleted at level i
  do {
```

```

level++;
terms[level][0] = 0;
for (i=0; i<numNodes; i++)
    if (cdt[i].numChild == 0 && cdt[i].mut == -1 && cdt[i].father != -1)
        if (cpt[i].numChild == 0)
            terms[level][++terms[level][0]] = i;
for (k=1; k<=terms[level][0]; k++) {
    cdt[cdt[terms[level][k]].father].numChild--; //delete from CDT
    cdt[terms[level][k]].father = -1;
    cpt[cpt[terms[level][k]].father].numChild--; //delete from CPT
    cpt[terms[level][k]].father = -1;
}
}while (terms[level][0]>0);
printf("\nPruned Merged Sets:\n");
printf("=====\n");
for (i=0; i<level; i++) {
    printf("Level %d => ", level-i);
    for (k=1; k<=terms[i][0]; k++)
        printf("%d ", terms[i][k]);
    printf("\n");
}
}

```

```

void PrintCT(int numNode, Node nodes[])
{int i;
for (i=0; i<numNode; i++) {
    if (nodes[i].numChild > 0) {
        printf("%d's father is %d\n", i, nodes[i].father);
        printf("    and has %d children\n", nodes[i].numChild);
    }
    else
        printf("%d has no children\n", i);
    if (nodes[i].mut != -1)
        printf("    and is mutually essential with %d\n", nodes[i].mut);
}
printf("\n");
}

```

```

main() {int numNodes;
Node *cpt, *cdt;
FILE *fpIn;
fpIn = fopen("cdt.data", "r");
fscanf(fpIn, "%d", &numNodes);
SkipToEndOfLine(fpIn);
cpt = (Node *)calloc(sizeof(Node), numNodes);
cdt = (Node *)calloc(sizeof(Node), numNodes);
ReadData(fpIn, numNodes, cdt);
}

```

```

fclose(fpIn);
printf("CDT:\n");
printf("====\n");
PrintCT(numNodes, cdt);
fpIn = fopen("cpt.data", "r");
fscanf(fpIn, "%d", &numNodes);
SkipToEndOfLine(fpIn);
ReadData(fpIn, numNodes, cpt);
printf("CPT:\n");
printf("====\n");
PrintCT(numNodes, cpt);
fclose(fpIn);
PruneSet(numNodes, cdt, cpt);
return(0);
}

```

Input Files

The following data are for the flowchart in Figure 4.1.

```

--- cdt.data ---
29          // #of nodes
0 4 2 4 26 27 // father node, #of children, and children
1 0 0        // node 1 is mutually essential with node 0
2 0 -1       // node 2 has no children and has no mutual essential.
3 0 0
4 4 5 6 8 10
5 0 -1
6 1 9
7 0 4
8 0 -1
9 0 -1
10 3 11 12 19
11 0 -1
12 3 13 14 18
13 0 -1
14 1 15
15 1 16
16 0 -1
17 0 15
18 0 -1
19 2 20 24
20 2 21 22
21 0 -1
22 0 -1

```

```

23 0 19
24 0 -1
25 0 0
26 0 -1
27 0 -1
28 0 0

--- cpt.data ---
29                                     //#of nodes
0 11 2 8 19 24 4 10 12 14 15 26 27 //node, #of children, and children
1 0 0
2 0 -1
3 0 0
4 3 5 6 9
5 0 -1
6 0 -1
7 0 4
8 0 -1
9 0 -1
10 0 -1
11 0 -1
12 0 -1
13 0 -1
14 0 -1
15 1 16
16 0 -1
17 0 15
18 1 13
19 5 11 18 22 20 21
20 0 -1
21 0 -1
22 0 -1
23 0 19
24 0 -1
25 0 0
26 0 -1
27 0 -1
28 0 0

```

Output

```

CDT:
====
0's father is 0
    and has 4 children
1 has no children

```

and is mutually essential with 0
2 has no children
3 has no children
and is mutually essential with 0
4's father is 0
and has 4 children
5 has no children
6's father is 4
and has 1 children
7 has no children
and is mutually essential with 4
8 has no children
9 has no children
10's father is 4
and has 3 children
11 has no children
12's father is 10
and has 3 children
13 has no children
14's father is 12
and has 1 children
15's father is 14
and has 1 children
16 has no children
17 has no children
and is mutually essential with 15
18 has no children
19's father is 10
and has 2 children
20's father is 19
and has 2 children
21 has no children
22 has no children
23 has no children
and is mutually essential with 19
24 has no children
25 has no children
and is mutually essential with 0
26 has no children
27 has no children
28 has no children
and is mutually essential with 0

CPT:

====

0's father is 0
and has 11 children

1 has no children
 and is mutually essential with 0
 2 has no children
 3 has no children
 and is mutually essential with 0
 4's father is 0
 and has 3 children
 5 has no children
 6 has no children
 7 has no children
 and is mutually essential with 4
 8 has no children
 9 has no children
 10 has no children
 11 has no children
 12 has no children
 13 has no children
 14 has no children
 15's father is 0
 and has 1 children
 16 has no children
 17 has no children
 and is mutually essential with 15
 18's father is 19
 and has 1 children
 19's father is 0
 and has 5 children
 20 has no children
 21 has no children
 22 has no children
 23 has no children
 and is mutually essential with 19
 24 has no children
 25 has no children
 and is mutually essential with 0
 26 has no children
 27 has no children
 28 has no children
 and is mutually essential with 0

Pruned Merged Sets:

=====

Level 7 => 2 5 8 9 11 13 16 21 22 24 26 27
 Level 6 => 6 15 18 20
 Level 5 => 14 19
 Level 4 => 12
 Level 3 => 10

Level 2 => 4
Level 1 => 0

Vita

Kandara was born on January 1, 1970, in Artvin, Turkey. He finished his undergraduate studies at Marmara University, Istanbul, Turkey, May 1991. He earned a master of science degree in systems science from Louisiana State University in December 1996. In January 1998 he came to Louisiana State University again to pursue graduate studies in computer science. He is currently a candidate for the degree of Doctor of Philosophy in computer science, which will be awarded in December 2003.