PROGRAM FLOW GRAPH DECOMPOSITION AS A MODEL OF SOFTWARE COMPREHENSION

By

CHARLOTTE P. REGSON

Bachelor of Science
University of Madras
Madras, India
1978

Bachelor of Science
Oklahoma State University
Stillwater, Oklahoma
1991

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July 1993

OKLAHOMA STATE UNIVERSITY

PROGRAM FLOW GRAPH DECOMPOSITION AS A MODEL OF SOFTWARE COMPREHENSION

Thesis Approved:

M. Samadzadeh-H.

Thesis Advisor

J Chandler

Blayne E. Mayfield

Thomas C. Collins

Dean of the Graduate College

ACKNOWLEDGEMENTS

I wish to express my gratitude to Dr. Mansur Samadzadeh for his continuous guidance and encouragement throughout this thesis work. His confidence in my ability and the moral support offered by him is greatly appreciated. I wish to thank Drs. Mayfield and Chandler for serving on my graduate committee, and for their comments and advice.

I would like to thank my brother Moses Vijayakumar for his moral and financial support which helped me to continue my studies. I would also like to extend my appreciation to my husband Randolph Regson Raj for his love and understanding.

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

Figure Page

CHAPTER I

INTRODUCTION

The maintenance phase of the software life cycle is the time period during which a software product performs useful work [Fairly 85]. The maintenance phase generally starts at the time of release, or even before, when on-site testing is in progress. Typically, the development cycle for a commercial software product spans 1 to 2 years, while the maintenance phase spans 5 to 10 years. Maintenance activities include correction of errors (corrective maintenance), implementation of design changes or enhancements (perfective maintenance), and adjustment of the software to changes in the environment (adaptive maintenance). There is evidence that maintenance costs exceed 60% of the total costs of software [Ghezzi et al. 91]. By keeping the maintenance expenditures low, literally billions of dollars can be saved each year. Software comprehension plays a critical role in the activities involved in the maintenance of software systems. The more a piece of software is understandable, the easier it is to modify or enhance, and hence to maintain it.

There are various graph models that can be associated with the static or dynamic behavior of programs. These graphs are derived based on the flow of control and/or data in programs. For instance, the flow of control in a program can be represented by a directed graph called the control flow graph of the program.

Several decomposition techniques of the flow graph of a program have been discussed

in the literature for various reasons. McCabe discusses the decomposition of the control flow graph of a program into basis paths or circuits in order to calculate the software complexity of the program using the cyclomatic number [McCabe 76]. Forman uses program decomposition into primes to solve the abstract data flow analysis problem [Forman 82]. The division of a control flow graph into intervals serves to put a hierarchical structure on the flow graph; this structure in turn is used in data flow analysis during program optimization [Aho et al. 88]. Bieman and Edwards propose the number of spanning trees in the data dependency graph of a program (DDG, see Section 2.2 for the definition) as a complexity measure (a result of implicit decomposition into spanning trees) [Bieman and Edwards 85]. Tamine represents the flow graph of a program in the form of a tree in order to arrive at a software complexity measure called "characteristic polynomial" of a program, which is sensitive to the nesting level of the program [Tamine 83].

This thesis concerns the different ways of decomposition of a program flow graph for the purpose of modeling software comprehension. Using decomposition as a model of software comprehension is not new [Samadzadeh and Edwards 88] [Samadzadeh and Nandakumar 92]. The novelty here is in the application of decomposition as a model of comprehension to a program flow graph. The basic idea here is the (successive) decomposition of some representation or abstraction of a program to model the classification part of software comprehension. George Miller, in his seminal paper [Miller 56], summarizes the essential motivation for chunking or decomposition as follows:

> The span of absolute judgement and the span of immediate memory impose severe limitations on the amount of information that we are able to receive, process, and remember. By organizing the stimulus input simultaneously into several dimensions and successively into a sequence of chunks, we manage to (or at least

stretch) this informational bottleneck.

Hence it can be argued that our ability to develop complex software is limited by the ability of the human mind to comprehend it, and the ability of the human mind, as Miller argued, is indeed limited [Yourdon 82].

Decomposition can therefore be considered to be an effective technique to comprehend software or at least to aid in the process of comprehension of software. This thesis discusses different decomposition techniques as an aid to and as a model of software comprehension and compares them based on the following dimensions:

(a) Complexity of calculation;

(b) Quantifiability - i.e., meaningful quantifiability of the decomposition process or the product;

(c) Composability - i.e., whether any of the resulting units of decomposition is reusable or not;

(d) Structured vs. non-structured graphs - i.e., whether each decomposition method is applicable to structured flow graphs only, or it is also applicable to non-structured flow graphs;

(e) Repeated decomposition - i.e., whether the decomposition process can be applied repeatedly;

(f) Availability (e.g., as a by-product of the compiling process);

(g) Formalism dependence - i.e., whether a particular decomposition method applies to a particular type of graph model or it is generalizable to different types of graph models as well;

(h) Scalability;

(i) Interdependence - i.e., whether each method is independent of other methods or it can

be obtained from one or more of the other methods;

(j) Automatibility;

(k) Uniqueness;

(l) Cover or partition - i.e., whether each decomposition serves as a cover or as a partition for the concerned graph model.

The organization of this thesis is as follows. Chapter II discusses the definitions of three graph models or abstract representations of programs. A number of decomposition techniques of a flow graph of a program and their uses are discussed in Chapter III. The different decomposition techniques are compared based on the above-mentioned criteria in Chapter IV, and finally Chapter V contains the summary and some possible areas of future work.

CHAPTER II

GRAPH MODELS

There are a number of graph models or abstract representations of a program that are defined and used in the literature for different purposes. For example, control flow graphs and data flow graphs are used in compilers during program optimization, and data dependency graphs can be used to measure data dependency complexity, etc. Different graph models or abstract representations of a program and their uses are discussed in this chapter.

2.1 Control Flow Graph

A control flow graph is a two-dimensional representation of a program that displays the flow of control of a program [Aho and Ullman 73]. Formally, the control flow graph of a program is a 4-tuple, F = (N, E, *a*, z), where N is a finite set of nodes, E is a finite set of directed edges ($E \subseteq N \times N$), *a* is the entry node whose indegree is zero, and z is the exit node whose outdegree is zero. Each node in N represents a basic block (a sequence of instructions with no branches) and each edge in E represents a possible block to block transfer. If a program has more than one entry point, a single entry node can be added to the graph with edges leading to the nodes representing the original entry points. If there is more than one terminal point, a terminal node can be added to the graph with edges leading from the original terminal points to the new terminal node. If any node

cannot be reached from the single entry node, or if any node cannot reach the single exit node, it is considered dead code and therefore it can be removed from the graph. Control flow graphs are studied for the purpose of analysis of a program's structure, conducting data flow analysis [Ottenstein 78], or deriving control flow complexity measures such as McCabe's cyclomatic complexity measure [McCabe 76]. Figure 1 shows the control flow graph corresponding to the given program segment in Pascal (each node in the control flow graph represents a single executable statement for the purpose of clarity).

```
1. Program example (input, output);
2. Var
3.   x, y: real;
4. begin
5.   readln(x);
6.   readln(y);
7.   while (x >= y) do
8.   begin
9.     x := x -y;
10.    readln(y);
11.  end;
12. end.
```

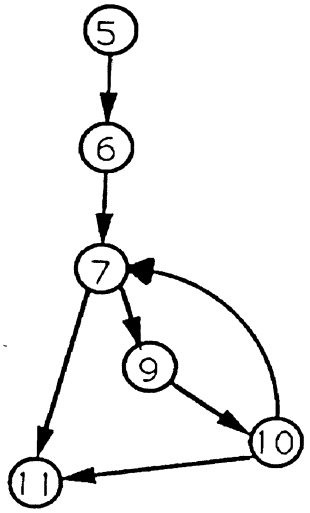


Figure 1. A program segment and its control flow graph

2.2 Data Dependency Graph

A data dependency graph (DDG) is a representation of a program that models the data dependencies within the program. A DDG can be used to derive measures of data dependency complexity [Bieman and Edwards 85]. A DDG is a directed graph in which

the nodes represent variable definitions and edges represent possible data dependencies. A variable definition is a statement that may alter or modify the value of a variable. Assignment statements, procedure calls, and input statements are examples of variable definitions. The edges of a DDG represent possible dependencies between definitions. For example, a statement S_n of the form A = f(B, C) yields a definition A_n for variable A. This definition depends on the values of B and C, and hence it gives rise to the edges (B_x, A_n) and (C_y, A_n), where B_x and C_y are the definitions of B and C that can reach statement S_n. The subscripts denote various definitions of the same variable, and they are numbered sequentially based on the relative position of the definition in the source code.

A data dependency can be a direct dependency or a control dependency. Direct dependencies include assignment statements, procedure calls, and iterative control structures. A variable definition in a statement S also depends on the variables used in the constructs that determine whether or not S will be executed, hence the control dependencies. Examples of DDGs are given in Section 3.4.

The above definition of a DDG can be used to derive data dependency complexity measures. The software complexity measure that is based on a DDG is the rooted spanning tree complexity [Bieman and Edwards 85]. The rooted spanning tree complexity of a DDG will be discussed in detail in Chapter III.

2.3 Program Dependence Graph

The program dependence graph (PDG) is a representation of a program depicting the data and control dependencies explicitly. A number of slightly different definitions have been provided in the literature for a program dependence graph depending on the context and each application. The definition presented here is by Horwitz et al. [Horwitz et al. 88].

The program dependence graph of a program is a directed graph in which the nodes represent the assignment statements and the control predicates that occur in the program. In addition, each program contains a distinguished node called the *entry node* and, corresponding to each variable x, there exist two nodes called initial definition of x, denoted by *init_defn(x),* and the final use of x, denoted by *fin_use(x).*

The edges of a PDG represent the control and data dependencies in the program. There is a *control dependence* edge from node X to Y, if X is a predicate node and Y is nested within the control construct represented by X. The control dependence edge is labeled by the truth value of the branch in which Y occurs.

```
(* c is a constant *)
begin
  bal := c;
  while bal >= 0 do
    begin
      read(wd);
      bal := bal - wd;
    end;
end;
```

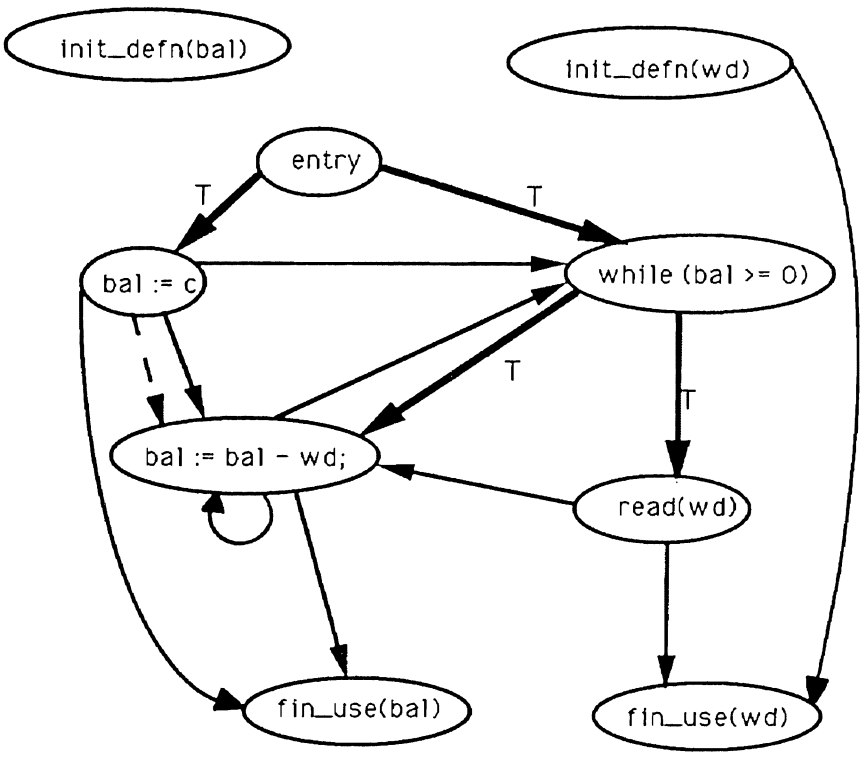


Figure 2. A program and its program dependence graph. (*The control* edges are represented by bold face arrows, *flow order* edges are represented by solid arrows and *def-order* edges are represented by dashed arrows.)

Data dependency edges are of two types: *flow order* and *def-order*. There is a flow dependence edge from node X to node Y, if there is a path in the control flow graph of the program through which the definition of a variable v at X can reach the use of v at Y. There is a *def-order* dependence edge from node X to node Y iff the following hold: 1) both X and Y define the same variable, 2) X and Y are in the same branch of any conditional statement that contains both of them, 3) there exists a node Z such that there exist flow dependence edges XZ and YZ, and 4) X occurs to the left of Y in the abstract syntax-tree of the program. Figure 2 shows a program and its PDG. It should be noted that the PDG of a program is a multigraph (i.e., a graph which can have multiple edges between some of the nodes). Data flow analysis is used to compute the data dependence edges of the PDG of a program [Horwitz et al. 88].

CHAPTER III

DECOMPOSITION TECHNIQUES

Different decomposition techniques of a flow graph of a program into various structures have been discussed in the literature. For a summary see [Regson and Samadzadeh 92]. Some of those techniques were briefly mentioned in the introduction. Various decomposition approaches of flow graphs of programs are discussed in detail in the following sections.

3.1 Decomposition of a Flow Graph into Basis Paths and Circuits

McCabe proposed the cyclomatic number, V(G), of a program's control flow graph as a graph-theoretic software complexity measure [McCabe 76]. For a control flow graph with e edges, n nodes, and p connected components, the cyclomatic complexity is V(G) = e - n + 2 * p. The cyclomatic number can be viewed in a number of ways including the maximum number of linearly independent circuits (or paths) in a strongly connected graph.

An arbitrary circuit (or path) in a strongly connected graph can be expressed as a linear combination of a basis set of independent circuits (or paths). Thus the cyclomatic number is the number of basis circuits (or paths) that can combined to make up any possible circuit (or path) in the graph. McCabe suggested that the number of basis paths or circuits, which is equivalent to the cyclomatic number, contributes to the complexity

of a program and that this number is indicative of how difficult the testing of the program will be [Mccabe 76].

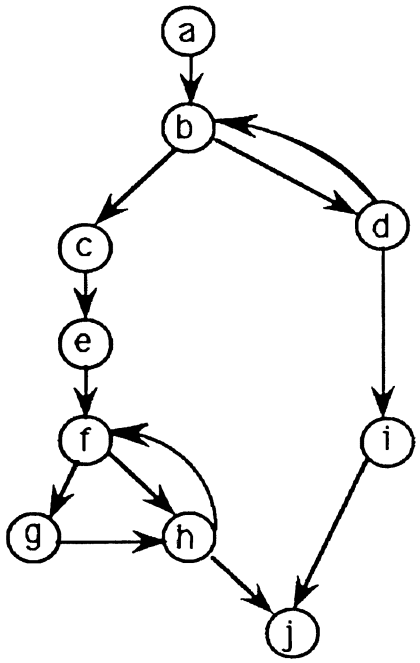


Figure 3. The control flow graph of a program

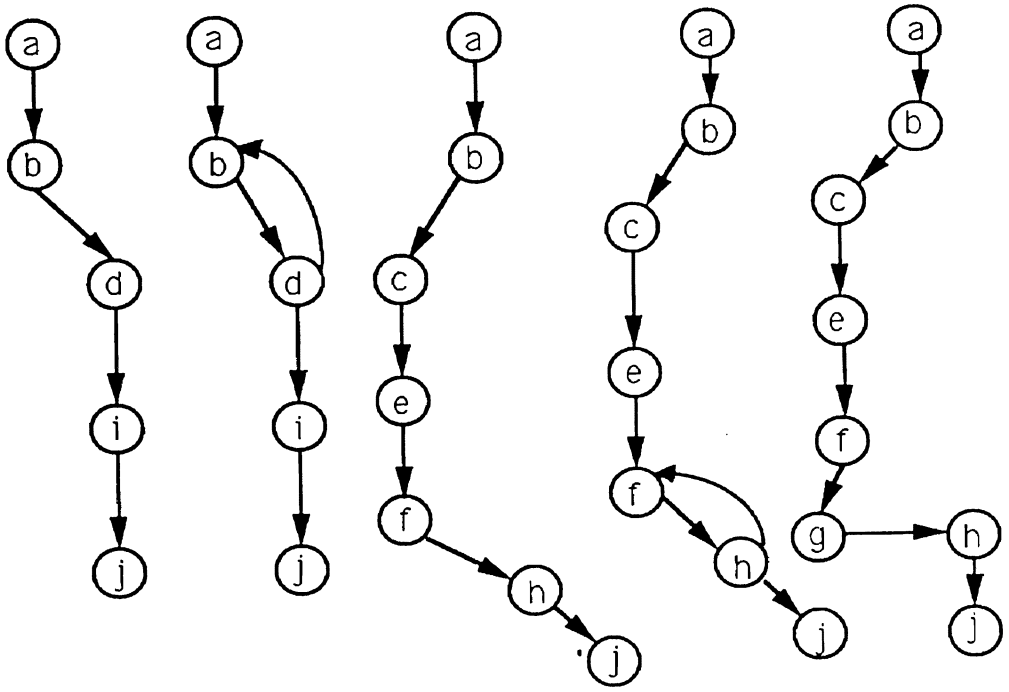


Figure 4. Five basis paths of the flow graph of Figure 3

Figure 3 depicits a control flow graph. The five paths in Figure 4 constitute one set of basis paths for the control flow graph of Figure 3. As an example, the non-basis path *abcefghfhj* can be expressed as a linear combination of the five basis paths given in Figure 4 as follows: *abcefghfhj* = *abcefhfhj* - *abcefhj* + *abcefghj*.

3.2 Decomposition of a Flow Graph into Dominators

If x and y are two (not-necessarily-distinct) nodes in a control flow graph F, then x is a dominator of y iff every path in F from its initial node to y contains x [Hecht 77].

The node x properly dominates node y iff $x \neq y$ and x dominates y. The node x directly dominates (or immediately dominates) node y iff a) x properly dominates y, and b) if w properly dominates y and $w \neq x$, then w (properly) dominates x.

Let DOM(y) = {x | x dominates y}. Let D be the dominance relation defined on N, $D \subseteq N \times N$, such that x = D(y) implies x dominates y. Some properties of the dominance relation D on a flow graph F are listed below [Hecht 77],

a) DOM(*a*) = {*a*}, where *a* is the start node of F.

b) D is a reflexive partial ordering.

c) The initial node *a* of F dominates all nodes of F.

d) All of the dominators of a node form a chain (i.e., a linear ordering).

e) Every node except *a* has a unique direct dominator.

Dominators are used during program optimization for elimination of common subexpressions, code motion, and loop discovery [Aho and Ullman 73]. A number of algorithms to find the dominators of a flow graph are discussed in the literature. Lengauer and Tarjan describe an efficient algorithm for finding dominators that is suitable for

practical use [Lengauer and Tarjan 79]. An example of a control flow graph and its dominating tree is given in Figures 5 and 6.

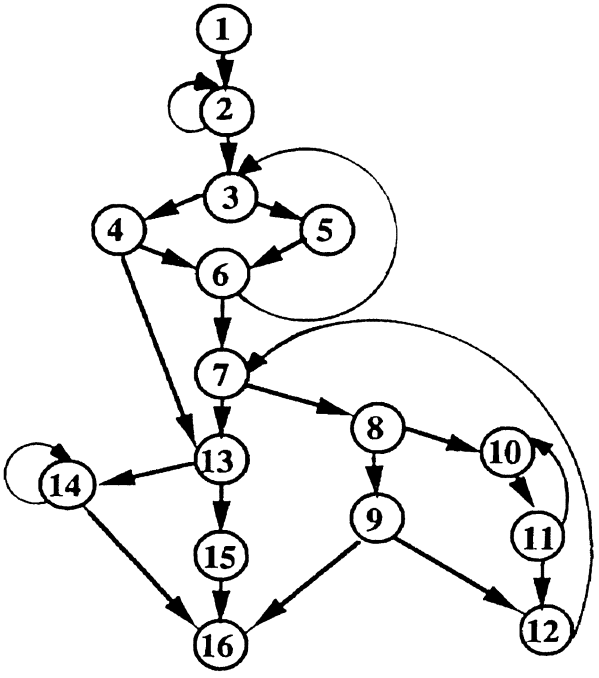


Figure 5. The flow graph of a program (Source: [Hecht 77])

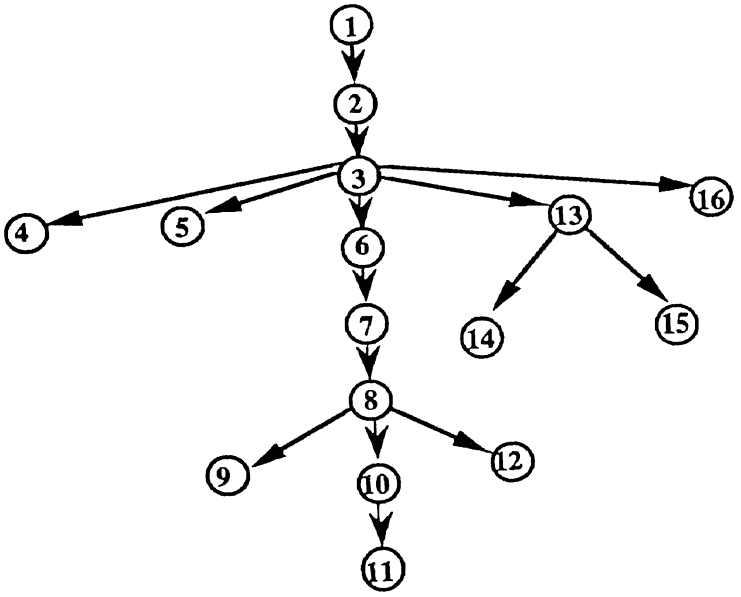


Figure 6. The dominator tree of the control flow graph of Figure 5 (Source: [Hecht 77])

3.3 Decomposition of a Flow Graph into Intervals

Given a control flow graph F with the initial node *a* and a node h of F, the interval with header h, denoted by I(h), is a subgraph of F which is defined as follows [Aho et al. 88]:

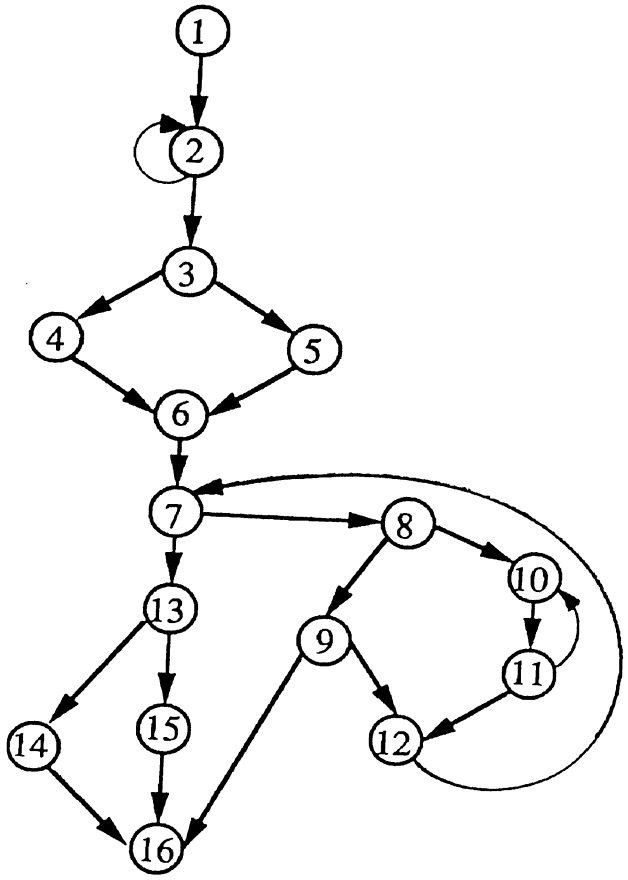
1. The node h is in I(h).
2. If all the predecessors of some node m $\neq a$ are in I(h), then m is in I(h).
3. Nothing else is in I(h).

Some properties of an interval I(h) of a flow graph F are listed below [Hecht 77].

1. Every arc entering a node of an interval I(h) from the outside enters the header h; that is, an interval is single-entry.
2. The header h dominates (with respect to all of F) every other node in I(h).
3. For each node h of a flow graph F, the interval I(h) is unique and independent of the order in which candidates for m in the definition of the interval are chosen.
4. Every cycle in an interval I(h) includes the interval header h.

An example of a flow graph of a program with intervals is given in Figure 7.

There is a certain way to choose interval headers so that a flow graph is uniquely partitioned into (disjoint) intervals [Hecht 77]. An algorithm to partition a flow graph into disjoint intervals is given by Hecht [Hecht 77]. The division of a flow graph into intervals serves to put a hierarchical structure on the flow graph, and the structure in turn facilitates the application of the rules for syntax-directed data flow analysis [Aho et al. 88]. Interval analysis (introduced by F. E. Allen [Allen 70] and J. Cocke [Cocke 70]) is used in intraprocedural data flow analysis [Hecht 77].



I(1) = {1}
I(2) = {2, 3, 4, 5, 6}
I(7) = {7, 8, 9, 13, 14, 15, 16}
I(10) = {10, 11}
I(12) = {12}

Figure 7. Example of a flow graph with intervals

The following definitions are mentioned in order to arrive at complexity measures based on intervals.

If F = (N, E, *a*, z) is a flow graph , then the derived flow graph of F, denoted by I(F), is defined as follows [Hecht 77].

a. The nodes of I(F) are intervals of F,

b. There is an edge from the node representing interval J to the node representing interval K, if there is any edge from a node in J to the header of K, and $J \neq K$.

c. The initial node of I(F) is $I(a)$.

The sequence $F = F_0, F_1, \ldots, F_k$ is called the derived sequence for F, iff $F_{i+1} = I(F_i)$ for $0 \leq i < k$, $k = 1, 2, \ldots$, $F_{k-1} \neq F_k$, and $I(F_k) = F_k$. F_k is called the limit flow graph of F. A flow graph is called reducible (an RFG), iff its limit flow graph is a single node with no edge. Otherwise, it is called irreducible or nonreducible. Let F' be F minus all of its self-loops. Then the length k of the derived sequence of F is defined [Hecht 77] as $k = 0$, if F' is the trivial flow graph; and $k \neq 0$, otherwise, such that

a) $F_0 = F'$,

b) $F_{i+1} = I(F_i)$, $i \geq 0$,

c) F_k is the limit flow graph of F, and

d) $F_k = F_{k-1}$.

Hecht states that the length of the derived sequence of an RFG corresponds intuitively to the maximum "nesting depth" of the loops of a reducible flow graph representing a computer program [Hecht 77].

The loop-connectedness of an RFG is the largest number of backward edges found in any cycle-free path in F. The Interval Derived Sequence Length and Loop-Connectedness are two complexity measures introduced by Hecht [Hecht 77] that are based on intervals. These measures are not widely referred to in the literature.

3.4 Decomposition of a Flow Graph into Spanning Trees

The number of spanning trees in the data dependency graph of a program has been proposed as a complexity measure [Bieman and Edwards 85].

(if-then-else)

1. if X < Y
2. A := B
3. else A := C
4. D := A

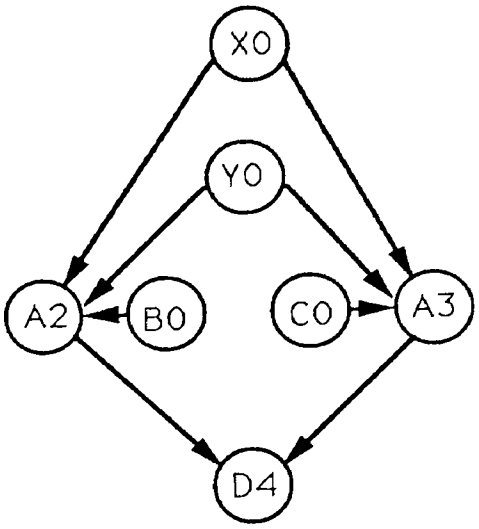


Figure 8. Data Dependency graph for a condition
(Source: [Bieman and Edwards 85])

(while loop)

1. A := B
2. while X > A do
3. A := F(A, B)
4. end while

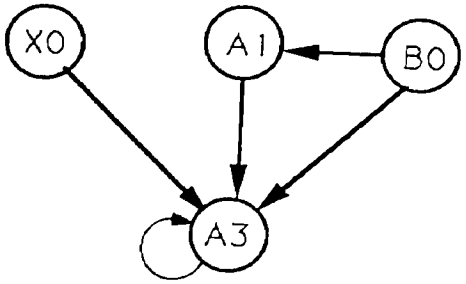


Figure 9. Data dependency graph for a loop
(Source: [Bieman and Edwards 85])

A data dependency graph (DDG) is a directed graph with each node representing a variable definition and each edge representing a possible data dependency. Figures 8 and

9 are two examples of DDGs corresponding to the segments of code given in each figure. The subscripts represent the source code lines in the definition and the subscript 0 represents the initial definition of each variable.

The rooted spanning tree complexity with root node x of a graph G is defined as the number of spanning trees with root x that can be constructed from the graph consisting of the nodes and the arcs of G that are successors of x [Bieman and Edwards 85]. The rooted spanning tree complexity can be considered a result of the implicit decomposition of a given DDG into spanning trees. The notion of using the number of spanning trees in a graph as a measure of complexity has been described in graph theory [Temperly 81].

The number of directed spanning trees with root r of a digraph with no self-loops is given by the minor of its indegree matrix, D(i,j), which results from the erasure of its rth row and column [Even 79] (see the APPENDIX for a definition of the indegree matrix). As an example, the spanning trees of the graph of Figure 10 is given in Figure 11.

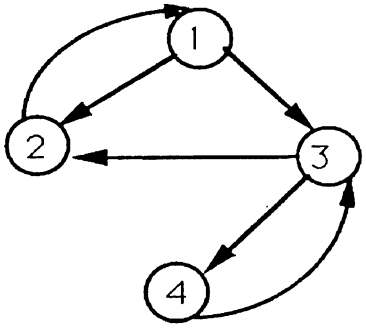


Figure 10. An example graph

$$D(i, j) = \begin{bmatrix} 1 & -1 & -1 & 0 \\ -1 & 2 & 0 & 0 \\ 0 & -1 & 2 & 1 \\ 0 & 0 & -1 & 1 \end{bmatrix}$$

The number of spanning trees rooted at node 1 is $\begin{vmatrix} 2 & 0 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{vmatrix} = 2$

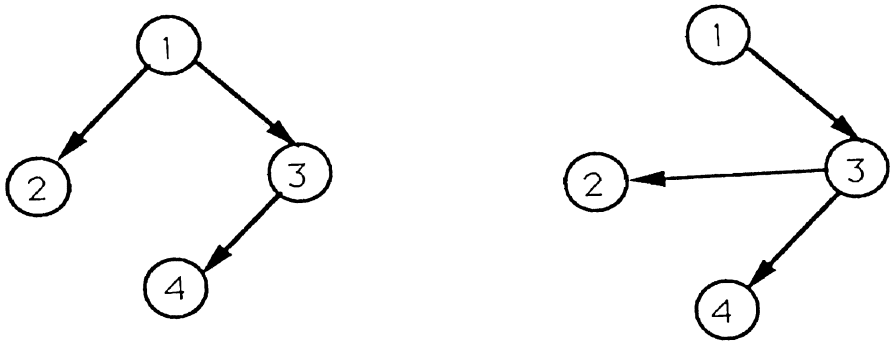


Figure 11. Two spanning trees rooted at node 1 of the graph of Figure 10

The definition slice graph of the DDG of a program P at node x is defined to be a subgraph of the DDG of P that includes nodes in the DDG of P that are predecessors of node x. The edges of the definition slice graph include all paths from predecessors to x [Bieman and Edwards 85]. Because programmers appear to decompose programs into slices when debugging [Weiser 82], the definition slice graph of a DDG at a particular node can be decomposed into spanning trees with the specified node as root, and the resulting spanning trees can be used for modeling the comprehension of a particular program segment.

Chase analyzed various published algorithms for finding all the spanning trees of a graph and compared them in terms of their efficiency [Chase 74]. As a result, he has come up with an algorithm, which he termed "new" and concludes that this algorithm is more efficient than all other algorithms. The complexity of the "new" algorithm is given by $t(e\sqrt{2})^{-n}$, where t is the number of spanning trees in the graph, e is a constant (≈

2.718), and n is the number of nodes in the graph.

3.5 Decomposition of a Flow Graph into Trees

Tamine makes use of the tree-like structure of a program to define a complexity measure that takes into account the nesting level of the structures of the program [Tamine 83]. In fact, the proposed complexity measure is an extension of the "characteristic polynomial" for the d-graph (see the APPENDIX for a definition) of a program proposed by Cantone et al. [Cantone et al. 83].

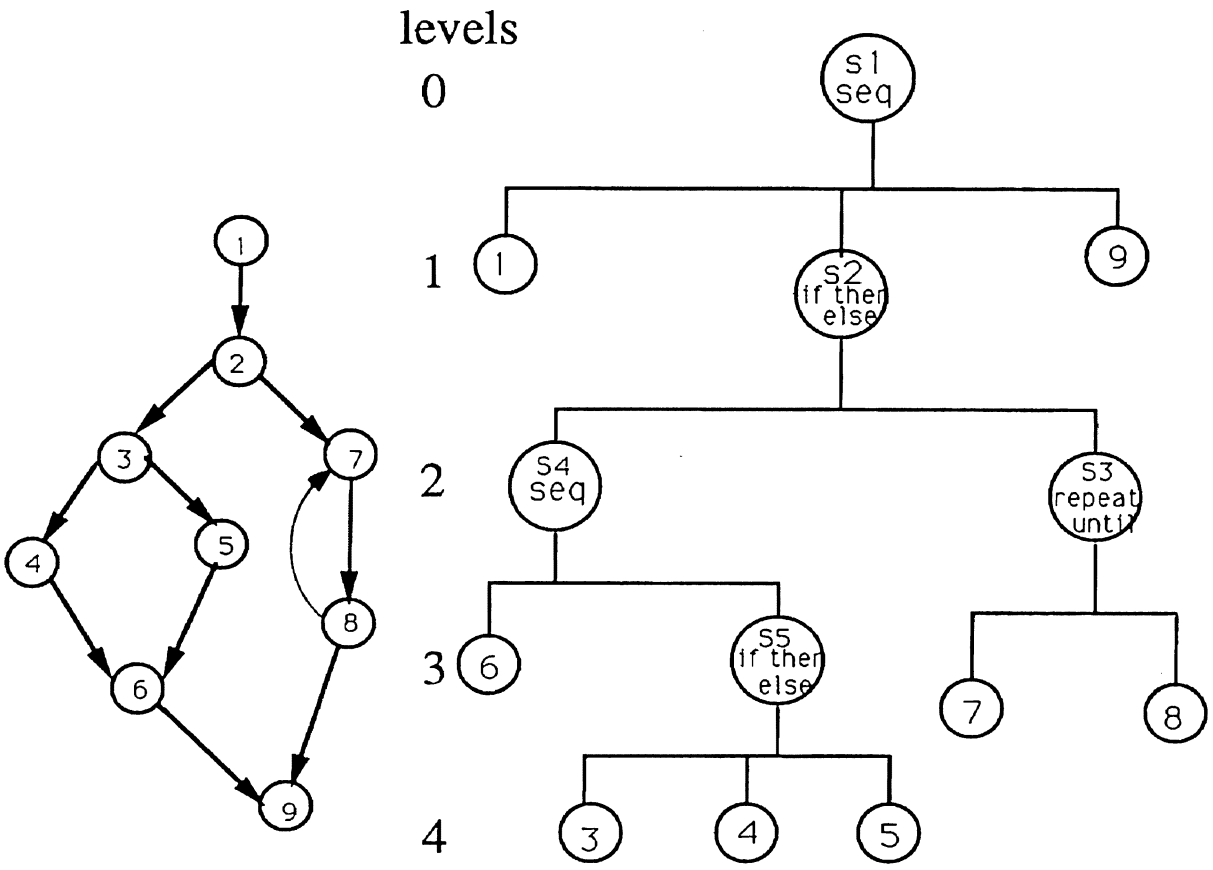


Figure 12. An example of a d-graph and relative nesting

Cantone et al. suggest that the nesting level of control structures influences the

complexity of the program, and hence they proposed a complexity measure called "characteristic polynomial" as follows [Cantone et al. 83]. The nesting of a program is a directed graph in which:

- a node represents a primitive node or a single-entry/single-exit component of the program and its quality (i.e., either a cyclic, selective, or serial structure);
- the edges represent the inclusion relations between the components of the nodes.

An example of the nesting of a program with its levels marked is given in Figure 12.

Each node of the program nesting graph is associated with a characteristic number (CN) according to the following rules [Cantone et al. 83].

1. For each primitive node (coinciding with a node of the flow graph), CN is 1;
2. For a (level J) structure of the sequential type, CN is given by the product of CNs of the structures and of the primitive nodes which are immediately (J + 1) contained in it;

primitive nodes and structures	CNs
1,2,3,4,5,6,7,8,9	1
S5	2
S4	2
S3	c
S2	2+c
S1	2+c

Figure 13. Characteristic numbers relative to the nesting structure in Figure 12

3. For a structure of the selective type, CN is given by the sum of CNs of the structures and of the nodes (with outdegree less than 2), which are immediately contained in it, increased by 1 in the case of **if-then**; and
4. For a cyclic structure, CN is given by the product of a weight *c* by CN of the unique structure or primitive node (with outdegree less than 2) immediately contained in it, or by CN + 1 if the cyclic structure in question is a **while-do**.

The CN of the unique node at level 0 is called the "characteristic number of a program" and, since it is a linear expression in *c*, it is called the "characteristic polynomial of a program". In Figure 13, the CNs of the nodes and structures of the program nesting of Figure 12 are listed.

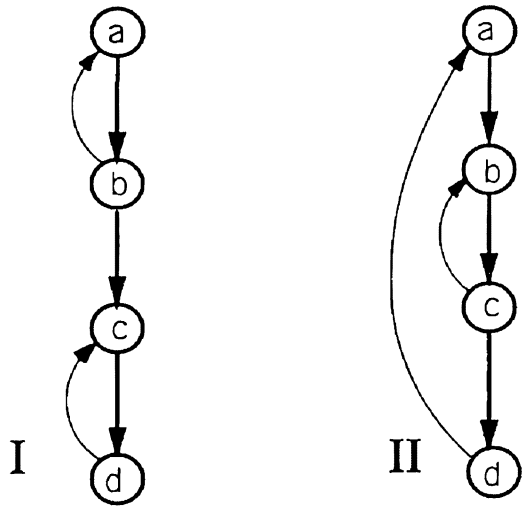
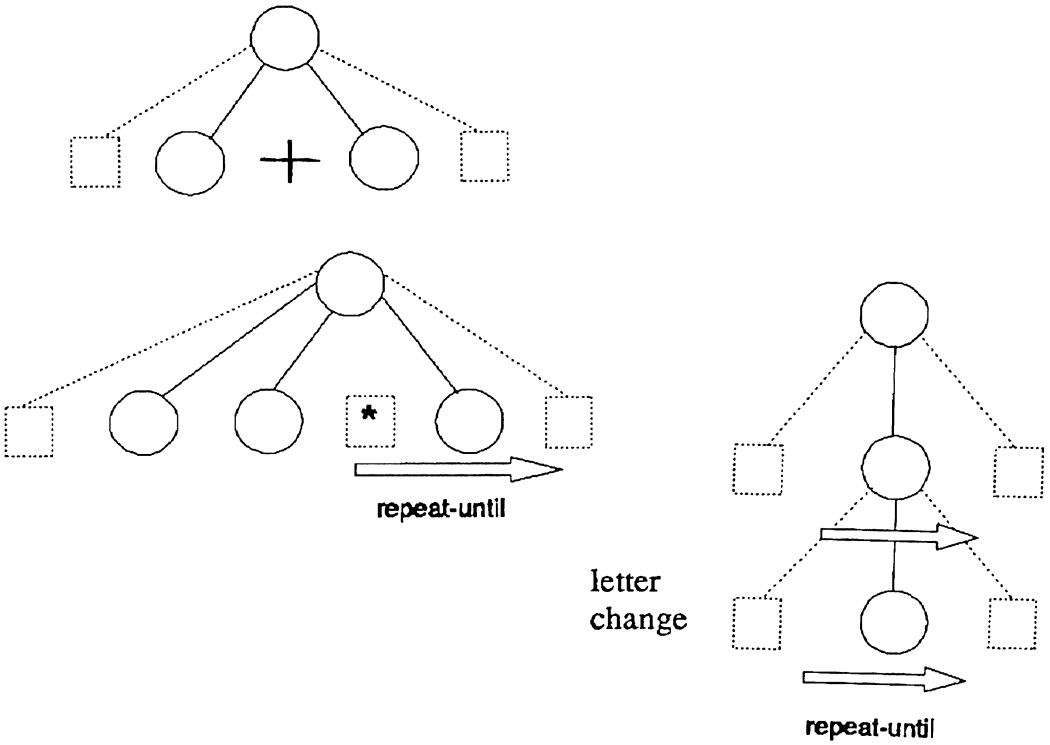


Figure 14. Two d-graphs of different complexities (Source: [Tamine 83])

Tamine argues that the above complexity measure is not sensitive to the nesting of cyclic structures, and is the same for equal number of cycles irrespective of whether they

are in series or nested [Tamine 83]. For example, in Figure 14 the characteristic polynomial (CP) is c^2 for both flow graphs I and II, but intuitively II is more complex than I. Tamine suggested to use different symbols for the nesting of cyclic structures starting from a and down. Therefore, in Figure 14, the CP for I and II are given by $CP_I = a^2$ and $CP_{II} = ab$, instead of $CP_I = c^2 = CP_{II}$, as suggested by Cantone et al.

Tamine proposed that, once the flow graph of a program is represented in the form of a tree (i.e., the nesting structure of a program), the "characteristic polynomial" (complexity measure) of the program can be calculated by the rules given in Figure 15. The dotted boxes in Figure 15 represent the begin and end nodes of the selection node, and the arrow represents repetition.



+: for selection
*: for repetition
letter change: for nesting of repetition

Figure 15. Rules for calculation of the characteristic polynomial

3.6 Decomposition of a Flow Graph into Prime Subgraphs

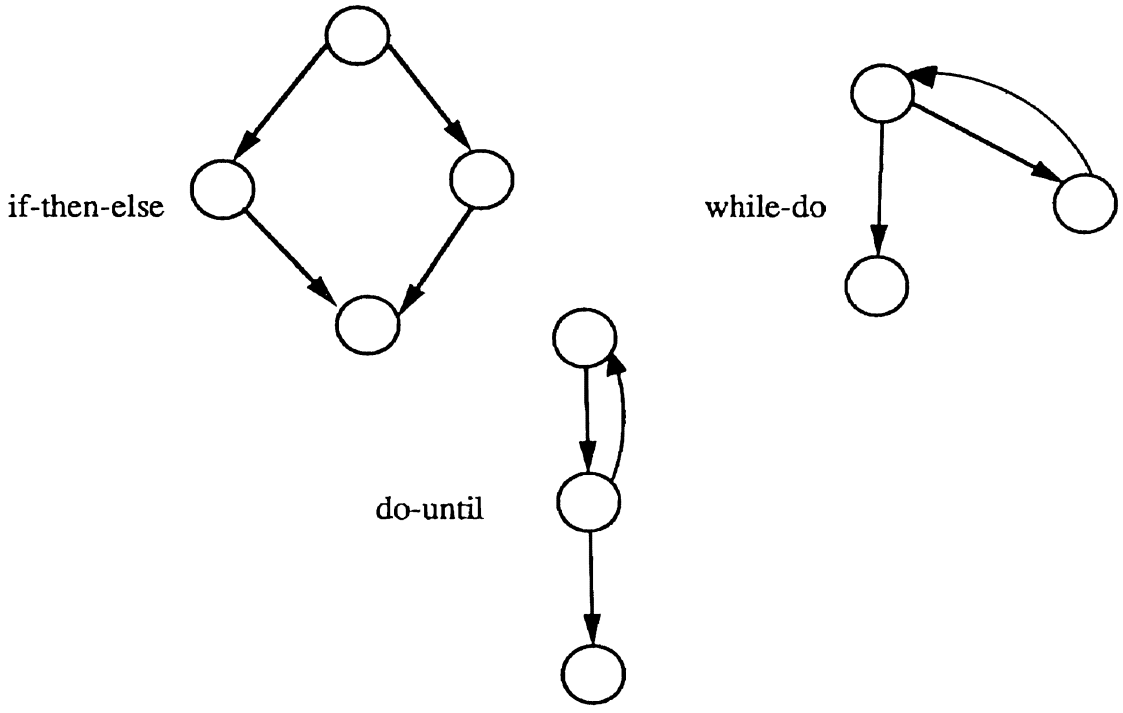
Informally, primes are single-entry/single-exit subflow graphs that are not further decomposable. Formal definition of a prime subgraph follows from the following definitions [Zuse 91].

--- A subflow graph F' = (N', E', *a*', z') of a flow graph F = (N, E, *a*, z) is a flow graph where F' is a subgraph of F, and the outdegree in F' of each node ($\neq$ z') in F' is the same as its outdegree in F (thus the only possible exit from F' to nodes in F - F' is via z').

--- If the only entry to F' - {z} from nodes in F - F' is via *a*', then F' is a single-entry subflow graph of F.

--- F' is a proper (single-entry) subflow graph of F, if F' is non-trivial and F $\neq$ F'.

--- A proper subflow graph F' = (N', E', *a*', z') of F = (N, E, *a*, z) is maximal (in F) if there is no proper subflow graph F" = (N", E", *a*", z") of F for which F' is a proper subflow graph of F".



A flow graph is a prime graph if it contains no proper single-entry subflow graphs.

It is important to note that all the common control structures are prime programs [Forman 82] as depicted in the last page.

An example of a program that is prime and a program that is not prime are given in Figures 16 and 17.

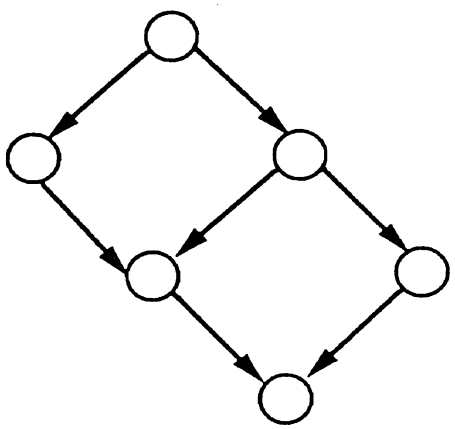


Figure 16. The flow graph of a program that is prime

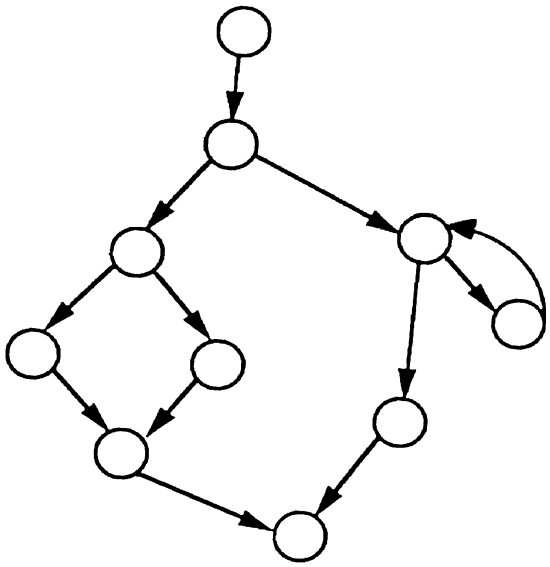


Figure 17. The flow graph of a program that is not prime

There are two methods of decomposing a flow graph into prime graphs: Forman's approach and the Fenton-Whitty scheme. The following two subsections present a brief discussion of these two methods.

3.6.1 Forman's Approach

Forman explains the decomposition of a flow graph into prime subgraphs using m-graphs [Forman 82] (see the APPENDIX for a definition). A subprogram cutset is a set of two arcs whose removal separates an m-graph into two blocks, the exterior block contains the entry and exit nodes while the interior block does not. An m-graph M is said to be prime if the only subprogram cutsets of M are the ones whose interiors contain only a single assignment node or of all the nodes of M excluding the entry and exit nodes.

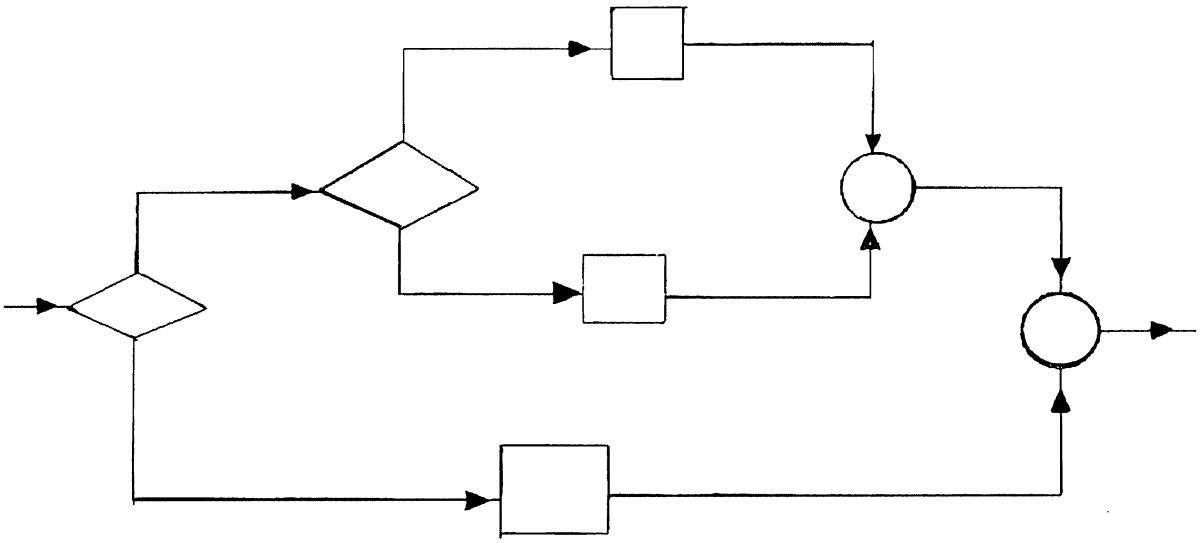


Figure 18. The m-graph of a program (Source: [Forman 82])

Forman proposed to decompose m-graphs into a hierarchy of primes. The hierarchy is formed by finding subprogram cutsets (whose interiors contain more than one node),

replacing their interior with a single assignment node that is called a *call node,* and making their interior an m-graph to which the *call node* points [Forman 82]. When this operation can no longer be performed upon the hierarchy, the result is called prime program decomposition because all the m-graphs in the hierarchy are prime. Figure 19 is the prime program decomposition of the m-graph given in Figure 18.

Prime program decomposition is employed to solve the problem of abstract data flow analysis [Forman 82]. A linear algorithm for finding prime program decomposition is given by Tarjan and Valdes [Tarjan and Valdes 80] which makes use of the algorithm to decompose a biconnected graph into triconnected components [Hopcroft and Tarjan 73]. In the algorithm given by Tarjan and valdes, they assume that the flow graph is biconnected, which is true in most of the cases.

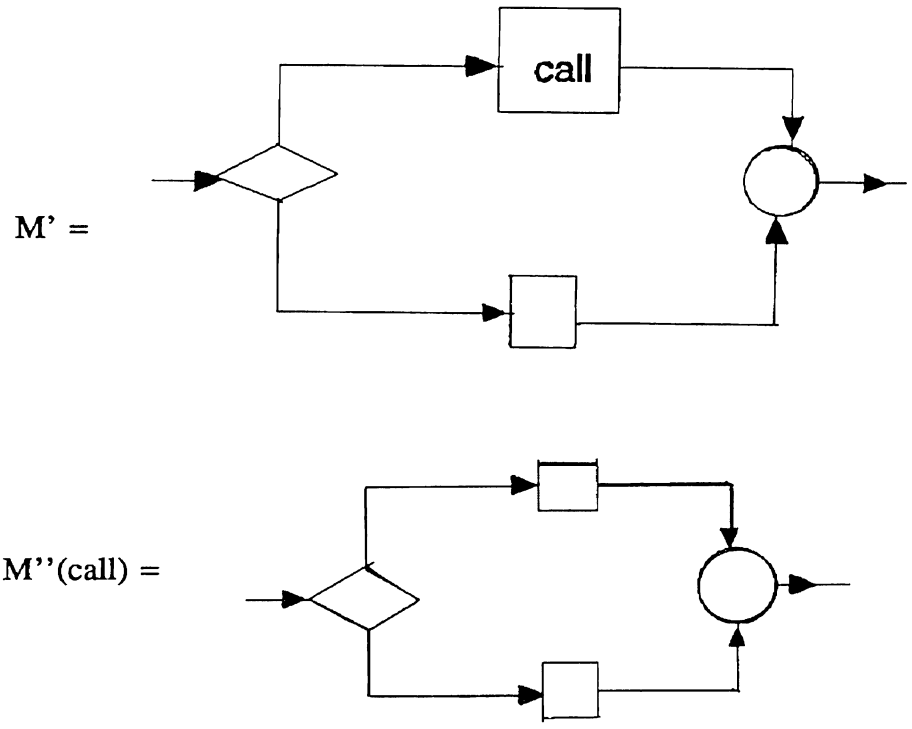


Figure 19. The prime program decomposition of the m-graph of Figure 18

3.6.2 The Fenton-Whitty Scheme

The Fenton-Whitty scheme [Alvey 88], which is employed to decompose a flow graph into prime subgraphs, differs from Forman's approach in that the outdegree of decision nodes does not change between the subflow graphs and the primes [Zuse 91]. The decomposition of F_2 in F_1 (where F_2 is a subflow graph of F_1) is the flow graph obtained by replacing F_2 with a single edge (x, z'), where z' is the exit node of F_2 and x is a new procedure node "replacing" F_2 [Fenton and Whitty 86]. The resulting flow graph is denoted by F_1(x for F_2) (i.e. the subflow graph F_2 has been replaced by the node x). Figures 20 and 21 contain an example that illustrates this process.

F =

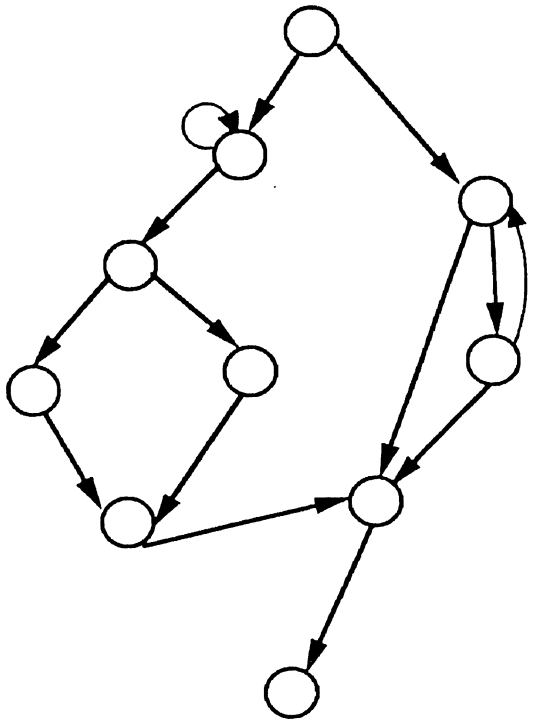


Figure 20. The flow graph of a program

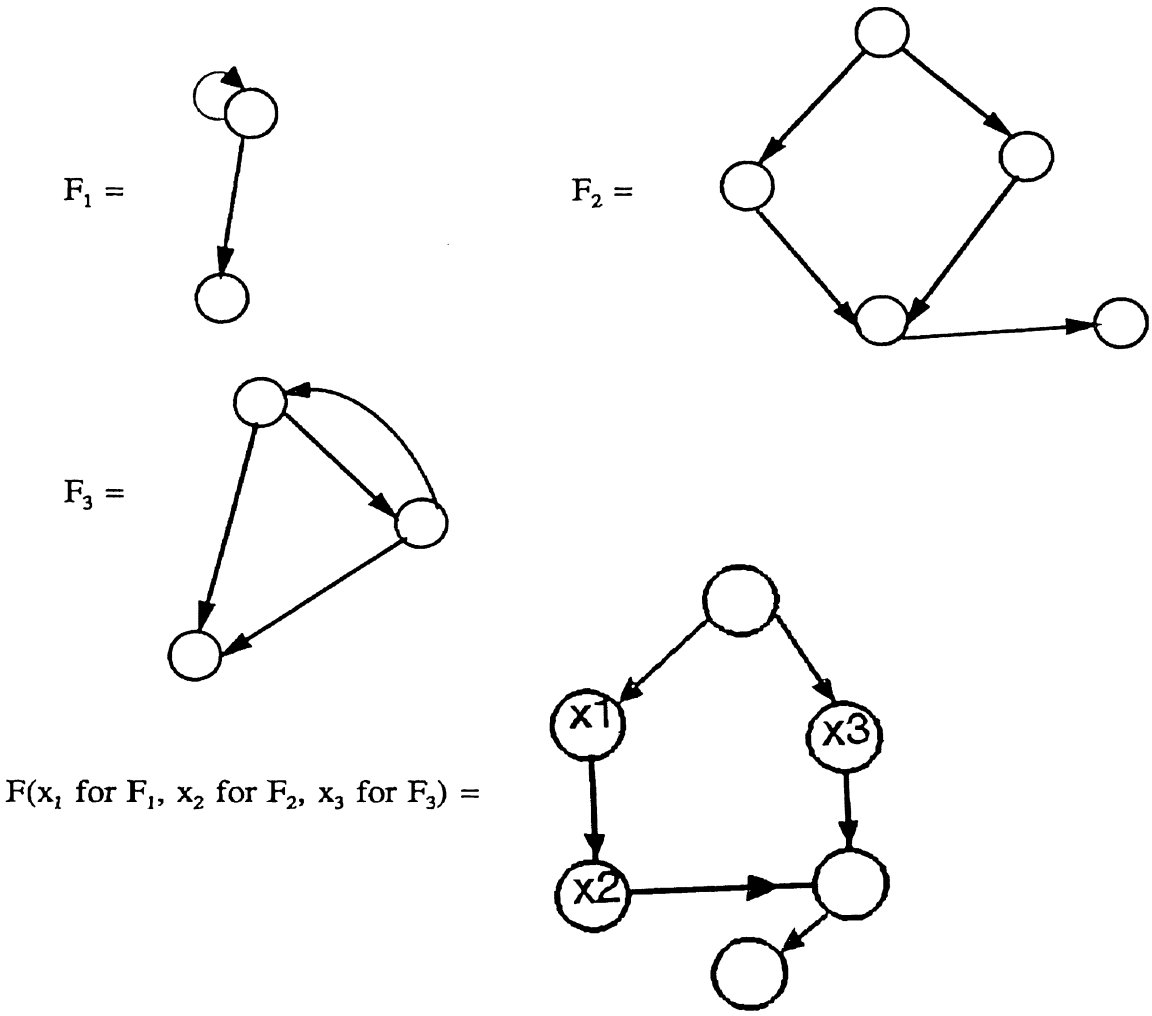


Figure 21. Decomposition of the flow graph of Figure 20 into prime subgraphs using the Fenton-Whitty scheme

According to the Fenton-Whitty scheme, the decomposition algorithm identifies the maximal subflow graphs of the flow graph and decomposes the flow graph into its underlying atom (i.e., a prime or P_n) as well as the subflow graphs which are nested within it [Alvey 88]. The subflow graphs are placed where they belong in the decomposition tree, the subflow graphs are further decomposed as above, and the results

arising from the decomposition are put into their places on the tree. This process is repeated until all leaves of the trees are atoms. Figure 22 shows a flow graph and its corresponding decomposition tree (the strings used as node labels for the trees represent certain types of flow graphs as explained in the APPENDIX under "certain common flow graphs" [Alvey 88]).

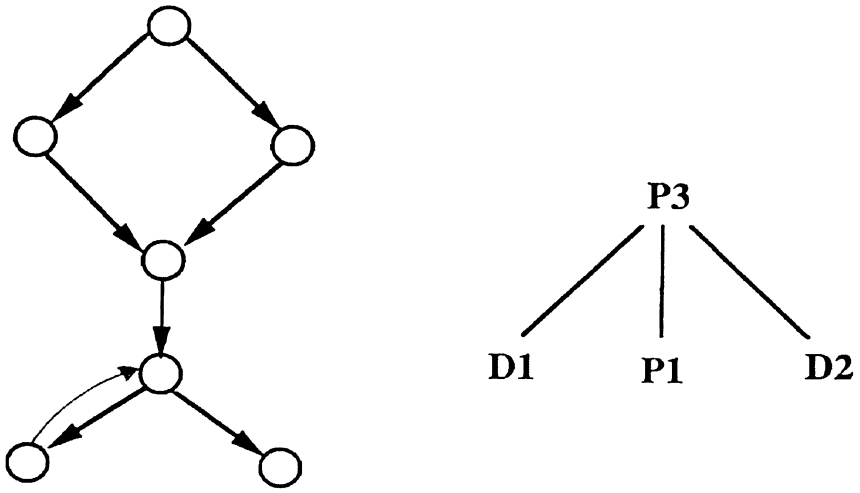


Figure 22. A control flow graph and its prime decomposition tree.

3.7 Decomposition of a Program into Slices

Another major technique, which is worth mentioning in this review, is decomposing a program into slices. Program slicing consists of decomposition of a program based on data flow and control flow analysis. A slice is a set of statements of the program that are not necessarily contiguous, but are related based on their flow of data [Weiser 84]. Program slicing was introduced by Weiser and he argued that programmers use slices in debugging programs [Weiser 82]. A program slice is a set of statements in a program that directly or indirectly influence the value of a variable in a statement.

The primary application of slicing is in debugging. A number of debugging tools have been devised using slicing [Nanja and Samadzadeh 90] [Samadzadeh and Wichaipanitch 93] [Samadzadeh and Hsiao 93]. Slices can also be used in testing, maintenance, parallelization, comprehending programs, and the integration of multiple versions of a program.

There are two major approaches to finding the slices of a program: one due to Weiser [Weiser 84] and the other due to Ottenstein and Ottenstein [Ottenstein and Ottenstein 84]. Weiser's approach uses the flow graph of a program (in which each statement represents a node). The data and control dependencies between statements are obtained using control and data flow analysis [Weiser 84]. To each statement (or node), except the procedure statements, are associated two sets of variables called *def* and *use*. *def* is the set of variables whose values have been changed by that statement, and *use* is the set of variables whose values are used by that statement [Hecht 77]. Also, to each statement or node x a set *rd*(x) is associated, which is the set of all nodes containing definitions that reach node x. The slice of a particular variable in a statement can be obtained using the sets *def*, *use*, and *rd*.

The approach due to Ottenstein and Ottenstein uses the program dependence graph (PDG) and reachability analysis. The edges of the PDG represent the control and data dependencies between the statements of a program. The slice of a particular statement S can be obtained from those nodes of PDG that can reach the statement S.

The above two approaches also differ in their slicing criterion (a factor based on which a slice is found), the former uses <statement, variables> while the latter uses <statement>.

There are a number of slicing algorithms that operate on different representations of

a program. Lakhotia generalized the program slicing algorithms that were originally defined for representations of programs, to operate on directed graphs by defining the notion of graph slicing [Lakhotia 91]. This generalization enables the above two slicing approaches to be modelled as abstract mathematical operations without any regard to a program's structure or its semantics. The following approach to slicing as a graph-theoretic operation is due to Lakhotia [Lakhotia 91] and is defined to operate over directed graphs. Lakhotia defines a modified version of Weiser's algorithm that uses the same slicing criterion as Weiser's algorithm. However, the algorithm uses reachability analysis to detect statements belonging to a slice. Thus, different slicing algorithms can use the same technique irrespective of the different representations of the program and the slicing criterion. This leads to abstract reachability analysis which is the technique for detecting statements in a slice, as an operation on directed graphs called *graph slicing*.

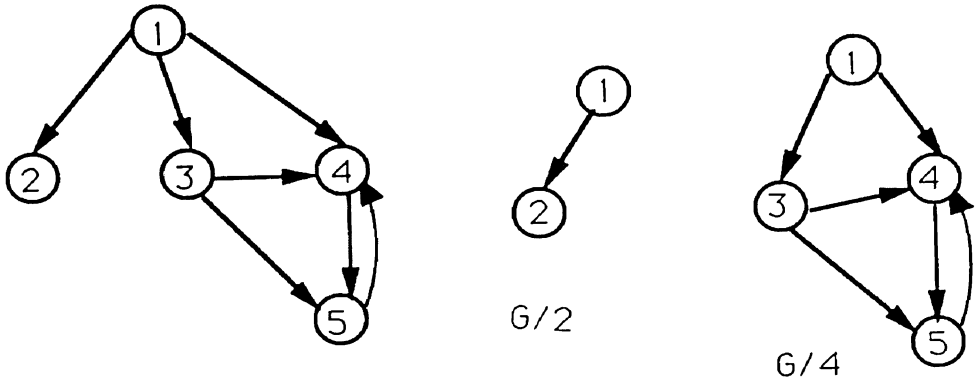


Figure 23. An example graph and its slices at different vertices.

A graph slice G/S of a graph G, over a set of nodes S, is a minimal unreachable subgraph (see the APPENDIX for a definition) of G with respect to S. The graph slice

G/S consists of the set of nodes $\{x \mid x \in S \vee (\exists$ a path from x to x_i in G, where $x_i \in S\}$ and the set of edges $\{(u,w) \mid u,v \in G/S \wedge (u,w) \in G\}$. Figure 23 shows a graph and its slices at different vertices.

The vertices belonging to a graph slice of a graph G can be obtained by determining the reachability matrix of G. From the reachability matrix of a graph, the nodes of the graph that have paths reaching a particular node can be determined. The edges of the graph slice can be obtained based on the above-mentioned set notation. The reachability matrix of G needs to be calculated only once for all the graph slices.

The notion of slicing was originally defined for a program as a syntactic decomposition technique. However, the preceding discussion as well as the obvious similarity between slicing and decomposition of a flow graph into spanning trees shows that slicing can be considered a graph decomposition method as well.

CHAPTER IV

COMPARISON OF DIFFERENT DECOMPOSITION TECHNIQUES

The different decomposition techniques of graph representations of a program (discussed in Chapter III) are compared in the following sections based on the dimensions mentioned in Chapter I, namely,

(a) Complexity of calculation;

(b) Quantifiability - i.e., meaningful quantifiability of the decomposition process or the product;

(c) Composability - i.e., whether any of the resulting units of decomposition is reusable or not;

(d) Structured vs. non-structured graphs - i.e., whether each decomposition method is applicable to structured flow graphs only, or it is also applicable to non-structured flow graphs;

(e) Repeated decomposition - i.e., whether the decomposition process can be applied repeatedly;

(f) Availability (e.g., as a by-product of the compiling process);

(g) Formalism dependence - i.e., whether a particular decomposition method applies to a particular type of graph model or it is generalizable to different types of graphs models as well;

(h) Scalability;

(i) Interdependence - i.e., whether each method is independent of other methods or it can be obtained from one or more of the other methods;

(j) Automatibility;

(k) Uniqueness;

(l) Cover or partition - i.e., whether each decomposition serves as a cover or as a partition for the graph model.

4.1 Complexity of Calculation

This section discusses the complexity of the various decomposition techniques of the graph models that were discussed in Chapter III. First, the complexity of the construction of the three graph models, namely, control flow graph, data dependency graph, and program dependence graph is discussed.

In order to generate a flow graph of a program, typically the given program is first rewritten in an intermediate form to express the implicit transfers of control (using *goto*'s). The algorithm to construct a flow graph first determines the basic blocks of the program and then determines the edges which represent the block-to-block transfer of control. Such an algorithm can be presented as follows [Aho and Ullman 73]:

--- The basic blocks are determined by finding the block entries. A statement S in a program P is a block entry if a) S is the first statement in P, b) S is labeled by an identifier which appears either after *goto,* in a *goto,* or after a conditional statement, or c) S is a statement immediately following a conditional statement.

--- The basic block with the block entry S consists of S and all the statements following S, a) up to but not including the next block entry or, b) up to and including a halt

statement. If nodes i and j of the flow graph correspond to blocks i and j of the program, an edge is drawn from node i to j if a) block j follows block i in the program, and the last statement in block i is not a *goto* or halt statement, or b) the last statement in block i is either *goto* L or it is if <condition> *goto* L, and L is the label of the first statement in block J. The node corresponding to the block containing the first statement of the program is the begin node of the flow graph.

It appears that the algorithm makes three passes through a program, it determines the basic block entries during the first pass, the basic blocks in the second pass, and finally connects the blocks by edges during the last pass. Once the given program is rewritten in an intermediate representation to express the transfers of control using *goto*'s, the complexity to find the control flow graph can be derived, based on the above discussion, to be 3n or O(n), where n is the number of lines in the program at the intermediate level.

Given the flow graph of a program, the worst case complexity of the algorithm given by Bieman and Edwards to construct a DDG (Data Dependency Graph) is $O(e^2)$, where e is the number of edges in the input flow graph [Bieman and Edwards 85].

The main concern in constructing the PDG (Program Dependence Graph) of a program is the amount of space required to represent a PDG, which depends on the dependence structure of the program. Ottenstein and Ottenstein suggest that the space requirements of the PDG of a program is approximately three times as much as a corresponding representation which represents each block as a DAG (Directed Acyclic Graph) [Ottenstein and Ottenstein 84]. The construction of the PDG of a program requires the program and the flow graph of the program as input. The cost of constructing the

control dependencies of the PDG is $O(n^2)$, where n is the number of nodes in the flow graph of a program.

The complexity to calculate the fundamental circuits in a graph is $O(n^3)$ in the worst case and $O(n^2)$ for a sparse graph, where n is the number of nodes in the graph [Gibbons 85].

Lengauer and Tarjan give a fast algorithm, whose time complexity is O(e logn), to calculate the dominators in a graph, where e and n are the number of edges and nodes of the graph [Lengauer and Tarjan 79].

The complexity to find the intervals in a graph is O(e), where e is the number of edges in the graph [Hecht 77].

The complexity to calculate one spanning tree of a graph is O(max(n,e)) [Gibbons 85], where n and e are the number of nodes and edges in the given graph. There are a number of algorithms discussed in the literature to find all the spanning trees of a graph. Chase analyzed the various available algorithms by comparing them in terms of their complexity [Chase 74]. Chase presents an improved algorithm whose complexity is given by $t(e\sqrt{2})^{-n}$, where t is the number of spanning trees in the graph, n is number of nodes in the graph, and **e** is a constant (≈ 2.718).

Given the control flow graph of a program, the algorithm to get the corresponding tree structure and the time complexity of finding the prime subgraphs using the Fenton-

whitty scheme are not available in the open literature.

The algorithm given by Tarjan and Valdes [Tarjan and Valdes 80] finds the prime subgraphs of a control flow graph using Forman's approach in O(n+e) time, where n and e are the number of edges and nodes of the graph.

The complexity of computing a slice depends on the approach adopted. Using Weiser's approach, the complexity of calculating a slice is O(n e loge), where n and e are the number of nodes and the number of edges in the flow graph [Weiser 84]. Once the PDG is constructed, Ottenstein and Ottenstein suggest that, using their approach, a linear time algorithm would be required to construct the slice of a program at a statement [Ottenstein and Ottenstein 84]. The nodes of a graph slice at node set S of a graph G can be obtained from the reachability matrix of G. The reachability matrix can be obtained using Warshall's algorithm [Skvarcius and Robinson 86]. The time complexity of Warshall's algorithm is $O(n^3)$, where n is the number of nodes of the graph.

4.2 Quantifiability

In this section, the quantifiability of the products of the different decomposition techniques that were described in Chapter III, are discussed. The point to consider is whether the resulting count is meaningful as some indication of the software development process such as conceptual complexity, difficulty of testing, or maintainability.

The number of basis paths or circuits in the control flow graph of a program gives the cyclomatic complexity of that program and it indicates how difficult it is to test a

program [McCabe 76]. The number of basis paths or circuits in a DDG (Data Dependency Graph) gives one of the data dependency complexities as viewed from a programmer's perspective [Bieman and Edwards 85].

The concept of dominators and post dominators of decision nodes in a flow graph is used in the calculation of a measure of software complexity called CSG due to Schmidt and Gong (see the APPENDIX for a definition) ([Schmidt 85a] [Schmidt 85b] as cited in [Zuse 91]). The complexity measure CSG is an extension of McCabe's cyclomatic complexity measure, which is extended to reflect the degree of nesting of the flow graph of a program.

The interval derived sequence length and loop-connectedness (defined in Section 3.3) are two software complexity measures that are based on intervals. Hecht suggests that the interval derived sequence length corresponds intuitively to the maximum "nesting depth" of the loops of a reducible computer program (i.e., its flow graph) [Hecht 77].

The number of spanning trees in a DDG (Data Dependency Graph) gives the rooted spanning tree complexity of that DDG (also referred to as the cyclical complexity of that DDG) [Bieman and Edwards 85]. This number can be an indicator of the software complexity or testing difficulty of a program.

The software complexity measure "characteristic polynomial", based on the tree structure of a program, is sensitive to the nesting of the control structures of a program [Cantone et al. 83] [Tamine 83]. Also, it seems that the height of the tree structure of a

program corresponds to the nesting level of the control structures in that program. The deeper the nesting of the control structures in a program, the taller will the height of the tree structure be, and hence it can be argued that the software complexity of a program increases with the height of the tree structure of that program.

A number of complexity measures that are based on prime program decomposition are discussed by Zuse [Zuse 91]. For instance, the number of primes in a flow graph has been proposed as a software complexity measure by Project SE/069: Structure-Based Software Measurement (SBSM) [Alvey 88], which is based on the prime program decomposition of the flow graph of a program using the Fenton-Whitty scheme.

A number of metrics based on slicing are suggested by Weiser ([Weiser 81] as cited in [Ottenstein and Ottenstein 84]). These metrics include 1) coverage, a measure of the length of slices compared to the length of the program; 2) overlap, a measure of the number of statements in a slice that belong to no other slice; 3) clustering, the degree to which slices are reflected in the original code layout; 4) parallelism, the number of slices with few statements in common; and 5) tightness, the number of statements in every slice. These metrics can be extended to graph slices also.

4.3 Composability

In this section, the composability of the various decomposition techniques is discussed. Composability is discussed in the sense of reusability of the decomposed units for the purpose of software comprehension. Also, composability is discussed to find out whether any of the resulting units of the different decomposition techniques can be reused

to compose the original graph model from which they were decomposed. Composability can also refer to the capability of the decomposed units to be used as reusable units in the construction of programs by first composing (or constructing) their graph models. Thus, a repository of decomposed units can serve as a software parts catalog stored in the form of flow graphs.

The basis circuits or paths of the flow graph of a program is a graph-theoretic concept that is not closely related to the semantic structure of the program. Hence, the potential for the reuse of the basis circuits or paths of the flow graph of a program, is not very high. When a flow graph is decomposed into a set of basis circuits or paths, the original undirected graph can be obtained by the operation of union. If the fundamental circuits are all directed circuits, then the union will be the original directed graph; whereas if the fundamental circuits contain semicircuits (circuits whose directions are not the same as those of the edges of the graph), then the directions in the union will be different from those in the original flow graph.

Dominance relationships in the flow graph of a program depend on each program's structure and they do not involve subgraphs of the flow graph. Hence, dominators are not good candidates for reusability. It is not always possible to get back the original graph from its dominators tree, as the dominator tree ignores loops. It is possible to have two flow graphs, one with a loop and the other without a loop, and arrive at the same dominator tree. For example, the flow graphs in Figure 24, give rise to the same dominator tree.

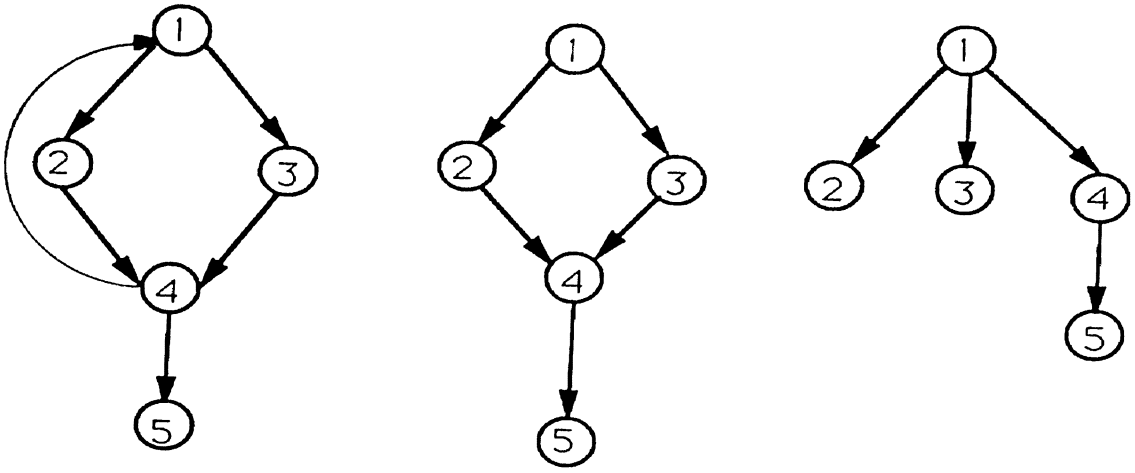


Figure 24. A single dominator tree corresponding to two different flow graphs

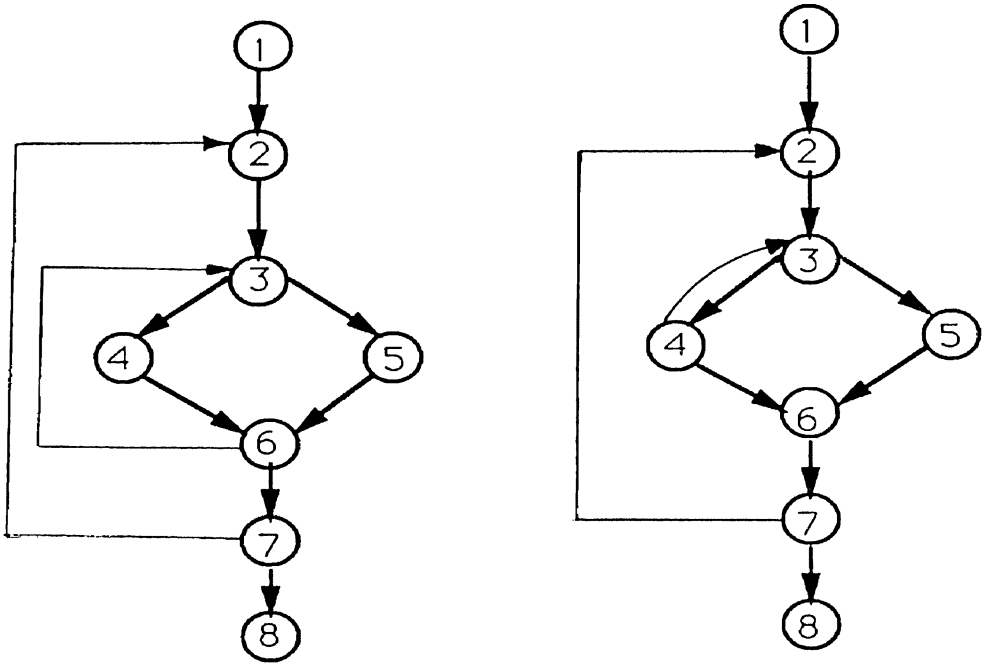


Figure 25. Two flow graphs with the same set of intervals

Intervals need not cover the edges of the flow graph of a program. Hence, even if the interval of one flow graph is similar to the interval of another flow graph, the edges corresponding to the intervals need not correspond to each other. Hence, the potential for reuse of intervals is not very high. Using the same argument that was used for dominators with respect to reconstructing the original flow graph, it can be shown that two different flow graphs can have the same set of intervals. Hence, the original flow graph cannot be constructed from the set of intervals. For example, the graphs in Figure 25 have the following set of intervals

I(1) = 1, I(2) = 2, I(3) = {3,4,5,6,7,8}.

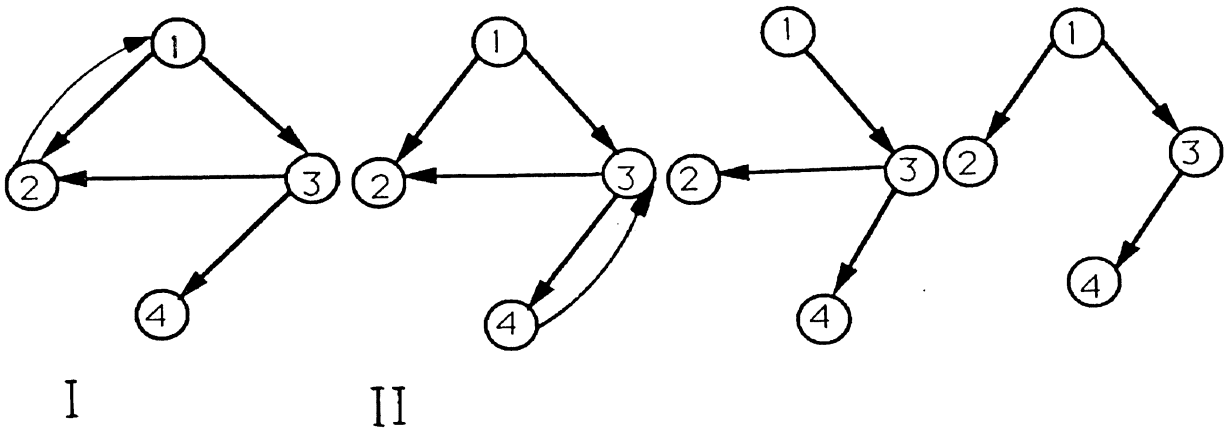


Figure 26. Two different graphs with the same set of spanning trees rooted at node 1

Similar to the argument presented for the reusability of basis circuits or paths, the spanning trees of the flow graph of a program need not correspond to the semantic structure of the program. Hence, it can be concluded that the spanning trees of the flow graph of a program will not be of much use in terms of reusability. Similar to the

argument presented for reconstructing the original flow graph from intervals, two different graphs can have the same set of rooted spanning trees. Hence, the original graph cannot be in general reconstructed from the set of decomposed rooted spanning trees. Figure 26 shows two graphs I and II, that have the same set of spanning trees rooted at node 1.

Corresponding to each program, there exists a unique tree structure. Hence, decomposition of the flow graph of a program into tree structure cannot be reused in any other situation. From the relative nesting (i.e., the tree structure) of a program, it is easy to get back the original d-graph.

Prime subgraphs of the flow graph of a program seem to be good candidates for reusability. Primes are single-entry/single-exit subgraphs of the flow graph of a program and hence, they can be replaced by single nodes. The reusability of prime subgraphs can be described as follows. Consider the flow graph of a program that contains primes and has a structure similar to the primes that occurred in the flow graphs of other programs that have already been comprehended. Hence, the primes of the flow graph of the program under consideration can be replaced by single nodes, because they have been comprehended as part of the flow graphs of other programs. Thus, the flow graph of the program under consideration can be systematically reduced during the comprehension process. Therefore, it can be concluded that decomposition into prime subgraphs has a high potential for reusability in the comprehension (and perhaps even construction) of software.

In the prime decomposition of a flow graph, when using Forman's approach, *virtual edges* (which are not edges of the original graph) are introduced in pairs with the same

labels. Thus, the original graph can be reconstructed from the decomposed primes by "gluing" them along the virtual edges with the same label, and then eliminating the virtual edges [Tarjan and Valdes 80]. In the prime program decomposition, when using the Fenton-Whitty scheme, the original graph can be obtained by nesting the prime subgraphs in their places in the corresponding prime decomposition tree [Alvey 88].

4.4 Structured vs. Non-Structured Graphs

Decomposition into basis paths and circuits, dominators, intervals, spanning trees, and prime subgraphs are defined on any single-entry/single-exit directed graphs. Hence, they can be applied to structured as well as non-structured flow graphs. Graph slices and slices are also defined on directed graphs, which correspond to some representation of a program. Hence, they are also applicable to both structured as well as non-structured graphs. Decomposition into trees is defined on d-graphs, which can only be the flow graphs of structured programs. Hence, it is applicable only to the flow graphs of structured programs.

4.5 Repeated Decomposition

Repeated decomposition is not meaningful in the cases of decomposition into basis paths and circuits, dominators, trees, or spanning trees.

Prime program decomposition involves repeated decomposition in building the prime decomposition tree and it was explained in detail in Chapter III.

The intervals of one flow graph can be considered as the nodes of another flow graph in which there is an edge between the intervals J and K iff J ≠ K and there is an edge from a node in J to the header of K, and this process may be performed repeatedly [Hecht 77]. The flow graph derived from the original flow graph is called a derived flow graph of the original flow graph. If $F = F_0, F_1, ..., F_k$, where k is an integer, is the derived sequence of a flow graph F (defined in Section 3.3), then F_1 is the derived flow graph of F_0, F_2 is the derived flow graph of F_1, and so on. F_k is called the limit flow graph of F. A flow graph is reducible (an RFG) iff its limit flow graph is a single node with no edge, otherwise it is called irreducible (an IRFG) or non-reducible. This repeated decomposition can be applied to the set of intervals of a flow graph in order to find out whether a flow graph is reducible or not.

A slice is in general, a reduced program of the original program, and it is an independent program which reproduces exactly a projection of the original program's behavior [Weiser 84]. Hence, a slice can be decomposed further into slices with respect to a vertex set belonging to the original slice. This process can be repeated. This repeated decomposition can be extended to graph slices as well. The meaning and the purpose of this repeated decomposition into slices, and the question of when to stop this repeated decomposition, has to be investigated further.

4.6 Availability

This section discusses whether the resultant units of the decomposition techniques of a program (or a program flow graph) are available as a product of some existing process.

The concept of "dominance" relationships is used in compilers in the control flow analysis during program optimization, and intervals are used in interval analysis for global data flow analysis [Aho et al. 88]. Hence dominators and intervals can be obtained as a by-product of the compiling process.

Ottenstein and Ottenstein recommend the PDG (Program Dependence Graph) of a program to be a good candidate as an internal program representation format in a software development environment [Ottenstein and Ottenstein 84]. If the PDG is used as the internal program representation in a software development environment, decomposition into slices will be available and can be used as a debugging aid.

Decomposition into basis paths and circuits, trees, spanning trees, and prime subgraphs are not available as a product of some existing process.

4.7 Formalism Dependence

This section discusses whether each type of decomposition is applicable to one type of graph model or is generalizable to other types of graph models as well.

The number of basis circuits or paths in a graph is equal to the cyclomatic number or nullity of the graph. Since the cyclomatic number is a graph-theoretic concept, which represents the measure of connectedness or redundancy of a graph, decomposition into basis circuits or paths can be applicable to any graph as a measure of complexity of the graph. Thus, decomposition into basis paths or circuits can be generalizable to different representations or graph models of a program.

The dominator tree for a control flow graph of a program was defined in Section 3.2. The DDG (Data Dependency Graph) of a program is usually not a single-entry/single-exit graph, and even if a single entry node and an exit node are introduced artificially, in almost all the cases the single entry node will be the immediate dominator of all the nodes of the graph. A similar argument is applicable to the PDG (Program Dependency Graph) of a program. Hence the concept of dominance or the decomposition into dominators is meaningful when applied to control flow graphs only.

Using an argument similar to the one presented above for the applicability of dominators to a DDG, each interval with a header h will contain only h as a member in most cases. Hence, decomposition into intervals is not meaningful for a DDG. A similar argument is applicable to the PDG of a program. Hence, decomposition into intervals is meaningful when applied to control flow graphs only.

The number of spanning trees in a graph can be considered as a measure of the complexity of the graph [Temperly 81]. Therefore, the decomposition of a graph into spanning trees makes sense for any graph model. Hence, decomposition into spanning trees is not only applicable to DDGs and PDGs, but it can also be generalized to any graph model.

Decomposition into a tree structure (i.e., relative nesting) is defined only for d-graphs (a graph which contains single-entry/single-exit subgraphs). Since DDGs and PDGs are not d-graphs, decomposition into a tree structure is not applicable to DDGs or PDGs. It is applicable only to control flow graphs of structured programs.

Prime subgraph decomposition involves decomposition of a flow graph into single-entry/single-exit subgraphs. The DDG of a program is not necessarily a single-entry/single-exit graph. All the nodes of the graph except the exit nodes will typically have outdegrees greater than one, and all the nodes except the entry nodes will have indegrees greater than one. Hence, it is quite unlikely to find a single-entry/single-exit subgraph in a DDG. Thus, the concept of prime subgraph decomposition of a DDG is not very meaningful. A similar argument is applicable to the PDG of a program.

4.8 Scalability

In this section, the effect of scalability on different decomposition techniques is discussed. Scalability in this context refers to the effect of the size of the input graph (or the program) on the decomposition process.

The flow graphs of computer programs are generally sparse because 1) binary branching is often used for flow of control, 2) programmers use sparse control flow structures for conceptual simplicity, and 3) most programs are written to solve real problems and hence do not generate complicated flow graphs [Hecht 77]. So O(e) can be assumed to be equal to O(n), where e and n are the number of edges and nodes of the flow graph of a program.

The complexity to calculate the fundamental circuits of a flow graph is $O(n^2)$ (since a flow graph can be considered to be sparse), where n is the number of nodes in the flow graph. Therefore, decomposition into circuits or paths seems to be sensitive to a change of scale of the flow graph (or the corresponding program).

Lengauer and Tarjan give a fast algorithm, whose time complexity is O(e logn), to calculate dominators in a graph, where e and n are the number of edges and nodes in the graph [Lengauer and Tarjan 79]. Therefore, decomposition into dominators also seems to be sensitive to the change in the size of the flow graph.

Since the cost of finding intervals in a graph is O(e), where e is the number of edges in the graph, it can be concluded that decomposition into intervals is relatively insensitive to a change in the size of the flow graph of a program.

The complexity of finding all the spanning trees of a graph is $t(e\sqrt{2})^{-n}$, where n is the number of nodes in the graph, **e** is a constant ($\approx$ 2.718), and t the number of spanning trees in the graph. As the number of nodes in the graph increases, the time complexity to find each spanning tree decreases exponentially but the total number of spanning trees increases in order of n^n. Hence, it can be concluded that an increase in the size of the flow graph is not advantageous to the decomposition of the flow graph into spanning trees as far as the computational cost is concerned.

The cost of finding the prime subgraphs of the flow graph of a program using Forman's approach is linear in time (i.e., O(n+e), where n and e are the number of nodes and edges of the flow graph). Hence, decomposition into prime subgraphs is not very sensitive to a change in the size of the input flow graph.

The computational cost of finding a slice based on Weiser's approach is O(n e loge), where n and e are the number of nodes and edges of the flow graph (in which each node

represents a statement in the program) [Weiser 84]. Therefore, it can be concluded that the decomposition of a program into slices, based on Weiser's approach, is sensitive to a change in the scale of the program. Provided the PDG of a program is already constructed, a slice based on Ottenstein and Ottenstein's approach can be obtained in linear time [Ottenstein and Ottenstein 84]. Hence, the decomposition of a program into slices, based on Ottenstein and Ottenstein's approach, is not very sensitive to an increase in the size of the program.

In the DDG (Data Dependency Graph) and PDG (Program Dependence Graph) models of a program, each assignment statement represents a node, and edges are introduced for data and control dependencies. As the size of a program increases, the sizes of DDG and PDG also increase rapidly. Ottenstein and Ottenstein comment that the space requirements of the PDG of a program is approximately three times that of the corresponding representation which represents each block as a DAG (Directed Acyclic Graph) [Ottenstein and Ottenstein 84]. Thus, it can be concluded that the sizes of the DDG and PDG of a program are sensitive to a change of the scale of the flow graph or of the program.

4.9 Interdependence

In this section, the interdependence among different decomposition techniques is discussed. The issue here is whether each decomposition technique depends on any other decomposition technique or is independent of other techniques.

The algorithm to construct the basis circuits of a graph starts with a spanning tree of

the graph [Gibbons 85]. Based on each spanning tree of a graph, one set of fundamental circuits of the graph can be constructed. The number of different sets of fundamental circuits in a graph is equal to the number of spanning trees of the graph. Hence, it can be argued that the decomposition into spanning trees and basis circuits or paths are dependent on each other.

Prime decomposition, using the Fenton-Whitty scheme, makes use of the concept of postdominators in building the prime decomposition tree [Alvey 88]. Therefore, decomposition into prime subgraphs depends on the decomposition into dominators.

The postdominator tree of the flow graph of a program is used during the construction of control edges in the PDG (Program Dependence Graph) of the program [Ferrante et al. 87]. Thus, program slicing based on Ottenstein and Ottenstein's approach is dependent on the decomposition into dominators, because program slicing due to Ottenstein and Ottenstein is based on the PDG of a program.

The other decomposition techniques, except the ones mentioned above, do not seem to depend on any other decomposition technique.

4.10 Automatibility

The automatibility of the different decompositions techniques are discussed in this section. The various decomposition techniques need some graph model as input, based on the particular decomposition considered. Hence, the automatibility of the construction of the three graph models (i.e., control flow graph, DDG, and PDG) is discussed first,

followed by the possibility of automating the various decomposition techniques.

The control flow graph of a program is used in flow analysis [Hecht 77] and can be obtained automatically given the program as input [Aho and Ullman 73]. Bieman and Edwards give an algorithm to find the DDG of a program, given the flow graph as input. They show that the DDG can be constructed automatically [Bieman and Edwards 85]. Ottenstein and Ottenstein recommend the PDG as a good candidate for intermediate program representation in a software development environment [Ottenstein and Ottenstein 84] and hence, the PDG can be constructed automatically, given the program as input.

Each set of basis circuits of a graph is based on a spanning tree of the graph. The spanning trees of a graph can be obtained automatically using depth first search or breadth search. The cost of finding the basis circuits of a flow graph is $O(n^2)$ (since the cost of finding the basis circuits for a sparse graph is $O(n^2)$ and the control flow graph of a program is usually sparse), where n is the number of nodes in the flow graph. Hence, it is feasible to decompose the flow graph of a program into basis circuits automatically.

Dominators and intervals are used in compilers during program optimization. Thus, decomposition into dominators and decomposition into intervals can be done automatically, given the flow graph of a program as input.

Chase coded the algorithm that he labels "new" for finding all the spanning trees of a graph [Chase 74]. The time complexity of the algorithm is $t(e\sqrt{2})^{-n}$, where t is the number of trees of the graph, **e** is a constant (≈ 2.718), and n is the number of nodes in

the graph. As n increases, the cost of finding one tree decreases exponentially but the number of trees increase in the order of n^n. Hence, we can conclude that it is feasible to automate the decomposition into spanning trees of the flow graph of a program in the case of flow graphs with relatively small number of nodes.

The prime subprogram decomposition, according to the Fenton-Whitty scheme, has been automated and the corresponding program has been embedded within the QUALMS software, which has been developed as part of the ALVEY project SE69 [Alvey 88]. Given the flow graph of a program, QUALMS displays the prime decomposition tree of the program and metricates it with a metric of the user's choice [Alvey 88]. Thus, the decomposition of the flow graph of a program into prime subgraphs, according to the Fenton-Whitty scheme, can be done automatically.

The decomposition of the flow graph of a program into prime subgraphs, using Forman's approach, uses the algorithm to find the triconnected components of a graph first. Hopcroft and Tarjan gave an algorithm to divide a graph into triconnected components in O(n+e) time, where e and e are the number of nodes and edges of the graph [Hopcroft and Tarjan 73]. They comment that the algorithm is efficient in practice and hence, it can be concluded that it is possible to automate the decomposition of a flow graph into prime subgraphs using Forman's approach.

Many tools have been devised that automatically find the slices of a program (e.g., see [Nanja and Samadzadeh 90], [Samadzadeh and Hsiao 93], and [Samadzadeh and Wichaipanitch 93]). Hence, the decomposition of a program into slices can be done automatically.

4.11 Uniqueness

This section discusses the uniqueness of the results of the various decompositions of the graph models of a program, i.e., it discusses whether the decomposition techniques yield a unique set of decomposed units.

The set of the basis circuits or paths of the flow graph of a program is not unique, since the set of basis circuits depends on the particular spanning tree of the graph being considered. The number of different basis paths or circuits is unique, but not the basis circuits or paths themselves.

From the definition of a dominator, it can be inferred that the dominator tree of the flow graph of a program is unique, i.e., the decomposition into dominators uniquely yields the dominator tree.

Allen gave an algorithm to obtain the intervals of the flow graph of a program uniquely, irrespective of the order in which the header nodes are selected. Hence, the set of intervals of the flow graph of a program is unique [Allen 70].

The decomposition of the flow graph of a program into spanning trees involves obtaining all the spanning trees of the graph. Hence, obviously, the decomposition obtains the set of all spanning trees of the graph, which is unique.

There is only one tree structure (relative nesting) that exists corresponding to a d-

graph of a program. Hence, the decomposition of the flow graph of a program into trees uniquely gives rise to a single tree structure.

In the decomposition into prime subgraphs using Forman's approach, the decomposition algorithm uses the triconnected components to build the prime decomposition tree [Tarjan and Valdes 80]. Hopcroft and Tarjan show that the triconnected components of a graph are unique [Hopcroft and Tarjan 73]. Hence, the decomposition of the flow graph of a program into prime subgraphs yields unique prime subgraphs. Fenton and Kaposi have proved that every flow graph has a unique decomposition into a hierarchy of primes. Hence, the decomposition into prime subgraphs according to the Fenton-Whitty scheme is also unique [Alvey 88].

4.12 Cover or Partition

In this section, the question of whether the decomposed units of different decomposition techniques form a partition or a cover for the original graph model, is discussed. Obviously, any partition is a cover. However, here the question is whether the units of decomposition overlap or not. The decomposed units can form a cover or partition for the nodes and edges of the graph, or only for the nodes of the graph.

McCabe shows that every circuit or path of the flow graph F of a program can be expressed as a linear combination of the fundamental circuits or paths of F [McCabe 76]. Hence, the union of the basis circuits or paths of the flow graph gives back the original flow graph, but the fundamental circuits or paths of the flow graph need not be disjoint. Thus, the set of basis circuits or paths of a flow graph of a program form a cover for the

flow graph (i.e., the set covers both the nodes and edges of the flow graph).

The decomposition into dominators and the decomposition into a tree structure do not involve subgraphs of the graph models and hence, the concept of partition or cover does not apply to these two decomposition techniques.

Allen shows that the set of intervals $I(h_1)$, $I(h_2)$, ..., $I(h_n)$ (where n is the number of intervals in the flow graph) of a flow graph F forms a partition for the nodes of F [Allen 70]. For all $i \neq j$, $I(h_i) \cap I(h_j) = \phi$, i.e., the intervals are disjoint and $U_i I(h_i) = F$. But the edges corresponding to the nodes belonging to the intervals need not form a cover or a partition for the flow graph. Thus, the set of intervals partitions only the nodes of the flow graph of a program.

Each spanning tree of a graph covers the original graph, i.e., it contains all the nodes of the graph (by the definition of a spanning tree). Hence, the set of all spanning trees of a graph can be considered as a cover for the nodes of the original graph. But the set of spanning trees of a graph need not cover the edges of the graph, as can be inferred from Figure 26.

The decomposition of a flow graph into prime subflow graphs, according to the Fenton-Whitty scheme, involves edge-disjoint subflow graphs [Fenton and Whitty 86]. Hence, the prime program decomposition forms a partition for the edges of the original flow graph. But the subflow graphs are not necessarily node-disjoint and therefore, the decomposition forms a cover for the nodes of the original flow graph.

The decomposition of the flow graph of a program into prime subgraphs, using Forman's approach, yields prime subgraphs with extra edges called "virtual edges", which are introduced in pairs during the decomposition process [Tarjan and Valdes 80]. The original flow graph can be reconstructed by "gluing" the components together along the virtual edges. Thus, the triconnected components (i.e., the prime subgraphs), excluding the virtual edges, form a partition for the edges of the original flow graph and a cover for the nodes of the flow graph; because the subgraphs are not node disjoint [Tarjan and Valdes 80].

CHAPTER V

SUMMARY AND FUTURE WORK

The main purpose of this thesis was to model software comprehension by the decomposition of an abstract representation of a program as a graph model. Three graph models of a program, namely, control flow graph, data dependency graph, and program dependence graph, were investigated. Seven different decompositions of one or more of these graph models were discussed. Finally, the different decomposition techniques were compared and analyzed based on a number of criteria that were identified specifically for the purpose of this comparison.

The major contributions of this thesis are: a) application of the notion of decomposition or chunking to various graph representations of a program as a model of software comprehension, and b) identification of the criteria in order to compare and contrast various graph decomposition techniques. It should be noted that it was not an objective of this work to identify the "best" or "worst" decomposition technique. Because of the various graph models involved and reasons for decomposition, the existence of an optimal decomposition technique is in general application-specific.

Possible future work to extend and utilize the work done in this thesis includes the following. Performing an empirical study by conducting experiments under a controlled environment on the different decomposition techniques. In such an experiment, a group of users (subjects) can be asked to use the different decomposition techniques, and studies

can be done to find out how easy it is to use each decomposition technique and the level of intuitiveness of each technique. Empirical studies can also be carried out to validate the work of this thesis as to the reusability of the decomposed units, scalability, and so on. Also, studies can be used to determine which of the decomposition techniques are helpful in aiding the comprehension of software.

Other possible future work to utilize the work done in this thesis includes developing an interactive graph decomposition tool which can display a desired decomposition of a given graph. The graph can be either the input to the tool or constructed by the tool.

REFERENCES

[Aho et al. 88] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, Reading, MA, 1988.

[Aho and Ullman 73] A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation and Compiling, Vol. II: Compiling*, Prentice-Hall, Englewood Cliffs, NJ, 1973.

[Allen 70] F. E. Allen, "Control Flow Analysis," *ACM SIGPLAN Notices*, Vol. 5, No. 7, pp. 1-19, July 1970.

[Alvey 88] Edited by: J. J. Elliot, N. E. Fenton, S. Linkman, G. Markham, and R. Whitty, "Structured-Based Software Measurement," *Alvey Project SE/069*, 1988, Department of Electrical Engineering, South Bank, Polytechnic, Borough Road, London, SE1 OAA, UK.

[Bieman and Edwards 85] J. M. Bieman and W. R. Edwards, Jr., "Modeling and Measuring Software Data Dependency Complexity," *Technical Report*, TR#85-18, Department of Computer Science, Iowa State University, Ames, IO, 1985.

[Cantone et al. 83] G. Cantone, A. Cimitile, and L. Sansone, "Complexity in Program Schemes: The Characteristic Polynomial," *ACM SIGPLAN Notices*, Vol. 18, No. 3, pp. 22-31, March 1983.

[Chase 74] S. M. Chase, Analysis of Algorithms for Finding All Spanning Trees of a Graph, Ph.D. Dissertation, Computer Science Department, University of Illinois at Urbana-Champaign, IL, 1974.

[Cocke 70] J. Cocke, "Global Common Subexpression Elimination," *ACM SIGPLAN Notices*, Vol. 5, No. 7, pp. 20-24, July 1970.

[Even 79] S. Even, *Graph Algorithms*, Computer Science Press, Rockville, MD, 1979.

[Fairly 85] R. E. Fairly, *Software Engineering Concepts*, McGraw Hill Book Company, New York, NY, 1985.

[Fenton and Whitty 86] N. E. Fenton and R. Whitty, "Axiomatic Approach to Software Metrication Through Program Decomposition," *The Computer Journal*, Vol. 29, No. 4, pp. 330-339, 1986.

[Ferrante et al. 87] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 3, pp. 319-349, July 1987.

[Forman 82] I. R. Forman, "Global Data Flow Analysis by Decomposition into Primes," *Proceedings of the Sixth International Conference on Software Engineering*, pp. 386-392, Tokyo, Japan, 1982.

[Ghezzi et al. 91] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Software Engineering*, Prentice-Hall, Englewood Cliffs, NJ, 1991.

[Gibbons 85] A. Gibbons, *Algorithmic Graph Theory*, Cambridge University Press, New York, NY, 1985.

[Hecht 77] M. S. Hecht, *Flow Analysis of Computer Programs*, Elsevier North-Holland, New York, NY, 1977.

[Hopcroft and Tarjan 73] J. E. Hopcroft and R. E. Tarjan, "Dividing a Graph into Triconnected Components," *SIAM J. Computing*, Vol. 2, No. 3, pp. 135-158, September 1973.

[Horwitz et al. 88] S. Horwitz, J. Prins, and T. Reps, "Integrating Non-Interfering Versions of Programs," *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pp. 133-145, San Diego, CA, 1988.

[Lakhotia 91] A. Lakhotia, Graph Theoretic Foundations of Program Slicing and Integration, CACS TR-91-5-5 (updated on March 16, 1992), The Center of Advanced Computer Studies, Lafayette, LA, March 1992.

[Lengauer and Tarjan 79] T. Lengauer and R. Tarjan, "A Fast Algorithm for Finding Dominators in a Flow graph," *ACM Transactions on Programming Languages and Systems*, Vol. 1, No. 1, pp. 121-141, July 1979.

[McCabe 76] T. J. McCabe, "A Complexity Measure," *IEEE Trans. on Software Eng.*, Vol. SE-2, No. 4, pp. 308-320, December 1976.

[Miller 56] G. A. Miller, "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," *The Psychological Review*, Vol. 63, pp. 81-97, 1956.

[Nanja and Samadzadeh 90] S. Nanja and M. H. Samadzadeh, "A Slicing/Dicing-Based Debugger for C," *Proceedings of the Eighth Annual Pacific Northwest Software Quality Conference*, Oregon Convention Center, Portland, OR, pp. 204-212, October 1990.

[Ottenstein 78] K. J. Ottenstein, Data-Flow Graphs as an Intermediate Program Form,

Ph.D. Dissertation, Computer Science Department, Purdue University, Lafayette, IN, August 1978.

[Ottenstein and Ottenstein 84] K. J. Ottenstein and L. M. Ottenstein, "The Program Dependence Graph in a Software Development Environment," *ACM SIGPLAN Notices*, Vol. 19, No. 5, pp. 177-184, May 1984.

[Regson and Samadzadeh 92] C. P. Regson and M. H. Samadzadeh, "Decomposition of Program Flow Graphs for Software Comprehension," *Proceedings of the Sixth Oklahoma Symposium on Artificial Intelligence*, pp. 97-105, Tulsa, OK, November 1992.

[Samadzadeh and Edwards 88] M. H. Samadzadeh and W. R. Edwards, Jr., "A Classification Model of Software Comprehension," *Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences (HICSS-21)*, Kailua-Kona, Hawaii, January 1988.

[Samadzadeh and Nandakumar 92] M. H. Samadzadeh and K. NandaKumar, "A Study of Software Metrics," *The Journal of Systems and Software*, Vol. 16, No. 3, pp. 229-234, November 1992.

[Samadzadeh and Hsiao 93] M. H. Samadzadeh and T. Hsiao, "An Interactive Parallel Program Slicer for the Hypercube," *Proceedings of the Twelfth Annual IEEE International Phoenix Conference on Computers and Communications (IPCCC'93)*, pp. 66-72, Tempe, AZ, March 1993.

[Samadzadeh and Wichaipanitch 93] M. H. Samadzadeh and W. Wichaipanitch, "An Interactive Debugging Tool for C Based on Dynamic Slicing and Dicing," *Proceedings of the Twenty-First Annual ACM Computer Science Conference (CSC'93)*, Edited by: Stan C. Kwasny and John F. Buck, pp. 30-37, Indianapolis, IN, February 1993.

[Schmidt 85a] M. Schmidt, "Ein Komplexitätsmaß basierend auf Entscheidüngen und Verschachtelugen," In: 13. Fachtagung Technische, Zuverlässigkeit., Nurnberg, VDE-Verlag GmbH Berlin, Offenbach, Mai 1985.

[Schmidt 85b] M. Schmidt, "A Complexity Measure Based on Selection and Nesting," *ACM SIGMETRICS - Performance Evaluation Review*, Vol. 13, No. 1, June 1985.

[Skvarcius and Robinson 86] R. Skvarcius and W. B. Robinson, *Discrete Mathematics with Computer Science Applications*, The Benjamin/Cummings Publishing Company, Inc., Mento Park, CA, 1986.

[Tamine 83] J. J. Tamine, "On the Use of Tree-Like Structures to Simplify Measures of Complexity," *ACM SIGPLAN Notices*, Vol. 18, No. 9, pp. 62-69, September 1983.

[Tarjan and Valdes 80] R. E. Tarjan and J. Valdes, "Prime Subprogram Parsing of a Program," *Proceedings of the Seventh Annual Symposium on the Principles of Programming Languages*, pp. 95-105, Las Vegas, NV, January 1980.

[Temperly 81] H. N. V. Temperly, *Graph Theory and Applications*, Ellis Horwood Limited, New York, NY, 1981.

[Weiser 81] M. Weiser, "Program Slicing," *Proceedings of the Fifth International Conference on Software Engineering*, pp. 439-449, San Diego, CA, March 1981.

[Weiser 82] M. Weiser, "Programmers Use Slices When Debugging," *Communications of the ACM*, Vol. 25, No. 7, pp. 446-452, July 1982.

[Weiser 84] M. Weiser, "Program Slicing," *IEEE Trans. on Software Eng.*, Vol. SE-10(4), pp. 352-357, July 1984.

[Yourdon 82] E. Yourdon, *Writings of the Revolution: Selected Readings on Software Engineering*, Yourdon Press, New York, NY, 1982.

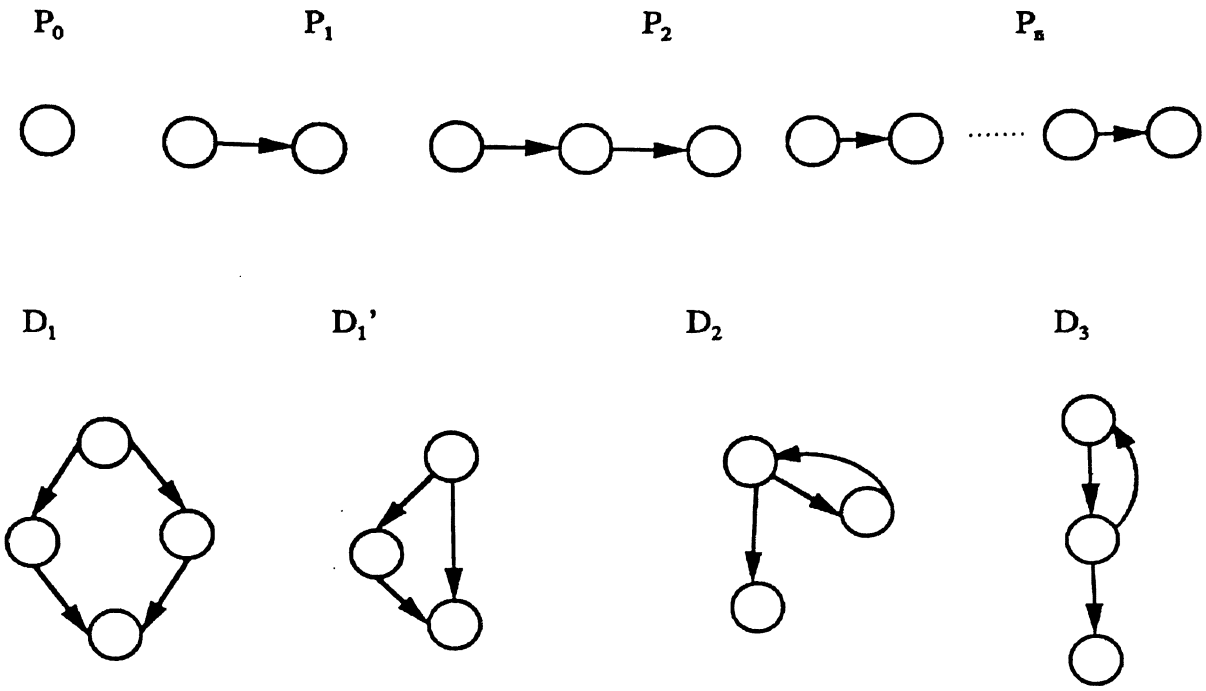
[Zuse 91] H. Zuse, *Software Complexity: Measures and Methods*, Walter de Gruyter, New York, NY, 1991.

APPENDIX

GLOSSARY

GLOSSARY

Common flow graphs. The following are some common flow graphs or constructs.



Circuit. A circuit is a path from a vertex to itself in which no edges in the path are repeated.

Cover. A set of subsets $\{B_i\}_{i \in I}$, where I is an indexing set, of a set B is said to cover B, if none of B_i's is empty and $\cup_i B_i = B$.

CSG. (A measure defined by Schmidt and Gong [Schmidt 85a] [Schmidt 85b]) For a flow graph F of a program, the measure CSG(F) = (cyclomatic complexity of F) + e, where $e = (e_1 + e_2 + ... + e_{|D|}) / |D|$, $0 \leq e \leq 1$ and $e_i = (1 - 1/d_i)$, with $1 \leq i \leq |D|$, d_i is the number of decision nodes plus 1 in the subgraph delineated by the decision node d_i and the postdominator of d_i which lies on a path to the exit node of F in the dominator tree of F, and D is the set of decision nodes in F.

DDG. Data Dependency Graph (see Section 2.2 for a definition).

d-graph. A d-graph program is composed of single-entry/single-exit subgraphs belonging to a well-defined finite set of types of single-entry/single-exit subgraphs. A subgraph of a d-graph is immediately contained in a (single-entry/single-exit) subgraph; a subgraph which contains only itself is a d-graph; conversely, a subgraph contains immediately primitive nodes of the d-graph and some subgraphs; a subgraph which does not contains

subgraphs contains only primitive nodes.

Decision Node. A node whose outdegree is greater than one.

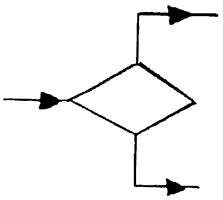
Digraph. A graph G = (V, E) consisting of a finite set V of elements called vertices and a finite set E of elements called edges; E is a relation on V.

Indegree Matrix. The indegree matrix D of a digraph G = (V, E), where v = {1, 2, ..., n}, is defined as

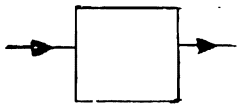
$$D(i,j) = \begin{cases} \text{in\_deg}(i), & \text{if } i = j \\ -k, & \text{if } i \neq j, \text{ where k is the number of edges in G from i to j} \end{cases}$$

k-connectedness. An undirected graph is k-connected if, for every pair of vertices v and w, there are k paths between v and w such that no vertex (except v and w) appears on more than one path. Biconnected means 2-connected and triconnected means 3-connected.

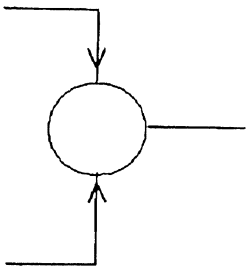
M-graph. A strongly connected graph which is composed of the following node types and contains a unique single-entry/single-exit node called an entry/exit node.



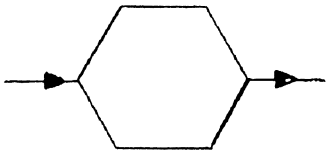
predicate node



assignment node



join node

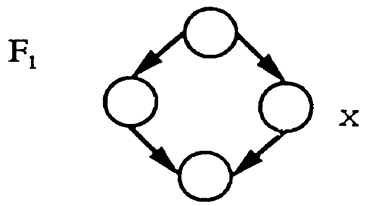


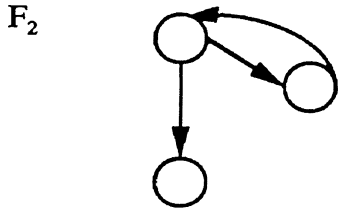
entry/exit node

Minimal Unreachable Subgraph. The subgraph G_1 of a graph G is unreachable from the rest of G, or simply G_1 is an unreachable subgraph, if $E \cap (\tilde{N}_1 \times N_1) = \phi$, where N is the set of nodes of G, E is the set of edges of G, N_1 is the set of nodes of G_1, and $\tilde{N}_1 = N - N_1$. G_1 is an unreachable subgraph of G with respect to a vertex set S, if $G_1 \cap G = G_1$, $G_1 \cup G = G$, and $S \subseteq N$. A graph is a minimal unreachable subgraph with respect to S, if

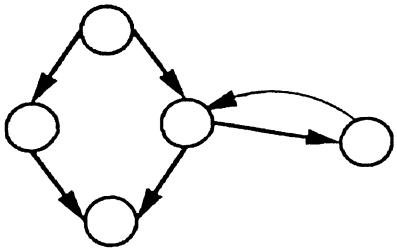
it is such a subgraph with a minimal number of nodes.

Nesting and Sequencing. Given a flow graph F_1 with a procedure node x, a second flow graph F_2 is nested on x, written $F_1(F_2$ on x), by deleting the unique arc leading out of x, identifying the stop node of F_2 with the node this arc led to in F_1 and identifying the start node of F_2 with x. Given two flow graphs F_1 and F_2, a flow graph that results from concatenating F_1 and F_2 is obtained by identifying the stop node of F_1 with the start node of F_2, and it is denoted by $seq(F_1,F_2)$. Examples of nesting and sequencing are given below.

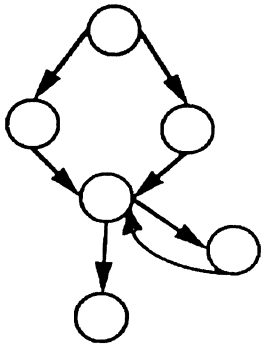




$F_1(F_2$ on x)



$seq(F_1,F_2)$



Partition. A set of subsets $\{B_i\}_{i\in I}$ of a set B is said to be a partition of B, if none of the B_i's is empty, $\cup_i B_i = B$, and $B_i \cap B_j = \phi$, for $i \neq j$ and i, j = 1, 2 ,....

Path. A path is a sequence of edges $e_1, e_2, \ldots, e_k$ such that the node at the head of each edge is the node at the tail of the next edge.

PDG. Program Dependence Graph (see Section 2.3 for a definition).

Postdominator. If x and y are two (not necessarily distinct) nodes in a flow graph F, then x is postdominator of y iff every path in F from y to the exit node contains x.

Reflexive Partial Ordering. A reflexive partial order on a set A is a relation R such that (1) R is transitive (aRb and bRc $\Rightarrow$ aRc, $\forall$ a,b,c $\in$ A) (2) R is reflexive (aRa, $\forall a \in$ A), and R is antisymmetric (aRb and bRa $\Rightarrow$ a = b, $\forall$ a,b $\in$ A).

RFG. Reducible Flow Graph (see Section 3.3 for a definition).

Software Complexity. The difficulty to maintain, change, and/or understand software.

Spanning Tree. A spanning tree of a graph G = (V, E) is a graph ST(G) = (V, E'), where E' is a subset of E. ST(G) is a tree that includes every node in G.

Strongly Connected Graph. A directed graph G is said to be strongly connected if, for every pair of vertices v and w, there exists a path from v to w and a path from w to v.

Tree. A graph G = (V, E) is called a tree if G is connected and acyclic.

Trivial Graph. A null graph ($V = E = \phi$) or an empty graph ($|V| = 1$ and $E = \phi$), or a connected graph with $|V| = 2$ and $|E| = 1$.

VITA

Charlotte P. Regson

Candidate for the Degree of

Master of Science

Thesis: PROGRAM FLOW GRAPH DECOMPOSITION AS A MODEL OF SOFTWARE COMPREHENSION

Major Field: Computer Science

Biographical:

Personal Data: Born in Vellore, India, April 9, 1958, daughter of Moses and Sophie Manickam. Married to Randolph Regson Raj on August 30, 1980.

Education: Graduated from Auxilium High School, Vellore, India, in May 1974; received Bachelor of Science degree in Mathematics from University of Madras, Madras, India, in May 1978; attended Stella Maris College, University of Madras, Madras, India, from 1978 - 1980. Received Bachelor of Science Degree in Computer Science from Oklahoma State University in May 1991. Completed requirements for the Master of Science degree at Oklahoma State University in July 1993.

Professional Experience: Research Assistant, Department of Electrical and Computer Engineering, Oklahoma State University, May 1991-August 1991, teaching Assistant, Computer Science Department, Oklahoma State University, September 1991-present.