INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.
2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.
3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of "sectioning" the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.
4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.
5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.

**University
Microfilms
International**
300 N. Zeeb Road
Ann Arbor, MI 48106

8416576

Zadeck, Frank Kenneth

**INCREMENTAL DATA FLOW ANALYSIS IN A STRUCTURED PROGRAM
EDITOR**

Rice University

PH.D. 1984

**University
Microfilms
International** 300 N. Zeeb Road, Ann Arbor, MI 48106

RICE UNIVERSITY

**Incremental Data Flow Analysis
in a
Structured Program Editor**

by

Frank Kenneth Zadeck

A Thesis Submitted
In Partial Fulfillment Of The
Requirements For The Degree

Doctor of Philosophy

Approved, Thesis Committee:

Kenneth W. Kennedy, Professor
Dept. of Mathematical Sciences
Chairman

Robert T. Hood, Assistant Professor
Dept. of Mathematical Sciences

J. Robert Jump, Professor
Dept. of Electrical Engineering

Houston, Texas

October, 1983

Incremental Data Flow Analysis
in a
Structured Program Editor

Frank Kenneth Zadeck

Abstract

Recent work in the area of software engineering has centered on the creation of sophisticated environments to enhance the development and maintenance of computer programs. One goal of this effort is to provide the programmer with information about the program early in the development cycle. With this information, the programmer can then correct design deficiencies and catch small programming errors.

One type of analysis that can be provided in these environments is *data flow analysis*. Data flow analysis can find many common programming errors such as uninitialized variables and calculated values that are never used. Additionally, this analysis can be used to maintain a data base that can aid the programmer in determining the effect that seemingly small changes may have on the entire program.

In an interactive programming environment, small changes are made to the program. Ideally, the data flow information for the problem should be updated as each small change is made. Traditionally, data flow techniques analyze the entire program for all variables at once, though the analysis can be performed for each variable separately. Updating the solution for each variable separately is a reasonable technique in an

interactive environment since many source changes are small and local.

By performing the analysis for each variable separately, the program flow graph can be modified in a manner that is unique for each variable. This transformation leads to the development of an analysis technique that is both linear in the size of the program and immune to the adverse effects that arise from flow graph irreducibility or deep loop nesting. The class of problems amenable to this technique is restrictive but does include problems as complex as the *Live Analysis* and *Reaching Definitions* problems. The output of the latter problem can then be used to build *Use-Defchains*.

In the incremental form of this algorithm, the solution can be updated for most changes by visiting only the part of the program flow graph in which the solution actually changes. For other types of changes, a larger area of the program flow graph must be visited, but this area is typically smaller than the entire program flow graph.

Acknowledgments

There are five groups that I would like to thank for their aid in my graduate work and the preparation of this dissertation.

My committee, Ken Kennedy, Bob Hood, Bob Jump, and Mike Donegon for their aid and advise.

Keith Cooper and Alan Demers whose careful understanding of my algorithms aided greatly in their development.

The National Science Foundation for their financial support through grant 81-04006.

Bill LeFebvre, Kim Taylor, and the people at Xerox for making the laser printer available and workable.

My family, for their long and continual support throughout my years as a student. (My mother always wanted a doctor for a son.)

Table of Contents

1	Introduction	1
2	Data Flow Analysis	8
2.1	The Data Flow Problem	8
2.2	Introduction to Lattice Theory	9
2.3	Lattice Theory Model for Data Flow Analysis	10
2.4	Introduction to Graph Theory	11
2.5	Graph Theory Model for Data Flow Analysis	11
2.6	Data Flow Problems	12
2.6.1	Forward Versus Backward Problems	12
2.6.2	Or Versus And Formulations	13
2.6.3	Clustered Versus Single Bit Problems	14
2.6.4	Rapid and Fast Problems	15
2.7	Data Flow Techniques	15
3	Data Flow Techniques to Enhance Software Reliability	18
3.1	The Limits of Data Flow Analysis	19
3.2	Anomalies Discovered by Dave	21
3.3	Other Data Flow Based Programming Environments	23
4	Partitioned Data Flow Analysis	24
4.1	The Global Algorithm - First Version	25
4.2	The Global Algorithm - Second Version	39
4.3	Correctness of the Partitioned Variable Technique	42
5	Partitioned Formulations of Common Problems	45
5.1	Data Structures	46
5.1.1	Input Structures	46
5.1.2	Output Structures	47
5.2	The Data Flow Problems	47
5.3	Problems not Solvable by PVT	49
6	Recent Work in Incremental Data Flow Analysis	52
6.1	Kou	52
6.2	Rosen	53

6.3	Babich and Jazayeri	54
6.4	Ryder	55
6.5	Wegman	56
7	Incremental Algorithms	57
7.1	Special Local Algorithms	58
7.2	Algorithms For Type I Changes	60
7.2.1	Addition of a Use	62
7.2.2	Deletion of a Use	63
7.2.3	Addition of a Def	63
7.2.4	Deletion of a Def	66
7.2.5	User Queries	67
7.2.6	Other Type I Operations	67
7.3	Algorithms For Type II Changes	68
7.3.1	Addition of an Edge	69
7.3.2	Deletion of an Edge	71
7.4	Algorithms for Type III Changes	73
7.4.1	Serial Splitting of a Block	74
7.4.2	Serial Combining of a Block	76
7.4.3	Remaining Operations	76
8	Techniques to Improve Performance of PVT	78
8.1	Comparison of PVT with Classical Techniques	78
8.1.1	Comparison of Single Variable Operations	78
8.1.2	Comparison of Multiple Variable Operations	79
8.1.3	Other Advantages of PVT over Classical Methods	81
8.1.4	Disadvantages of the Partitioned Variable Technique	82
8.1.5	Asymptotic Complexities of Various Algorithms	82
8.2	Adding Knowledge About Program Structure	83
8.2.1	A Technique To Improve Use and Def Density	85
9	Conclusions	87

9.1	Areas for Future Research	87
9.1.1	Better Local Algorithms	87
9.1.2	Developing Type II Algorithms For Ryder's Work	88
9.1.3	The Display Issue	88
9.1.4	More Complex Analysis	91
9.1.5	Integration With Universal Data Flow Analysis Algorithms	92
9.2	Conclusions	92

Table of Figures

2.1	Lattice Formulations for <i>And</i> and <i>Or</i> Problems	13
3.1	Weakness of Data Flow Based Error Checking	20
3.2	Errant Loop	21
4.1	Original Program Flow Graph	27
4.2	Partitioning of Nodes At <i>S</i> sites	28
4.3	The Numbering of <i>B</i> sites	31
4.4	Location of Strongly Connected Regions	33
4.5	Reduction of Strongly Connected Regions	34
4.6	Propagation of Information by Depth First Search	36
4.7	Mapping Information Back	38
4.8	Algorithm for <i>SetInfo</i>	40
4.9	Algorithm for <i>SearchC</i>	41
4.10	Algorithm for <i>SearchD</i>	42
5.1	Formulations for Common Data Flow Problems	49
7.1	Algorithm for <i>SearchL</i>	59
7.2	Algorithm for <i>AddUse</i>	62
7.3	Algorithm for <i>DelUse</i>	63
7.4	Algorithm for <i>AddDef</i>	65
7.5	Algorithm for <i>DelDef</i>	66
7.6	Algorithm for <i>AddEdge</i>	70
7.7	Algorithm for <i>DelEdge</i>	72
7.8	Algorithm for <i>SerSplit</i>	75
7.9	Algorithm for <i>SerCombine</i>	76
8.1	Comparison of Data Flow Techniques	83
9.1	Editing Display Problem	89

Chapter 1

Introduction

In the early days of computing, a large staff of people was required to program, maintain, and run a computer. Excessive physical size, poor reliability, and lack of speed made early computers very expensive to operate. At that time, the programming costs were far less than the operational costs. As a consequence of this economic difference, the emphasis of software development was on making programs execute faster rather than on making computers easier to use.

Computer hardware has improved substantially in the last thirty years. The mean time between hardware failures has increased from a matter of minutes in the 1940's to weeks, and in some cases months, with today's machines. At the same time, the speed of computers has risen by orders of magnitude.¹ The cost of computers has dropped to the point where, for as little as one hundred dollars, a computer can be purchased that rivals in processing power and speed the largest machine available in the 1940's. While the cost of computing power has been falling, the cost of developing software has steadily risen. The reasons for this rise in cost are varied.

First, the salaries for professional workers have risen much faster than the general cost of living. This rise has been especially dramatic for computer programmers as the demand for programmers has been rising

¹There are several references that give good overviews of the development of computer hardware and software. One such source is section 1.3 of *The Structure of Computers and Computations*, by D. J. Kuck [Kuck 78].

faster than their supply.²

Second, a great deal of software written today is developed by specialists in science and engineering. These people are highly trained and therefore demand high salaries. The need for solutions to nontrivial problems makes this use of specialists necessary.

Third, the task of developing software has not been completely automated and is still very labor intensive. There remains a belief that simply making programming languages look like English will alleviate the need for programmers [Ehrm 79]. While it is true that the act of programming would be simpler if languages were more uniform and natural, there is much more to programming than dealing with syntax.³ There is still research being done on making programming constructs model human thought. The focus has been to study the way nonprogrammers attempt to solve programming-like problems and to modify programming languages to model these processes [BoSo 83].

Lastly, as computers have become more powerful, they have been expected to handle larger and more complex problems. Some large systems of programs, such as typical operating systems, contain hundreds of

²Magazines such as *Electronics*, *Computerworld*, and *Datamation* regularly have articles that lament the shortage of programmers and give estimates of the projected number of programmers needed and the salaries that these people can command.

³Every once in a while someone claims to have developed a programming mechanism so powerful that programmers are no longer necessary. These claims have never been borne out by close inspection. Chapter 1 of *The Psychology of Computer Programming: Essays on Software Engineering* by G. M. Weinberg [Wein 71] states that an excellent example of this type of claim was the development effort of the Cobol language. The primary justification for this effort was that because of Cobol's English-like syntax, executives would be able to understand and write programs with no help from programmers. Instead, we have developed a large body of programmers who can understand only Cobol.

thousands of lines of code, which is far too much for a single person to develop or understand. These large programs require a large support staff, not only for development, but for ongoing maintenance. In the second chapter of "The Mythical Man Month", Brooks [Broo 79] suggests that the amount of work needed to develop a program grows as an exponential, rather than linear, function of the number of lines of code. This exponential rise is largely due to the amount of programmer interaction required when many programmers are involved.

This steady increase in the cost of programming, coupled with a steady decrease in the cost of machine cycles, has historically been the motivation for new types of developmental software. As the ratio of software costs to hardware costs has risen, economic necessity has fueled an ongoing search for more sophisticated development tools. Throughout the short history of computing, as the cost of programming has risen compared to the cost of machine cycles, more cycles have been used to develop and support programming. We are now on the verge of an era where hardware can support more sophisticated software. This software, if well designed, should be able to slow the rise in the cost of developing software.

The continual change in the economics of computing has not gone unnoticed by either the computing industry or the academic community. The goal of lowering the cost of software has been approached in many different ways. It seems clear that this problem is so large and diverse that there is no comprehensive solution. There are, however, many partial solutions that can, in the proper combination, be expected to slow the rise in software development costs.

One trend in the software development area has been to develop techniques to detect errors.⁴ These techniques can be grouped into four very broad, general categories:

- (1) *Constructive testing* - The program is analyzed to mechanically generate test cases that test each executable path through the program [Huan 79].
- (2) *Destructive testing* - A program that is very close to being correct is mutated. The generated mutant programs are run on a set of test data. If the program is correct and the test data is complete, all mutants will produce incorrect results. [BDLS 80]
- (3) *Proving programs correct* - A mechanical process is used, though sometimes with human intervention, to prove that the text of the program matches a very high-level specification. If this can be done, the program is considered correct [BoMo 79].
- (4) *Identifying Anomalies* - The text of the program is checked to see if it matches certain criteria. Any section of the program that fails to match any of the criteria is flagged for later scrutiny.

In the past, the last of these categories has been the most successful. This is because fairly simple criteria can be used to find a large number of errors.⁵ Examples of this technique abound: lexicographical syntax checkers, semantic checkers, and analysis packages such as Lint [John 78] and Dave [FoOs 76].

⁴A distinction is made between *testing* and *debugging*. The goal of testing is to detect errors by discovering their effects. The goal of debugging is to search for their associated cause [ClRi 83].

⁵It can be argued that this success is the result of lower expectations for practitioners in this category.

A second trend in the area of software development has been to perform as much error detection as early as possible in the program development cycle. Recently, there has been great interest in the development of techniques for editing higher level representations of objects. In the area of programming, interest has centered on having the user directly manipulate statements and blocks of the program rather than dealing with the program as a simple string of text. The *Cornell Program Synthesizer* [TeRe 81], *Mentor* [DHKL 75], *Interlisp* [Tiet 77], *Gandalf* [HaNo 82], *Chill* [RuMo 81], and *Rⁿ* [HoKe 83] are all examples projects that utilize these concepts. This interest has been motivated largely by the changes in the economics of computing. The cost of CPU cycles has decreased to the point where it is very reasonable to consider doing things interactively which would have been considered prohibitively expensive only a few years ago.

One of the effects of this work on structured program editors has been a renewed interest in algorithms for *data flow analysis*. Data flow analysis is a tool that has been used to gather information about computer programs. This information has typically been used to modify the program to allow it to run faster [Hech 77] [MuJo 81].

Unfortunately, existing flow analysis algorithms are not well suited for an interactive environment. There has been some work done on converting existing batch algorithms into an incremental form [Rose 81] [Ryde 82] [Wegm 83]. The presumption has been made that modifying existing algorithms to maintain the same structures that the batch algorithms create is the most efficient mechanism for dealing with small changes in a

program. This work has met with limited success and is reviewed in chapter 6.

The ideas presented in this dissertation form the basis for a partial solution to the software development problem. Specifically, this dissertation develops much of the technology necessary to use data flow analysis to provide programmers with much information about the programs they are developing. This information can then be presented to the programmer while the program is being edited. This technique would allow a common class of programming errors to be detected and corrected early in the program development cycle. Early availability of this information should enhance programmer productivity.

The main result of this dissertation is the development of an algorithm for incremental global data flow analysis. This algorithm is asymptotically efficient in its global version and is shown to be suitable for collecting and maintaining information that is useful in the detection of several common programming errors. The work is presented in the following order:

Chapter 2 contains a characterization of existing data flow analysis techniques.

Chapter 3 contains a discussion of the type of information that can be found by the techniques discussed later in this dissertation. This chapter also contains an overview of other systems that have been used to enhance the reliability of software through the use of data flow analysis techniques.

Chapter 4 contains a global algorithm for data flow analysis that uses a non-standard approach. This algorithm is shown to be capable of solving a large and useful class of data flow problems in an efficient manner.

Chapter 5 contains formulations for several common data flow problems that can be solved by the techniques presented here.

Chapter 6 contains an overview of work that has been done in the area of incremental data flow analysis.

Chapter 7 contains algorithms that collect and update data flow information in an incremental manner. These algorithms also detect data flow anomalies in the program.

Chapter 8 contains a modification of the algorithms presented in chapters 4 and 7 to enhance the performance of those algorithms. Additionally, this chapter contains a comparison of the asymptotic complexities of most data flow algorithms.

Chapter 9 contains a list of other topics for future research and general conclusions.

Chapter 2

Data Flow Analysis

Data flow analysis has long been one of the fundamental tools used in compiler optimization. Information about the ways that variables are computed and used in computer programs is produced by these analysis techniques. This information can then be used to optimize code produced by compilers. The importance of compiler optimization has sustained continuing research into methods to more efficiently produce this information.

2.1. The Data Flow Problem

While there are major differences in each of the algorithms developed, there is also much similarity. There have been several survey works, notably Kennedy [Kenn 81]¹ and Hecht [Hech 77], that compare various features of the different data flow algorithms. Each of the algorithms attempts to solve essentially the same problem. The basic data flow problem is this:

Given a control flow structure, the object is to discover the nature of the data flow, i.e. which definitions of program quantities can affect which uses, within the program.

¹This paper, "A Survey of Data Flow Analysis Techniques", by K. W. Kennedy [Kenn 81] has been reprinted as the first chapter of *Program Flow Analysis* edited by S. S. Muchnick and N. D. Jones [MuJo 81].

2.2. Introduction to Lattice Theory

The data flow problem can be formalized by casting it in the structure of *lattice theory*. This section and the following section have been essentially taken from Kam and Ullman, [KaUl 76]. Some definitions have been modified or expanded for use here.

A *semilattice* L is a set L with a binary *meet* operation, $\wedge$, such that for all $a, b, c \in L$:

$$\begin{array}{ll}
 a \wedge a = a & \text{idempotent} \\
 a \wedge b = b \wedge a & \text{commutative} \\
 a \wedge (b \wedge c) = (a \wedge b) \wedge c & \text{associative} \\
 a \geq b \Leftrightarrow a \wedge b = b \\
 a > b \Leftrightarrow a \geq b \text{ and } a \neq b \\
 a \leq b \Leftrightarrow a \wedge b = a \\
 a < b \Leftrightarrow a \leq b \text{ and } a \neq b
 \end{array}$$

Lattices may have:

$$\begin{array}{ll}
 \perp \wedge a = \perp & \text{bottom element} \\
 \top \wedge a = \top & \text{top element}
 \end{array}$$

It is assumed here that all semilattices have a *bottom element*.

Given a semilattice, L , a sequence $a_1 \dots a_j$ of elements of L is said to be a *chain* if for $1 \leq i \leq j$ we have $a_i > a_{i+1}$.

L is said to be *bounded* if for each $a \in L$ there is a constant b such that any chain beginning with a has length at most b .

If L is bounded, then we can take the meet over countably infinite sets if we define:

$$\bigwedge_{a \in S} a \text{ where } S = \{a_1, a_2 \dots\} \text{ to be } \lim_{n \rightarrow \infty} \bigwedge_{1 \leq i \leq n} a_i$$

The fact that L is bounded assures us that the limit does exist.

2.3. Lattice Theory Model for Data Flow Analysis

Following [Kild 73] we treat data flow analysis problems as follows. We choose a semilattice L and attach to its elements a *meaning*, normally data which could reach a point in a flow graph. We associate with each node of the flow graph a function f from L to L which intuitively represents how data is transformed when control passes through the block of code represented by that node.

In what follows, we find it necessary to consider the set of all functions which could be associated with some node of the flow graph. That is, having selected a semilattice L and some intended meaning for lattice elements, the admissible functions are those which reflect the actions of straight line blocks of code on elements of L . We abstract the notation of such a set of functions in the following definition.

Given a bounded semilattice L , a set of functions F on L is said to be an *admissible set of functions for L* if and only if the following conditions are satisfied:

- (1) Each $f \in F$ distributes over meet, $\wedge$

$$(\forall x, y \in L)(\forall f \in F) [f(x \wedge y) = f(x) \wedge f(y)]$$

- (2) There exists an identity function i in F , such that,

$$(\forall x \in L)[i(x) = x]$$

- (3) F is closed under composition, i.e. $f, g \in F \Rightarrow f \circ g \in F$, where

$$(\forall f, g \in F)[f \circ g(x) = f(g(x))]$$

- (4) For each $x \in L$, there exists a finite subset $H \subseteq F$ such that

$$x = \bigwedge_{f \in H} f(\perp)$$

2.4. Introduction to Graph Theory

It is convenient to model programs and describe algorithms using the graph theory model.

A *Directed Graph*, $G = (N, E)$ has a set of *nodes* N and a set of *edges* E , where $E \subseteq N \times N$. In this dissertation, all graphs, and therefore all edges, are *directed*.

The term, *edge*, is synonymous with the term, *arc*. An *out-edge* of a node is an edge that emanates from that node. An *in-edge* of a node is an edge that terminates at that node.

A *path* $p = p_0 \dots p_k$, $k \geq 0$, is a sequence of nodes such that $\forall i \mid 0 \leq i \leq k-1, (p_i, p_{i+1}) \in E$. A path $p = p_0 \dots p_k$ is a *path of length k* from p_0 to p_k and is written as $p_0 \rightarrow p_k$.

$\forall (i, j) \mid (0 \leq i < j \leq k) p_i$ is a *predecessor* of p_j and p_j is a *successor* of p_i . If $j = i + 1$ p_i is an *immediate predecessor* of p_j and p_j is an *immediate successor* of p_i .

A *cycle* is a path $c = p_0, p_1, \dots, p_k$, where $p_0 = p_k$. A graph with no cycles is said to be *acyclic*. A *strongly connected component* of G is a set of nodes, V where $V \subseteq N: \forall (u, v) \in V \Leftrightarrow u \rightarrow v$ and $v \rightarrow u$.

2.5. Graph Theory Model for Data Flow Analysis

A graph that represents the control flow of a computer program is called a *program flow graph*. A directed graph $G = (N, E, n_0)$ represents the program.

N	is the list of nodes. Each node in this graph typically corresponds to a single basic block in the program ² .
E	is the list of edges. The edges represent the block to block transfers.
n_0	is the initial node in the graph. This node represents the entry point for the routine.

Two special notations for E will be used in this dissertation:

$Succ(x)$ is the set of arcs, leaving each node x , that travel in the direction of control flow.

²This is not strictly true. Alternative strategies are discussed in section 8.2.

$Pred(x)$ is the set of edges that is obtained if each edge in the *Succ* set is reversed. Each edge in *Pred* runs in the direction against control flow.

2.6. Data Flow Problems

Data flow problems have been formulated to solve a variety of problems in compiler optimization and program reliability. The problems can be divided using various criteria into many different categories. The categories can affect such quantities as:

- (1) The time and space required to solve the problem.
- (2) The precision of the information obtained.
- (3) Applicable algorithms.
- (4) Uniqueness of the solution.

2.6.1. Forward Versus Backward Problems

Data flow problems can be divided into categories depending on the direction that information must be propagated to produce the desired result.

- (1) *Forward flow problems* are those which, given a point in the program, ask what can happen before control reaches the point (i.e., what definitions can affect computations at that point).
- (2) *Backward flow problems* are those which, given a point in the program, ask what can happen after control leaves that point (i.e. what uses can be affected by computations at that point).

A very few algorithms, such as *T1-T2* analysis by Ullman [Ullm 73], have not been shown capable of solving backward problems.

2.6.2. Or Versus And Formulations

Data flow problems can be classified according to the type of information desired. All problems fall into one of the following two classes:

- (1) *And, Set Intersection, or Must* - This class of problem attempts to discover what information *must* reach a certain point in the program. The solution technique begins with the assumption that all information reaches a given node and lets the propagation process delete information as it is found to be false.
- (2) *Or, Set Union, or May* - This class problem attempts to discover what information *may* reach a certain point in the program. The solution technique begins with the assumption that no information reaches a given node, and lets the propagation process add information as it is found to be true.

In terms of the lattice theory model of data flow, *and* and *or* problems are characterized as in figure 2.1.

Lattice Operator	<i>And</i> Operator	<i>Or</i> Operator
$\wedge$	$\wedge$	$\vee$
$\supseteq$	$\supseteq$	$\subseteq$
$>$	$\supset$	$\subset$
$\equiv$	$\subseteq$	$\supseteq$
$<$	$\subset$	$\supset$
$\perp$	0	1
$\top$	1	0

Figure 2.1: Lattice Formulations for *And* and *Or* Problems

The *set intersection* class can be transformed into *set union* problems through the use of DeMorgan's laws.

2.6.3. Clustered Versus Single Bit Problems

Problems can be classified as to the amount of information that is found.

- (1) *Single Bit Formulation* - The goal of single bit problems is to discover *if* information is true or false at a given node in the program. In the *set union* class of problems, the goal is to prove *if* a given attribute of information exists along *any* of the paths that reach a given node. For the *set intersection* class of problems, the goal is to prove *if* the information exists along all paths that reach a given node. Since only a single attribute is being propagated, only a single bit is required for each node to store the information.
- (2) *Clustered Formulation* - The goal of problems formulated in this manner is to discover *where* information that reaches a given node comes from, i.e. the source node of the information being propagated. Each potential source node is given a unique position in a bit vector of information to be propagated.

There are several common data flow problems that are *clustered* and belong to the *set union* class of problems. It is difficult to imagine a useful *set intersection* problem that is clustered.

In this dissertation, an algorithm is presented to directly solve *clustered* problems. This algorithm produces a solution to this class of problems in time that is linear in the size of the flow graph. These algorithms do not require flow graph reducibility. A similar technique, capable of solving only the single-bit problems, has been presented by Kou [Kou 77] and Babich and Jazayeri [BaJa 78].

2.6.4. Rapid and Fast Problems

Graham and Wegman [GrWe 76] characterize a data flow problem as being *fast* if the problem satisfies the following criterion in addition to the admissible functions presented in section 2.3:

$$(\forall f \in F)(\forall x \in L)[f(f(x)) \geq f(x) \wedge x]$$

They present an algorithm to efficiently solve problems of this class in *almost* linear time.

Kam and Ullman [KaUl 76] characterize a data flow problem as being *rapid* if the problem satisfies the following criterion in addition to the admissible functions:

$$(\forall f, g \in F)(\forall x \in L)[f(g(\perp)) \geq g(\perp) \wedge f(x) \wedge x]$$

Kam and Ullman's problems can be solved efficiently by the iterative algorithm.

The rapid problems are a subset of fast problems.

2.7. Data Flow Techniques

The gathering of information to solve data flow problems is accomplished in two phases. The program is subdivided into basic blocks, possible block to block transfers are noted, and program loops are found. This phase is known as *control flow analysis*. Next the information about how uses and definitions relate to one another is gleaned in the *global data flow analysis* phase.

Classical techniques can be loosely characterized by the following paradigm:

- (1) Partition the program into small sections of code. These sections are normally single entry, single exit sections of assignment statements called *basic blocks*.
- (2) Collect local information about each basic block. This information consists of which variables are defined and which variables are used in the block. This information is collected using basically ad hoc techniques.
- (3) Build a graph of the program. The nodes of this graph typically correspond to the basic blocks. The edges represent the executable connections between the nodes.
- (4) Find an ordering for the nodes in the flow graph. This step is where most methods differ from one another. Depending on the data flow technique, some nodes may be listed more than once.³

In many algorithms this and the next steps are combined. The ordering used in this step is found by doing reductions on the flow graph. As sets of nodes are reduced, the information is propagated for that local area. The common feature of these algorithms is that a single ordering is found. Information in the form of single, wide bit vector is propagated using this ordering.

³Nearly always, a node may be visited more than once. For a *rapid* problem, as defined by Kam and Ullman [KaUl 76], the iterative algorithm halts in $d(G) + 3$ passes, where $d(G)$ is the maximum depth of loop nesting. If $d(G)$ is 1, this means at least four passes of the algorithm are required for some questions and assignments of local information. If the problem is not *rapid*, many more passes may be required.

- (5) Propagate the local information for each node to the other nodes in the flow graph. This propagation step is done for all variables in parallel. The propagation proceeds using the ordering established in the previous step. The propagation continues until the information stops changing⁴. For the problems addressed by this dissertation, the information to be propagated is represented by bit vectors.

This dissertation investigates an algorithm to produce data flow information using a different paradigm. Instead of developing a single propagation order to be used for the entire bit vector, an optimal propagation order for small *clusters* of bits is defined separately. The optimal ordering for each cluster is discovered during one pass and the information is propagated during a second pass. For each cluster, each pass requires $O(n + e)$ time, where n is the number and e is the number of edges in the graph. The idea of solving a separate data flow problem for each variable in the program was originally suggested by Kou [Kou 77] and Babich and Jazayeri [BaJa 78]. The algorithm they present solves only the live-dead problem and is not developed into a general algorithm. The advantages and disadvantages of the technique presented in this dissertation, the Partitioned Variable Technique, are discussed in section 8.1.

⁴Several techniques that establish the ordering for the nodes in the graph have the property that after a fixed number of passes the information will have reached a fixed point.

Chapter 3

Data Flow Techniques to Enhance Software Reliability

Fosdick and Osterweil developed a system called *Dave* [FoOs 76] to use data flow analysis for purposes other than optimization. This system is batch oriented and requires that the entire source code be reprocessed each time any of it changes. *Dave* produces a massive listing that details the results of the analysis.

The basic scheme of *Dave* is to use data flow techniques to process the entire program. The analysis performed attempts to detect paths of execution that may result in the program producing undefined or misleading results. Such paths are called *data flow anomalies*. If a data flow anomaly is found, the path of the anomaly is reconstructed and printed out.

In *Dave*, the *decision algorithm*, which detects anomalies, is inexpensive to execute by comparison to the *printing algorithm*, which determines where the anomalies occur. The choice of decision algorithms was made independently of the choice of printing algorithms. These decisions were based simply on time, space, ease of programming, and generality of algorithm.

The shortcomings of *Dave* are few and easy to understand. During the early 1970's when the study was undertaken, the inexpensive computational power was not available to consider developing an interactive system. Also, the study was made easier by using a proven data flow technique rather than developing an incremental technique necessary to make an online system reasonable.

One of the positive results of the study was the development of reasonable expectations as to what type of information data flow analysis could provide. These results should not be considered all inclusive. Several people have investigated other problems that can be addressed effectively by these techniques [Will 79] [Masi 80].

3.1. The Limits of Data Flow Analysis

Data flow analysis cannot determine that a program is correct, it can only indicate that the program may be incorrect. This distinction is significant. Data flow analysis is capable of determining that if the code is executed under a given set of conditions, the code will produce erroneous results. Data flow analysis cannot discover, in general, that the program will ever execute in that manner, nor can it prove that if it executes in another manner it will produce correct results. A similar problem occurs while detecting syntax errors in a normal compiler; a program that is syntactically correct will not necessarily run correctly, but one with syntax errors will definitely run incorrectly. The main purpose of providing the output of a data flow analysis phase to the user is not to prove the program correct, but to find as many mistakes in the program at the earliest possible point in the development cycle.

Two examples shown in figures 3.1 and 3.2 will illustrate these points. These examples have been motivated from the survey article [FoOs 76].

{ Assume that a is undefined on entry to this block. }

```

begin
(1)  if (  $b$  )
      then do
(2)     $y \leftarrow x^0$           {  $y \geq 0$  }
      end
      else do
(3)     $y \leftarrow x + 1$ 
(4)     $a \leftarrow y$           { define  $a$  }
      end

(5)  if (  $y < 0$  )
(6)  then print  $a$ 
end

```

Figure 3.1: Weakness of Data Flow Based Error Checking

Note that the path (1)(2)(5)(6) in figure 3.1 is impossible since (6) can only be executed if y is negative and (2) guarantees that y is non-negative. If that path were to be executed, the undefined value of a would be printed in (6). *Dave*, and probably any system based solely on data flow analysis, would report that this path exists and that an abnormality is present.¹ It is up to the programmer, in a system such as *Dave*, to prove that this path is not executable.

¹It is possible that symbolic interpretation could prove that this path is not executable. This could be very costly in terms of execution time. Systems have been proposed that address such problems by using a mixture of data flow analysis and symbolic interpretation. One such system has been proposed by Williams and is based on an extended version of the Pascal language [Will 79]. Another system has been proposed by Holley and Rosen [HoRo 80].

```
begin
  a ← 1
(1)  while ( a > 0 ) do
      b ← b + a
    end
end
```

Figure 3.2: Errant Loop

For a second example, consider loop termination. Loop (1) of figure 3.2 will never terminate because there are no statements inside the loop that can modify the termination condition, $a > 0$, of the loop. If the body of the loop does not contain an assignment to any variable that is in the expression for loop termination, the loop is clearly in error. But having such an assignment by no means guarantees termination.

3.2. Anomalies Discovered by Dave

The *Dave* system has a decision algorithm to discover data flow anomalies. This decision algorithm is broken into three phases. The first phase collects global² information and makes it available to the second phase. The second phase uses the information provided by the first phase to propagate the information along the call graph of the program. The output of this second phase is then used to increase the precision of the global information found by the first phase. This is accomplished by using the

²In the area of data flow analysis the term *local* means for a single basic block, *global* means single procedural, and *universal* means interprocedural. This use of nomenclature is unfortunate and misleading to the casual observer.

universal information as a starting point and recalculating the global information.

The types of anomalies found by *Dave* can be found by building *chains*. A single chain is a link from a node with one attribute to a node with another attribute. Building chains is the process of discovering all such links.

The types of chains used by *Dave* are:

- (1) *Def-Use* chains. A link is made from each definition site for a variable to every site where that variable may be used.
- (2) *UnDef-Use* chains. A link is made from each site where the value of a variable becomes undefined to the site where that variable may be used.

The actual anomalies found by *Dave* fall into two categories:

- (1) *Forward errors*. In this type of analysis, each *definition site* is checked to see that the value calculated at that point *may* actually be used. While it is not necessarily an error to have useless code, the existence of such code is indicative of many potential program errors. The *Def-Use* chains are used here.
- (2) *Backward errors*. In this type of analysis, each *use site* is checked to see that a value *must* be defined on each path that reaches it. In most languages, the use of undefined variables is a semantic error.³ The *UnDef-Use* chains are used here.

³Most compilers do little or nothing to aid the programmer in finding this type semantic error. An exception is the IBM PL/I optimizing compiler. When all optimization is enabled, the compiler warns the user of possible uninitialized variables [IBM 76].

This type of analysis can also make available data flow specific cross-reference tables. In these tables, information is available such as the location of all *Def* sites that produced values for each *Use* and the location of each *Use* site that can be reached by each *Def* site.

3.3. Other Data Flow Based Programming Environments

Masinter at Xerox has added several types of program checking to the Lisp based *Scope Programming Environment* [Masi 80]. In this environment, several types of errors, including type errors, can be detected. The intent of *Scope* is much different than that of *Dave*. The analysis in *Scope* is totally query driven. No information is available to the user unless the user specifically asks for it. This design decision was made for performance considerations. It was felt that maintaining the data flow information in an up-to-date manner would provide a user interface that was unacceptably slow. Therefore, no algorithmic research was done to produce an incremental algorithm of sufficient efficiency. Instead, the strategy of *lazy evaluation* was used to defer any overhead until the user actually requests the information.

The undesirable side effect of a lazy evaluation based system is that no checking is automatically done. The programmer must guess what harmful side effects he may have introduced by making a change. He must then ask *Scope* to check a particular condition. No secondary effects are found automatically. On the other hand, the user does have the ability to query an interactive cross reference facility of the same power as the *Dave* system.

Chapter 4

Partitioned Data Flow Analysis

The technique presented here does not solve all data flow problems. It does solve a limited subset of problems in time linear in the size of the problem, for general flow graphs. The data flow problems that can be solved by this technique are characterized as follows:

Let L be a semilattice for a data flow problem, and let M be a function space over L . We say that (L, M) is *cluster partitionable* if:

(1) $\exists$ semilattices $L_1 \dots L_n$ $n \geq 1$ | any $l \in L$ can be represented as a tuple $\langle l_1 \dots l_n \rangle$ where:

(a) $l_i \in L_i \forall 1 \leq i \leq n$

(b) $u = \langle u_1 \dots u_n \rangle, v = \langle v_1 \dots v_n \rangle \Rightarrow u \wedge v = \langle u_1 \wedge v_1 \dots u_n \wedge v_n \rangle$

(2) For each $f \in M \exists (f_1 \dots f_n) \mid \forall l = \langle l_1 \dots l_n \rangle \in L$

(a) $f(l) = \langle f_1(l_1) \dots f_n(l_n) \rangle$ where $f_i(l_i) \in L_i \forall 1 \leq i \leq n$

(b) each $f_i \forall 1 \leq i \leq n$ is of the form:

$f_i(v) = c \quad \text{or} \quad f_i(v) = v \wedge c \quad \text{where } c \text{ is a constant.}$

Such functions are called *simple meet linear*.

Rule (1) allows the data flow problem to be *partitioned* into a series of independent problems called *clusters*. The algorithm given here solves the data flow problem for a single cluster. Therefore, a separate pass of the algorithm is required for each cluster of the problem.

For most common data flow problems a separate cluster must be solved for each variable in the program. Therefore, this method is called the *Partitioned Variable Technique*, or *PVT* for short.

The global algorithm for the Partitioned Variable Technique will be presented here. A more concrete version is given later. The first

presentation will show the algorithm as a series of independent steps. Each of these steps may require a full pass through the flow graph. Each step in this first presentation is motivated and analyzed separately for asymptotic time bounds.

The next chapter describes several common data flow problems that can be solved using this algorithm. The chapter also contains examples of data flow problems that cannot be solved by this algorithm with some insights as to why this algorithm is inadequate for these problems.

4.1. The Global Algorithm - First Version

This algorithm must be executed once for each cluster v .

This data flow technique differs from other data flow techniques in that the flow graph is manipulated based on the information contained within each node.

In data flow graphs, the nodes typically correspond to the basic blocks of the program. These basic blocks contain a group of statements. The first step of the algorithm addresses the problems that result from the ordering of statements within the block.

There are three types of sites that can exist within each node.¹

B sites These sites *begin* information chains. Information is propagated from nodes that contain *B sites*. All nodes that contain *B sites* are called *B nodes*.

E sites These sites *end* information chains. Information is propagated to nodes that contain *E sites*. All nodes that contain *E sites* are called *E nodes*.

S sites These sites *stop* the propagation of information chain.

¹See the next chapter for examples.

All nodes that do not contain any *B sites*, *E sites*, or *S sites* are *P nodes*. *P nodes* propagate information without changing it.

In some problems, the *B sites* may also be *S sites*. When this occurs, the *B site* is defined to be logically before the *S site*. In some problems, the *E sites* may also be *S sites*. When this occurs, the *E site* is defined to be logically after the *S site*.

- (1) Produce a graph G_1 from $G(N, E)$.² In G_1 , every node N in G that contains an *S site* is split into two nodes. The first new replacing N contains all information that occurs before the first *S site* for node N . It also contains all of the in-edges that entered node N . The second new node replacing N contains all information that occurs after the last *S site* for node N in G . This new node also contains all of the out-edges that exited node N .

Within a block, all *E sites* that occur before the first *S site*, all *B sites* that occur after the last *S site*, and all *B sites* and *E sites* that occur after the first *S site* but before the last *S site* are to be ignored.³ The original flow graph is shown in figure 4.1. Figure 4.2 shows the graph with nodes split at the *S sites*.

²In this dissertation, the root of the flow graph is dealt with in a cavalier manner. Real flow graphs from programs are generally rooted. The root is typically the entry node to the routine. The algorithms presented in this dissertation will propagate the information to the entire graph regardless of the reachability of any node from the root. In data flow problems where this behavior is critical, a separate prepass over the graph should be made to produce a new graph that is rooted.

³These sites to be ignored will only affect chains that are local to a single basic block. These chains can be built and maintained by algorithms that solve the local data flow problem. This simplification allows the asymptotic complexity of this algorithm to be based on the size of the node set and edge set.

Figure 4.1: Original Program Flow Graph

Figure 4.2: Partitioning of Nodes At S sites

In this algorithm, information is propagated from *B sites* to *E sites*. This propagation proceeds *against* the edges of the flow graph.

From the point of view of the information chains for v , the graphs G and G_1 are equivalent. By definition, no information may pass through the *S site*.

The lattice model can account for this splitting of nodes. By viewing the operation at a node not as a single event but as a continuous series of events, we can model the process of building chains as a series of function calls as we pass through a node. In the area of the node that contains no *B sites*, *E sites*, or *S sites* the function is the identity function:

$$f(v) = v$$

B sites add information to what is being propagated through the node.

Whenever a *B site* is encountered, the function is:

$$f(v) = v \wedge c$$

S sites inhibit any information from being propagated through themselves. Whenever an *S site* is encountered, the function is:

$$f(v) = c$$

The propagation function for a node without any *S sites* is:

Let b be the set of *B sites* in node n .

$$f(v) = (\bigwedge_{i \in b} c_i) \wedge v$$

The propagation function for the first new node with an *S site* is:

Let b be the set of *B sites* in node n that occur before the first *S site* in node n .

$$f(v) = (\bigwedge_{i \in b} c_i)$$

The propagation function for the second new node with an *S site* is:

$$f(v) = v$$

This step takes no more than $O(N)$ time and can at most double the size of the graph. The order of visitation of nodes is irrelevant.

- (2) Assign a unique number to each *B site* of v . This number will be needed in steps (3) and (4). This step can be done in $O(N)$ time since the graph can be traversed in any convenient order. This is shown in figure 4.3.

Figure 4.3: The Numbering of B sites

- (3) Produce the graph G_2 from G_1 so that G_2 is a *directed acyclic graph*. The graph, G_1 is a *directed graph*. This can be accomplished by collapsing each *cycle* or *strongly connected component* into single nodes. These new single nodes of G_2 contain all of the in-arcs and out-arcs of each of the members of the cycle in G_1 . Figure 4.4 shows the graph with the strongly connected region identified. Figure 4.5 show the graph after that region has been collapsed to a single node.

The numbering of the *B sites* is used here so that each *B site* can be sorted out later. When a region is collapsed, all *B sites* in that region are inherited by the new node. The numbering still allows each *B site* to remain unique.

An algorithm developed by Tarjan [Tarj72] finds the strongly connected components of a directed graph. This algorithm works in $O(N + E)$ time. A version of this algorithm can also be found as Algorithm 5.4 on pages 189-195 of *The Design and Analysis of Computer Algorithms* [AHU174] by Aho, Hopcroft and Ullman. This latter version has been modified slightly for use in the next section.

B sites *E sites*

Figure 4.4: Location of Strongly Connected Regions

B sites *E sites*

Figure 4.5: Reduction of Strongly Connected Regions

This step is justified by noting that every *B site* that is reachable from any node in a cycle is reachable from every node in the cycle since execution may proceed around cycles in the graph an arbitrary number of times. No cycle contains an *S site* since all *S sites* were split in step (1).

The function at the new node is:

$$f(v) = (\bigwedge_{i \in old} c_i) \wedge v$$

This step is the key to the whole algorithm. It sets this method apart from all other methods for solving data flow problems. This step only works because we are solving a single cluster and can therefore modify the edges of the graph in a manner that is dependent on the content of the nodes of the graph.

- (4) Traverse the graph G_2 in depth first search order. Begin the depth first search at the *E sites*. Do not visit any node more than once. (Use a simple *Visited* field at each node to keep track of this.) Propagate the information by doing the meet operation as the depth first search returns. The nodes will be visited in *postorder*.⁴ This visitation scheme imposes the same ordering as topological sorting.

Figure 4.6 show the information after the propagation phase. The propagated data is shown in italic numbers.

⁴Visit all sons of *N* before visiting *N*.

Figure 4.6: Propagation of Information by Depth First Search

The function executed at each node is:

Let s be the set of immediate successors of node n .

$$v_n \leftarrow (\bigwedge_{i \in s} f_i(x_i)) \wedge f_n(x_n)$$

This operation takes time $O((N + E) * (\text{information evaluation time}))$.

- (5) Map the information back into the original flow graph. The bit vectors of all nodes in a collapsed region are all the same. This is done in figure 4.7.

None of the above steps requires more than $O((N + E) * (\text{information evaluation time}))$. The entire algorithm must be executed once for each cluster. This makes the overall worst case complexity of $O((N + E) * (\text{information evaluation time}))$.

B sites *E sites* *S sites*

Figure 4.7: Mapping Information Back

4.2. The Global Algorithm - Second Version

The second presentation will demonstrate the algorithm at a more concrete level. In this version, several of the steps of the previous presentation have been merged.

This version deals only with a subset of the program graph. The calculation of which nodes are in the subset is presented in the chapter 7. For the purposes of the presentation here, it should be assumed that the entire graph is being processed.

The *SetInfo* routine is the calling routine for *SearchC* and *SearchD*. This routine has seven parameters. These parameters define the data flow problem and the area of the graph to be visited.

<i>NodeList</i>	The list of nodes that may be searched.
<i>EdgeList</i>	The set of edges that connect the nodes. This list may be either the <i>Successor</i> set or the <i>Predecessor</i> set depending on whether the problem is a <i>backward problem</i> or a <i>forward problem</i> .
<i>B</i>	The set of nodes that contain <i>B sites</i> for the cluster.
<i>E</i>	The set of nodes that contain <i>E sites</i> for the cluster.
<i>S</i>	The set of nodes that contain <i>S sites</i> for the cluster.
<i>Info</i>	The information to be updated by this call.

Two data structures used in *SetInfo* must be explained here. For an explanation of *DFNumber* and *LowLink*, see pages 189-195 of [AHU1 75].

<i>Visited</i>	An instance of this variable is required for each node in the graph. The variable can have one of four possible values. The outside value is used to indicate that the node is outside of the area that is to be addressed by this call. ⁵ All nodes within the affected area are initialized to new . As the nodes are visited by the algorithm to reduce the strongly connected regions they are placed on a stack and given the onstack value.
----------------	---

⁵This is used when the algorithm is used in the incremental form. This is addressed in chapter 7.

As the nodes are popped off of the stack, they are given the value *old* so that they will not be visited again.

MapToNew This structure maps the node indices from the original program flow graph to the new directed acyclic graph.

```

procedure SetInfo(NodeList, EdgeList, B, E, S, Info)
  { Output - Info }

  Count  $\leftarrow$  1
  empty Stack
  Visited[ * ]  $\leftarrow$  outside
  NewEdgeList[ * ]  $\leftarrow$   $\emptyset$ 

  for each node in NodeList do
    Visited[ node ]  $\leftarrow$  new
  endfor

  { Each edge is visited exactly once by the following loop }
  for each node in  $E \cap \text{NodeList}$  do
    NewEdgeList[ node ]  $\leftarrow$  EdgeList[ node ]
    for each n in EdgeList[ node ] do
      if ( Visited[ n ] = new )
        then call SearchC( n )
      endif
    endfor
  endfor

  NewVisited[ * ]  $\leftarrow$  outside
  for each node in NodeList do
    NewVisited[ MapToNew[ node ] ]  $\leftarrow$  new
  endfor

  { Do the depth first search }
  for each node in  $E \cap \text{NodeList}$  do
    call SearchD( MapToNew[ node ] )
  endfor

  { Copy the newly calculated Info back. }
  { This step does the work of (5) in the first cut. }
  for each node in NodeList do
    Info[ node ]  $\leftarrow$  NewInfo[ MapToNew[ node ] ]
  endfor
end

```

Figure 4.8: Algorithm for *SetInfo*

```

{ This algorithm implements steps (1) and (3) of the previous presentation. }
{ Step (3), the numbering of B sites is done at some other level. }
procedure SearchC( node )
    DFNumber[ node ] ← Count
    Count ← Count + 1
    LowLink[ node ] ← DFNumber[ node ]
    push node on Stack
    Visited[ node ] ← onstack

    { This conditional replaces step (1) in the previous presentation. }
    { This conditional stops all searches past a S site. }
    if ( node ∉ S )
        then do
            for each s in EdgeList[ node ] ∩ NodeList do
                if ( Visited[ s ] = new )
                    then do { Edge has not been visited }
                        call SearchC( s ) { Recursive call }
                        LowLink[ node ] ← min ( LowLink[ node ], LowLink[ s ] )
                    enddo

                    else do { Node has already been visited }
                        if ( ( DFNumber[ s ] < DFNumber[ node ] )
                            and ( Visited[ s ] = onstack ) )
                            { Mark as member of same region. }
                            then LowLink[ node ] ←
                                min ( DFNumber[ s ], LowLink[ node ] )
                        enddo
                endfor
            enddo

    NewInfo[ node ] ← Info[ Node ]
    if ( LowLink[ node ] = DFNumber[ node ] )
        then do { Everything on the stack belongs to a single }
            repeat { strongly connected region }
                pop x from Stack
                Visited[ x ] ← old
                MapToNew[ DFNumber[ x ] ] ← DFNumber[ node ]
                NewEdgeList[ node ] ← NewEdgeList[ node ] ∪ EdgeList[ node ]
                if ( x ∈ B )
                    then NewInfo[ node ] ← NewInfo[ node ] ∧ x
                until ( x = node )
            enddo
    end

```

Figure 4.9: Algorithm for SearchC

SearchD propagates information from either the leaves or nodes that are reachable by one arc from a node in the *NodeList*.

```

{This algorithm implements step (4) of the previous presentation. }
procedure SearchD( node )

    NewVisited[ node ]  $\leftarrow$  old

    for each n in NewEdgeList[ node ] do
        Newn  $\leftarrow$  MapToNew[ n ]
        if (NewVisited[ Newn ] = new )
            call SearchD( Newn )

        if (NewVisited[ Newn ] = outside )
            { Look outside the affected area to find the info to propagate upward }
            then NewInfo[ node ]  $\leftarrow$  NewInfo[ node ]  $\wedge$  Info[ n ]

            { Look at the newly calculated son's value }
            else NewInfo[ node ]  $\leftarrow$  NewInfo[ node ]  $\wedge$  NewInfo[ Newn ]

    endfor
end

```

Figure 4.10: Algorithm for *SearchD*

List Version of Recursive Depth First Search Routine

4.3. Correctness of the Partitioned Variable Technique

The goal of any data flow analysis algorithm is to determine the *meet over all paths* solution for the problem. Kam and Ullman [KaUl 77] describe this solution as follows:

Given a gdfap,⁶ the *meet over all paths* (MOP) solution for a program can be interpreted informally as the calculation for each statement in the program of the maximum information which is true along every possible execution path from the starting point of the program to that particular statement.

The normal method of assessing correctness of a data flow technique is to

⁶Global data flow analysis problem.

prove that the technique calculates the *meet over all paths* solution to the problem if that solution exists. Kildall [Kild 73] has show that the MOP solution exists for all problems that are distributive⁷ and that this solution is equivalent to the *maximum fixed point* solution.

Theorem 1 proves that the Partitioned Variable Technique calculates the meet over all paths solution.

Theorem 1:

Propagation of information using a postorder traversal of a directed acyclic graph produces a meet over all paths (MOP) solution.

Base Case:

On the graph with one node, the information at that node satisfies the MOP criteria. Since there are no edges in this graph, the information at the node is correct.

Induction Step:

Assume that on all graphs of size $k - 1$, the algorithm correctly propagates the information such that each node's information satisfies the MOP criteria.

Take a graph of size k . The execution of the algorithm for the k th node does not affect the MOP solution for the nodes with postorder numbers less than k . The information at nodes with postorder numbers less than k cannot involve the k th node, since (1) the graph is visited in postorder and (2) the graph is acyclic and

⁷Rule 1 of the characterization of admissible functions given in section 2.3 requires that the data flow problem be distributive. Kam and Ullman [KaUl 77] show that unique MOP solutions do not necessarily exist for problems that are not distributive.

therefore has no backedges.

This the induction hypotheses holds for the execution of the algorithm on the first $k - 1$ postordered nodes of the graph produces correct results.

The execution of the algorithm for node k causes the information to be propagated to this node as the meet of the information at each of the k th nodes immediate successor nodes. This satisfies the MOP criteria since each of the immediate successor nodes satisfied that criteria.

Chapter 5

Partitioned Formulations of Common Problems

There are several intraprocedural problems that are formulated as a collection of equations and are implemented as bit vectors. All of these problems meet Kam and Ullman's criteria for admissible functions given in section 2.3 and their criteria for rapid problems given in section 2.6.4. Each of the problems given here is also a *clustered* problem as defined in section 2.6.3 and therefore must be solved by a technique that can propagate a bit vector for each variable. The information produced by a solution to each of these problems is used to build some type of information chain.¹

There are five problems presented in this chapter. Each problem has an analogue that is formulated as a *single bit* problem. All of the problems presented here belong to the *set union* class of problems. This is not the result of any restriction; there simply do not appear to be any *set intersection* problems that have meaningful *clustered* formulations. The problems presented here are:

- (1) Reaching Definitions.
- (2) Live Variables.
- (3) UnSafe Uses.
- (4) Forward Printing.
- (5) Backward Printing.

¹See chapter 3 for a description of information chains.

5.1. Data Structures

5.1.1. Input Structures

The following information is required as input to the data flow algorithms discussed here. All of the information is local to and must be constructed from a single basic block. It is assumed that some front end program builds these structures in batch environments or maintains these structures in interactive environments.

<i>Def</i> [<i>v</i>]	is a list of nodes for variable <i>v</i> , whose values are changed by the execution of this block. This includes targets of assignment statements and parameters to input statements.
<i>Mod</i> [<i>v</i>]	is a list of nodes for variable <i>v</i> , whose values may be changed by the execution of this block. Assignment into single values of arrays and variables passed to subroutines are two examples of statements that can generate <i>Mods</i> . ²
<i>UnDef</i> [<i>v</i>]	is a list of nodes for variable <i>v</i> , whose values become undefined by statements in the node <i>x</i> . Loop index variables in languages such as BCPL and Fortran become undefined in the exit node from the loop. Also, all local variables are initially undefined on entry to the subroutine and become undefined on exit from the subroutine. ³
<i>Use</i> [<i>v</i>]	is a list of nodes for variable <i>v</i> , that are referenced by statements in node <i>x</i> . This set is typically made from all variables that appear in any expression, in parameters to subroutines, and in output statements.

²Parameters to subroutines must be put into both the *Use* and *Mod* set unless other information is known about the use of that parameter within the subroutine.

³This is not true for static variables. In some implementations of Fortran, all variables are static and there is no need for an *UnDef* set at all.

5.1.2. Output Structures

The following information is produced by the algorithms discussed here.

$Defs[v, x]$	contains the list of definition sites for variable, v , that are reachable from node x . This set is calculated top-down and is the output of the <i>Reaching Definitions</i> problem.
$Mods[v, x]$	contains the list of modification sites for variable, v , that are reachable from node x . This set is calculated top-down and is the output of the <i>Reaching Definitions</i> problem.
$UnDefs[v, x]$	contains the list of undefinition sites for variable, v , that reach node x . This set is calculated top-down.
$Uses[v, x]$	contains the list of use sites for variable, v , that are reachable from node x . This set is calculated bottom-up and is the output of the <i>Live Variables</i> problem.

5.2. The Data Flow Problems

- (1) *Reaching Definitions* - Determine the set, $Defs$, for each node, x , of variable definitions that can enter node x . The term *variable definition* means a statement that can potentially modify the value of a variable. This is a *forward* problem and the output is typically used to build *Use-Definition Chains*⁴ which are used in compiler optimization [Hech 77].
- (2) *Live Variables* - Determine the set, $Uses$, for each node, x , of variables that are *live* or may be used after control passes out of node x . This problem is the *backward* analogue of the Reaching Definitions Problem and its output can also be used to build *Use-Definition Chains* [Hech 77].

⁴See section 3.2 for a description of chains.

In addition to being used in compiler optimization, the information produced by a solution of this problem can be directly used to discover if the output of a definition site is ever used. This has applicability in the area of software reliability since unused definitions may be the result of programming errors.

- (3) *Unsafe Uses* - Determine the set, *UnDefs*, for each node x of variable undefinitions that can reach node x . A use of a variable is *unsafe*, for each node x , if there exists a path that the program may take to node x , where the variable has not been defined. By characterizing the problem in this manner, it is not only possible to determine if the *use* is safe, but the location of the kill site is determined if the *use* is unsafe. This problem is associated with reliability of software [FoOs 76].
- (4) *Value Printing Problem* - This problem is broken into two subproblems, the *Forward Printing Problem* and the *Backward Printing Problem*. The output of these two problems is then processed to find the list of nodes that a particular value of a single variable is live within. These two problems were defined by Fosdick and Osterweil in the *DAVE* system. To date their only use has been in the area of reliability [FoOs 76].

These two problems have never been formulated for solution by a classical technique. The technique used by Fosdick and Osterweil is similar to the technique proposed by Kou. This technique was used because the number of *Use-Def* pairs that were needed was small compared to the number of possible *Use-Def* pairs in a given program.

In the *Forward Printing Problem*, a list of nodes for each *Def*, *Mod* or *UnDef* in the program is created. The list contains each node that is reachable from each beginning node without passing through another *Def*, *Mod* or *UnDef*.

In the *Backward Printing Problem*, a list of nodes for each *Use* in the program is created. The list contains each node that is reachable from each *Use* node without passing through another *Def*, *Mod* or *UnDef*.

Table 5.1 summarizes the formulations of these data flow problems into the partitioned framework.

Problem	Class	B sites	E sites	S sites
Reaching Definitions	<i>Pred</i>	$Def \cup Mod$	$Use \cup Mod$	$Def \cup UnDef$
Live Variables	<i>Succ</i>	$Use \cup Mod$	$Def \cup Mod$	$Def \cup UnDef$
UnSafe Uses	<i>Pred</i>	$UnDef$	$Use \cup Mod$	$Def \cup UnDef$
Forward Printing	<i>Pred</i>	$Def \cup Mod$	<i>Node</i>	$Def \cup UnDef$
Backward Printing	<i>Succ</i>	$Use \cup Mod$	<i>Node</i>	$Def \cup UnDef$

Figure 5.1: Formulations for Common Data Flow Problems

5.3. Problems not Solvable by PVT

As explained in the previous chapter, the PVT algorithm works by dividing the problem into small *clusters* that are solved separately. The key property of a cluster is that each *strongly connected component* of a cluster has the same information available for each node of that component. This allows the strongly connected regions to be reduced to single nodes, transforming the program flow graph into a directed acyclic graph. A data flow problem that is amenable to this technique is called a *partitionable problem*.

While it is true that the PVT algorithm can solve most of the *clustered* problems that are *rapid*, it should not be inferred by this that the PVT

algorithm is powerful enough to solve all problems in these classes. The following problem is *rapid* but cannot be solved by PVT:⁵

Suppose you wish to find the length of the shortest path from each node in a flow graph to every other node in the data flow graph. The formulation of this problem is as follows:

$\wedge = \text{min}$ for each element in the cluster.

There are no *P sites* or *S sites*. The operation at each *B site* is a *meet* with the constant vector:

$[\infty, \infty, \dots, 0, \dots, \infty]$

where the number in the *i*'th position is 0 for node *i*.

The formulation of this problem does not satisfy the definition of PVT since an additional operation must be done at the *P nodes*. The value of each path must be incremented by 1 to accumulate the path length. Intuitively, this problem cannot be solved by the PVT since information is lost when the strongly connected regions are collapsed and this information, the length of the shortest path, is utilized by this problem. This problem can be solved by the techniques of Kam and Ullman or Graham and Wegman.

As a second example consider a common data flow problem that is not *distributive* such as constant propagation. This problem is described by Hecht [Hech 77] as:

Try to discover at the top of each node in the flow graph, those variables that are assigned the same constant value on all paths from the initial node to the top of a given node.

This problem fails to be *fast* or *rapid* since it has the property that new information may be acquired each time propagation proceeds around any

⁵Kildall [Kild 73] describes a problem similar to this that uses the minimum path distance. Whenever all registers are busy and contain live expressions, the register containing the live expression with the largest distance to its occurrence is displaced.

loop. Additionally, this problem has another characteristic that cannot be handled by PVT, namely, it fails to be partitionable into clusters. In the constant propagation problem, whenever a constant is propagated to a new node, that propagation may make other constant propagations possible. There is no way to make the clusters independent of each other.

While there are many problems that the PVT cannot solve, the PVT does provide a mechanism to efficiently solve a rich set of data flow problems that are useful in many areas.

Chapter 6

Recent Work in Incremental Data Flow Analysis

The first paper reviewed here is the work by Kou. This work does not deal at all with incremental changes. The paper is included here because modifications to and extensions of his techniques are used heavily by the algorithms presented in this dissertation. The second paper, one by Rosen, sets a formal basis to evaluate incremental algorithms.

Of the five papers discussed here, three of them are significant works that directly address the subject of incremental global data flow analysis. The second and third papers do not deal with the whole incremental update problem. Their area of study is limited to non-structural program modifications. The fourth work covers all types of modifications but only over a small subset of program graphs.

6.1. Kou

Kou [Kou 77] developed an algorithm for solving the *single bit* version of the reaching definitions problem. His solution technique involves solving a graph problem for each variable in the program. By operating on each variable separately, the data flow graph can be modified to remove edges out of nodes that contain definition points.

Since this algorithm solves only *single bit* or *if* problems,¹ it needs only to traverse the graph from the definition sites until he finds (1) another definition site, (2) the end of a chain, or (3) a node that has been visited. By following this algorithm, all of the nodes that have live values for a

¹See section 2.6.3 for a discussion of *if* verses *where* problems.

particular variable have been marked.

This work is important because it is the first to develop a global algorithm that is based on solving each variable separately. This allows several common data flow problems to be solved in linear time on general program flow graphs.

This algorithm has not been used in practice because compilers typically need the solution for all variables at once and a separate pass for each variable has appeared too costly. This dissertation is the first to investigate this algorithm, the *multi-bit* modification, and modifications to limit this algorithm to affected area.

6.2. Rosen

Rosen [Rose 81] has developed a technique for transforming *exhaustive* algorithms to *demand* algorithms. The term *exhaustive* describes algorithms that must recalculate from scratch any time the input changes. The term *demand* describes algorithms that can update a part of the solution after some number of changes have been made to the input. The technique attempts to transform exhaustive algorithms so that the complexity is based on the size of the affected region rather than the size of the input. The technique is specifically targeted to transform data flow algorithms.

The basis of this technique is to cause the algorithm to produce a list of affected blocks for each variable. When a block is modified, only those variables whose status in that block have changed need have their information updated. Since a list of the affected blocks for each variable has been maintained, it is known which blocks need to be examined to

update the data flow database. The number of blocks affected should be much smaller than the number of blocks in the program since data flow information is typically sparse.

Rosen's paper does not present an algorithm for demand data flow analysis nor does it give an algorithm for transforming a given exhaustive algorithm into a demand algorithm. The technique cannot predict the time complexity of the resulting algorithm, but only that it will be a function of the affected region. The complexity of the algorithm will depend on the type and precision of the information desired and the specific structures used to store the information. The paper is significant in that it describes a framework in which different algorithms can be compared.

The algorithms presented in the next chapter fall into the third category of *incremental* algorithms. This class of algorithms keep the solution to the problem correct as the input changes. The algorithms presented in the next chapter could be modified to become *demand* algorithms. The decision to develop them as *incremental* algorithms was based on the environment in which they are intended to work.

6.3. Babich and Jazayeri

Babich and Jazayeri [BaJa 78] develop essentially the same algorithm as Kou in the first part of their paper. The algorithm differs from Kou in that theirs performs the analysis on a parse tree, rather than the typical low level, representation of the program.

In the second part of their paper they modify the algorithm to work in a demand environment. The key result of this work is that for problems such as *live-dead* analysis, if information is desired for a particular node, a

search is started at that node and ends anywhere the value is redefined. This result is used in this dissertation to find the affected area of a change.

6.4. Ryder

Ryder, in her PhD. dissertation [Ryde 82], has made an attempt to solve the incremental update version of the global data flow problem. She has modified several classical data flow algorithms to work in an incremental fashion. The data flow algorithms specifically addressed in her thesis are: Allen-Cocke Interval Analysis, Hecht-Ullman T1-T2 Analysis, and Tarjan Interval Analysis. It is implied that the methods used could be easily extended to other data flow techniques.

The work that Ryder has done is limited to a subset of the changes required in many environments. Only *Type I* changes,² i.e. local, single variable changes are considered. There are no provisions for dealing with any changes to the structure of the program flow graph. This places severe restrictions on the environments in which this work can be used. In the area of programming environments, it is required that the user be able to make changes that affect the program structure and still have the reliability information produced quickly.

There is no implication that this work could not be extended to handle *Type II* and *Type III* changes. This was simply outside the scope of the topics that Ryder wished to tackle for her dissertation.

²See the next chapter for the definition of *Type I* changes.

6.5. Wegman

Wegman [Wegm 82] has developed an algorithm for processing data flow information in $k \log(k)$ time, where k is the size of the program graph. The algorithm works for restricted languages, specifically those that do not have `goto`'s. The importance of the algorithm is that the output can be processed in time k to answer many questions that are directly applicable to this dissertation.

There is no mention in the paper of how much work would be required to update the processed description quickly for small changes in the graph.

Chapter 7

Incremental Algorithms

The purpose of this chapter is to provide algorithms to maintain the data flow information in the presence of incremental changes to the program. These algorithms also perform the analysis required to detect many anomalies in the program. It is expected that the implementor of a structured program editor will incorporate the operations provided here to suit the needs of his particular environment.

The algorithms presented here are intended to respond to changes that fall into three broad categories:

- (1) *Type I* operations are changes to individual references in the program.
- (2) *Type II* operations are changes that add or delete edges (control flow paths) to the program.
- (3) *Type III* operations are changes that add or delete basic blocks to the program.

Two data flow problems discussed in chapter 5, the *Live Variable Problem* and the *UnSafe Uses Problem* will be used as examples throughout this chapter.

Several sets are maintained by the incremental algorithms defined in this chapter. These sets are *Use*, *Def*, *UnDef*, and *Mod* defined in section 5.1.1 and *Uses*, *Defs*, *UnDefs*, and *Mods* defined in section 5.1.2.

7.1. Special Local Algorithms

The following procedure, *SearchL*, is used to find the area affected by a single change in either the forward or backward direction. *SearchL* calculates a list of nodes that are affected by a change at a particular section of the graph. This list is then used as input by the higher level algorithms. These higher level algorithms do specific operations at each of the nodes.¹

This procedure begins the search at some node specified at a higher level. This node is typically the one that is currently being edited by the user, and is identified as being pointed to by the *cursor*. *SearchL* only pursues a single value.² The procedure has five parameters:

<i>NodeList</i>	The list of nodes that may be searched.
<i>EdgeList</i>	The set of edges that connect the nodes. This list may be either the <i>Successor</i> set for <i>backward problems</i> or the <i>Predecessor</i> set for <i>forward problems</i> .
<i>S</i>	The set of nodes that contain <i>S sites</i> for the cluster.
<i>c</i>	The node pointed to by the editor cursor.
<i>Visited</i>	The list of nodes that have been visited.

¹In actual implementations, the visiting would be integrated into the higher level routines. The separation is made here for clarity.

²The term *value* means the result of a single definition point of a single variable.

```

procedure SearchL(NodeList, EdgeList, S, c, Visited)

    { output: Visited }

    Visited  $\leftarrow c$                                 { Set of nodes that have been visited. }
    Work  $\leftarrow \text{EdgeList}[c]$                     { Work is the worklist of nodes. }

    while ( Work  $\neq \emptyset$  ) do
        choose any t from Work
        Work  $\leftarrow \text{Work} - t$ 
        Visited  $\leftarrow \text{Visited} + t$ 

        if ( t  $\notin S$  )
            then Work  $\leftarrow \text{Work} \cup ( \text{EdgeList}[t] - \text{Visited} )$ 

    endwhile
end

```

Figure 7.1: Algorithm for *SearchL*

There is a simple conceptual picture that describes the nodes that are visited. The edges of the flow graph can be viewed as pipes that connect the nodes of the flow graph. In these pipes, the fluid can only travel in the direction of the arc. Each node that contains an *S site* for *v* cannot pass any fluid. The algorithm, *SearchL* visits all of the nodes that would be wet if a source of water were connected at node *c*.

The worst case complexity of *SearchL* is $O(\text{NodeList} + \text{EdgeList})$. The use of *Visited* assures that no node is visited more than once. The *Visited* field of each node must be checked once for each in-edge. These algorithms only modify the graph for a single value of a single variable.

The expected case complexity should be much better than the number of nodes in the graph. *SearchL* visits only those nodes that a single value

may flow through. While it is true that sometimes the entire graph must be visited, many values are short-lived.

The algorithm *SetInfo*, presented earlier, is used to propagate information from several *B sites* to other nodes. *SetInfo* visits only those nodes that are discovered by the call to *SearchL*.

The output of these algorithms is different from the output of a classical data flow procedure. Instead of producing all the data flow information that is associated with all variables, these algorithms produce the information associated with a small set of values for a single variable.

There are two changes in the description of *SetInfo* that are required by the local algorithms. First, when *SetInfo* is called to propagate information over the *Succ* set, it must update three sets, *Defs*, *UnDefs*, and *Mods*, rather than only one. The use of the three sets is indicated by concatenating these sets in the parameter position. In a real implementation, there would be separate versions for both the forward and backward cases, thereby avoiding this problem. Second, the meet operator is *set union* for the problems solved here.

7.2. Algorithms For Type I Changes

Type I operations are changes that do not affect control flow or block structure of the program. While each change is confined to a single basic block, the effect of a single change extends over a wider area, possibly the entire program. Any modification to an assignment or input-output statement is a *Type I* change. Additionally, any change to the conditional expression in **while**, **for**, or **if** statement fall into this category.

- (1) *Type I* algorithms examine only at a single variable permitting them to use the actual *Use*, *Def*, *Mod* and *UnDef* information for that variable to pick an optimal path through the flow graph at run time.
- (2) *Type I* algorithms do not look at all the values of the variable. The algorithms only deal with those values that either affect or are affected by the part of the program pointed to by the editing cursor.
- (3) *Type I* algorithms will only track this value through part of the program. This part of the program is the extent that a single value of the variable is live.

The following variables reflect the state of the editor. They are manipulated indirectly by the user and are used as input parameters to the routines that respond to *Type I* changes.

- c* is the node, or basic block, that is currently being edited. This value is automatically modified by the editor in response to actions by the user.
- v* is the variable to be tracked through the flow graph. This is the name of the variable whose use or definition has been modified by the user.

Algorithm *SearchL* provides a set of nodes that may affect or be affected by a change at node *c*. The next four algorithms use *SearchL* to check the consistency of the program for the following four editing operations:

- (1) Addition of a Use
- (2) Deletion of a Use
- (3) Addition of a Definition
- (4) Deletion of a Definition
- (5) User Queries

For each of these cases an algorithm must be executed to assure that the program state is still consistent. If any inconsistencies are found, the editor should make the user aware of the potential problem.

7.2.1. Addition of a Use

The procedure to add a *Use* to a block is to propagate the *Use* to the *Predecessor* nodes that are reachable from the changed node, *c*. The only error condition that must be checked is to make sure that the *UnDefs* set for node *c* is empty. A nonempty *UnDefs* set indicates the existence of a path where the new *Use* may reference an undefined variable.³

```

procedure AddUse( c, v )
  { Use algorithm SearchL to assure that all paths that }
  { precede this node define v. }
  call SearchL( G, Pred, Def[ v ]  $\cup$  UnDef[ v ], c, Visited )

  { Add this Use to the local block information. }
  Use[ v ]  $\leftarrow$  Use[ v ] + c

  { Add this Use to the Uses structure for each pred. }
  for each node in Visited do
    Uses[ node, v ]  $\leftarrow$  Uses[ node, v ] + c
  endfor

  for each node in UnDefs[ c, v ] do
    print
    "Killed value from node node for variable v may reach node (c)."  

  endfor
end

```

Figure 7.2: Algorithm for *AddUse*

Addition of a *Use* to a Basic Block

³This path may not be executable due to constraints that cannot be detected by data flow analysis. See section 4.1.

7.2.2. Deletion of a Use

The procedure to delete a *Use* from a basic block is similar to the procedure to add a *Use*. The difference in the algorithms stems only from the error conditions that must be checked. When deleting the *Use*, it is important to check that each *Def* that can reach node *c* still has valid *Uses*.

```

procedure DelUse( c, v )
  { Use algorithm SearchL to find all sites that may define v. }
  call SearchL( G, Pred, Def[ v ]  $\cup$  UnDef[ v ], c, Visited )

  { Delete this Use from the Uses structure for each defining node. }
  for each node in Visited do
    Uses[ node, v ]  $\leftarrow$  Uses[ node, v ] - c
  endfor

  for each node in Defs[ c, v ] do
    if ( Uses[ node, v ] =  $\emptyset$  )
      then print
        "The value calculated at node node for v is never used."
    endif
  endfor

  { Delete this Use from the local block information. }
  Use[ v ]  $\leftarrow$  Use[ v ] - c
end

```

Figure 7.3: Algorithm for *DelUse*

Deletion of a *Use* from a Basic Block

7.2.3. Addition of a Def

There is a problem associated with adding a *Def* to a block. When a *Def* is added, it may break some or all use-definition chains for that variable that pass through that block. There are no data structures in this algorithm for encoding the various paths that may be available around the *Def* that is to be added. Therefore *SetInfo* must be called to fix up the chains. The algorithm that is used to add a *Def* works by deleting all old chains that pass

through the node, and then rebuilding the chains. The chains are rebuilt by propagating the chains that border the affected area into the affected area.

The algorithm must ensure that all of the *Def* sites that can reach *c* still have valid uses. This same test must also be made for the newly added *Def*.

```

procedure AddDef( c, v )
  { Add the Def to the current node. }
  Def[ v ]  $\leftarrow$  Def[ v ] + c

  { Delete all the backwards Uses for v that pass through c. }
  call SearchL( G, Pred, Def[ v ]  $\cup$  UnDef[ v ], c, Visited )

  for each node in Visited do
    Uses[ node, v ]  $\leftarrow$  Uses[ node, v ] - Uses[ c, v ]
  endfor

  { Rebuild the Uses. }
  B  $\leftarrow$  Visited  $\cap$  ( Use[ v ] )
  E  $\leftarrow$  Visited  $\cap$  ( Def[ v ]  $\cup$  UnDef[ v ]  $\cup$  Mod[ v ] )
  S  $\leftarrow$  Visited  $\cap$  ( Def[ v ]  $\cup$  UnDef[ v ] )
  call SetInfo( Visited, Succ, B, E, S, Uses )

  Defs[ c, v ]  $\leftarrow$  Defs[ c, v ] + c
  for each node in Defs[ c, v ] do
    if ( Uses[ node, v ] =  $\emptyset$  )
      then print
        "The value calculated at node node for v is never used."
    endif
  endfor

  { Delete all the forward Defs for v that pass through c. }
  call SearchL( G, Succ, Def[ v ]  $\cup$  UnDef[ v ], c, Visited )

  for each node in Visited do
    Defs[ node, v ]  $\leftarrow$  Defs[ node, v ] - Defs[ c, v ]
  endfor

  { Rebuild the Defs, UnDefs, and Modes. }
  B  $\leftarrow$  Visited  $\cap$  ( Def[ v ]  $\cup$  UnDef[ v ]  $\cup$  Mod[ v ] )
  E  $\leftarrow$  Visited  $\cap$  ( Use[ v ] )
  S  $\leftarrow$  Visited  $\cap$  ( Def[ v ]  $\cup$  UnDef[ v ] )
  call SetInfo( Visited, Pred, B, E, S, Defs || UnDefs || Modes )
end

```

Figure 7.4: Algorithm for *AddDef*

Addition of a *Def* to a Basic Block

7.2.4. Deletion of a Def

The procedure to delete a *Def* from a basic block is simpler than adding the *Def*. Once the effect of the *Def* is removed by visiting the *Successors*, the *Uses* that were blocked by the *Def* must be propagated to the *Predecessors* of *c*. Checking that no *UnDefs* can reach the newly exposed *Uses* is the only error that must be checked.

```

procedure DelDef( c, v )
  { Delete the Def from the current node. }
  Def[ v ]  $\leftarrow$  Def[ v ] - c

  { Propagate all Defs for v to the successors of c. }
  call SearchL( G, Succ, Def[ v ]  $\cup$  UnDef[ v ], c, Visited )

  for each node in Visited do
    Defs[ node, v ]  $\leftarrow$  Defs[ node, v ] + Defs[ c, v ] - c
    UnDefs[ node, v ]  $\leftarrow$  UnDefs[ node, v ] + UnDefs[ c, v ]
  endfor

  { Propagate all Uses for v to the predecessors of c. }
  { This information was blocked by the old def. }
  call SearchL( G, Pred, Def[ v ]  $\cup$  UnDef[ v ], c, Visited )

  if ( Uses[ c, v ]  $\neq \emptyset$  )
    then do
      for each node in Visited do
        Uses[ node, v ]  $\leftarrow$  Uses[ node, v ] + Uses[ c, v ]
      endfor

      for each node in UnDefs[ c, v ] do
        print
        "Killed value from node node for variable v may pass thru node c."
      endfor
    enddo
  end

```

Figure 7.5: Algorithm for *DelDef*

Deletion of a *Def* from a Basic Block

7.2.5. User Queries

The term *user queries* is a generic term which encompasses all operations where the user wishes to inquire about information that is maintained by the system. In these operations, the information that is being maintained by the system is made available to the user. Examples of these queries are:

Print all the locations that can use this definition for *v*.

Print all the locations that can define the value for *v* that is used here.

These queries are not of the type that can be determined by inspection of the symbol table since they require knowledge of the flow of values for each variable. These queries can be very important to the user. The output of these queries can provide information that can aid the user assessing in the effects of small changes in the program.⁴

Each of these queries can be implemented by trivial calls to *SearchL*. This procedure is used extensively by the other *Type I* algorithms to provide this exact information for internal use.

7.2.6. Other Type I Operations

Other operations that fall into the *Type I* category are:

- (1) Addition of a *Mod*.
- (2) Deletion of a *Mod*.

The algorithms that relate to *Mods* are easy to derive in a manner similar to the algorithms for adding and deleting *Defs*.

⁴The *Scope* system by Masinter [Masi 80] provides a similar feature for a Lisp based system. See chapter 3.

7.3. Algorithms For Type II Changes

Type II changes are all control flow changes. These changes occur as changes in the edges in the program flow graph. *Type II* changes are the result of operations on `goto` statements and statements that implement restricted branches.

- (1) *Type II* algorithms look at all variables that can have live values in the region changed.
- (2) *Type II* algorithms do not look at all values of the variable. The algorithms only deal with those values that either affect or are affected by the part of the program pointed to by the editing cursor.
- (3) *Type II* algorithms will only track these values through part of the program. This part of the program includes only the area in which a single computed value is live.

Type II algorithms are invoked by specifying pointers to two basic blocks. These parameters point to the source and destination of the edge being manipulated.

cin is a pointer to the source node that is affected by editing operations where more than one node is involved. Operations that make arbitrary changes in the flow graph must be specified by a source node *cin* and a destination node *cout*.

cout is a pointer to the destination node that is affected by editing operations where more than one node is involved.

Algorithms are given for two general problems:

- (1) Addition of an arbitrary edge.
- (2) Deletion of an arbitrary edge.

7.3.1. Addition of an Edge

The routine *SearchL* is not used when adding an edge to a graph. *SearchL* has been merged into *AddEdge* in order to allow additional criteria for selecting the visitation order. These additional criteria keep subtrees from being visited where the information is already correct. These already correct subtrees are the result of other paths through the tree that carry the same information as the newly added edge. The criterion applied is based on the observation that propagations need not be done if the information to be propagated is a subset of the information already there.

Once the edge is added to the *EdgeList*, the values that are propagated forward, *Defs*, *UnDefs*, and *Mods* are available at the source node, *cin*, and must be propagated to all nodes that are reachable from the destination node, *cout*.

```

procedure AddEdge( cin, cout )

  for each v in VarList
    Visited  $\leftarrow$  cout                                { Set of nodes that have been visited. }
    Work  $\leftarrow$  Succ[ cout ]                        { Work is the worklist of nodes. }

    while ( Work  $\neq$   $\emptyset$  ) do
      choose any node from Work
      Work  $\leftarrow$  Work - node
      Visited  $\leftarrow$  Visited + node

      if ( ( v  $\notin$  ( Def  $\cup$  UnDef ) )
        or ( UnDefs[ node, v ]  $\supset$  UnDefs[ cin, v ] )
        or ( Defs[ node, v ]  $\supset$  Defs[ cin, v ] )
        or ( Mods[ node, v ]  $\supset$  Mods[ cin, v ] ) )
        then Work  $\leftarrow$  Work  $\cup$  ( Succ[ node ] - Visited )

      UnDefs[ node, v ]  $\leftarrow$  UnDefs[ node, v ]  $\cup$  UnDefs[ cin, v ]
      Defs[ node, v ]  $\leftarrow$  Defs[ node, v ]  $\cup$  Defs[ cin, v ]
      Mods[ node, v ]  $\leftarrow$  Mods[ node, v ]  $\cup$  Mods[ cin, v ]
    enddo

    Visited  $\leftarrow$  cin
    Work  $\leftarrow$  Pred[ cin ]

    while ( Work  $\neq$   $\emptyset$  ) do
      choose any node from Work
      Work  $\leftarrow$  Work - node
      Visited  $\leftarrow$  Visited + node

      if ( ( v  $\notin$  ( Def  $\cup$  UnDef ) ) or ( Uses[ node, v ]  $\supset$  Uses[ cout, v ] ) )
        then Work  $\leftarrow$  Work  $\cup$  ( Pred[ node ] - Visited )

      Uses[ node, v ]  $\leftarrow$  Uses[ node, v ]  $\cup$  Uses[ cin, v ]
    enddo
  endfor
end

```

Figure 7.6: Algorithm for *AddEdge*

Add an edge to the graph

7.3.2. Deletion of an Edge

The same feature that makes for a faster algorithm when adding an edge causes a certain amount of difficulty when deleting an edge. Because it is unknown from which path a value comes, a conservative strategy must be used to update the information caused by deleting an edge. The strategy used is to remove all information that is propagated from the deleted edge, and then to rebuild the information in the affected area by propagating the information from the in-edges of that affected region. The errors that must be detected stem from *Defs* that do not have any *Uses* after the edge has been deleted.

The *DelEdge* algorithm is the only algorithm that is not linear in the size of the affected area. Every other algorithm presented in this dissertation has the property that no node is visited unless the information at that node is changed in some matter. This algorithm visits extra nodes in some cases.

An interesting point to note is that it is not necessary to check that each *Use* has a *Def* site that reaches it. This is true since the three sets, *Defs*, *UnDefs*, and *Mods* are propagated in the forward direction and cover all paths. As long as the *Use* site is reachable, in the standard graph theory sense, there must either be a path that contains a *Def* or *Mod* which is acceptable, or the path is reachable by an *UnDef*, in which case the path had already been detected as a possible error path.

```

procedure DelEdge( cin, cout )

  for each v in VarList
    { Delete all the backwards Uses for v that pass through cin. }
    call SearchL( G, Pred, Def[ v ]  $\cup$  UnDef[ v ], cin, Visited )

    for each node in Visited do
      Uses[ node, v ]  $\leftarrow$  Uses[ node, v ] - Uses[ cin, v ]
    endfor

    { Rebuild the Uses. }
    B  $\leftarrow$  Visited  $\cap$  ( Use[ v ] )
    E  $\leftarrow$  Visited  $\cap$  ( Def[ v ]  $\cup$  UnDef[ v ]  $\cup$  Mod[ v ] )
    S  $\leftarrow$  Visited  $\cap$  ( Def[ v ]  $\cup$  UnDef[ v ] )
    call SetInfo( Visited, Succ, B, E, S, Uses )

    for each node in Defs[ cin, v ] do
      if ( Uses[ node, v ] =  $\emptyset$  )
        then print
          "The value calculated at node node for v is never used."
      endif
    endfor

    { Delete all the forward Defs for v that pass through cout }
    call SearchL( G, Succ, Def[ v ]  $\cup$  UnDef[ v ], cout, Visited )

    for each node in Visited do
      Defs[ node, v ]  $\leftarrow$  Defs[ node, v ] - Defs[ cout, v ]
      Mods[ node, v ]  $\leftarrow$  Mods[ node, v ] - Mods[ cout, v ]
      UnDefs[ node, v ]  $\leftarrow$  UnDefs[ node, v ] - UnDefs[ cout, v ]
    endfor

    { Rebuild the Defs, UnDefs, and Mods. }
    B  $\leftarrow$  Visited  $\cap$  ( Def[ v ]  $\cup$  UnDef[ v ]  $\cup$  Mod[ v ] )
    E  $\leftarrow$  Visited  $\cap$  ( Use[ v ] )
    S  $\leftarrow$  Visited  $\cap$  ( Def[ v ]  $\cup$  UnDef[ v ] )
    call SetInfo( Visited, Pred, B, E, S, Defs  $\parallel$  UnDefs  $\parallel$  Mods )

  endfor
end

```

Figure 7.7: Algorithm for *DelEdge*

Delete an edge from the graph

7.4. Algorithms for Type III Changes

Type III changes are modifications that add or delete nodes from the program flow graph. *Type III* changes differ from *Type II* changes in that *Type III* changes always modify the number of basic blocks in the program while *Type II* changes only deal with interconnection changes among the basic blocks.

Type III operations occur when:

- (1) Addition of a **goto** statement into or from the interior of a *basic block*. That basic block must be *serially split* into two blocks at the point where the new edge enters.
- (2) Deletion of a **goto** statement from the head of a *basic block*. If there were only two in-edges into that block and the remaining in-edge is the only out-edge from another block, the two blocks can be *serially combined*.
- (3) Addition or deletion of a section of code. Code is added to the program by asking the editor to add an empty template at the cursor position. In these operations, the user fills in the body of the template. *Type I* and *Type II* operations are used to fill in these templates.

The operation of adding the template is implemented by adding a series of empty basic blocks in the area pointed to by the editing cursor. This operation of adding the templates can be implemented by three primitive operations: *serial splitting*, *parallel splitting*, and *addition of self-loop*.

A basic block is a section of code that satisfies the following constraint:

If *any* statement in the block is executed, *all* statements in the block are executed.

There are three operations over basic blocks that are used to grow a program. These operations also have inverse operations that are used to shrink a program. Unlike other operations that have been considered here, the process of growing the flow graph is a perfect inverse of the process of shrinking the flow graph.

The operations considered in this section are:

- (1) Serial splitting of a basic block.
- (2) Combining two serial basic blocks.
- (3) Parallel splitting of a basic block.
- (4) Combining two parallel basic blocks.
- (5) Addition of a self-loop block.
- (6) Deletion of a self-loop block.

7.4.1. Serial Splitting of a Block

The term *serial splitting* means to split a single block where the resultant blocks are connected such that after control leaves the first resultant block, it will enter the second resultant block. When a block is split, the first few statements of the split block go into the first resultant block and the remaining statements go into the second block. If the assumption is made that the first resultant block retains the label of the old block, then all that must be done in the splitting operation is to relabel the *Uses*, *Defs*, *Mods*, and *UnDefs* that are propagated from the resultant block.⁵

⁵A better version of this program would pick the resultant block with the smallest number of program references. This would result in a slight, nonasymptotic, speedup of this operation.

The only work that is done in either serial splitting or combining of basic blocks is to renumber the information in the basic blocks that are reachable for the block being changed. There are no error conditions to be checked.

```

procedure SerSplit( OldBlock, NewBlock, SplitPtr )

    call SerSplitSet(Use, Uses, Pred)
    call SerSplitSet(Def, Defs, Succ)
    call SerSplitSet(Mod, Mods, Succ)
    call SerSplitSet(UnDef, UnDefs, Succ)

    procedure SerSplitSet(Set, Sets, Edges)
        for each v in SymbolTable do
            if (( OldBlock  $\in$  Set[ v ] ) and ( Loc( v ) > SplitPtr ))
                then do
                    Set[ v ]  $\leftarrow$  Set[ v ] - OldBlock + NewBlock

                    call SearchL( G, Edges, Def[ v ]  $\cup$  UnDef[ v ], NewBlock, V )

                    for each node in V do
                        Sets( node, v )  $\leftarrow$  Sets( node, v ) + NewBlock - OldBlock
                    endfor
                enddo
            enddo
        end
    end

```

Figure 7.8: Algorithm for *SerSplit*

Serial Splitting of a Basic Block

7.4.2. Serial Combining of a Block

As mentioned earlier, the *SerCombine* routine is the exact inverse of the *SerSplit* routine.

```

procedure SerCombine( OldBlock, DelBlock )

  call SerCombineSet(Use, Uses, Pred)
  call SerCombineSet(Def, Defs, Succ)
  call SerCombineSet(Mod, Mods, Succ)
  call SerCombineSet(UnDef, UnDefs, Succ)

  procedure SerCombineSet(Set, Sets, Edges)

    for each v in SymbolTable do
      if ( DelBlock  $\in$  Set[ v ] )
        then do
          Set[ v ]  $\leftarrow$  Set[ v ] - DelBlock + OldBlock

          call SearchL( G, Edges, Def[ v ]  $\cup$  UnDef[ v ], OldBlock, V )

          for each node in V do
            Sets( node, v )  $\leftarrow$  Sets( node, v ) + NewBlock - DelBlock
          endfor
        enddo
      enddo
    end
  end

```

Figure 7.9: Algorithm for *SerCombine*

Serial Combining of Two Basic Blocks

7.4.3. Remaining Operations

The operations to parallel split and parallel combine basic blocks, and the operations to add and delete self-loop blocks are null operations with respect to the data flow information. This is a result of the restriction that all blocks acted on in this manner must be empty. While this restriction may seem unreasonable at first, it must be realized that these operations

only result from the addition or deletion of templates from the program. Templates must be empty. The only operation that must be done is to allocate or free a new basic block. The implementation of this is outside the scope of this presentation.

The algorithms that could be derived for more of these operations become equivalent to doing a series of *Type I* operations to delete the information from one block, followed by a series of *Type I* operations to reinsert the information in the other block. The purpose of this presentation is to describe a series of primitive operations out of which any high level operation can be built, rather than to provide concrete algorithms for all possible situations.

Chapter 8

Techniques to Improve Performance of PVT

This chapter is divided into two sections. The first is a comparison of the Partitioned Variable Technique with classical techniques. The second section describes a modification to alleviate some shortcomings of the PVT.

8.1. Comparison of PVT with Classical Techniques

A set of algorithms has been presented to perform all the operations that are required to keep data flow information correct as it is being changed. This information is then used to detect several common programming errors. The technique presented uses a framework that is different from the classical data flow frameworks typically used. Classical data flow techniques can be used to provide the same information.

The advantages and disadvantages of using the PVT in a real editor with real programs are not readily apparent. The following sections attempt to objectively assess the benefits of this new approach in several settings that arise in the context of structured program editing.

8.1.1. Comparison of Single Variable Operations

When *Type I* modifications (single variable) are made to the program, the PVT only visits those nodes of the flow graph where the data flow information actually changes. When a classical data flow technique is used, such as interval analysis, it is possible to identify those intervals that contain the affected area. No finer level of information beyond the identified intervals is available. The data flow equations can then be resolved for that variable using each of the nodes in the identified

intervals.¹ Since only one variable is being updated, the width of the bit vectors that would be used for a classical technique are small. Therefore, only simple operations must be performed for each node using either type of algorithm.

The identified intervals cannot be any smaller than the area identified by the PVT since the PVT visits the minimum area. The identified intervals may be much larger since interval analysis, and in general all classical techniques, fail to make use of the definition and use of single variables within the intervals.

Since these types of operations only affect a single variable, the PVT can be expected to provide performance superior to any classical technique for this class of problems.

8.1.2. Comparison of Multiple Variable Operations

In *Type II* modifications (multiple variable), the choice of algorithms is not so clear. Again it is possible to envision techniques that would discover the affected regions using classical data flow techniques.² These regions would also always be at least as large and probably larger than the areas found by the partitioned technique. But it is misleading to conclude on this basis alone that the PVT is superior to a technique built on a classical data flow technique. While it is true that the number of operations for the partitioned variable technique will never be larger than the number of

¹For the purposes of this discussion, interval analysis will be used as an example of a classical data flow technique. Ryder has shown that affected areas of nodes can be identified for most classical techniques.

²The only work to date on the updating of data flow information with respect to changes in the control flow structure that I am aware of is by Mark Wegman [Wegm 83]. This work is reviewed in detail in section 6.3.

operations for any classical technique, the nature of these operations must be carefully investigated.

In a classical strategy, the work involved consists of simple bit vector operations. The bit vector is as wide as the number of definition (or use) sites in the program. The number of bit vector operations required is a linear function of the number of nodes in the affected area.

In the partitioned variable strategy, the work involved consists of solving a problem similar to the single variable case for each variable that passes through the nodes being modified. For each variable, the number of nodes may be, and typically should be, somewhat smaller than the number of nodes for a classical technique. Additionally the bit vector to be solved may be smaller since it is known which *use* or *def* sites can be involved.

The savings in the size of the problem must be balanced against the larger cost to visit a node for the PVT. While each operation using either strategy can be done in a constant amount of time, it should be expected that wide bit vector operations, while providing no asymptotic speedup, provide a very efficient implementation vehicle on a Von Neumann type machine.³ The decision is not clear as to which technique is best. One useful criterion to assess the speed of the PVT in incremental applications would be to estimate the density of *Defs* for an average variable. As the density of *Def* sites per variable increases, the average size of the affected area for each change will decrease. Classical data flow techniques are insensitive to this density and are only affected by changes in the control

³See "On Live-Dead Analysis for Global Data Flow Problems" by L. T. Kou [Kou 77] for a detailed comparison of a similar technique with classical data flow algorithms using several machine models.

flow of the program. A technique that can positively affect this density is discussed in detail in a later section of this chapter.

A consideration must be made regarding the frequency of occurrence of *Type II* operations. The only *Type II* operations that are expensive are the general case, addition and deletion of edges. These cases only occur when *goto* statements are added or deleted or if sections of the flow graph are moved around. These operations should be expected to occur with much less frequency than the other *Type II* operations or any of the *Type I* operations.

8.1.3. Other Advantages of PVT over Classical Methods

- (1) The algorithm works over all flow graphs. While studies have shown that most programs have flow graphs that are reducible,⁴ it is normally not a very attractive possibility to implement a production program that only works some of the time.^{5 6}
- (2) The PVT is easy to understand and lends itself to a straightforward implementation.

⁴Kennedy and Zucconi conducted an empirical study at Rice University which showed that 94% of all graphs were reducible and 88% were SSFG reducible [KeZu 77]. Knuth [Knut 70] looked at 50 Fortran programs and found that all were reducible. Hecht observes (in a footnote on page 77 of [Hech 77]) that many of the example programs in *The Art of Computer Programming Vol I. Fundamental Algorithms* [Knut 74] by Knuth are not reducible.

⁵Tarjan's Balanced Path algorithm works on non-reducible flow graphs. The algorithm is slower in this case.

⁶Algorithms that do not work on reducible flow graphs can be modified to handle non-reducible graphs by using one of two techniques. The first is called *node splitting* [CoSc 70]. A copy is made of one of the nodes in the irreducible region in an attempt to break the region. This algorithm can be costly in time and space. A second technique used is to solve the irreducible region using the iterative method. Since the irreducible region is typically small, this does not tend to be very expensive.

8.1.4. Disadvantages of the Partitioned Variable Technique

- (1) Classical techniques can be used to solve a wider variety of data flow problems than the proposed technique. Many applications require the solution of several different data flow problems.⁷ It is therefore convenient to implement the solution of all of the data flow problems using the same technique.
- (2) Since the PVT does not reduce the flow graph, it can not be said to "understand" the program. This problem will be discussed in detail in the next section.

8.1.5. Asymptotic Complexities of Various Algorithms

In this dissertation, several common data flow problems are shown to be *partitionable* and may be solved by the *Partitioned Variable Method*. The time complexity to solve a partitionable data flow problem using various choices of the propagation algorithm is summarized in figure 8.1. Examples of problems are presented in chapter 5. The speed shown is the worst case time complexity. Using any classical technique, the time shown is the time of propagation phase multiplied by the width of the bit vector.

⁷For many data flow problems, the typical solution technique is to build *Use-Def* chains and use these chains as input to the solution algorithm. This technique converges faster than using the original flow graph for many problems.

Worst Case Time Complexity to Solve Bit Vector Problems			
Algorithm	Speed	Graph Class	Reference
Iterative	$e \times (d + 3) \times b$	all	[Hech 77]
Interval	$e \times d \times b$	reducible	[Alle 70]
Balanced tree	$e \times (\log e) \times b$	reducible	[Ullm 73]
Path compression	$e \times (\log e) \times b$	reducible	[GrWe 76]
Node listing	$e \times (\log e) \times b$	reducible	[Kenn 75]
Balanced path	$e \times \alpha(e, e) \times b$	reducible	[Tarj 75]
Grammar	$n \times b$	SSFG reducible	[KeZu 77]
High-level	$n \times b$	parse tree	[Rose 77]
Partitioned variable	$(n + e) \times b$	all	Presented here

where:

n	Number of nodes in graph
e	Number of edges in graph ⁸
d	Maximum nesting of loops
b	Width of bit vector
α	Inverse Ackerman's function

Figure 8.1: Comparison of Data Flow Techniques

8.2. Adding Knowledge About Program Structure

The idea investigated here to improve the efficiency of PVT is the use of a higher level representation of the program than the basic block. A major failing of partitioned variable data flow analysis is that it does not make use of the program structure. This deficiency manifests itself in the incremental environment in a fairly conservative and possibly slow algorithm to delete arbitrary edges from a program flow graph. The intent of this section is to investigate a technique that can be used to improve the

⁸It is virtually always assumed that the number of edges in a data flow graph is roughly the same as the number of nodes in the graph. While it is theoretically possible to have graphs where the number of edges is the square of the number of nodes, the program that yields such a graph could only be produced by a very clever and malicious programmer.

performance of PVT. The hope is to enhance the performance of the PVT to a point where it is very attractive for all situations that may arise in a structured program editor.

The mechanism used to enhance performance is raising the density of *Def sites* for each variable. The most obvious way to increase the *Use* and *Def* density is to attempt a reduction on the program flow graph. This reduction would replace groups of nodes by a single new node. This new node would then be labeled with information that summarized the effects of execution through that group of nodes.

While this strategy seems similar to those used by other data flow techniques, there are several important differences:

- (1) One goal in the development of this idea is to use a method to abstract the program to a higher level that is tolerant of unstructured programming constructs. While it is true that our goal is to find regions that occur in a large number of graphs, there is no expectation that real programs can be completely reduced by this technique. The method that is developed here is tolerant of regions that do not fit our model of reducibility.

Many different graph transformation strategies could be used. The PVT algorithm is uniquely amenable to this type of arrangement. Since the PVT method does not attempt to understand the structure of the program flow graph, any transformations that can be made on the flow graph before the PVT algorithms are applied will work.

- (2) The reduction strategy presented here is not an attempt to improve the asymptotic complexity of this data flow technique. Since the PVT

technique is linear for all flow graphs, there is little reason to expect improvements here. Instead, the goal is to reduce the search that must be done for each variable in the program to those areas of the program where that variable is used, rather than those areas where it is just live.

8.2.1. A Technique To Improve Use and Def Density

The method that is proposed reduces the single-entry, single-exit (hereafter *SESE*) regions of the graph to single nodes. These regions are prevalent in "structured programs" but unstructured constructs such as **goto** statements can destroy these regions.⁹

SESE regions occur naturally in program without **gotos** as:

- (1) **do** or **while** loops.
- (2) **if-then** or **if-then-else** blocks.
- (3) **case** Statements.
- (4) Single Basic blocks.

Additionally, any of the above constructs that contain **goto** statements that have both source and destination local to that construct may also be reduced as an SESE region. This means that SESE regions are not necessarily composed of SESE regions.¹⁰

It is exactly the prevalence of SESE regions in real programs that is used to develop a fast algorithm to find and reduce these regions. Rather

⁹Mark Wegman [Wegm 83] has investigated algorithms restricted solely to these constructs. His work is reviewed in section 6.3.

¹⁰There is one other programming construct that can give rise to an SESE region, the **goto** loop. Programming using this style is very much out of vogue. The technique presented in this dissertation will not find this type of region. But existence of this construct will not inhibit this technique from finding other legitimate SESE regions.

than looking to graph theory to find such an algorithm, it is proposed to look at the programming constructs themselves to aid in their reduction.

Given the existence of a parser and a routine to discover the basic block structure of the program, it is trivial to locate the candidate SESE regions. Once a candidate region is located, all that is required is to check for the existence of errant *gotos*. The block can then be marked as to whether or not it is an SESE region.

This type of algorithm is also well suited for an efficient implementation in a structured editor. If the block is an SESE region, then that group of nodes is labeled with information that summarizes the region. This summation consists of a list of *Uses*, *Defs*, *Mods*, and *Undefs* in that region for each variable in the program. If that region is clear of any reference to a variable, then that region can be treated as a single node by all the search routines. If the region is not clear, then the nodes in that region must become candidates in the search in the affected area for that variable.

The benchmarks that apply to classical data flow algorithms do not apply to this method since the performance of the PVT is tied to the variable structure. The purpose of this reduction is to skip over regions of the program where the variable is not used. This technique is well suited to programs that have a high degree of data locality. This reduction does not affect the asymptotic complexity of the program, therefore a rigorous analysis of this technique is difficult.

Chapter 9

Conclusions

9.1. Areas for Future Research

9.1.1. Better Local Algorithms

There are three major weaknesses of the local algorithms presented in this dissertation. Each should be addressed by more research in the area. Of course, these are not all the weaknesses of the algorithms, but it is likely that the solution of any of these weaknesses would result in better real performance of the Partitioned Variable Technique.

- (1) *Better Edge Deletion.* The algorithm to delete edges from the graph may not be optimal since some nodes whose information does not change may be visited. It may be true that no algorithm exists that has this property of optimality.
- (2) *Recalculation of Strongly Connected Regions.* The local algorithms recalculate these regions for the affected area each time the algorithm is called. It may be possible to develop incremental algorithms to maintain these strongly connected regions and thereby avoid this recalculation.
- (3) *Better Global Algorithm.* If an algorithm were developed that was able to utilize for one variable some of the structure information developed for another variable, the global algorithm would be improved. This could also provide a speedup to the *Type II* local algorithms since each *Type II* change requires updating the solution for all variables in the program.

9.1.2. Developing Type II Algorithms For Ryder's Work

As mentioned in section 6.2, Ryder's work does not include any algorithms for handling *Type II* or *Type III* changes. Proof that efficient algorithms can be developed to solve these problems would make comparison with techniques developed here realistic.

It is also not clear on what data flow problems Ryder's technique will be effective. The examples in Ryder's dissertation are for the *Live-Dead* problem. In no place does Ryder explore behavior of her algorithm on more difficult data flow problems even though she claims correctness of her work over a wide variety of problems.

9.1.3. The Display Issue

There are two problems that become obvious with respect to the display of data flow information. The first deals with how soon and how often an error message is to be reported. The second deals with how to handle errors that are not correctable.

In structured editors that deal primarily with syntactic and semantic errors, such as the Cornell Synthesizer, the syntactic tokens exist in one of two states; either it is syntactically correct or it is null. Null tokens are used as placeholders and are treated as syntactically correct unless an attempt is made to execute them. This allows the code to be saved or even executed before it is finished. It is impossible to insert a token that is syntactically incorrect.¹

¹In the Cornell Synthesizer, a syntactically incorrect token can be inserted since expressions are entered free format. The expression is then immediately parsed. If an error is found in an expression, it must be corrected before the cursor can be moved.

This simple technique is not adequate for some logic errors. Logic errors can be the result of the subtle interaction of many program elements over a possibly wide area of the program. Take the following three statements of code as an example. (Assume that there are no flow of control changes, or assignments to or uses of x in the intervening sections of code.)

(1) $x = a$
 ...
(2) $y = x$
 ...
(3) $x = b$

Figure 9.1: Editing Display Problem

If statements (1) and (3) are entered at a time before statement (2), the editor must report that the value of x in (1) is unused. After (2) has been entered, the editor will notice that the problem has been corrected and may delete the message. The difficult question here is how to present the message to the user so that he receives the information in a timely manner and is not overburdened with misleading error messages.

Another example can be found by examining the results of a programmer who fills in the loop conditions before filling in the body of the loop. Until the loop body is completed, the loop condition variables are not likely to be modified by the statements inside the loop. This kind of error detection is useful in that it can detect simple programming errors very

early. But it needs to be done in such a manner as to not produce a message so early as to be ignored. This kind of behavior can be expected to occur fairly often for people who develop code by writing stubs in a top down manner and then filling in the stubs from the bottom up, behavior encouraged by template based editing. The messages produced would be dependent on the order of completion of the stubs. One of the worst mistakes in designing such a system is to produce so many meaningless error messages that the user will probably ignore the important ones.

One possible way to handle early editing errors is to turn off error detection until the editing process is somewhat mature. The real power of a system like this is to aid in understanding the affect of small changes in a large program. As the program nears completion, the user should then turn on message production and fix all the errors. For all future editing sessions on that section of code, the errors should be reported as they are discovered.

As stated above, the data flow information will not be precise enough to give truly reliable information. The editor will, in certain cases, warn the user of a possible error that the user can prove is of no concern. The editor must have some mechanism to allow the user to suppress a particular error message. The editor should print the path in question and allow the user to "promise" that path can never be taken.

While these are not the only problems that will require attention to make a successful display, they will probably be the most vexing. Unfortunately, these are not the type of problems that have clearly defined

algorithmic solutions. While it is possible to consider many alternatives, the best choice can only be made after understanding the weaknesses of several bad solutions.

9.1.4. More Complex Analysis

More types of error analysis could be performed. The search for new classes of error analysis could take several different paths:

- (1) *More Sophisticated Techniques.* These new techniques could include, but should not be limited to limited forms of symbolic interpretation that could prove certain paths are not executable [HoRo 80].
- (2) *Construct-Specific Error Detection.* Several program constructs have some redundancy in their coding. Inconsistencies in these redundant features can be found and may indicate the existence of errors in the code. An example of this type of redundancy was given in the errant loop example of section 3.1.

There are some problems with doing more complex analysis. The analysis may be so time consuming that it is impractical to perform in an interactive environment. Of course what is impractical with the hardware of today may be practical with the next generation of hardware.

There appears to be a direct relation between the sophistication of the error analysis and the locality of the error. As the errors searched for become more subtle, larger areas of the program can be expected to contribute to the analysis. As the sophistication of the analysis increases, the problems with the presentation of these errors, as discussed in the previous section, would increase.

9.1.5. Integration With Universal Data Flow Analysis Algorithms

If the PVT algorithms were integrated with algorithms for interprocedural data flow analysis, the precision of the information produced would be improved. Without this universal information, very conservative estimates must be used at procedure boundaries. The integration could possibly proceed using the same three step method used in *Dave*.² Algorithms to produce and incrementally update data flow information across procedure boundaries have been proposed by Cooper [Coop 83].

9.2. Conclusions

In this dissertation, a new algorithm has been presented to perform data flow analysis over a limited set of problems. The algorithm is capable of building *Use-Definition Chains* and solving many similar problems. A precise characterization of problems solved has also been demonstrated. The asymptotic complexity of the global algorithm is linear in the number of edges in the program flow graph.

An incremental version of this algorithm has also been given. The suitability of this algorithm to detect many common programming errors in a high level program editor has also been explored.

The complexity of the incremental algorithms fall into two categories. In one category, the algorithm must visit only those nodes where the data flow solution must actually change. In the second category, a slightly larger number of nodes must be visited, but this number should be less than the number of nodes in the graph. In both categories, each node must

²See section 3.2.

be visited a fixed number of times for each in-edge, thereby retaining the linear complexity of the algorithm.

Unlike many other algorithms for data flow analysis, this algorithm, in both the incremental and global versions, is unaffected by any complexity in structure in the program flow graph.

Bibliography

- [AHU1 74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
- [Alle 70] F. E. Allen. Control flow analysis. *Sigplan Notices*, July 1970.
- [BaJa 78] W. A. Babich and Mehdi Jazayeri, The method of attributes for data flow analysis (two parts). *Acta Informatica* 10 1978, 245-272. See also Tr-77-007, Department of Computer Science, University of North Carolina.
- [BDLS 80] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sawyer. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. *Conference Record of the Seventh ACM Symposium on Principles of Programming Languages*, January 1980, pp. 220-233.
- [BoMo 79] R. S. Boyer and J. S. Moore. *A Computational Logic*, Academic Press, 1979.
- [BoSo 83] J. Bonar and E. Soloway. Uncovering principles of novice programming. *Conference Record of the Tenth ACM Symposium on Principles of Programming Languages*, January 1983, pp. 10-13.
- [Broo 79] F. P. Brooks Jr. *The Mythical Man Month*, Addison Wesley, 1979.
- [ClRi 83] L. A. Clarke and D. J. Richardson. The application of error-sensitive testing strategies to debugging. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, August 1983, pp 45-52.
- [Coop 83] K. D. Cooper. Interprocedural data flow analysis in a programming environment. PhD. dissertation, Mathematical Sciences Department, Rice University, Houston, Texas, 1983.
- [CoSc 70] J. Cocke and J. T. Schwartz. *Programming Languages and Their Compilers; Preliminary Notes*, New York: Courant Institute of Mathematical Sciences, New York University, 1970.
- [DHKL 75] V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, and J. J. Levy. A structure oriented program editor: A first step toward computer assisted programming. *International Computing Symposium 1975*, Elsevier North-Holland, 1975, pp 113-120.

- [Ehrm 79] J. R. Ehrman. The babel of application development tools. *Proceedings of the Application Development Symposium of SHARE*, October 1979, pp 31-39.
- [FoOs 75] L. D. Fosdick and L. D. Osterweil. Data flow analysis in software reliability. *Computing Surveys*, Vol 8, No 3, September 1976, pp. 305-330.
- [GrWe 76] S. L. Graham and M. Wegman. A fast and usually linear algorithm for global flow analysis. *Journal of the Association for Computing Machinery*, Vol. 23, No. 1, January 1976, pp. 172-202.
- [HaNo 82] A. N. Habermann and D. S. Notkin. The gandalf software development environment. research report, Computer Science Department, Carnegie-Mellon University, Pittsburg, Pa., January 1982.
- [Hech 77] M. S. Hecht. *Flow Analysis of Computer Programs*, Elsevier North-Holland, 1977.
- [HoKe 83] R. T. Hood and K. W. Kennedy. A programming environment for Fortran. Research report, Department of Mathematical Sciences, Rice University, Houston, Texas, 1983.
- [HoRo 80] L. H. Holley and B. K. Rosen. Qualified data flow problems. *Conference Record of the Seventh ACM Symposium on Principles of Programming Languages*, January 1980, pp. 68-82.
- [Huan 79] J. C. Huang. Detection of data flow anomaly through program instrumentation. *IEEE Transactions on Software Engineering*, Vol. Se-5, No. 3, May 1979, pp. 226-236.
- [IBM 76] OS PL/I Optimizing Compiler: programmer's guide. IBM SC33-0006-4, 1976.
- [John 78] S. C. Johnson. Lint, a C Program Checker. Berkley Unix documentation volume 2c. July 1978.
- [KaUl 76] J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the Association for Computing Machinery*, Vol. 23, No. 1, January 1976, pp. 158-171.
- [KaUl 77] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica* 7, Springer-Verlag 1977, pp. 305-317.

- [Kenn 75] K. W. Kennedy. Node listing applied to data flow analysis. *Conference Record of the Second ACM Symposium on Principles of Programming Languages*, January 1975, pp. 10-21.
- [Kenn 81] K. W. Kennedy. A survey of data flow analysis techniques. *Program Flow Analysis*, edited by S. S. Muchnick, N. D. Jones. Prentice-Hall, 1981.
- [KeZu 77] K. W. Kennedy and L. Zucconi. Applications of a graph grammar for program control. *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, January 1977, pp. 72-85.
- [Kild 73] G. A. Kildall. A unified approach to global program optimization. *Conference Record of the First ACM Symposium on Principles of Programming Languages*, October 1973, pp. 194-206.
- [Knut 71] D. E. Knuth. An emperical study of Fortran programs. *Software Practice and Experience*, 1:12, pp. 105-134.
- [Knut 74] D. E. Knuth, *The Art of Computer Programming Vol. I. Fundamental Algorithms*, 2nd Edition, Addison-Wesley, 1974.
- [Kou 77] L. T. Kou. On live-dead analysis for global data flow problems. *Journal of the Association for Computing Machinery*, Vol. 24, No. 3, July 1977, pp. 473-483.
- [Kuck 78] D. J. Kuck. *The Structure of Computers and Computations*, John Wiley & Sons, 1978.
- [Masi 80] L. M. Masinter. Global program analysis in an interactive environment. Xerox Report SSL-80-1, January 1980.
- [MevD 82] N. Meyrowitz and A. van Dam. Interactive editing systems (two parts). *Computing Surveys*, Vol. 14, No. 3, September 1982, pp 321-415.
- [MuJo 81] S. S. Muchnick and N. D. Jones. *Program Flow Analysis*, Prentice-Hall, 1981.
- [Rose 77] B. K. Rosen. High level data flow analysis. *Communications of the ACM*, Vol. 20, No. 10, October 1977, pp. 712-724.
- [Rose 81] B. K. Rosen. Linear cost is sometimes quadratic. *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages*, January 1981, pp. 117-124.

- [RuMo 81] A. Rudmic and B. Moore. The Chill compiling system: Towards a Chill programming environment. *Proceedings of the Fourth International Conference of Software Engineering for Telecommunications Software*, 1981, pp. 187-190.
- [Ryde 82] B. G. Ryder. *Incremental Data Flow Analysis Based on a Unified Model of Elimination Algorithms*. PhD dissertation, DCS-TR-117, Rutgers University, September 1982.
- [Tarj 72] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing* 1:2, 1972, pp 146-160.
- [Tarj 75] R. E. Tarjan. Application of path compression on balanced trees. Technical report STAN-75-512 (1975), Computer Science Department, Stanford University, Stanford, Ca.
- [Teit 77] W. Teitelman. A display-oriented programmer's assistant. *Proceedings of the Fifth International Joint Conference of Artificial Intelligence*, 1977, pp 905-915.
- [TeRe 81] R. T. Teitelbaum and T. Reps. The Cornell Program Synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24-9, September 1981, 563-573.
- [Ullm 73] J. D. Ullman. Fast algorithms for the elimination of common subexpressions. *Acta Informatica* 2, Springer-Verlag 1973, pg 191-213.
- [Wegm 83] M. Wegman. Summarizing graphs by regular expressions. *Conference Record of the Tenth ACM Symposium on Principles of Programming Languages*, January 1983, pp. 203-216.
- [Wein 71] G. M. Weinberg. *The Psychology of Computer Programming: Essays on Software Engineering*, Van Nostrand Reinhold, 1979.
- [Will 79] G. Williams. Program checking. *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages*, January 1979, pp. 13-25.