A Generalized Theory of Bit Vector Data Flow Analysis

UDAY P. KHEDKER and DHANANJAY M. DHAMDHERE
Indian Institute of Technology

The classical theory of data flow analysis, which has its roots in unidirectional flows, is inadequate to characterize bidirectional data flow problems. We present a generalized theory of bit vector data flow analysis which explains the known results in unidirectional and bidirectional data flows and provides a deeper insight into the process of data flow analysis. Based on the theory, we develop a worklist-based generic algorithm which is uniformly applicable to unidirectional and bidirectional data flow problems. It is simple, versatile and easy to adapt for a specific problem. We show that the theory and the algorithm are applicable to all bounded monotone data flow problems which possess the property of the separability of solution.

The theory yields valuable information about the complexity of data flow analysis. We show that the complexity of worklist-based iterative analysis is same for unidirectional and bidirectional problems. We also define a measure of the complexity of round-robin iterative analysis. This measure, called *width*, is uniformly applicable to unidirectional and bidirectional problems and provides a tighter bound for unidirectional problems than the traditional measure of *depth*. Other applications include explanation of isolated results in efficient solution techniques and motivation of new techniques for bidirectional flows. In particular, we discuss edge-splitting, edge placement and develop a feasibility criterion for decomposition of a bidirectional flow into a sequence of unidirectional flows.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers*; *Optimization*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—complexity of proof procedures

General terms: Algorithms, Theory

Additional Key Words and Phrases: Data flow frameworks, data flow analysis, bidirectional data flows

1. INTRODUCTION

Data flow analysis is the process of collecting information about the uses and definitions of data items in a program. This information is put to a variety of uses, *viz.* program design, debugging, optimization, maintenance

Authors' address: Department of Computer Science & Engineering, Indian Institute of Technology, Bombay 400 076, India. Email: uday@cse.iitb.ac.in, dmd@cse.iitb.ac.in
A near look-alike copy of the paper published in ACM Transactions on Programming Languages and Systems, Vol. 16, No. 5, September 1994, Pages 1472-1511.

and documentation. Compilers typically use data flow analysis to collect information for the purpose of code optimization.[1]

Data flows used in code optimization mostly involve *unidirectional* dependencies, i.e. the data flow information available at a node in the program flow graph is influenced either by its predecessors or by its successors. Such data flows can be readily classified into *forward* and *backward* data flows [Aho, Sethi, and Ullman 1986]. In *bidirectional* problems, the information available at a node depends on its predecessors as well as its successors. The Morel and Renvoise Algorithm for partial redundancy elimination [Morel and Renvoise 1979] (also called MRA), is a representative bidirectional problem. The advantage of bidirectional problems is that they unify several optimizations reducing both the size and the running time of an optimizer. For example, MRA unifies the traditional optimizations of code movement, common subexpression elimination, and loop optimization. The Composite Hoisting and Strength Reduction Algorithm [Joshi and Dhamdhere 1982a; Joshi and Dhamdhere 1982b] unifies code movement, strength reduction and loop optimization.

Though bidirectional data flow problems have been known for over a decade, it has not been possible to explain the intricacies of bidirectional flows using the traditional theory of data flow analysis. Although a fixed point solution for a bidirectional problem exists, the flow of information and the safety of an assignment can not be characterized formally. Because of this theoretical lacuna, efficient solutions to bidirectional problems have not been found though some isolated and ad hoc results have been obtained [Dhamdhere 1988a; Dhamdhere and Patil 1993; Dhamdhere, Rosen, and Zadeck 1992].

In this paper we present a theory which handles unidirectional as well as bidirectional data flow problems uniformly. Apart from explaining the known results in unidirectional and bidirectional flows, it provides deeper insights into the process of data flow analysis. Though the exposition of the theory is based on the *bit vector problems*, the theory is applicable to all bounded monotone data flow problems which possess the property of separability of solution. Several proofs have been omitted from the paper for brevity; they can be found in [Khedker and Dhamdhere 1992].

Section 2 introduces MRA which is used as a representative example throughout the paper. Section 3 reviews the classical theory of data flow analysis. Section 4 defines bit vector problems formally, generalizes the traditional concepts and provides generic data flow equations which facilitate uniform specification of data flow problems. A worklist-based generic algorithm is developed in section 5. Arising out of a generalized theory, it is uniformly applicable to unidirectional and bidirectional data flow problems. This section also analyses the performance of the generic algorithm and shows that the complexity of the worklist-based iterative analysis is same for unidirectional and bidirectional problems.

[1]Data flow analysis can be either *inter-procedural* or *intra-procedural*. We restrict ourselves to the latter in this paper except that the inter-procedural information at the entry/exit of a procedure is considered for analyzing data flows within the procedure.

Section 6 discusses several applications of the generalized theory in the complexity of data flow analysis. A new measure called the *width* (w) of a graph for a data flow framework is defined which is shown to bound the number of iterations of round-robin analysis for unidirectional and bidirectional problems. We show that the width provides a tighter bound for unidirectional problems than the traditional measure of *depth*. This section also explains several known results in the solution of bidirectional data flows, *viz.* edge-splitting, edge placement and develops a feasibility criterion for decomposition of bidirectional flows into a sequence of unidirectional flows.

Section 7 discusses the significance and applicability of the results presented in this paper.

2. BIDIRECTIONAL DATA FLOWS : AN EXAMPLE

This section introduces the Morel and Renvoise Algorithm (MRA) [Morel and Renvoise 1979] for partial redundancy elimination which is used as a representative bidirectional problem throughout the paper.

MRA unifies the traditional optimizations of code movement, common subexpression elimination, and loop optimization. The importance of the MRA framework lies in the fact that unification of many classical optimizations reduces the size as well as the running time of an optimizer; a 35% reduction in the size and a 30% to 70% reduction in the execution cost has been reported in the literature [Morel and Renvoise 1979]. It has been implemented in at least two important production compilers (MIPS and PL.8) and has inspired several other unifications [Chow 1988; Dhamdhere 1988a; Joshi and Dhamdhere 1982a; Joshi and Dhamdhere 1982b].

The data flow properties and the data flow equations for MRA are given in Figure 1. Note that PPIN_i is the bit vector for node i which represents the property PPIN for *all expressions*, whereas PPIN_i^l is the bit representing the expression e_l.

Local property ANTLOC_i^l represents *local anticipability*, i.e. existence of an upwards exposed expression e_l in node i, while TRANSP_i^l reflects *transparency*, i.e. the absence of definition(s) of any operand(s) of e_l in the node. The global property of *anticipability* ($\text{ANTIN}_i^l/\text{ANTOUT}_i^l$) indicates whether expression e_l is very busy at the entry/exit of node i — a necessary and sufficient condition for the safety of placing an evaluation of e_l at the entry/exit of the node [Kennedy 1972]. Equations 1 and 2 do not use $\text{ANTIN}_i^l/\text{ANTOUT}_i^l$ properties explicitly; they are implied by $\text{PPIN}_i^l/\text{PPOUT}_i^l$ properties. The data flow property of *availability* ($\text{AVIN}_i^l/\text{AVOUT}_i^l$) is computed using the classical forward data flow problem [Aho, Sethi, and Ullman 1986]. The partial redundancy of an expression is represented by the *partial availability* of the expression (PAVIN_i^l) at the entry of node i. PPIN_i^l indicates the feasibility of placing an evaluation of e_l at the entry of i while PPOUT_i^l indicates the feasibility of placing it at the exit. Computations of an expression e_l are inserted at the exit of node i if $\text{INSERT}_i^l = \mathbf{T}$. REDUND_i^l indicates that the upwards exposed occurrence of e_l in node i is redundant and may be deleted.

LOCAL DATA FLOW PROPERTIES :

ANTLOC_i^l	Node i contains a computation of e_l, not preceded by a definition of any of its operands.
COMP_i^l	Node i contains a computation of e_l, not followed by a definition of any of its operands.
TRANSP_i^l	Node i does not contain a definition of any operand of e_l.

GLOBAL DATA FLOW PROPERTIES :

$\text{AVIN}_i^l/\text{AVOUT}_i^l$	e_l is available at the entry/exit of node i.
$\text{PAVIN}_i^l/\text{PAVOUT}_i^l$	e_l is partially available at the entry/exit of node i.
$\text{ANTIN}_i^l/\text{ANTOUT}_i^l$	e_l is anticipated at the entry/exit of node i.
$\text{PPIN}_i^l/\text{PPOUT}_i^l$	Computation of e_l may be placed at the entry/exit of node i.
INSERT_i^l	Computation of e_l should be inserted at the exit of node i.
REDUND_i^l	First computation of e_l existing in node i is redundant.

DATA FLOW EQUATIONS :

$$\text{PPIN}_i = \text{PAVIN}_i \cdot (\text{ANTLOC}_i + \text{TRANSP}_i \cdot \text{PPOUT}_i) \cdot \prod_{j \in pred(i)} (\text{AVOUT}_j + \text{PPOUT}_j) \tag{1}$$

$$\text{PPOUT}_i = \prod_{k \in succ(i)} (\text{PPIN}_k) \tag{2}$$

$$\text{INSERT}_i = \text{PPOUT}_i \cdot \neg\text{AVOUT}_i \cdot (\neg\text{PPIN}_i + \neg\text{TRANSP}_i)$$

$$\text{REDUND}_i = \text{PPIN}_i \cdot \text{ANTLOC}_i$$

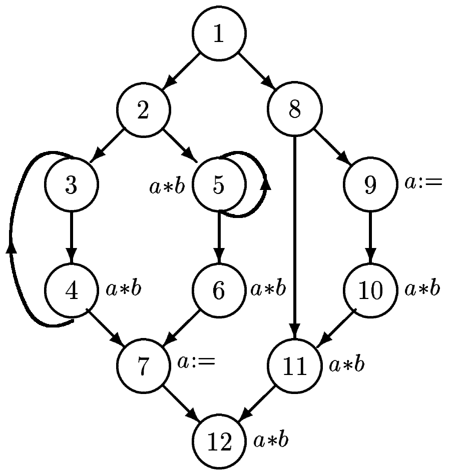
Fig. 1. The Morel-Renvoise Algorithm

The PPIN_i equation is slightly different from the original equation in MRA; the term $\text{ANTIN}_i \cdot (\text{PAVIN}_i + \neg\text{ANTLOC}_i \cdot \text{TRANSP}_i)$ in the original MRA equations is replaced by the term PAVIN_i to prohibit redundant hoisting when the expression is not partially available. The PAVIN_i term represents the *profitability* of hoisting in that there exists at least one possible execution path along which the expression is computed more than once. The other two terms in the PPIN_i equation represent the *feasibility* of hoisting.

Bidirectional dependencies of MRA arise as follows : *Redundancy* of an expression is based on the notion of availability of the expression which gives rise to forward data flow dependencies (reflected by the $\prod$ term in the PPIN_i equation). The *safety* of code movement is based on the notion of anticipability of the expression which introduces backward dependencies in the data flow problem (reflected by the $\prod$ term in the PPOUT_i equation).

Example 1. Consider the program flow graph in Figure 2. The partial redundancy elimination performed by MRA subsumes the following three optimizations :

—*Loop Invariant Movement*: The computations of $a * b$ in node 4 and node



Node	Transp	Antloc	Pavin	Avout	Ppin	Ppout	Insert	Redund
1	T	F	F	F	F	F	F	F
2	T	F	F	F	F	T	T	F
3	T	F	T	F	T	T	F	F
4	T	T	T	T	T	F	F	T
5	T	T	T	T	T	T	F	T
6	T	T	T	T	T	F	F	T
7	F	F	T	F	F	T	T	F
8	T	F	F	F	F	F	F	F
9	F	F	F	F	F	F	F	F
10	T	T	F	T	F	F	F	F
11	T	T	T	T	F	T	F	F
12	T	T	T	T	T	F	F	T

Fig. 2. Program flow graph and properties for example 1

5 are hoisted out of the loops and are inserted in node 2 (REDUND$^l_4$, REDUND$^l_5$, and INSERT$^l_2$ are **T**).

—*Code Hoisting* : The partially redundant computation of $a*b$ in node 12 is hoisted to node 7. As a result of suppressing this partial redundancy, the path 1-8-11-12 would have only *one* computation of $a*b$; the unoptimized program has two.

—*Common Subexpression Elimination* : The totally redundant computation of $a*b$ in node 6 is deleted as an instance of common subexpression elimination.

Note that the partially redundant computation $a*b$ in node 11 is not suppressed since hoisting it to node 8 would be unsafe — the path 1-8-9 had no computation of $a*b$ in the original program. □

- BASIC LOAD STORE INSERTION ALGORITHM (LSIA) [Dhamdhere 1988b]

$$\begin{aligned}
\mathrm{SPPIN}_i &= \prod_{j \in pred(i)} (\mathrm{SPPOUT}_j) \\
\mathrm{SPPOUT}_i &= \mathrm{DPANTOUT}_i \cdot (\mathrm{DCOMP}_i + \mathrm{DTRANSP}_i \cdot \mathrm{SPPIN}_i) \cdot \\
&\quad \prod_{k \in succ(i)} (\mathrm{DANTIN}_k + \mathrm{SPPIN}_k)
\end{aligned}$$

- COMPOSITE HOISTING AND STRENGTH REDUCTION ALGORITHM (CHSA) [Joshi and Dhamdhere 1982a; Joshi and Dhamdhere 1982b]

$$\begin{aligned}
\mathrm{NOCOMIN}_i &= \mathrm{CONSTA}_i \cdot \mathrm{NOCOMOUT}_i + \\
&\quad \sum_{j \in pred(i)} \mathrm{CONSTB}_i \cdot \mathrm{NOCOMOUT}_j \\
\mathrm{NOCOMOUT}_i &= \mathrm{CONSTC}_i + \mathrm{CONSTD}_i \cdot \mathrm{NOCOMIN}_i + \\
&\quad \sum_{k \in succ(i)} \mathrm{CONSTE}_i \cdot \mathrm{NOCOMIN}_k
\end{aligned}$$

Fig. 3. Data flow equations of some other bidirectional problems

Example 2. Bidirectional data flows have also been used in register assignment and strength reduction optimizations. Figure 3 presents the data flow equations of two such algorithms. The SPPIN/SPPOUT problem of LSIA performs sinking of STORE instructions using partial redundancy elimination techniques [Dhamdhere 1988b]. The NOCOMIN/NOCOMOUT problem of CHSA is used to inhibit the placement of an update computation following a high strength computation [Joshi and Dhamdhere 1982a; Joshi and Dhamdhere 1982b]. □

3. NOTIONS FROM CLASSICAL DATA FLOW ANALYSIS

This section presents an overview of the classical theory of data flow analysis and compares various solution methods and their complexities. Our description is based mostly on Graham and Wegman [1976]; Hecht [1977]; Marlowe and Ryder [1990]. A more detailed treatment can be found in Aho, Sethi, and Ullman [1986]; Graham and Wegman [1976]; Hecht [1977]; Kam and Ullman [1977]; Kildall [1973]; Marlowe and Ryder [1990]; and Rosen [1980]. The concluding part of this section motivates the need for a more general setting.

3.1 Preliminaries

A data flow framework is defined as a triple $\mathbf{D} = <\mathcal{L}, \sqcap, \mathcal{F}>$ (Figure 4). Elements in $\mathcal{L}$ represent the information associated with the entry/exit of a basic block. $\sqcap$ is the set union or intersection operation which determines the way the global information is combined when it reaches a basic block. A function $f_i \in \mathcal{F}$ represents the effect on the information as it flows through

DATA FLOW FRAMEWORK : $\mathbf{D}$

$\mathbf{D} = <\mathcal{L}, \sqcap, \mathcal{F}>$, where

- ▷ $<\mathcal{L}, \sqcap>$ is a semilattice such that :
  - $\mathcal{L}$ is a partially ordered set (often finite).
  - $\sqcap$ is a binary meet operation which is commutative, associative, and idempotent.
  - The partial order (denoted $\sqsubseteq$) is reflexive, antisymmetric, and transitive.
    $\forall\, a, b \in \mathcal{L} : a \sqsubseteq b$ *iff* $a \sqcap b = a$
  - There are two special elements *top* (denoted $\top$) and *bottom* (denoted $\bot$).[a]
    $$\forall\, a \in \mathcal{L} : \begin{array}{l} a \sqcap \top = a \\ a \sqcap \bot = \bot \end{array}$$
  - $\mathcal{L}$ has finite height (i.e. length of every strictly descending chain $a \sqsubset b \sqsubset \cdots \sqsubset z$ is finite).
    If the length of every strictly descending chain is bounded by a constant, say H, we say that $\mathcal{L}$ has *strictly finite height*, or simply *height* H.
- ▷ $\mathcal{F} \subseteq \{f : \mathcal{L} \to \mathcal{L}\}$ is a class of functions such that :
  - $\mathcal{F}$ contains an identity function $\imath$.
    $\forall\, a \in \mathcal{L},\ \imath(a) = a$
  - $\mathcal{F}$ is closed under composition.
    $\forall\, f_1, f_2 \in \mathcal{F} :\ f_1 \circ f_2 \in \mathcal{F}$
  - $\forall\, a \in \mathcal{L},\ \exists f \in \mathcal{F}$ such that $a = f(\bot)$
- ▷ $\mathbf{D}$ is *monotone* if and only if :
  $\forall\, a, b \in \mathcal{L}, \forall\, f \in \mathcal{F} :\ f(a \sqcap b) \sqsubseteq f(a) \sqcap f(b)$. This is same as
  $a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$
- ▷ $\mathbf{D}$ is *distributive* if and only if :
  $\forall\, a, b \in \mathcal{L}, \forall\, f \in \mathcal{F} :\ f(a \sqcap b) = f(a) \sqcap f(b)$
- ▷ $\mathbf{D}$ is *k-bounded* if and only if :
  $$\forall\, f \in \mathcal{F},\ \exists k : \prod_{i=0}^{k-1} f^i = f^* \text{ where,}$$
  $f^{j+1} = f \circ f^j$, $f^0 = \imath$, and f^* denotes $f^0 \sqcap f^1 \sqcap f^2 \sqcap \cdots$

INSTANCE OF A DATA FLOW FRAMEWORK : $\mathbf{I}$

$\mathbf{I} = <G, M>$, where

- ▷ $G = <N, E, n_0>$ is a control flow graph where N is the set of nodes representing basic blocks, E is the set of edges, and n_0 is a unique entry node with in-degree zero.
- ▷ $M : N \to \mathcal{F}$ maps the nodes in N to functions in $\mathcal{F}$. It is extended to paths as follows :
  - If $\rho = (n_0, n_1, \cdots, n_i)$ is a *path* in G then
    $M(\rho) = M(n_{i-1}) \circ \cdots \circ M(n_1) \circ M(n_0)$
  - If ρ is a null path then $M(\rho)$ is an identity function.

[a] In some cases $\top$ may not exist. However, it can always be added artificially. Such a $\top$ may not be a natural element of $\mathcal{L}$ but it helps in performing data flow analysis.

Fig. 4. Data Flow Framework

basic block i.[2]

A data flow framework is characterized by any or all of the following :

—Algebraic properties of functions in $\mathcal{F}$ (*viz.* monotonicity, distributivity, continuity etc. [Hecht 1977]).

—Finiteness properties of functions in $\mathcal{F}$ (*viz.* boundedness [Marlowe and Ryder 1990], fastness [Graham and Wegman 1976], rapidity [Kam and Ullman 1977] etc.).

—Finiteness properties of $\mathcal{L}$ (*viz.* height [Hecht 1977; Marlowe and Ryder 1990]).

—Partitionability properties of $\mathcal{L}$ and $\mathcal{F}$ [Zadeck 1984].

There is an important subclass of k-bounded partitionable problems[3] called the *bit vector problems* [Hecht 1977] which has been extensively discussed in the literature [Dhamdhere, Rosen, and Zadeck 1992; Hecht 1977; Marlowe and Ryder 1990; Muchnick and Jones 1981], though it is defined only informally (*viz.* in Hecht [1977] and Zadeck [1984]). We provide a formal definition in section 4.1 and use it in the exposition of our theory.

3.2 Data Flow Equations

To formulate a data flow problem, the data flow properties associated with each node of the flow graph are represented as variables which, as noted earlier, are elements in $\mathcal{L}$. Interdependencies of the values of these variables give rise to simultaneous equations. Thus, solving a data flow problem reduces to solving a system of simultaneous equations.

A data flow problem is posed as a pair $<Q, X_0>$, where Q is a system of equations parameterized by the nodes of the flow graph and whose terms may include constants. These constants may represent information derived from other data flow problems. $X_0 : N \rightarrow \mathcal{L}$ is a conservative initialization. For the entry node it is usually, though not always, $\perp$. For the non-entry nodes, such an estimate is almost always $\top$ and is needed in the case of iterative methods only.

Let *pred*(i) and *succ*(i) denote the set of predecessors and successors of node i. The equations $X = Q(Y)$ have the following form[4] :

$$\text{IN}(i) = \begin{cases} X_0(n_0) & \text{if } i = n_0 \\ \displaystyle\prod_{j \in pred(i)} M(j)(\text{IN}(j)) & \text{otherwise} \end{cases} \tag{3}$$

Note that the equations may well be written in terms of information at the node exit [Ryder and Paull 1986]. Alternatively, both IN and OUT may

[2]Alternatively, the functions can be associated with in-edges(out-edges) of node i for forward(backward) flow problems.

[3]We use the terms *data flow problem* and *data flow framework* interchangeably, though the latter is more formal.

[4]Though we present the definitions for forward problems only, analogous definitions exist for backward problems.

be used. Further, the function M may be dropped and the node numbers may be used as subscripts of $f \in \mathcal{F}$ as shown below.

$$\mathrm{IN}_i = \begin{cases} X_0(n_0) & \text{if } i = n_0 \\ \displaystyle\bigsqcap_{j \in pred(i)} (\mathrm{OUT}_j) & \text{otherwise} \end{cases} \tag{4}$$

$$\mathrm{OUT}_i = f_i(\ \mathrm{IN}_i\ ) \tag{5}$$

We use this form in the paper.

3.3 Solutions of a Data Flow Problem

The solution of a data flow problem is an assignment of values $X : N \to \mathcal{L}$ to the nodes of the flow graph.

An assignment SA is *safe* if the information at a node does not exceed the information that can be gathered along any path from n_0 to that node [Graham and Wegman 1976], i.e.

$$\forall\, i \in N \ :\ \mathrm{SA}(i) \sqsubseteq M(\rho)(X_0(n_0))$$

where ρ is a path from n_0 to i. A safe assignment guarantees the correctness of optimizations; an unsafe assignment may result in semantics changing optimizations.

The *Meet Over Paths* solution of a data flow problem represents the information reaching a basic block along *all possible* program paths [Aho, Sethi, and Ullman 1986; Hecht 1977; Marlowe and Ryder 1990]. Let *paths*(i) denote the set of all paths from n_0 to i. Then,

$$\forall\, i \in N \ :\ \mathrm{MOP}(i) = \bigsqcap_{\rho \in paths(i)} M(\rho)(X_0(n_0))$$

Note that MOP is the maximum safe assignment.

An assignment FP is a *fixed point* of an instance of a data flow framework [Graham and Wegman 1976] if :

$$\forall\, i \in N - \{n_0\},\ \forall\, j \in pred(i) \ :\ \mathrm{FP}(i) \sqsubseteq M(j)(\mathrm{FP}(j))$$

and $\mathrm{FP}(n_0) = X_0(n_0)$. It is easy to see that FP is a fixed point of equation 3.

A fixed point guarantees the consistency of information associated with the nodes of the flow graph. A *Maximum Fixed Point*, MFP, contains all other fixed points. It can be shown that MFP is contained in MOP. Thus, every fixed point is a safe assignment, though not vice-versa.

An assignment X is *acceptable*[5] if and only if it is safe and contains all fixed points of Q. For the monotone data flow problems, MOP and MFP typically exist. For the distributive problems, MFP is always equal to MOP. Since MFP represents the maximum information that can be gathered in practice, *the goal of data flow analysis can also be defined as finding MFP.* Though this means that we may not be able to capture all information for the non-distributive problems, it does not matter since no algorithm capable of

[5]Marlowe and Ryder [1990] use this term for a slightly different notion; we follow Graham and Wegman [1976].

computing MOP for all instances of arbitrary monotone data flow problems exists anyway [Kam and Ullman 1977].

3.4 Performing Data Flow Analysis

There are two broad categories of the approaches to data flow analysis : *iterative* methods and *elimination* methods.

The iterative method of data flow analysis solves the system of equations by initializing the node variables to some conservative values and successively recomputing them till a fixed point is reached. The *round-robin* version recomputes the data flow properties of all the nodes repeatedly, till the values stabilize. If the size of the bit vector is r, there are $O(n{\cdot}r)$ properties. Thus, in the worst case $O(n \cdot r)$ iterations may be needed. Each iteration involves computation of the properties for n nodes. If all the r bits can be processed in one step, the complexity becomes $O(n^2{\cdot}r)$. In the unidirectional problems the flow is in one direction only, hence the nodes can be visited in the postorder or reverse postorder depending upon the direction. Thus $d{+}2$ iterations are sufficient, where d is the depth[6] of the flow graph [Aho, Sethi, and Ullman 1986; Hecht 1977]. Hence the complexity is $O((d{+}2) \cdot n \cdot r)$. The *worklist* version visits the nodes selectively and involves $O(n{\cdot}r)$ work [Hecht 1977].

Elimination methods reduce the amount of effort required to solve a data flow problem by utilizing the structural properties of a flow graph [Allen and Cocke 1977; Graham and Wegman 1976; Hecht 1977; Muchnick and Jones 1981; Tarjan 1981a]. The flow graph is reduced to one node by successive applications of graph transformations which use *graph parsing* or *graph partitioning* to identify regions to obtain a derived graph. The data flow properties of a node in a region are determined from the data flow properties of the region's header node. This enables delayed substitution of some values in the simultaneous equations. For unidirectional flow problems, these methods are typically $O(\mathrm{N})$, where N is the total number of nodes in the sequence of reduced graphs. A comparison of various elimination methods appears in Ryder and Paull [1986]. It has been shown that the elimination methods cannot be extended to general bidirectional data flow problems, though they have been used to solve a restricted class of bidirectional problems [Dhamdhere and Patil 1993].

For programs with $r = O(n)$, all the r bits can not be processed in one step; processing the r bits of a bit vector would itself require $O(n)$ steps. Hence the bounds on the iterative methods are $O(n^4)$, $O(n^2 \cdot (d{+}2))$ and $O(n^2)$, while the elimination methods are $O(N \cdot n)$.

3.5 Limitations of the Classical Theory

The limitations of the classical theory of data flow analysis are easy to trace. It is based on strictly unidirectional flow — information reaches one end of a basic block, flows through it, and emanates from the other end. As a consequence, the information flows from a node either to its predecessors or

[6]Not to be confused with the *nesting depth.*

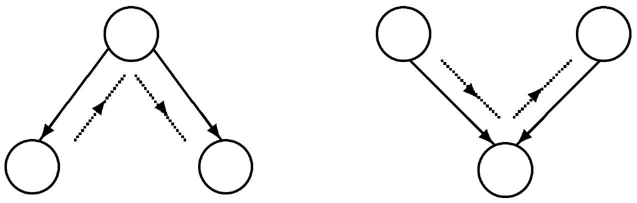


Fig. 5. Sibling and spouse effects

its successors.

In bidirectional problems, apart from the above flows, the following kinds of information flow may exist (refer to Figure 5) :

—information at one successor of a node may influence the information at another successor of the same node, and

—information at one predecessor of a node may influence the information at another predecessor of the same node.

We term these as the *sibling effect* and the *spouse effect* respectively. (Two nodes are *siblings* if they have a common predecessor; we call them *spouses* if they have a common successor.) The traditional theory fails to characterize these flows.

Example 3. In example 1, $\text{ANTLOC}_9^l = \text{TRANSP}_9^l = \mathbf{F}$, consequently PPIN_9^l becomes $\mathbf{F}$. This makes $\text{PPOUT}_8^l = \mathbf{F}$ which sets $\text{PPIN}_{11}^l = \mathbf{F}$. This flow from the entry of 9 to the entry of 11 is an example of the sibling effect. $\text{PPOUT}_8^l = \mathbf{F}$ makes $\text{PPOUT}_{10}^l = \mathbf{F}$ via PPIN_{11}^l; this is an example of the spouse effect. □

4. A GENERALIZED THEORY OF DATA FLOW ANALYSIS

We define the bit vector frameworks, generalize the notions of edge, node, and path flow functions, characterize the safety of an assignment, and propose generic data flow equations.

4.1 Preliminary Concepts

4.1.1 *Traversals.* We define a flow graph by $G = < N, E, \mathit{entry}(G), \mathit{exit}(G) >$ where $\mathit{entry}(G)$ and $\mathit{exit}(G)$ denote the (non-null) sets of entry and exit nodes, i.e. nodes with zero in-degree and zero out-degree, respectively.

A *program point* refers to the entry/exit of a basic block. For a basic block i, its entry and exit points are denoted by $\mathit{in}(i)$ and $\mathit{out}(i)$ respectively. Two program points are *neighbours* if they are adjacent in G. Thus $\mathit{in}(i)$ is a neighbour of $\mathit{out}(j)$ where $j \in \mathit{pred}(i)$. By definition, $\mathit{in}(i)$ is a neighbour of $\mathit{out}(i)$. The neighbour relations are symmetric.

Given a depth first spanning tree of G, we differentiate between *back edges* and non-back edges; we term the latter as *forward edges.*[7] Thus the term

[7]We use the terms *forward edges* and *back edges* as synonyms of *advancing edges* and

forward edges, as used in this paper, includes the conventional notions of forward as well as cross edges [Aho, Sethi, and Ullman 1986]. A *forward traversal* along an edge is a tail-to-head traversal of the edge, while a *backward traversal* is a head-to-tail traversal. We use the following notations :

—T_e^f/T_e^b: Forward/backward traversal along an edge.

—T_f^f/T_f^b: Forward/backward traversal along a forward edge.

—T_b^f/T_b^b: Forward/backward traversal along a back edge.

A forward edge traversal indicates traversal *along* the direction of control flow whereas a backward edge traversal indicates a traversal *against* the direction of control flow.

4.1.2 *Data Flow Information and Data Flow Properties.* When the sets of information are implemented as bit vectors, each bit represents a data flow property. There is one bit vector for the entry and one for the exit of each node. The lattice elements $\top$ and $\bot$ are "all bits true" or "all bits false" depending on $\sqcap$. We speak of TOP as the value of an individual property in a $\top$ bit vector, and BOT as the value of a property in the $\bot$ bit vector.

The data flow properties associated with program points $in(i)$ and $out(i)$ are denoted by IN_i and OUT_i respectively.

Definition 1. (*Inflow/outflow Properties*). Data flow properties that represent the information reaching/emanating from a node are called inflow/outflow properties.

The inflow/outflow properties are associated with the entry/exit of a node depending upon the direction of the flow. From equations 4 and 5, it is evident that for a forward problem IN_i represents the inflow properties while OUT_i represents the outflow properties of node i. However, for a backward problem, OUT_i represents the inflow properties whereas IN_i represents the outflow properties.

Example 4. For the problem of Reaching Definitions, REACH_IN_i represents the inflow properties while REACH_OUT_i represents the outflow properties. However, in the case of Live Variables, LIVE_OUT_i represents the inflow properties and LIVE_IN_i represents the outflow properties. Note that for bidirectional problems (*viz.* MRA), the same property may be an inflow as well as an outflow property. For instance, PPIN_i^l may become $\mathbf{F}$ due to PPOUT_j^l where $j \in pred(i)$, thus PPIN_i^l represents an inflow property. On becoming $\mathbf{F}$, PPIN_i^l may cause $\text{PPOUT}_{j'}^l$ to become $\mathbf{F}$, for some predecessor j'; here PPIN_i^l represents an outflow property. $\Box$

The program point for a property p is denoted by *program_point*(p). Two properties belonging to different program points are called *corresponding* properties if they represent information about the same data item, *viz.* the

retreating edges respectively. We prefer the former because they express the intuitive notion of direction more clearly. For non-reducible flow graphs, a back edge in this paper means a retreating edge.

same variable or the same expression. Two corresponding properties are *neighbours* of each other if their program points are neighbours in G. If a property p' influences the value of a neighbouring property p through a flow function h, it is denoted by $p \leftarrow h(p')$. If p and p' belong to neighbouring nodes i and j, p is an inflow property of i while p' is the corresponding outflow property of j. If p and p' belong to the same node, p is an outflow property whereas p' is the corresponding inflow property.

4.1.3 *Bit Vector Frameworks.*

Definition 2. (*Separability of Solution*[8]). A data flow framework $\mathbf{D} = <\mathcal{L}, \sqcap, \mathcal{F}>$ possesses the property of the separability of solution if $\exists$ semilattices $\mathcal{L}_1, \mathcal{L}_2, \cdots, \mathcal{L}_n$ such that an element $X \in \mathcal{L}$ can be represented by a tuple $< X^1, X^2, \ldots, X^n >$ where $X^i \in \mathcal{L}_i,\ 1 \le i \le n$ and :

(1) $\forall\, X, Y \in \mathcal{L} :\ X \sqcap Y\ \equiv\ < X^1 \sqcap Y^1, X^2 \sqcap Y^2, \ldots, X^n \sqcap Y^n >$

(2) $\forall h \in \mathcal{F}, h(X) \equiv < h^1(X^1), h^2(X^2), \ldots, h^n(X^n) >$ *where* $h^i \colon \mathcal{L}_i \to \mathcal{L}_i, 1 \le i \le n$.

(3) *Some* $\mathcal{L}_i$ *has height* H *and all other* $\mathcal{L}_i$ *have height at most* H.

Elements in each $\mathcal{L}_i$ represent different values of a *single* data flow property. The first two conditions ensure the independence of data flow properties while the third condition ensures that a property may assume at most $H+1$ distinct values during data flow analysis. Note that the separability of solution implies that a *factorization* exists for $(\mathcal{L}, \mathcal{F})$ [Rosen 1980], the *effective* height of $\mathcal{L}$ is H [Rosen 1980], and the functions in $\mathcal{F}$ are $(H+1)$-bounded [Marlowe and Ryder 1990].

It is easy to see that the bit vector problems satisfy all the three conditions : Data flow properties, represented by single bits, are independent of each other and each property may have two distinct values. Define bit functions *start*, *stop*, *propagate*, and *negate* such that for a bit b, *start* $(b) = \mathbf{T}$, *stop*$(b) = \mathbf{F}$, *propagate*$(b) = b$, and *negate*$(b) = \neg b$ [Dhamdhere, Rosen, and Zadeck 1992]. Let X^i be the i^{th} bit in a bit vector X. Let the size of the bit vector be k.

Definition 3. (*Bit vector function*). A bit vector function h is a mapping from $\{\ \mathbf{T}, \mathbf{F}\ \}^k$ to $\{\ \mathbf{T}, \mathbf{F}\ \}^k$ such that h can be written as a tuple of bit functions $h \equiv < \mathcal{B}_h^1, \mathcal{B}_h^2, \ldots, \mathcal{B}_h^k >$ where $\mathcal{B}_h^i$ is the bit function for the i^{th} bit, i.e. if $Y = h(X)$, then

$$\begin{aligned} X &\equiv\ < X^1, X^2, \ldots, X^k > \\ Y &\equiv\ < Y^1, Y^2, \ldots, Y^k > \quad \textit{where},\ \ Y^i \equiv \mathcal{B}_h^i(X^i),\ \ 1 \le i \le k. \end{aligned}$$

LEMMA 1. *A bit vector function h is monotonic if and only if it does not negate any bit.*

[8]This notion is analogous, though not identical, to Zadeck's notion of *cluster partitionability* [Zadeck 1984].

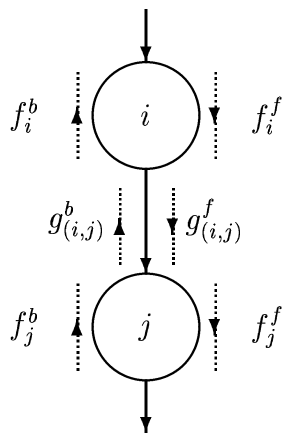


Fig. 6. Flow functions

Definition 4. (*Bit vector framework*). A data flow framework is bit vector framework if and only if all flow functions are monotonic bit vector functions.

LEMMA 2. *A bit vector function h is a monotonic bit vector function if and only if it can be expressed in the form $h(X) = C_1 + C_2{\cdot}X$ where $C_1, C_2, X \in \{\mathbf{T},\ \mathbf{F}\}^k$.*

LEMMA 3. *A bit vector framework is fast (i.e. 2-bounded).*

The influence of p' on p through a function h is denoted by $p \leftarrow \mathcal{B}_h(p')$ where the context demands a bit function and by $p \leftarrow h(p')$ where the context demands a bit vector function.

4.2 Characterizing the Flow of Information

4.2.1 *The Notion of Information Flow.* Since $x \sqcap \text{TOP} = x$ and $x \sqcap \text{BOT} = \text{BOT}$, a TOP value for a data flow property is an intermediate value until the data flow analysis is completed whereas BOT is a final value even during analysis. Thus, a BOT value implies a useful item of information from the viewpoint of data flow analysis, whereas a TOP value implies that such information can not be concluded during analysis. For iterative data flow analysis, the data flow properties are initialized to TOP for all nodes (except for the graph entry/exit nodes, which may have other values). Some properties change to BOT due to the local effect of computations in a node/along an edge, *viz.* when a definition is generated, or an expression is killed. These properties, in turn, change the neighbouring properties to BOT.

Definition 5. (*Information flow*). Information flows from a program point u to a program point v when a property at u, on becoming BOT, causes the corresponding property at v to become BOT.

Note that the information flow is transitive. It is shown in [Khedker and Dhamdhere 1992] that the incorporation of information flows due to all BOT properties in the program flow graph leads to a fixed point of the data flow equations.

4.2.2 *Flow Functions.* There are two fundamental kinds of flows in a data flow analysis problem :

(1) Information flows *within* a node, i.e. between the entry and exit of the node :
Represented by *node flow* functions $f \in \mathcal{F}$. These are the traditional transfer functions.

(2) Information flows along an edge :
Represented by *edge flow* functions g. We define a new set $\mathcal{G}$ to contain g.

The mapping between nodes and node flow functions is defined by the function $M_{\mathcal{F}} : N \to \mathcal{F}$ while the mapping between edges and edge flow functions is defined by the function $M_{\mathcal{G}} : E \to \mathcal{G}$. As is customary, we drop these mappings and subscript the flow functions directly by nodes/edges as the case may be. Thus, the node flow function for node i is denoted by f_i while the edge flow function for an edge $e = (i, j)$ is denoted by g_e or $g_{(i,j)}$.

The flow functions are determined directly from the data flow equations governing a problem. We refer to the functions by their type names f and g respectively, with an appropriate superscript f or b to indicate whether the flow is in the forward or the backward direction.

If a particular flow does not exist in a data flow problem, the corresponding function is the constant function $\top$. For example if forward flow of information *through* a node does not exist, f^f is $\top$. Similarly, if forward flow of information reaching a node entry (and hence the forward confluence of information) does not exist, g^f is $\top$. Analogous remarks hold for f^b and g^b.

Example 5. Table I summarizes the flow functions for some data flow problems. Note that there is no backward flow in the case of reaching definitions while there is no forward flow in the case of live variables. □

Definition 6. (*Non-singular data flow problem*). A data flow problem is non-singular if it involves more than one distinct confluence operator.

Clearly, a non-singular data flow problem is not a data flow framework as defined in Figure 4.

Example 6. All examples considered in this paper are singular except the Modified MRA (MMRA, for short) data flow problem [Dhamdhere 1991] which, apart from the Π term of MRA, contains a Σ term to inhibit redundant code movement (Figure 7). □

4.2.3 *Information Flow Paths.* Let T_{e_i} denote a traversal along the graph edge e_i, i.e. T_{e_i} can be T_e^f / T_e^b.

Definition 7. (*Information flow path*). An information flow path *(*ifp*)* is a sequence of edge traversals $T_{e_1}, T_{e_2}, \ldots, T_{e_k}$ along which information can flow during data flow analysis.

We use the notation $< u, v, \rho >$ for an *ifp* ρ from program point u to a program point v. Note that an *ifp* may not follow a graph theoretic path.

Table I. Examples of Flow Functions

- **Reaching Definitions**

$h(X)$	$Y \leftarrow h(X)$	X
$f_i^f(X) = \text{REACH\_GEN}_i + \neg\ \text{REACH\_KILL}_i \cdot X$	REACH_OUT_i	REACH_IN_i
$f_i^b(X) = \top$	REACH_IN_i	REACH_OUT_i
$g_{(j,i)}^f(X) = X$ (i.e. identity function $\imath$)	REACH_IN_i	REACH_OUT_j
$g_{(i,k)}^b(X) = \top$	REACH_OUT_i	REACH_IN_k

- **Live Variables**

$h(X)$	$Y \leftarrow h(X)$	X
$f_i^f(X) = \top$	LIVE_OUT_i	LIVE_IN_i
$f_i^b(X) = \text{LIVE\_GEN}_i + \neg\ \text{LIVE\_KILL}_i \cdot X$	LIVE_IN_i	LIVE_OUT_i
$g_{(j,i)}^f(X) = \top$	LIVE_IN_i	LIVE_OUT_j
$g_{(i,k)}^b(X) = X$ (i.e. identity function $\imath$)	LIVE_OUT_i	LIVE_IN_k

- **Morel-Renvoise Algorithm (MRA)**

$h(X)$	$Y \leftarrow h(X)$	X
$f_i^f(X) = \top$	PPOUT_i	PPIN_i
$f_i^b(X) = \text{ANTLOC}_i + \text{TRANSP}_i \cdot X$	PPIN_i	PPOUT_i
$g_{(j,i)}^f(X) = \text{AVOUT}_j + X$	PPIN_i	PPOUT_j
$g_{(i,k)}^b(X) = X$ (i.e. identity function $\imath$)	PPOUT_i	PPIN_k

- **Basic Load Store Insertion Algorithm (LSIA)**

$h(X)$	$Y \leftarrow h(X)$	X
$f_i^f(X) = \text{DCOMP}_i + \text{DTRANSP}_i \cdot X$	SPPOUT_i	SPPIN_i
$f_i^b(X) = \top$	SPPIN_i	SPPOUT_i
$g_{(j,i)}^f(X) = X$ (i.e. identity function $\imath$)	SPPIN_i	SPPOUT_j
$g_{(i,k)}^b(X) = \text{DANTIN}_k + X$	SPPOUT_i	SPPIN_k

- **Composite Hoisting and Strength Reduction Algorithm (CHSA)**

$h(X)$	$Y \leftarrow h(X)$	X
$f_i^f(X) = \text{CONSTD}_i \cdot X$	NOCOMOUT_i	NOCOMIN_i
$f_i^b(X) = \text{CONSTA}_i \cdot X$	NOCOMIN_i	NOCOMOUT_i
$g_{(j,i)}^f(X) = \text{CONSTB}_i \cdot X$	NOCOMIN_i	NOCOMOUT_j
$g_{(i,k)}^b(X) = \text{CONSTE}_i \cdot X$	NOCOMOUT_i	NOCOMIN_k

$$\begin{aligned}
\mathrm{PPIN}_i \;=\; & \mathrm{PAVIN}_i \cdot (\,\mathrm{ANTLOC}_i \;+\; \mathrm{TRANSP}_i \cdot \mathrm{PPOUT}_i) \,\cdot \\
& \prod_{j \,\in\, pred(i)} (\,\mathrm{AVOUT}_j \;+\; \mathrm{PPOUT}_j) \,\cdot \\
& \sum_{j \,\in\, pred(i)} (\,\mathrm{PPIN}_j \cdot \neg\mathrm{ANTLOC}_j \;+\; \mathrm{AVOUT}_j) \\
\mathrm{PPOUT}_i \;=\; & \prod_{k \,\in\, succ(i)} (\,\mathrm{PPIN}_k)
\end{aligned}$$

Fig. 7. The MMRA Equations [Dhamdhere 1991]

Table II. Some examples of information flow paths

Problem	Function types	Information flow paths
Reaching Def.	f^f, g^f	$(T_e^f)^+$
Live Variables	f^b, g^b	$(T_e^b)^+$
MRA	f^b, g^b, g^f	$((T_e^b)^+(T_e^f \mid \epsilon))^+(T_e^b)^*$
LSIA	f^f, g^f, g^b	$((T_e^f)^+(T_e^b \mid \epsilon))^+(T_e^f)^*$
CHSA	f^f, f^b, g^b, g^f	$T_e^f(T_e^b \mid T_e^f)^*$

Where convenient, we will represent an *ifp* as a sequence of nodes, leaving the traversal of the edges connecting these nodes implicit. Since *ifp*'s can be statically determined from the flow functions, they follow a pattern which can be described by a regular expression; Khedker and Dhamdhere [1992] describe a procedure to construct the regular expression representing an *ifp* pattern. These regular expressions should be contrasted with Tarjan's *path expressions* [Tarjan 1981b]; the latter are restricted to graph theoretic paths and can not be used to characterize *ifp*'s arising out of sibling/spouse effects.

Example 7. Table II contains examples of the *ifp* patterns. These patterns provide valuable insights about how the information could flow in a given data flow problem. □

Example 8. Consider the graph in Figure 2. Some sequences of edge traversals which may form *ifp*'s, and the data flow problems in which these *ifp*'s are valid, are :

- $(1,8,11,12) = T_f^f\ T_f^f\ T_f^f\ T_f^f$: Reaching Definitions
- $(11,10,9,8) = T_f^b\ T_f^b\ T_f^b\ T_f^b$: Live variables
- $(5,2,3,4,7) = T_f^b\ T_f^f\ T_b^b\ T_f^f$: MRA
- $(7,4,3,2) = T_f^b\ T_b^f\ T_f^b$: LSIA
- $(2,5,6,7,4,3) = T_f^f\ T_f^f\ T_f^f\ T_f^b\ T_f^b$: CHSA

□

For bit vector frameworks, *ifp*'s are necessarily acyclic. Note, however, that the underlying graph theoretic path may be cyclic since a node may appear in the path once for its entry point and once for its exit point.

4.2.4 *The Path Flow Function.* Consider an *ifp* $< u, v, \mathcal{P}_r >$ from a graph entry/exit node to the entry/exit of a node r. For such an *ifp*, $u \in \{in(n_0), out(n_x)\}$ where $n_0 \in entry(G)$, $n_x \in exit(G)$, and $v \in \{in(r), out(r)\}$. Let $\mathcal{P}_r$ be the sequence $q_1, q_2, \cdots, q_k, q_{k+1} = T_{e_1}, T_{e_2}, \cdots, T_{e_k}$. Then, $q_1 \in \{n_0, n_x\}$ and $q_{k+1} = r$. Consider an *ifp* fragment $< u, v', \rho >$ of $\mathcal{P}_r$, terminating with edge $e_i = (q_i, q_{i+1})$, such that $v' = in(q_{i+1})$ if $T_{e_i} = T_e^f$ and $v' = out(q_{i+1})$ if $T_{e_i} = T_e^b$.

Let $flow_i$ denote the path flow function of ρ (i.e. the *ifp* terminating with e_i). We define

$$flow_1 = \begin{cases} g_{e_1}^f \circ f_{n_0}^f & \text{if } T_{e_1} = T_e^f \ \ (\text{i.e. } u = in(n_0)) \\ g_{e_1}^b \circ f_{n_x}^b & \text{if } T_{e_1} = T_e^b \ \ (\text{ i.e. } u = out(n_x)) \end{cases}$$

Since $e_{i+1} = (q_{i+1}, q_{i+2})$, information flows from q_{i+1} to q_{i+2}. $flow_{i+1}$ is obtained by composing the functions $f_{q_{i+1}}$ and $g_{e_{i+1}}$ with $flow_i$, as shown in Table III. Note that if $v = out(r)$ and $T_{e_k} = T_e^f$ then there is a forward flow through node r. Similarly, if $v = in(r)$ and $T_{e_k} = T_e^b$ then there is a backward flow through node r. Thus, path function for the *ifp* $\mathcal{P}_r$ is :

$$\text{FLOW}_{\mathcal{P}_r} = \begin{cases} f_r^f \circ flow_k & \text{if } T_{e_k} = T_e^f \ \text{ and } v = out(r) \\ f_r^b \circ flow_k & \text{if } T_{e_k} = T_e^b \ \text{ and } v = in(r) \\ flow_k & \text{otherwise} \end{cases}$$

In a unidirectional problem, the typical sequence of edges in a path is either $(T_e^f)^*$ or $(T_e^b)^*$. In either case, the information *necessarily* flows through all intermediate nodes. It can be verified from Table III that in such a case, the edge flow functions appear in composition with the node flow functions and never in isolation from them. Thus there is no need to treat them separately. Under such circumstances, the flow can be adequately characterized by functions which could be associated with the nodes or edges interchangeably. $\mathcal{F}$ has been the set of such functions in the classical theory.

However, in bidirectional problems the sibling/spouse effects exist and the information may flow from $in(q_i)$ to $in(q_{i+2})$ via $out(q_{i+1})$ or $out(q_i)$ to $out(q_{i+2})$ via $in(q_{i+1})$. Though the information flows along the edges $e_i = (q_i, q_{i+1})$ and $e_{i+1} = (q_{i+1}, q_{i+2})$, it does not flow *through* node q_{i+1}. Thus, unlike the unidirectional problems, the edge and node flows must be represented distinctly. It is easy to see that all the above flows are handled uniformly by the generalized path flow function.

4.3 Specification of a Data Flow Problem

A data flow problem is completely specified by the pair $\mathbf{S} = < Q, X_0 >$, where Q is the system of equations and X_0 is the set of initial values for the properties of the nodes.

4.3.1 *Data Flow Equations.* The entry/exit properties of node i can be computed from various flows as follows :

$$\text{IN}_i = \text{IN}_i^f \sqcap \text{IN}_i^b \sqcap \text{CONST\_IN}_i$$

Table III. Computing the flow function

	$flow_{i+1}$	
	$T_{e_{i+1}} = T^f_e$	$T_{e_{i+1}} = T^b_e$
$T_{e_i} = T^f_e$	$g^f_{e_{i+1}} \circ f^f_{q_{i+1}} \circ flow_i$ (Forward Flow)	$g^b_{e_{i+1}} \circ flow_i$ (Flow between spouses)
$T_{e_i} = T^b_e$	$g^f_{e_{i+1}} \circ flow_i$ (Flow between siblings)	$g^b_{e_{i+1}} \circ f^b_{q_{i+1}} \circ flow_i$ (Backward Flow)

$$\text{OUT}_i = \text{OUT}^f_i \sqcap \text{OUT}^b_i \sqcap \text{CONST\_OUT}_i$$

CONST_IN/CONST_OUT are the *constant* properties which represent the information already known concerning the entry/exit of a node. However, unlike the local properties of a node, the constant properties typically represent *lower order* data flow properties, i.e. properties computed by an earlier data flow analysis. If no such information is involved in the problem, the CONST_IN/CONST_OUT properties are $\top$.

Example 9. For all unidirectional problems referred in this paper, the CONST_IN and CONST_OUT properties are $\top$. However, for MRA, CONST_IN is PAVIN while CONST_OUT is $\top$. For CHSA, CONST_IN is $\top$ while CONST_OUT is CONSTC. □

$\text{IN}^f_i/\text{OUT}^f_i$ and $\text{IN}^b_i/\text{OUT}^b_i$ represent the contributions from the forward and backward flows respectively. Clearly, $\text{IN}^f_i/\text{OUT}^b_i$ represent the inflow while the $\text{IN}^b_i/\text{OUT}^f_i$ represent the outflow component of the entry/exit information of node i. They are computed as follows :

$$\begin{aligned}
\text{IN}^f_i &= \bigsqcap_{j \in pred(i)} g^f_{(j,i)}(\text{OUT}_i) \\
\text{IN}^b_i &= f^b_i(\text{OUT}_i) \\
\text{OUT}^b_i &= \bigsqcap_{k \in succ(i)} g^b_{(i,k)}(\text{IN}_k) \\
\text{OUT}^f_i &= f^f_i(\text{IN}_i)
\end{aligned}$$

Thus, the data flow equations become

$$\text{IN}_i = \bigsqcap_{j \in pred(i)} g^f_{(j,i)}(\text{OUT}_j) \sqcap f^b_i(\text{OUT}_i) \sqcap \text{CONST\_IN}_i \quad (6)$$

$$\text{OUT}_i = \bigsqcap_{k \in succ(i)} g^b_{(i,k)}(\text{IN}_k) \sqcap f^f_i(\text{IN}_i) \sqcap \text{CONST\_OUT}_i \quad (7)$$

Equations 6 - 7 are the generic data flow equations. Specific problems can be treated as special cases of these equations.

4.3.2 *Initialization.* We define X_0 to consist of two classes of values : *Boundaryinfo* and *Initinfo*.

Boundaryinfo contains values specifying the interprocedural information reaching the entry/exit of the graph. For an entry node i, $Boundaryinfo_i$ specifies the value associated with $in(i)$ while for an exit node, $Boundaryinfo_i$

Table IV. *Boundaryinfo* values for local variables/expressions involving local variables.

Data Flow Problem	Direction	$\sqcap$	$\top$	$\perp$	*Boundaryinfo*$_i$	
					$i \in entry(G)$	$i \in exit(G)$
Reaching Definitions	Forward	OR	$\overline{\text{F}}$	$\overline{\text{T}}$	$\overline{\text{F}}$	$\overline{\text{F}}$
Live Variables	Backward	OR	$\overline{\text{F}}$	$\overline{\text{T}}$	$\overline{\text{F}}$	$\overline{\text{F}}$
Available Expressions	Forward	AND	$\overline{\text{T}}$	$\overline{\text{F}}$	$\overline{\text{F}}$	$\overline{\text{T}}$
Very Busy Expressions	Backward	AND	$\overline{\text{T}}$	$\overline{\text{F}}$	$\overline{\text{T}}$	$\overline{\text{F}}$
Dead Variables	Backward	AND	$\overline{\text{T}}$	$\overline{\text{F}}$	$\overline{\text{T}}$	$\overline{\text{T}}$
MRA	Bidirectional	AND	$\overline{\text{T}}$	$\overline{\text{F}}$	$\overline{\text{F}}$	$\overline{\text{F}}$

specifies the value associated with $out(i)$. These values are important for the correctness of any optimization based on the solution of the data flow problem. A wrong specification may lead to an unsafe solution and may thus lead to an incorrect optimization.

Boundaryinfo is determined as follows :

—*Boundaryinfo*$_i$ is $\top$, if either $i \in entry(G)$ and the forward confluence does not exist, or $i \in exit(G)$ and the backward confluence does not exist.

—If $i \in entry(G)$ and the forward confluence exists, or $i \in exit(G)$ and the backward confluence exists, *Boundaryinfo*$_i$ is determined by interprocedural information if available, else, by the semantics of the data flow problem as explained in the following.

Let $\overline{\text{T}}$ and $\overline{\text{F}}$ denote "all bits **T**" and "all bits **F**" respectively. Table IV provides *Boundaryinfo* for some representative data flow problems for local data items (i.e. local variables/expressions involving local variables). Consider the example of live variable analysis. For local variables, the values in *Boundaryinfo* are determined as follows : The information being gathered is a set of predicates, each of which represents that a variable is live. For local variables, all these predicates are false at the exit of a program, hence the values in *Boundaryinfo* are $\overline{\text{F}}$ for the exit nodes. Since there is no forward flow, *Boundaryinfo* values for entry nodes are $\top$ which is $\overline{\text{F}}$ for a union problem.

Initinfo specifies values for the internal nodes of the program flow graph. These are required in the case of iterative methods only. Using the confluence operator as a criterion, the *Initinfo* values are defined to be $\top$; anything else might lead to a fixed point lower than MFP. Given correct *Boundaryinfo*, *Initinfo* influences the quality of information (vis-a-vis the maximality of information), but not its safety.

The distinction between *Boundaryinfo* and *Initinfo* is usually not made in the literature, leading to avoidable confusion. Although *Boundaryinfo* has no connection with the confluence operator, the values in *Boundaryinfo* are often recommended as $\perp$.[9] To correctly determine the *Boundaryinfo*, we only need to ask the following two questions : (*i*) What is the information

[9]Hecht [1977] specifies $\perp$, Marlowe and Ryder [1990] cautiously mention "often $\perp$"

being gathered ? and (*ii*) What is the information available from the caller procedure ?

Example 10. Consider the problem of dead variable analysis which is a dual of the problem of live variable analysis. It is an intersection problem in which a predicate indicates that a variable is dead. For local variables, all predicates are true at the exit of a program. Hence the values in *Boundaryinfo* are $\overline{\mathrm{T}}$ for the exit nodes.

This should be contrasted with the problem of Very Busy Expressions for which the *Boundaryinfo* values are $\overline{\mathrm{F}}$ for the expressions involving local variables. Both the problems are backward intersection problems with an identical form of data flow equations, yet they have different *Boundaryinfo* values. □

Clearly, it is incorrect to link the confluence operator with *Boundaryinfo*.

4.4 Solutions of Data Flow Problems

As noted in section 3, an acceptable solution is characterized by the safety and maximality of fixed point.

4.4.1 *MOP Solution.* Let $const(u)$ return the value of the constant property associated with program point u. For all p, p' and $h \in \mathcal{F} \cup \mathcal{G}$ such that $p \leftarrow h(p')$, we replace h in FLOW by $h \sqcap const(program\_point(p))$. Let $< u, v, \mathcal{P}_r >$ be denoted by $\mathcal{P}_r^{in}$ if $v = in(r)$ and by $\mathcal{P}_r^{out}$ if $v = out(r)$. Further, let η denote *Boundaryinfo* at the program point u which belongs to a graph entry/exit node.

A safe assignment SA : $N \rightarrow \mathcal{L}$ is a function with two components, $<$ SIN, SOUT$>$ such that, for a node r, and all $\mathcal{P}_r^{in}$ /$\mathcal{P}_r^{out}$:

$$\mathrm{SIN}(r) \sqsubseteq \mathrm{FLOW}_{\mathcal{P}_r^{in}}(\eta) \tag{8}$$

$$\mathrm{SOUT}(r) \sqsubseteq \mathrm{FLOW}_{\mathcal{P}_r^{out}}(\eta) \tag{9}$$

Maximum SA represents the MOP solution. Any solution which is not contained in MOP is unsafe.

The *ifp*'s reduce to graph theoretic paths for unidirectional problems. For a forward unidirectional problem, $\mathcal{P}_r^{in}$ reduces to graph paths from $in(n_0)$ to $in(r)$ where n_0 is an entry node, and $\mathcal{P}_r^{out}$ does not exist. Thus, the path flow function $\mathrm{FLOW}_{\mathcal{P}_r^{in}}$ reduces to $M(\rho)$ defined in Figure 4 and used for characterizing MOP solution in section 3.3. Analogous remarks hold for backward problems.

4.4.2 *Fixed Point Solution.* A fixed point solution is the fixed point of equations 6 - 7. Maximum fixed point is obtained by setting *Initinfo* values to $\top$.

For a forward unidirectional problem, the generic data flow equations reduce to equations 4 and 5. Analogous remarks hold for backward problems.

while Aho, Sethi, and Ullman [1986] specify it as $\top$ with a remark that the values may be $\perp$ in some cases.

4.5 Characteristics of Data Flow Frameworks

Let $p \leftarrow \mathcal{B}_h(p')$. If p' is BOT, it may cause p to become BOT.

All bit vector problems have the following important characteristics :

—*MBVP* : A property changes from TOP to BOT *only.*

—*SBVP* : $\forall\ p', \forall\ h$ such that $p \leftarrow h(p')$, if p' causes p to become BOT, it does so *on its own and not in combination with other corresponding properties.*

These characteristics arise from monotonicity and singularity respectively. This follows from the facts that :

(1) For convergence on MFP, *Initinfo* is $\top$. Since the functions involved are monotonic, if there is a change, it must be from TOP to BOT only.

(2) Data flow frameworks have been defined in terms of a semilattice which implies a unique confluence operator.

Since unidirectional flow problems typically have only one confluence, *SBVP* was never emphasized or mentioned explicitly in the literature. In the case of bidirectional problems, if all confluences merge the global information using the same boolean operator, as is the case in MRA, *SBVP* holds automatically.

Example 11. As noted in example 6, MMRA uses two confluences. Consequently, *SBVP* is violated : $\text{PPIN}_j = \mathbf{F}$ can not set the PPIN_i of a successor i to $\mathbf{F}$ on its own, but may do so in combination with PPIN of other predecessors of i. □

Lemma 4. *All bit vector data flow frameworks possess the SBVP property.*

There are two important implications of *MBVP* and *SBVP* :

(1) A considerable reduction in the work for performing data flow analysis is possible because :
   (a) *MBVP* guarantees that a property p which has become BOT need not be recomputed as it has attained its final value.
   (b) *SBVP* guarantees that all neighbouring properties can be *refined*, rather than recomputed, to incorporate the effect of a property changing to BOT.[10]
   Let $X(u)$ and $X'(u)$ represent the old and new values at program point u, and let v' be the program point at which values have changed. Equation 10 defines recomputation, while equation 11 defines refinement :

$$X'(u) = \prod_{v \in neighbours\ of\ u} h(X'(v)) \sqcap \text{CONST}(u) \qquad (10)$$

$$X'(u) = X(u) \sqcap h(X'(v')) \qquad (11)$$

[10]Refinement is not a new concept; it can be traced in the worklist-based iterative algorithm in [Hecht 1977]. Here we just make it explicit.

When contrasted with recomputation, which uses the values of *all* neighbouring properties, refinement decreases the amount of work by a factor that depends on the number of in-edges/out-edges of a node. Lemma 5 shows the equivalence between refinement and recomputation.

(2) *MBVP* guarantees the termination of data flow analysis — the values of properties change in one direction only.

LEMMA 5. *Refinement and recomputation yield identical results for singular data flow problems.*

5. PERFORMING DATA FLOW ANALYSIS

5.1 Wordwise Analysis

Since the size of a bit vector may vary with the size of the program, we propose to process the bit vectors in parts. Further, we partition the problem of data flow analysis so as to process a specific part of a bit vector, rather than the entire bit vector, in each step. We select a part, process it all over the graph and then select the next part which needs to be processed. At one extreme, each part may consist of one bit as in [Dhamdhere, Rosen, and Zadeck 1992]; at the other extreme, each part may be the largest chunk of a bit vector which can be processed in one machine operation, which typically is a *machine word.* We follow the latter approach, which we term *wordwise analysis.*

Wordwise analysis results in considerable savings in the work to be performed since all parts may not require processing for all nodes of the graph. More formally, let N be the set of nodes, and M, the set of words. The set $N \times M$ is partitioned into two subsets, the set NM_p which requires processing, and the set NM_{np} which does not. Wordwise analysis implies selecting all entries for a specific word from NM_p and processing them. The traditional approach of processing all words of a bit vector partitions only N. Let N_p and N_{np} be the partitions. Since *all* words in M are processed for each node in N_p, it results in more work.

Henceforth, the discussion will be in terms of properties belonging to $<u, m>$ where u is a program point and m is a word.

5.2 The Basic Algorithm

Figure 8 contains the basic algorithm which is a generalization of the worklist-based iterative method. The work is divided in two phases : *initialization* and *propagation*, performed by the procedures *init* and *settle* respectively.

5.2.1 *Initialization.* As noted in section 4.2, information flow is initiated by the properties whose initial values are BOT due to local effects of computations existing within a node/along an edge. The *Initial Trigger Set*, denoted TR_0, contains all such properties.

$$\begin{aligned} Bprops \;=\; \{p \mid \;& \text{either } program\_point(p) = in(i), i \in entry(G) \text{ or} \\ & program\_point(p) = out(i), i \in exit(G)\} \end{aligned}$$

```
1.  procedure dfa()
2.  {    init()
3.       settle()
4.  }
5.  procedure init()
6.  {    for each word $m$
7.           for each node $i$ of the graph
8.               Set all inflow properties in word $m$ to TOP
9.               Compute the outflow properties in word $m$
10.              if any property is BOT then          /⋆ it belongs to $TR_0$ ⋆/
11.                  Insert $i$ in the worklist for word $m$
12. }
13. procedure settle()
14. {    for each word $m$
15.          for each node $i$ in the worklist of $m$
16.              propagate($i, m$)
17. }
18. procedure propagate($i, m$) /⋆ propagate effect of outflow properties of $i$ ⋆/
19. {    for each neighbouring node $k$ of $i$
20.          Refine the inflow properties of $k$ in word $m$        /⋆ using $g_e$ ⋆/
21.          Compute the outflow properties of $k$ in word $m$      /⋆ using $f_k$ ⋆/
22.          if some property changes to BOT then
23.              propagate($k, m$)
24. }
```

Fig. 8. The Basic Algorithm

$$TR_0 = \{p \mid p \leftarrow \mathcal{B}_h(p') \text{ such that } \mathcal{B}_h(\text{TOP}) = \text{BOT or } p \in Bprops \text{ and } p = \text{BOT}\} \tag{12}$$

After incorporating the initialization in equations 6 and 7, TR_0 can be computed from the following equations :

$$\text{IN}_i = \begin{cases} Boundaryinfo_i & \text{if } i \in entry(G) \\ \prod_{j \in pred(i)} g^f_{(j,i)}(\top) \sqcap f^b_i(\top) \sqcap \text{CONST\_IN}_i & \text{otherwise} \end{cases} \tag{13}$$

$$\text{OUT}_i = \begin{cases} Boundaryinfo_i & \text{if } i \in exit(G) \\ \prod_{k \in succ(i)} g^b_{(i,k)}(\top) \sqcap f^f_i(\top) \sqcap \text{CONST\_OUT}_i & \text{otherwise} \end{cases} \tag{14}$$

Procedure *init* constructs TR_0 by computing the outflow properties for each node i. If any property p in word m is BOT, node i is inserted in the worklist for word m.

5.2.2 *Propagation.* Propagation selects a node from the worklist of a given word and propagates the transitive influence of its BOT properties. Effectively, many bits in a word are processed simultaneously. The outflow properties in the current word of a node's bit vector may change the inflow properties of neighbouring nodes, which are refined to incorporate their influence. From these inflow properties, the corresponding outflow properties of the node are computed. If any outflow property changes to BOT, it becomes a candidate for propagation whose influence is propagated to its neighbours by the recursive call in procedure *propagate*.

5.3 A Generic Algorithm for Data Flow Analysis

The generic algorithm embodies two major deviations from the traditional algorithms :

(1) Wordwise analysis : This reduces the amount of work required for data flow analysis.

(2) Distinction between entry and exit points of a node : This is necessary for the treatment of bidirectional flows.

Figure 9 provides the algorithm in terms of equations 6, 7 and 13, 14 for the IN and OUT properties of a node. The specific points to be noted are :

—Some speedup can be achieved by accumulating many changes for a pair $<u, m>$ before refining the properties of neighbouring program points. Hence the pairs with fewer BOT properties are processed later by maintaining the worklists in sorted order according to the number of BOT properties.

—For refinement of properties during propagation, it is sufficient to apply a function h' instead of $h(Z) = A + B \cdot Z$ where h' is
  —For intersection problems : $\forall Z \in \mathcal{L}$, $h'(Z) = A + Z$.
  —For union problems : $\forall Z \in \mathcal{L}$, $h'(Z) = B \cdot Z$.
  Thus, only two operations are required per function application (this includes one operation for the meet) instead of three.

—The recursive calls during propagation have been eliminated by inserting a node in the worklist during propagation also. Apart from eliminating the overheads associated with recursion, this aids in delayed propagation.

The generic algorithm is uniformly applicable to unidirectional and bidirectional flows. Further, a data flow analysis algorithm for a given problem can be automatically constructed from the data flow equations without the knowledge of the semantics of the underlying problem.

5.4 Performance Analysis of the Generic Algorithm

Since the size of a bit vector may vary with the size of the program (section 5.1), the unit of work for performance analysis should be the work required to process *one word* rather than the work required to process one bit vector. Hence, in the following, an *operation* refers to one bit vector operation on the properties located in one word of a node. The bounds derived in this section assume that the number of edges, e, is $O(n)$ where n denotes the number of nodes. Figure 10 gives the notations used in this section.

To estimate the complexity, we develop a notion of *orthogonality* in bit vector processing. Let n_1 and n_2 be some two nodes in the worklist for the word being processed. The order in which the effect of n_1 and n_2 is

```
procedure dfa()
{   init()
    settle()
}
procedure init()
{   for each word $m$
        for each node $i$
            if $i \in entry(G)$ then
                $\text{IN}_i = Boundaryinfo_i$
            else
                $\text{IN}_i = \bigsqcap_{j \in pred(i)} g^f_{(j,i)}(\top) \sqcap f^b_i(\top) \sqcap \text{CONST\_IN}_i$
            if any property in $\text{IN}_i$ is BOT then     /* it belongs to $TR_0$ */
                Insert $< i, in(i) >$ in $\text{LIST}_m$
            if $i \in exit(G)$ then
                $\text{OUT}_i = Boundaryinfo_i$
            else
                $\text{OUT}_i = \bigsqcap_{k \in succ(i)} g^b_{(i,k)}(\top) \sqcap f^f_i(\top) \sqcap \text{CONST\_OUT}_i$
            if any property in $\text{OUT}_i$ is BOT then /* it belongs to $TR_0$ */
                Insert $< i, out(i) >$ in $\text{LIST}_m$
}
procedure settle()
{   for each word $m$
        while $\exists$ an entry <node, program_point> in $\text{LIST}_m$
            Delete <node, program_point> from $\text{LIST}_m$
            if program_point = in(node) then
                propagate_in(node, m)
            else propagate_out(node, m)
}
procedure propagate_in(i, m)
{   $\text{OUT}_i = \text{OUT}_i \sqcap f^f_i(\text{IN}_i)$                 /* refinement using $f^f_i$ */
    if any property in $\text{OUT}_i$ becomes BOT then
        Insert $< i, out(i) >$ in $\text{LIST}_m$ if not already present
    for all $j \in pred(i)$
        $\text{OUT}_j = \text{OUT}_j \sqcap g^b_{(j,i)}(\text{IN}_i)$          /* refinement using $g^b_{(j,i)}$ */
        if any property in $\text{OUT}_j$ becomes BOT then
            Insert $< j, out(j) >$ in $\text{LIST}_m$ if not already present
}
procedure propagate_out(i, m)
{   $\text{IN}_i = \text{IN}_i \sqcap f^b_i(\text{OUT}_i)$                 /* refinement using $f^b_i$ */
    if any property in $\text{IN}_i$ becomes BOT then
        Insert $< i, in(i) >$ in $\text{LIST}_m$ if not already present
    for all $k \in succ(i)$
        $\text{IN}_k = \text{IN}_k \sqcap g^f_{(i,k)}(\text{OUT}_i)$          /* refinement using $g^f_{(i,k)}$ */
        if any property in $\text{IN}_k$ becomes BOT then
            Insert $< k, in(k) >$ in $\text{LIST}_m$ if not already present
}
```

Fig. 9. A Generic Algorithm for Data Flow Analysis

n	number of nodes.
e	number of edges.
no_w	number of words.
d_in_i	the in-degree of node i.
d_out_i	the out-degree of node i.
$\mathcal{B}_j$	$\{ k \mid k$ is a bit in word j and figures in TR_0 for some node $\}$.
b_j	$\lvert \mathcal{B}_j \rvert$.
n_j	number of nodes in the worklist for word j after TR_0 construction.
$op\_prop^j_k$	number of operations performed while processing properties corresponding to the bit k in $\mathcal{B}_j$.
max_op_prop	$\max(op\_prop^j_k) \quad 0 \le k \le b_i,\ 1 \le j \le no\_w.$
$op\_word^j_i$	number of operations performed while propagating the effect of word j of node i.
max_op_word	$\max(op\_word^j_i) \quad 0 \le j \le no\_w,\ 1 \le i \le n$
t_op_j	total number of operations required to refine the properties of word j.
t_work	total work performed during propagation.

Fig. 10. Notations used for performance analysis

propagated does not matter; final result is a superposition of the effects of various nodes regardless of the order in which they are processed. Similarly, since the bits are independent, the words may also be processed in any order. Thus, while propagating the effect of a node on some worklist, we may safely ignore the presence of other nodes in the same worklist, and all nodes in all other worklists. If we can estimate the amount of work done for one node in one worklist, it is easy to estimate the total work done — it is the sum of the work done for all nodes in all worklists.

Note that due to the orthogonality among words and among the nodes in one worklist, all changes that have to take place in a given word due to a given node in the worklist, will take place simultaneously. If some bit changes later, it must be due to the effect of some other node in the same worklist. This argument forms the basis of the following lemma.

LEMMA 6. *After constructing TR_0, the effect of the BOT properties of a given node in the worklist for a given word can be completely propagated in $O(n)$ operations.*

PROOF. Let node i be reached while propagating the effect of properties in word m of some node r. Without loss of generality, assume that the effect of IN_i properties is propagated first. Since two operations are required for refinement (section 5.3), the effect of changes in IN_i can be propagated in $2 \cdot d\_in_i + 2$ operations — $2 \cdot d\_in_i$ operations for refining the OUT properties of predecessors and two operations for refining OUT_i. The effect of the BOT properties in OUT_i, if any, can be propagated in $2 \cdot d\_out_i$ operations. Thus, the maximum number of operations performed on visiting node i is $2 \cdot d\_in_i + 2 \cdot d\_out_i + 2$. In the worst case all the nodes may have to be visited.

Hence,

$$max\_op\_word = \sum_{i=1}^{n}(2\cdot d\_in_i + 2\cdot d\_out_i + 2) = 4\cdot e + 2\cdot n$$

which is $O(n)$. □

LEMMA 7. *The effect of a given property in TR_0 can be completely propagated in $O(n)$ operations.*

PROOF. As in lemma 6, it can be shown that $max\_op\_prop = 4\cdot e + 2\cdot n$. □

LEMMA 8. *$O(n^2)$ operations are needed to propagate the influence of all properties in* TR_0.

PROOF. Propagating the influence of the properties of a node may subsume the influence of the properties of some other node. Thus,

$$\begin{aligned} t\_op_j &\leq n_j \cdot max\_op\_word \\ t\_work &= \sum_{j=1}^{no\_w} t\_op_j \\ &\leq \sum_{j=1}^{no\_w} n_j \cdot max\_op\_word \\ &\leq max\_op\_word \cdot \sum_{j=1}^{no\_w} n_j \end{aligned} \tag{15}$$

Further, since all properties in a word are processed simultaneously,

$$\begin{aligned} t\_op_j &\leq b_j \cdot max\_op\_prop \\ t\_work &= \sum_{j=1}^{no\_w} t\_op_j \\ &\leq \sum_{j=1}^{no\_w} b_j \cdot max\_op\_prop \\ &\leq max\_op\_prop \cdot \sum_{j=1}^{no\_w} b_j \end{aligned} \tag{16}$$

Both *max_op_word* and *max_op_prop* are $4\cdot e + 2\cdot n)$. Hence, it follows from 15 and 16 that,

$$t\_work \leq 4\cdot e + 2\cdot n \cdot \min(\sum_{j=1}^{no\_w} n_j, \sum_{j=1}^{no\_w} b_j) \tag{17}$$

The worklist length for a word is $O(n)$ hence $\sum n_j$ could be $\mathrm{O}(n^2)$. Since no_w is $\mathrm{O}(n)$, $\sum b_j$ is $\mathrm{O}(n)$ and hence, $\min(\sum n_j, \sum b_j)$ is $O(n)$. Since $e = O(n)$, t_work is $\mathrm{O}(n^2)$. □

LEMMA 9. TR_0 *can be constructed in $O(n^2)$ operations.*

PROOF. It is evident from equations 13 and 14 that $3\cdot d\_in_i + 3$ operations are required to compute the values of IN_i — two operations for computing each $g^f_{(j,i)}(\top)$, two operations for computing $f^b_i(\top)$ and $d\_in_i + 1$ operations for meet. Similarly, $3\cdot d\_out_i + 3$ operations are required to compute the values of OUT_i. Hence the total number of operations for one word is

$$\sum_{i=1}^{n}(3\cdot d\_in_i + 3\cdot d\_out_i + 6) = 6\cdot(e+n)$$

which is $O(n)$. Since there are $O(n)$ words, the complexity is $O(n^2)$. □

THEOREM 1. *Total work done by the proposed algorithm is $O(n^2)$.*

A closer look at the proofs of lemmas 8 and 9 reveals that refinement does not seem to play any role in the complexity. Since e is $O(n)$, it is perfectly valid to assume that the degree of a node is bounded by a constant. In such a case even if the properties are recomputed, the order of the work involved remains the same. However, refinement has a practical significance as it reduces the number of operations by a constant factor.

5.5 Performance in Practical Situations

Though the theoretical complexity of the algorithm is $O(n^2)$, there are several reasons to believe that the performance would be better in practice.

5.5.1 *Initialization.* The edge flow functions for almost all known data flow frameworks are either identity functions or functions of the form $h(X) = A + X$ for the intersection problems (*viz.* MRA and LSIA) and $h(X) = A \cdot X$ for union problems (*viz.* CHSA). Thus $h(\top)$, required for the TR_0 construction (equations 13, 14), is almost always $\top$ and the number of operations per word reduces from $3\cdot d\_in_i + 3\cdot d\_out_i + 6$ to 3 — two operations for the node flow function and one for the meet with constant properties. For the unidirectional problems, since CONST_IN/CONST_OUT are typically $\top$, the number of operations further reduces to 2. These operations incorporate the local effect of a node and represent the minimum amount of work that must be done for any data flow problem. Note that though the bound on initialization is $O(n^2)$, it requires only one traversal over the graph.

5.5.2 *Propagation.* Let K be defined as follows :

$$K = \min\Big(\sum_{j=1}^{no\_w} n_j,\ \sum_{j=1}^{no\_w} b_j\Big) \tag{18}$$

Then the bound on the propagation becomes $O((e+n)\cdot K)$.

After the initialization is performed, we can evaluate K and determine a more realistic bound on the work required by propagation. In the best case, both no_w and K might be 1, in which case the work is $O(n)$. The actual work performed by the propagation is likely to be even better than the estimate in terms of K :

—We *do not* process the individual bits but words of bits.

δ_f	:	Traversal along a forward edge in direction δ
δ_b	:	Traversal along a back edge in direction δ
δ_f^-	:	Traversal along a forward edge in direction δ^-
δ_b^-	:	Traversal along a back edge in direction δ^-
δ_G	:	Traversal over the graph in direction δ
δ_G^-	:	Traversal over the graph in direction δ^-

Fig. 11. Generic notation for various traversals.

—In practice, the effect of the BOT properties of a node in the worklist *may not* propagate over the entire graph.

—Since propagation is delayed as far as possible, the effect of some nodes may be subsumed by propagation for other nodes. Hence the effect of all nodes in the worklist may not have to be propagated separately.

—Some other heuristics can be employed for the worklist organization to select appropriate nodes for propagation. In the case of MRA, forward node flow does not exist (i.e. f_k^f is $\top$) hence the information flow is predominantly backwards. Thus, it may be beneficial to process the nodes in the postorder as it may propagate the effect of some changes more rapidly.

6. APPLICATIONS OF THE GENERALIZED THEORY

This section discusses several interesting applications of the generalized theory [Dhamdhere and Khedker 1993]. Some general concepts which form the basis of these applications are presented in section 6.1. Section 6.2 defines the *width of a graph* which is shown to bound the number of iterations of round-robin analysis. Section 6.3 discusses the efficiency of data flow analysis and explains several known results using the theory.

6.1 Concepts and Definitions

For simplicity, we represent the forward and backward traversals generically by δ. For example, if δ is the forward direction, then a T^f is replaced by δ while a T^b is replaced by δ^-. Figure 11 summarizes the generic notation. We extend the notation of edge traversals to the traversals over the graphs. Thus T_G^f indicates a graph traversal in reverse postorder while T_G^b indicates graph traversal in postorder.[11] One traversal over the graph implies one iteration of the round-robin analysis.

A graph traversal cannot realize the effect of all kinds of edge traversals. The following definition captures the relationship between edge and graph traversals.

Definition 8. (*Conforming and non-conforming edge traversals*). For a δ_G traversal, δ_f and δ_b^-edge traversals are conforming edge traversals while δ_f^- and δ_b are non-conforming edge traversals.

[11]For a node i, T_G^f visits $in(i)$ followed by $out(i)$ while T_G^b visits $out(i)$ followed by $in(i)$.

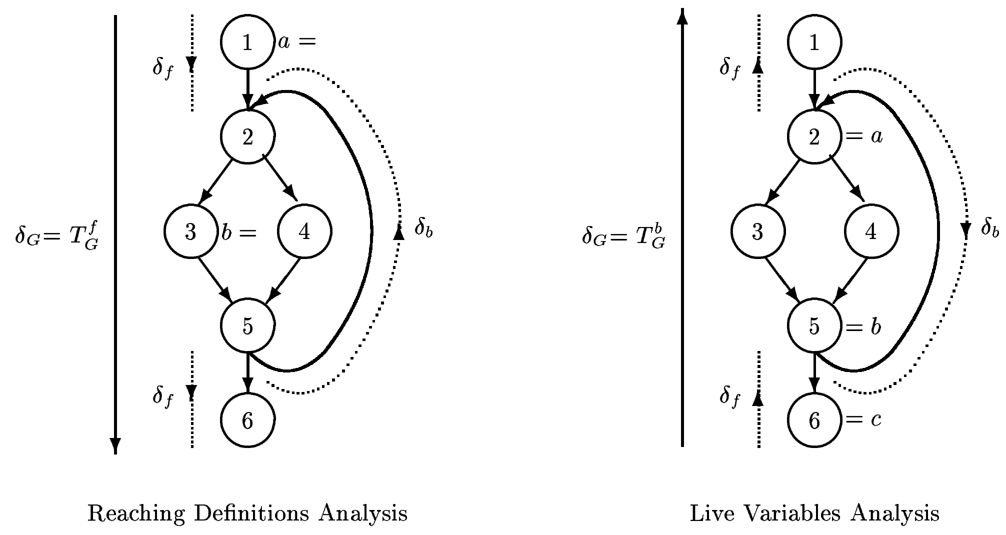


Fig. 12. Conforming and non-conforming edge traversals

A graph traversal realizes the effect of conforming edge-traversals, but fails to realize the effect of non-conforming edge traversals. This is illustrated in the following example.

Example 12. Consider program flow graphs in Figure 12. For the reaching definitions analysis, the graph is traversed in a reverse postorder. The fact that the definition of a in node 1 reaches all other nodes (via T^f_f edge traversals) is known in the first iteration over the graph. Similarly, the definition of b in node 3 is known to reach node 5 (T^f_f traversal) in the same iteration. However, the fact that this definition also reaches node 2 along the back edge (T^f_b traversal) is known only in the next iteration. Thus an T^f_b edge traversal is non-conforming whereas an T^f_f traversal is conforming.

Analogous comments hold for the graph for live variable analysis with T^b replacing T^f. In this case, the fact that b is live at $out(1)$ is known in the first iteration but its liveness at $out(5)$ is known in the next iteration only. □

An *ifp* consists of conforming and non-conforming edge-traversals.

Definition 9. (*Span*). A span is a maximal sequence of conforming edge traversals in an ifp.

Spans are separated by a non-conforming edge traversal and vice-versa. Thus, two successive non-conforming edge traversals have a *null span* between them. Further, an information flow path may begin and/or end with a null span.

The information along a span can be propagated in one δ_G traversal; the same graph traversal also realizes the information flow along the preceding non-conforming edge traversal.

Definition 10. (*Segment*). A segment is a maximal sequence of edge traversals in the same direction.

Successive T_e^f's constitute a *forward* segment while successive T_e^b's constitute a *backward* segment. A segment may be *bounded* or *unbounded*.

Example 13. Consider a T_G^f graph traversal and an *ifp*

$$\overbrace{\underbrace{T_f^f\ T_f^f\ T_f^f}\ \underline{T_b^f}}\ \underbrace{\overbrace{T_b^b}\ \ \overbrace{T_f^f}}\ \overbrace{\underline{T_f^b}\ \underline{T_f^b}}$$

The underbraces denote the (non-null) spans; overbraces, segments; and underscores denote the non-conforming edge traversals. Note that there is a null span between two successive T_f^b's. □

Example 14. From Table II, it is clear that the *ifp*'s of unidirectional data flow problems consist of a single unbounded segment. MRA and CHSA have *unbounded* backward segments. CHSA has unbounded forward segments too, while MRA has a bounded forward segment consisting of a single edge traversal. □

6.2 Complexity of Round Robin Iterative Data Flow Analysis

Definition 11. (*Information preserving path*). An ifp is an information preserving path (*ipp*) if all flow functions in the ifp are identity functions.

The edge flow functions of the type g^f (g^b) are said to be *clustered* if the information flow is identical for all out-edges (in-edges) of a node. For an *ifp* $< u, v, \rho >$, let $length(\rho)$ and $width(\rho)$ denote the total number of edge flow functions and the number of edge flow functions along non-conforming edge traversals, respectively.

Definition 12. (*Bypassed information flow path*). An ifp $< u, v, \rho_1 >$ is said to be bypassed by $< u, v, \rho_2 >$ if the edge flow functions are clustered, and

(1) either ρ_2 is an ipp or $length(\rho_2) = 1$, and
(2) $width(\rho_2) < width(\rho_1)$.

Intuitively, $< u, v, \rho_1 >$ is bypassed by $< u, v, \rho_2 >$ if the same information is guaranteed to flow along ρ_2 and $length(\rho_2) < length(\rho_1)$.

In practical data flow problems, bypassing usually occurs due to $length(\rho_2) = 1$.

Definition 13. (*Width*). The width w of a graph G for a data flow framework with respect to a traversal δ_G is the maximum number of non-conforming edge traversals along an ifp, no part of which is bypassed.

If we represent the number of spans by s then $s = w + 1$ for the width determining path.

THEOREM 2. *$w+1$ iterations are sufficient for the round-robin algorithm to converge on a fixed point.*

The information flow can be initiated only after the IN and OUT properties of all nodes are computed to determine the information originating within each node (i.e. the information represented by TR_0). This is achieved in the first iteration. The same iteration also realizes the propagation of information along a non-null span (if any) at the beginning of an *ifp*. However, every non-conforming edge traversal, and the span that follows it, requires a separate iteration. Thus, $w + 1$ iterations are sufficient for information propagation along the width determining path.

Now consider an *ifp* $< u, v, \rho_1 >$ such that $width(\rho_1) > w$. This is possible only if a section ρ' of ρ_1 is bypassed by another *ifp* ρ''. Let $width(\rho')$ be w', $width(\rho'')$ be w'' and $width(\rho_1)$ be w_1. Then $(w_1 - w') + w'' \leq w$. Again $w + 1$ iterations suffice for information to propagate from u to v.

Note that the width w of a graph G is defined for a data flow framework $\mathbf{D}$ and not for an instance of $\mathbf{D}$. For a particular instance $\mathbf{I} = < G, M >$, the number of may well be less.

Example 15. Consider the graph in Figure 2. For MRA, we choose $\delta_G = T_G^b$. T_b^b and T_f^f are the non-conforming edge traversals. The width for MRA is 3 along the width determining path (5,2,3,4,7,6) $= T_f^b \underline{T_f^f}\ \underline{T_b^b}\ \underline{T_f^f}\ T_f^b$. □

6.2.1 *The Width and the Depth.* Depth (d) is defined as the maximum number of back edges along any acyclic path [Aho, Sethi, and Ullman 1986]. To use the notion of width for unidirectional flows, choose δ as the natural direction of the flow in the problem. There are no flows along δ_f^- and δ_b^- edge traversals, and the only non-conforming edge traversal is δ_b. Since the width considers only those paths which do not have bypassed fragments, $w \leq d$. Thus, width provides a tighter bound on the number of iterations.

Example 16. Consider a *spiral graph* [Biswas, Bhattacharjee, and Dhar 1980] whose depth increases linearly with the nesting depth.[12] Here $d = 3$, while $w = 1$ for a unidirectional problem with clustered edge flow functions since every part of an *ifp* beginning on a back edge is bypassed, *viz.* path $(5, 2, 6)$ is bypassed by the path $(5, 6)$. This explains the observation that the number of iterations remains constant even as the size of the spiral graph grows. □

Further, the notion of depth assumes a fixed pattern for information flow governed by the directed paths in the flow graph, hence it is only applicable to unidirectional data flow problems.

6.3 Efficiency of Data Flow Analysis

When applied to the efficiency of data flow analysis, the generalized theory :

—motivates efficient solution techniques *viz.* interval analysis technique for MRA [Dhamdhere and Khedker 1993], and the method of alternating iterations,

—explains several known results in bidirectional flows.

In this section, we motivate the method of alternating iterations and explain some known results.

[12]Spiral structures result from **repeat** ... **until** loops with premature exits.

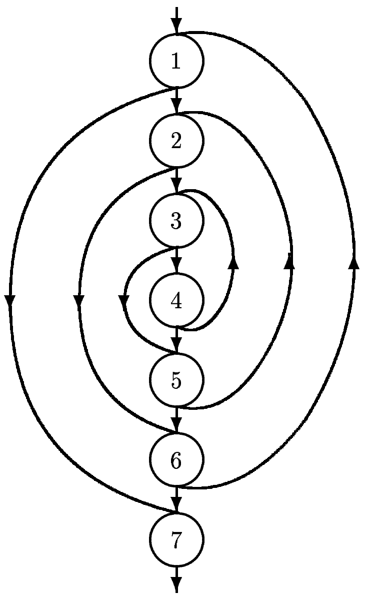


Fig. 13. A Spiral Graph

6.3.1 *Choice of Direction in Graph Traversal.* Consider a data flow problem whose *ifp*'s have unbounded segments in one direction and bounded segments in the other direction. Recall that $w = \#\delta_b + \#\delta_f^-$, i.e. width has contributions from the back edges in the δ segments and forward edges in the δ^- segments. $\#\delta_f^-$ is likely to be smaller when the segments in the δ^- direction are bounded rather than unbounded. Hence the appropriate direction for graph traversal is the one that makes the bounded segments lie in the δ^- direction, and unbounded segments, in the δ direction.

Example 17. MRA has unbounded backward segments but bounded forward segments, hence backward graph traversal would require fewer iterations than forward traversal (the experimental results are reported in [Dhamdhere and Khedker 1993]). For unidirectional problems, the favoured direction of traversal is trivially the direction of the data flow. □

6.3.2 *.Alternating iterations For problems with unbounded segments in both directions, alternating the direction of graph traversal between successive iterations can effectively reduce the solution complexity. This can be explained as follows : A segment in the δ direction may consist of a number of spans separated by non-conforming edges, which may themselves form sizable spans in the δ^- direction. Since the effect of a span in the δ direction is incorporated by a single iteration in the δ direction, alternating the direction of graph traversal between successive iterations would yield better results. Thus, the alternating iterations approach is clearly warranted in the case of CHSA.

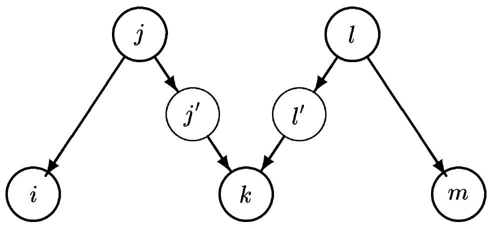


Fig. 14. Edge splitting

Let s_ρ be the number of non-null spans along an *ifp* ρ. The width of ρ for alternating iterations is defined as follows : $width_a(\rho) = 2s_\rho + c$, where $c = 1$ if ρ ends with a non-null span, else $c = 0$. The number of iterations for the method of alternating iterations is then $w_a + 1$ where w_a is defined analogous to w, *viz.* $w_a = \max(width_a(\rho))\ \forall \rho$, where ρ is an *ifp*, no part of which is bypassed.

6.3.3 *Reducing the Complexity by Reducing Width.* When a data flow problem has bounded segments in one direction, and unbounded segments in the other direction, complexity of the data flow analysis can be reduced by attempting to *truncate* the information flow paths. Consider an *ifp* $\rho = (\cdots, e_i, e_{i+1}, e_{i+2})$, where edge e_{i+1} constitutes a bounded segment of length 1, i.e. e_i and e_{i+2} belong to the segments in the opposite direction. Truncation can be effected by transforming the program flow graph or the data flow equations so as to terminate each *ifp* analogous to ρ *before* it reaches the edge e_{i+2}. Effectively, ρ is split into two *ifp*'s ρ_1 and ρ_2. Since $width(\rho_1), width(\rho_2) \leq width(\rho)$, this could reduce the width. In this section, we present two transformations based on this approach.

6.3.4 *.Edge splitting
An edge which runs from a *branch node* (i.e. a node with more than one successor) to a *join node* (i.e. a node with more than one predecessor) is called a *critical edge*. It has been reported [Dhamdhere, Rosen, and Zadeck 1992] that when such an edge is split by inserting a new node, the solution complexity of MRA is reduced. The following lemma captures the influence of edge splitting on the solution complexity of MRA.

LEMMA 10. *Following edge splitting, MRA can be solved with the complexity of a unidirectional problem.*

As a result of edge splitting the MRA *ifp*'s become $(T_e^b)^+(T_e^f \mid \epsilon)$ for the transformed graph, instead of the original $((T_e^b)^+(T_e^f \mid \epsilon))^+(T_e^b)^*$.

6.3.5 *.Edge placement The technique of *edge placement* eliminates a partial redundancy of an expression e in node i, which cannot be safely hoisted into a predecessor j, by creating a synthetic node along the edge (j, i) and hoisting e into it [Dhamdhere 1988a]. Unlike edge splitting, however, a synthetic node is conceptual; it does not participate in data flow analysis. It becomes real only when a computation is inserted in it during optimization phase following the data flow analysis.

Use of edge placement in MRA results in elimination of the Π term from the PPIN$_i$ equation. It thus transforms the data flow, rather than the flow graph, to achieve the same effect as edge splitting, *viz.* restricting the number of δ_f^- traversals along any *ifp* to zero. Solution efficiency vis-a-vis MRA is guaranteed by the fact that the resulting data flow is simply (backwards) unidirectional in nature.

6.3.6 *Decomposing Bidirectional Flows into Unidirectional Flows.* Decomposition of a bidirectional data flow problem into a sequence of unidirectional problems (i.e. solving a bidirectional problem as a sequence of cascaded unidirectional problems) is motivated by the desire to reduce the amount of work or to improve the understandability of the data flow involved. Prior work on decomposition has been ad hoc and/or directed at specific bidirectional data flow problems [Dhamdhere 1988a; Dhamdhere, Rosen, and Zadeck 1992; Knoop, Ruthing, and Steffen 1992]. In this section we provide a condition for the decomposability of a bidirectional data flow problem.

Observation 1. For a program graph G, $w^u \leq w^b$ where w^u and w^b are the widths of G for arbitrary unidirectional and bidirectional data flows with respect to δ_G such that δ_G is along the natural direction of data flow for the unidirectional problem. □

LEMMA 11. *It is feasible to decompose a bidirectional data flow problem into a sequence of unidirectional data flow problems if and only if the number of segments in every information flow path for the data flow problem is bounded by a constant.*

PROOF. Let δ be the direction of the first segment in an information flow path. Information propagation along the segment can be realized by a unidirectional data flow problem which has δ as its natural direction of flow. Information flow along the following segment would require a unidirectional problem in the opposite direction, etc. Thus, the number of unidirectional problems required will equal the number of segments, which should be bounded by a constant for the decomposition to be feasible. Further, the order of solving the unidirectional problems will have to be the same as the order of segments in the information flow paths. □

COROLLARY 1. *MRA cannot be solved by cascaded unidirectional problems.*[13]

The *ifp*'s of MRA have the form $((T_e^b)^+(T_e^f \mid \epsilon))^+(T_e^b)^*$, thus they may consist of an unbounded number of forward and backward segments. Similar statements hold for the LSIA and CHSA problems.□

COROLLARY 2. *It is possible to decompose MRA if edge splitting is performed.*

Following section 6.3, the information flow paths in the resulting program flow graph can be characterized by the regular expression $(T_e^b)^+(T_e^f \mid \epsilon)$. Since the number of segments can at most be 2, it is possible to solve MRA

[13]unless edge splitting is performed.

by cascaded unidirectional problems. Further, since the second (i.e. the forward) segment has a length ≤ 1, it is possible to solve MRA on a graph in which critical edges have been split as a backward problem followed by a *forward correction* [Dhamdhere, Rosen, and Zadeck 1992].□

COROLLARY 3. *It is not possible to decompose CHSA even if edge splitting is performed.*

Since both forward and backward segments are unbounded for CHSA, edge splitting does not truncate any *ifp* and the *ifp* pattern remains $T_e^f(T_e^b \mid T_e^f)^*$. Hence the number of segments remains unbounded.□

7. DISCUSSION

Classical data flow analysis uses two key features : *properties of graph structures* and *patterns of information flow.* Different methods blend and use these features in different ways. Elimination methods use graph regions to divide the data flow problems into smaller subproblems, while round-robin methods follow postorder or reverse postorder for graph traversal. Worklist versions do not use properties of the graphs but follow the data flow pattern dynamically.

One common thread in these techniques is a strictly unidirectional flow of information, which is a most fundamental assumption in the classical theory of data flow analysis. This assumption results in a theory which makes no distinction between the edge and node flows. Hence it cannot handle the spouse and sibling effects which arise when information flows in both the directions. We eliminate this restriction by proposing a more general concept of information flow. Simple as it may seem, this generalization has far reaching consequences, both on the theory and practice of data flow analysis.

On the theoretical side, the contributions of this research include a generalized characterization of information flow and safety of an assignment. These results are based on sound theoretical foundations and are applicable to unidirectional as well as bidirectional data flow problems. This unification shows that the bidirectional problems are inherently no more complex than the unidirectional problems. Such a result – though quite significant – should not come as a surprise. Both the unidirectional and the bidirectional problems need to compute the same number of properties and a property changes only once regardless of the direction of information flow.

On the practical side, we propose a worklist-based generic algorithm, which has a unique advantage in that after TR_0 computation, it is possible to estimate a *more realistic bound* on the work that might be needed for a particular instance of the problem. If K (defined in section 5.5) is small, which is quite likely in practice, we can proceed without any fear of the quadratic behaviour of the algorithm. Though the theoretical complexity of propagation is $O(n^2)$, refinement reduces the number of operations further by a constant factor.

The algorithm computes maximally consistent (i.e. MFP) and comprehensive (i.e. MOP) information for all singular bit vector data flow frameworks.

Apart from the generic algorithm, the theory has several interesting applications. We explore the complexity of data flow analysis using the generalized theory. The most important outcome of this application is the notion of the width of a graph which is shown to bound the number of iterations required for round-robin iterative data flow analysis. This notion is uniformly applicable to unidirectional and bidirectional flows and provides a more accurate bound than the traditional notion of the depth of a graph. More importantly, width provides the first (strict) bound on the round-robin analysis of bidirectional flows. Other applications include explanation of isolated results in efficient solution techniques and motivation of new techniques for bidirectional flows. In particular, we discuss edge-splitting, edge placement and develop a feasibility criterion for decomposition of a bidirectional flow into a sequence of unidirectional flows.

7.0.7 *.Applicability of the generalized theory Though the exposition of the theory in this paper is restricted to bit vector problems only, it is applicable to all bounded monotone data flow frameworks which possess the property of separability of solution. Let the effective height of $\mathcal{L}$ be H. Thus, a property may assume at most $H + 1$ values during data flow analysis. This has the following consequences :

—*MBVP* now implies : A node variable changes from X_1 to X_2 where $X_1 \sqsupset X_2$.
—The notion of information flow now becomes : Information flows from a program point u to a program point v when a change in a property at u causes the corresponding property at v to change.
—A property may change H times rather than only once, hence a program point may appear H times in an *ifp*.

These changes do not constrain any proposition in the theory except that a property may have to be processed H times rather than once. The generic algorithm is also applicable to such problems by

—replacing the single bit representation of a data flow property by a suitable data structure.
—replacing the words “is BOT” and “becomes BOT” by “is not TOP” and “changes” respectively.[14]

If H is independent of n (i.e. the number of nodes), the complexity of the algorithm remains same.

ACKNOWLEDGMENTS

The authors wish to thank the referees for many helpful suggestions.

REFERENCES

Aho, A. V., Sethi, R., and Ullman, J. D. 1986. *Compilers – Principles, Techniques, and Tools.* Addison-Wesley.

[14]TOP and BOT retain their usual meanings i.e. TOP refers to the value of a property in a variable represented by the $\top$ element of the lattice while BOT refers to the value of a property in a variable represented by the $\perp$ element of the lattice.

Allen, F. E. and Cocke, J. 1977. A program data flow analysis procedure. *Communications of ACM 19,* 3, 137–147.

Biswas, S., Bhattacharjee, G. P., and Dhar, P. 1980. A comparison of some algorithms for live variable analysis. *International Journal of Computer Mathematics 8,* 121–134.

Chow, F. C. 1988. Minimizing register usage penalty at procedure calls. In *Proceedings of SIGPLAN'88 Symposium. on Compiler Construction*, pp. 85–94. Also Published as *SIGPLAN Notices*, 23(7).

Dhamdhere, D. M. 1988a. A fast algorithm for code movement optimization. *ACM SIGPLAN Notices 23,* 10, 172–180.

Dhamdhere, D. M. 1988b. Register assignment using code placement techniques. *Computer Languages 13,* 2, 75–93.

Dhamdhere, D. M. 1991. Comments on practical adaptation of the global optimization algorithm by Morel & Renvoise. *ACM Transactions on Programming Languages and Systems 13,* 2, 291–294.

Dhamdhere, D. M. and Khedker, U. P. 1993. Complexity of bidirectional data flow analysis. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages*, pp. 397–408.

Dhamdhere, D. M. and Patil, H. 1993. An elimination algorithm for bidirectional data flow analysis using edge placement technique. *ACM Transactions on Programming Languages and Systems 15,* 2, 312–336.

Dhamdhere, D. M., Rosen, B. K., and Zadeck, F. K. 1992. How to analyze large programs efficiently and informatively. In *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

Graham, S. and Wegman, M. 1976. A fast and usually linear algorithm for global data flow analysis. *Journal of ACM 23,* 1, 172–202.

Hecht, M. S. 1977. *Flow Analysis of Computer Programs.* Elsevier North-Holland Inc.

Joshi, S. M. and Dhamdhere, D. M. 1982a. A composite algorithm for strength reduction and code movement : part I. *International Journal of Computer Mathematics 11,* 1, 21–44.

Joshi, S. M. and Dhamdhere, D. M. 1982b. A composite algorithm for strength reduction and code movement : part II. *International Journal of Computer Mathematics 11,* 2, 111–126.

Kam, J. B. and Ullman, J. D. 1977. Monotone data flow analysis frameworks. *Acta Informatica 7,* 3, 305–318.

Kennedy, K. 1972. Safety of code movement. *International Journal of Computer Mathematics 3*, 112–130.

Khedker, U. P. and Dhamdhere, D. M. 1992. A generalized theory of data flow analysis. Technical report TR-070-92, Department of Computer Science and Engineering, Indian Institute of Technology, Bombay.

Kildall, G. 1973. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM Symposium on Principles of Programming Languages*, pp. 194–206.

Knoop, J., Ruthing, O., and Steffen, B. 1992. Lazy code motion. In *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.* Also Published as *SIGPLAN Notices*, 27(7).

Marlowe, T. J. and Ryder, B. G. 1990. Properties of data flow frameworks. *Acta Informatica 28*, 121–163.

Morel, E. and Renvoise, C. 1979. Global optimization by suppression of partial redundancies. *Communications of ACM 22,* 2, 96–103.

Muchnick, S. S. and Jones, N. D. 1981. *Program Flow Analysis : Theory and Applications.* Prentice-Hall, Inc.

Rosen, B. K. 1980. Monoids for rapid data flow analysis. *SIAM Journal of Computing 9,* 1, 159–196.

Ryder, B. G. and Paull, M. C. 1986. Elimination algorithms for data flow analysis. *ACM Computing Surveys 18*, 277–316.

Tarjan, R. E. 1981a. Fast algorithms for solving path problems. *Journal of ACM 28,* 3, 594–614.

Tarjan, R. E. 1981b. A unified approach to path problems. *Journal of ACM 28,* 3, 577–593.

Zadeck, F. K. 1984. Incremental data flow analysis in a structured program editor. In *Proceedings of SIGPLAN'84 Symposium. on Compiler Construction*, pp. 132–143. Also Published as *SIGPLAN Notices*, 19(6).