# INCREMENTAL DATA FLOW ANALYSIS

Barbara G. Ryder

Department of Computer Science
Rutgers University
New Brunswick, New Jersey 08903

**Abstract**

In this paper we present **ACINCF** and **ACINCB**, incremental update algorithms for forward and backward data flow problems, which are based on a linear equations model of Allen/Cocke interval analysis [Allen 77, Ryder 82a]. We have studied their performance on a robust structured programming language *L*. Given a set of localized program changes in a program in *L*, we can identify *a priori* the nodes in its flow graph whose corresponding data flow equations will be affected by the changes. We can characterize these affected nodes by their corresponding program structures and their relation to the original change sites.

## 1. Introduction

A global data flow algorithm gathers information about the definition and use of data in a program or a set of programs. The algorithm is usually applied on some intermediate form of a program. It may be a flow graph which describes the control flow among basic blocks in a procedure [Hecht 77]. It also can be a call graph which describes the calling relations between procedures in a program [Allen 74, Ryder 79] or a parse tree representation of a procedure [Farrow 75, Kennedy 77]. In all of these, we have a digraph representation of control flow. Each node has associated data flow constants which describe how the code at the node affects data flow in the program.



Data flow analysis algorithms gather this local information and infer the global data flow from it. This global information then is specialized to provide data flow information with respect to any node in the digraph.

An incremental update algorithm for data flow analysis modifies a known data flow solution to reflect changes in a problem; that is, an incremental update algorithm obtains the new solution of an altered problem without application of the algorithm that originally solved it. We have designed and analyzed incremental update data flow algorithms based on elimination methods [Ryder 82a]. Given a program and a known data flow problem solution, the effects of a set of localized program changes can be determined without full re-analysis of the program by some global data flow algorithm. The ease of incremental updating depends upon the data flow algorithm involved, the type of data flow problem (i.e., forward or backward), the program changes allowed (i.e., control flow changes that affect the structure of the digraph or changes that affect the local data flow characteristics of a node) and the structure of the digraph (i.e., the presence of nested loops, the edges/nodes ratio etc.).

By developing models of elimination algorithms, which show how they solve the systems of linear equations that describe the data flow problems, our investigations were focused on the question "can we allow a small change in a structured system of linear equations, whose solution is already known, and find the effects of that change without totally re-solving the system?" [Ryder 82b]

Our model of Allen/Cocke interval analysis traces the solution of a sequence of progressively smaller systems of linear equations. By incrementalizing this model, we developed incremental update algorithms **ACINCF** and **ACINCB** for forward and backward data flow problems, respectively [Ryder 82c] [1]

We studied the performance of **ACINCF** and **ACINCB** on a robust, Algol-like structured programming language *L*, with loop exit structures similar to those of Sail [Reiser 76]. We identified program structures that affect the complexity of incremental updating and established the extent of this effect offered by combinations of such structures. Specifically, for reducible digraphs we have shown that the elimination phase of **ACINCF** updates on each linear system in the derived sequence, a set of interval head equations and **at most** the equations in **one** interval A similar result holds for **ACINCB**.

The depth of loop nesting directly affects the complexity of incremental updating; it also is a key element in the calculation of the worst case complexity of Allen/Cocke interval analysis. We showed that for a reducible flow graph with loop nesting depths bounded by a constant, disregarding the work of interval finding, Allen/Cocke interval analysis has an **O(n)** worst case complexity bound [Ryder 82a]. Carrying the analysis of our incremental algorithms further, we considered program changes within one interval in a nested loop in a program in *L* and characterized the set of all variables whose equations may be affected, in terms of each variable's corresponding program structure and its relation to the original change site Our result enables us to analyze a flow graph in *L* with a set of possible program changes identifying *a priori* nodes whose equations will be affected by these changes. Thus, we ascertain all the data flow solutions affected by the changes.

At the outset of this research, we found no previous published work in incremental data flow analysis; communication with F. Allen confirmed this fact [Allen 79]. At the Eighth Annual ACM Symposium on the Principles of Programming Languages in 1981, B Rosen emphasized the need for incremental update algorithms in general; he used global data flow algorithms for illustration. We concur in his opinion of the inappropriateness of the conventional worst case error bounds for these algorithms. Our theoretical studies of algorithm performance on *L* provide better insight into incremental algorithm complexity.

There are many applications for incremental global data flow algorithms. Interprocedural data flow analysis, in which the call graph of a program is analyzed with respect to data flow in global variables and parameters is an obvious application area [Allen 74, Ryder 74]. Interval analysis based interprocedural data flow algorithms exist [Allen 73, Sharir 79]. Often in the development of large software systems, the set of procedures remains constant while the data flow characteristics of some procedures change. **ACINCF** and **ACINCB** can be adapted to handle these changes and to perform incremental interprocedural data flow analysis. Source-to-source transformation systems depend on data flow information to validate the triggering of certain transformations that subsequently change the source code and may change its data flow characteristics [Burstall 71, Kibler 77, Loveman 77, Paige 77]. Currently, *ad hoc* methods are used to accommodate data flow changes, incrementalizing data flow algorithms that use a parse tree representation of the program would be preferable [Babich 78a, Babich 78b, Rosen 77] as in [Reps 82]. Interactive programming environments are popular useful tools for software development [Alberga 81]. Here, data flow information can aid in debugging and documentation.

In the remaining sections of this paper, first we present an overview of our linear equations model of Allen/Cocke interval analysis. Second, we summarize the

[1] We also modelled Hecht/Ullman T1-T2 analysis [Ullman 73] and Tarjan interval analysis [Tarjan 74, Tarjan 79] and designed **HUINC**, an incremental algorithm based on the Hecht/Ullman algorithm, which was compared with **ACINCF**.

derivations of **ACINCF** and **ACINCB**. Third, we show why worst case error analysis is inappropriate for incremental analysis. Fourth, we explain our results on the performance of our algorithms on reducible flow graphs in general and on flow graphs of programs in $\mathcal{L}$ in particular. Fifth, we outline our plans for the implementation of our algorithms. Finally, we summarize the results presented here which appear in detail in [Ryder 82a].

## 2. Allen/Cocke Interval Analysis Model

In this section we define the linear data flow equations of interest. Then we describe our model of Allen/Cocke interval analysis which shows how Gaussian elimination techniques are used to solve the system of linear data flow equations [Ryder 82b].

We are interested in those data flow problems that are defined by a system of linear equations involving the binary operators of union and intersection. We use the term **linear** to refer to a system of equations defined using two binary operators whose properties enable us to apply Gaussian elimination techniques to obtain a solution [Paull 82]. Depending upon the implementation, the solution of these equations can involve bit vector or set operations. Each variable in the linear system of **n** equations is associated with a unique node in the digraph of **n** nodes.[2] Each edge in the digraph corresponds to a term in the linear system.

The general form of a forward data flow equation is:

$$X_m = \underset{j \in \mathbf{pred}\{m\}}{\phi} \{ a_{m,j} \cap X_j \cup b_{m,j} \} \cup c_m \qquad (1)$$

where $\phi$ can be union or intersection, $a_{m,j}$ $b_{m,j}$ $c_m$ are constants relating to data flow through nodes m and j, and **pred{m}** is the set of immediate predecessor nodes of m in the digraph. For every term $X_j$ in equation 1 there is a corresponding edge (j,m) in the digraph. The general form of a backward data flow equation is similar to equation 1; the difference is that the limit on $\phi$ is $j \in$ **succ{m}** where **succ{m}** is the set of immediate successor nodes of m in the digraph.

The following description of our model of Allen/Cocke interval analysis shows how a system of these linear data flow equations is solved. We partition the variables into subgroups called **intervals** [Allen 71]. Each equation in an interval is reduced by variable substitutions to a linear function of one variable, the **interval head variable**. The interval headed by node h is denoted $I_h$. When the equations in an interval are written according to a linear order of the variables called **interval order** [Allen 71], their coefficient matrix has a lower triangular structure for a forward data flow problem and an upper triangular structure for a backward data flow problem. Thus, forward substitution or backward substitution within an interval accomplishes the reduction of the equations in that interval.

The equation of each interval head variable is reduced by variable substitutions to a linear function of other interval head variables. We form the derived linear system defined by these reduced interval head variable equations. We define intervals and repeat this process which is analogous to the variable substitution phase of Gaussian elimination. Finally, one variable remains; the solution for this variable is found and then we begin a process which is analogous to the back substitution phase of Gaussian elimination.[3] We identify each solution with an interval head variable in the previous linear system and then use its value to back substitute in the reduced equations in the previous system. In a forward data flow problem, this process involves substituting the solution for the interval head variable into all reduced equations of that interval. In a backward data flow problem, this process involves substituting the solution for each interval head variable into all reduced equations which are dependent upon it. In the latter case, the solution for a variable is a linear

[2] Because of this one-to-one relation between nodes and variables, we can use these terms interchangeably.

[3] We assume the digraph is reducible; irreducible graphs are rare and can be handled by node splitting techniques [Hecht 77, Kennedy 74].

combination of interval head variable solutions rather than a linear function of one interval head variable solution as in the former case. We repeat this back substitution process until all solutions in all the linear systems have been obtained

### 3. Overview of ACINCF and ACINCB

Given a localized set of program changes, our incremental update algorithms **ACINCF** and **ACINCB** consist of two phases corresponding to the two phases of our model in Section 2.. First, all coefficients and constants in all data flow equations affected by the changes are recalculated. Second, all affected solutions are recalculated.

In Figure 1 we show the coefficient structure of the equations of a forward data flow problem for the variables in $I_h$ assuming the variables are ordered in an interval order. Possibly non-zero coefficients are indicated by x or *. Given a change at node $m \in I_h$ the * indicate the region in which coefficients are possibly affected. We must recalculate the reduced equations in this region when necessary. Then we must check to see if there has been an interval head variable equation affected by these changes. If no interval head variable equation has been affected, we are finished tracing coefficient/constant changes. Otherwise, there is an interval head variable $X_j$, such that the reduced equation for $X_j$ is the equation for $X_y$, where y represents $I_j$ in the derived linear system. In that system, we find the interval containing y, $I_r$. Then we can find the reduced equations in $I_r$ that are affected by this change in the equation for $X_y$ as previously. We must iterate this process through the sequence of systems of linear equations until all coefficient/constant changes have been propagated as far as possible.

Eventually, either we will find a linear system where no interval head variable equation is affected, or we will reach the last system of equations. In the former case, we can re-perform the back substitutions in the changed reduced equations in this system, obtaining new solutions. In the latter case, we can solve the equation for the last variable in the system, obtaining a new solution. In either

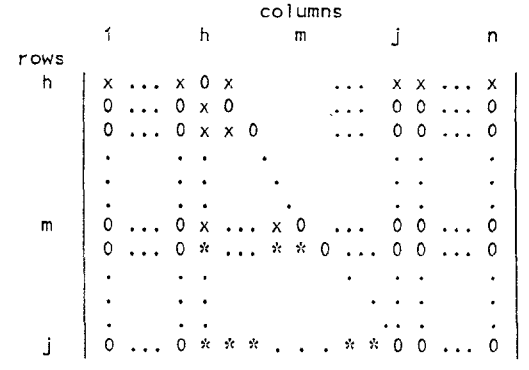


**Figure 1:** Affected Coefficients, Forward Problem

case, we identify each changed solution in the final system with an interval head variable solution in the previous system. We substitute each changed interval head solution in the reduced equations for all variables within that interval, obtaining all solutions in the interval. This back substitution process continues through the sequence of linear systems in the reverse of derived order, updating all solutions corresponding to changed equations and/or changed interval head variable solutions.

Our incremental update algorithm **ACINCB** is similar to **ACINCF**; the differences deal with the dissimilarities in the coefficient matrix structure and the reduced equation form. Figure 2 is analogous to Figure 1, showing the coefficient structure of the equations of a backward data flow problem for the variables in $I_h$, assuming the variables are ordered in an interval order.

Given a change at node $m \in I_h$ the * indicate the region in which coefficients are possibly affected. We must recalculate the reduced equations in this region when necessary. Then we must check to see if there has been an interval head variable equation affected by these changes. Because an interval is a single-entry, connected subgraph there can be only one interval head equation affected by these changes, namely the equation of $X_h$. As in the forward case, the reduced equation for $X_h$ is the equation of $X_y$ where y represents $I_h$ in the derived linear

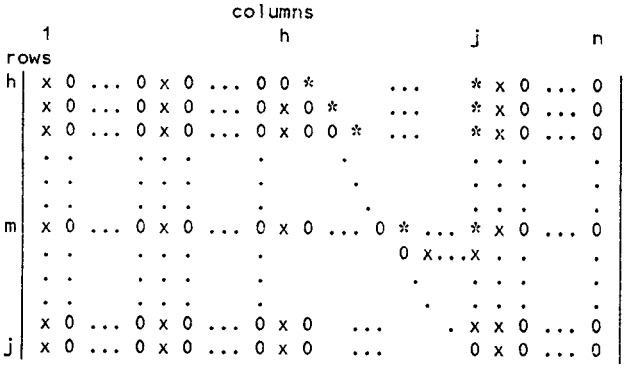


**Figure 2:** Affected Coefficients, Backward Problem

system. We find the interval containing y in the derived system, $I_r$. Then we can find the reduced equations in $I_r$ that are affected by this change in the equation for $X_y$ as previously. We must iterate this process through the sequence of systems of linear equations until all coefficient/constant changes have been propagated as far as possible.

Finally, we reach the last linear system and solve for the final variable. We identify this solution with an interval head variable solution in the previous system. Then we must recalculate all solutions in the previous system corresponding to reduced equations containing that interval head variable. We also recalculate all solutions in the previous system corresponding to an equation whose coefficients/constants have been changed by the change propagation phase. This back substitution process continues through the sequence of linear systems in the reverse of derived order, updating all solutions corresponding to changed equations and/or changed interval head variable solutions.

## 4. Complexity

Worst case error analysis is inappropriate for incremental update algorithms. There are pathological digraphs on which incremental updating is tantamount to re-performing the data flow analysis algorithm. Ullman presented such a digraph of **n** nodes [Ullman 73] shown in Figure 3 for n=8; the annotations represent definitions of

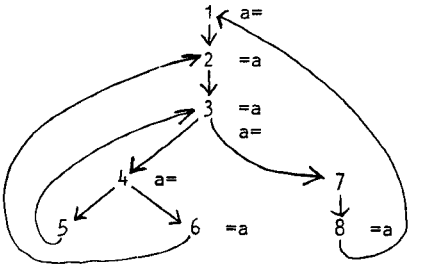


**Figure 3:** Pathological Digraph for Allen/Cocke Algorithm

variable a (a=) and uses of variable a (=a).

Consider applying **ACINCF** to update the solution of the reaching definitions problem [Hecht 77] on this digraph. Deletion of the definition of variable **a** at node 3 requires the recalculation of all the reduced equations in all the linear systems associated with this example, it also causes all the solutions to be recalculated. On a graph of this type with n nodes, the worst case complexity of **ACINCF** and that of Allen/Cocke interval analysis is bounded above by $O(n^2)$. However, such heavily nested loop structures are uncommon in modern programming language usage [Allen 79, Elshoff 76, Kennedy 77, Knuth 71, Robinson 76]. Therefore, this is not an appropriate measure of the complexity of **ACINCF**. We have shown that on a reducible digraph of **n** nodes with loop nesting depth bounded by a constant, the equation solution work of Allen/Cocke interval analysis exhibits linear worst case performance (i.e., **O(n)**) [Ryder 82a].

Since a loop is a strongly connected component of the digraph, it is reasonable that a program change in a loop may affect every solution in that loop. The back substitution work necessary to obtain those new solutions is proportional to the number of nodes in the loop The equation update work is bounded as well. Given any reducible digraph with a localized set of changes (i.e., all changes within one interval in the original linear system), the effects generated by these changes on any of the derived systems of equations are limited. That is, for a forward data flow problem in any derived system the equations

affected consist **only** of a set of interval head equations and, **at most**, the equations in **one** interval in the system For a backward data flow problem in any derived system the equations affected consist **only** of an interval head equation and, **at most**, the equations in **one** interval in the system [Ryder 82c].

To refine our understanding of the equation updating process we defined a robust, structured programming language *L* which consists of straight-line code (e.g., **assignment**, i/o statements), **while** statements, compound **if** statements, **done** <**label**> statements and **continue** <**label**> statements. We assumed semantics for these statements similar to those in Sail [Reiser 76]. The **done** <**label**> statement causes control to pass to the statement following the while loop labeled <label> or following the syntactically innermost while loop containing the **done** statement, if <**label**> is null. The **continue** <**label**> statement causes control to pass to the test of the while loop labeled <label> or to the test of the syntactically innermost while loop containing the **continue** statement, if <**label**> is null.

In order to state our complexity results, we used the term **g-loop** to refer to an interval in a program written in *L* noting that virtually but not all intervals correspond to while loops. If g-loop h is syntactically nested within loop w, then g-loop h is a **descendant g-loop** of loop w; by syntactic nesting we mean that there is a path from the entry node of loop w to the entry node of g-loop h in the digraph and a path from the entry node of g-loop h to the entry node of loop w. Likewise, loop w is a **parent loop** of g-loop h. If g-loop h is syntactically nested within loop w and is **not** nested within any other loop that is nested in loop w, then g-loop h is an **immediate descendant g-loop** of loop w; loop w is the **immediate parent loop** of g-loop h If two g-loops have the same immediate parent loop, they are **sibling g-loops**. A g-loop q is a **right (left) sibling g-loop** of g-loop h, if both are sibling g-loops such that there is a path from an exit of g-loop h(q) to the entry node of g-loop q(h). We abbreviate these terms **rsib** and **lsib**.

We categorized g-loops with respect to when all their variables are eliminated from the sequence of linear systems G-loop s is a **right greater (equal to) sibling** of g-loop h if g-loop s is an rsib of g-loop h and all the variables in g-loop h are eliminated before (at the same time as) all the variables in g-loop s We abbreviate these terms **rgsib** and **resib**. Corresponding definitions exist for **lgsib** and **lesib**. In Figure 4, loop w is a parent loop of g-loops h, s, t and q and an immediate parent loop of all but g-loop t. G-loop q is an resib of g-loop h, g-loop h is an lesib of g-loop q. g-loop s is an rgsib of g-loop h and g-loop s is an lgsib of g-loop q. As seen in Figure 4, these sibling relations roughly correspond to a measure of the loop nesting depth within a g-loop. When all of the variables in a g-loop have been eliminated from the linear system, we call that g-loop **collapsed** in that system.

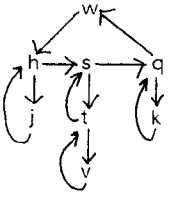


**Figure 4:** Examples of G-loops

Given a program written in *L* with a forward data flow problem solution and a set of localized program changes, we characterized the equations affected by these changes in terms of their corresponding program structures and the sites of the original changes. Also, we indicated the degree of increased complexity introduced by loop exit statements used singly or in concert.

Figure 5 depicts our results for **ACINCF** on a k-nested g-loop in a program in *L*. Each triangle with top vertex labelled p represents a set of paths through the interval $I_p$. The entry nodes of the nested loops are $p_k,\ldots,p_2,p_1=r$ ordered from outermost to innermost. If program changes occur in $I_r$ then the variables whose equations may be affected correspond to nodes along the dashed paths in Figure 5 If no **done** statements occur

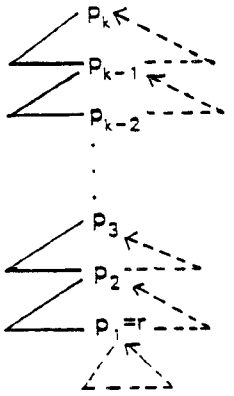


**Figure 5:** **ACINCF** on a k-nested Loop

within the loop $p_k$, these variables correspond to the entry nodes of: rgsibs or resibs of g-loop r, parent loops of g-loop r, **or** rgsibs or resibs of parent loops of g-loop r. If **done** statements do occur, the variables correspond to the entry nodes of: rsibs of g-loop r, parent loops of g-loop r, **or** rsibs of parent loops of g-loop r.

Figure 6 shows the same k-nested loop structure as Figure 5. It illustrates the behavior of **ACINCF** during a step of the change propagation phase. Nodes corresponding to variables with affected equations are indicated by dashed circles or lie on dashed paths. Assume the equation of $X_q$ was changed in the previous linear system and that g-loop q is collapsed in the current linear system. Consider which equations can be affected by the changes in the equation for $X_q$.[4] There are three cases:

1. If there are no **done** or **continue** statements in loop $p_k$, then variables corresponding to entry nodes of the immediate parent loop of g-loop q, loop $p_{n+1}$, or rgsibs of q can be affected.

2. If there are no **done** statements, but there are **continue** statements, then possibly affected variables correspond to the entry nodes of rgsibs of g-loop q and a subset of $\{p_{n+1}, p_{n+2}, \ldots, p_k\}$.

3. If there are **done** and **continue** statements in loop $p_k$, then possibly affected variables correspond to entry nodes of rgsibs of g-loop q, a subset of $\{p_{n+1}, p_{n+2}, \ldots, p_k\}$ **and/or** nearest rsibs of a subset of $\{p_{n+1}, p_{n+2}, \ldots, p_k\}$.

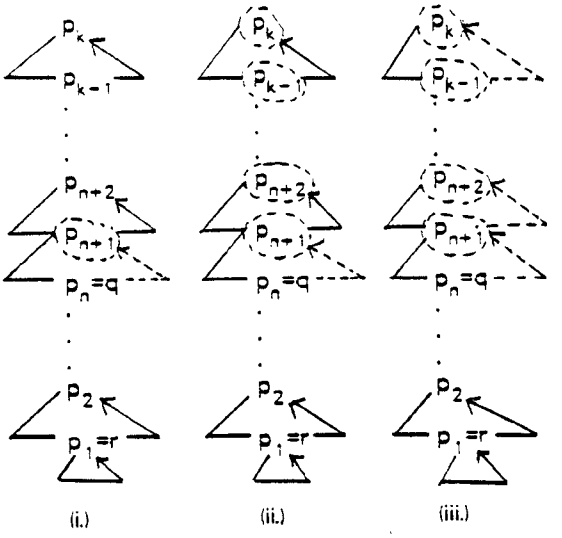


**Figure 6:** Snapshot of **ACINCF**

Figure 7 is analogous to Figure 5 and depicts our results for **ACINCB** on a k-nested g-loop in a program in $\mathcal{L}$. If program changes occur in $l_r$ then the variables whose equations may be affected correspond to nodes along the dashed paths in Figure 7. Any variables affected correspond to the entry nodes of a parent loop of g-loop r, an lgsib or lesib of a parent loop or an lgsib or lesib of g-loop r itself. In this case, the presence of **done** and **continue** statements does not affect our result.

Figure 8 shows the same k-nested loop structure as Figure 7. It illustrates the behavior of **ACINCB** during a step of the change propagation phase. Assume the equation of $X_q$ is changed in the previous linear system and g-loop q is collapsed in the current linear system. Irrespective of the type of loop exit statements in loop $p_k$, the same variables can be affected as a result of this change because the data flow information travels in the

[4] The code in loop $p_k$ determines which of these possibly affected variables are actually affected by this change.

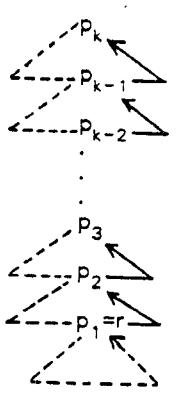


**Figure 7:** **ACINCB** on a k-nested Loop

direction which is the reverse of the control flow. The nodes corresponding to these affected variables are indicated by dashed circles or lie on dashed paths in Figure 8. They are the entry nodes of $p_{n+1}$ the immediate parent loop of g-loop q or lesibs or lgsibs of g-loop q.

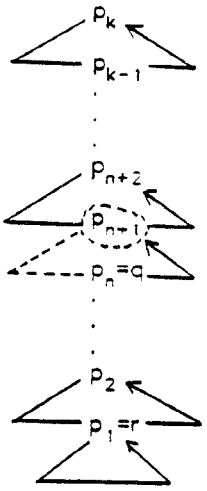


**Figure 8:** Snapshot of **ACINCB**

The analytic results described in this section enable us to perform *a priori* analysis of the data flow effects of program changes. They limit the variable substitution work of **ACINCF** and **ACINCB** to a prescribed set of equations related to the g-loop structure of the program near the changes [Ryder 82c].

## 5. Future Work

The obvious next step in our work is to implement **ACINCF** and **ACINCB** for a widely used high level programming language such as Pascal or C and gather empirical "average complexity" information on algorithm performance. We also can gather more current information on programming language usage to augment the empirical studies cited here. This will enable us to concentrate our attention on program changes which occur sufficiently often to insure that our efforts to accommodate them in incremental updating will "pay off".

We envision the use of our incremental update algorithms as part of an interprocedural data flow analysis tool. Our experience with the PFORT Verifier attests to the need for even the most rudimentary data flow information with respect to interprocedural analysis of software systems [Ryder 74, Ryder 79]. In software maintenance there is a need to delineate the scope of a system change; today often a "let's try it and see" attitude prevails. Large applications often maintain histories of source code changes during system development and maintenance. Studies of these histories would yield information about the kinds of changes large systems are likely to undergo. This information alone would be valuable to software designers, as the transformation of a set of algorithms and data structures into a working software system is not well understood in large practical applications, although various software design techniques exist.

## 6. Summary

We have presented incremental update algorithms for data flow analysis based on Allen/Cocke interval analysis. We have shown the inappropriateness of worst case error bounds for these incremental algorithms. We have complexity results for our algorithms on a robust, structured programming language *L*, which enable us *a priori* to characterize those variables whose equations are affected by a set of localized program changes with respect to their corresponding program structures and the original site of the changes. These results verify the desirability of incremental update algorithms for data flow

analysis. We have indicated areas of application of these methods. We have outlined our implementation plans for continuing work in this area.

## 7. Acknowledgments

We acknowledge the help and support of Marv Paull, our thesis adviser, throughout our research. We also thank F. Allen, J Ferrante, B. Rosen and M. Wegman for their helpful comments on this presentation.

## References

[Alberga 81] Alberga, C. N., Brown, A. L., Leeman, G. B., Mikelsons, M. and Wegman, M. N. A Program Development Tool In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 92-104. Association for Computing Machinery-SIGPLAN, January, 1981.

[Allen 71] Allen, F. E. A Basis for Program Optimization. In *Proceedings of 1971 IFIP Congress*, pages 385-390. Institute of Electrical and Electronics Engineers, Inc., North Holland Publishing Company, Amsterdam, Holland, 1971.

[Allen 73] Allen, F. E. and Schwartz, J. T. Determining the Data Relationships in a Collection of Procedures. 1973.

[Allen 74] Allen, F. E. Interprocedural Data Flow Analysis. In *Proceedings of 1974 IFIP Congress*, pages 398-402. Institute of Electrical and Electronics Engineers, Inc., North Holland Publishing Company, Amsterdam, Holland, 1974.

[Allen 77] Allen, F. E. and Cocke, J. A Program Data Flow Analysis Procedure. *Communications of the ACM* 19(3):137-147, 1977.

[Allen 79] Allen, F. E. private communication.

[Babich 78a] Babich, W. A. and Jazayeri, M. The Method of Attributes for Data Flow Analysis, Part I: Exhaustive Analysis. *Acta Informatica* 10:245-264, 1978.

[Babich 78b] Babich, W. A. and Jazayeri, M. The Method of Attributes for Data Flow Analysis, Part II: Demand Analysis. *Acta Informatica* 10:265-272, 1978.

[Burstall 71] Burstall, R. and Darlington, J. A Transformation System for Developing Recursive Programs. *Journal of the ACM* 24(1):44-67, January, 1971.

[Elshoff 76] Elshoff, J. A Numerical Profile of Commercial PL/I Programs. *Software Practice and Experience* 6(4):505-525, 1976.

[Farrow 75] Farrow, R., Kennedy, K. and Zucconi, L. Graph Grammars and Global Program Data Flow Analysis. In *Proceedings of Seventeenth Annual IEEE Symposium on the Foundations of Computer Science*, pages 42-56. Institute of Electrical and Electronics Engineers, Inc., November, 1975.

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

[Kennedy 74] Kennedy, K. Schaeffer's Node Splitting Algorithm. SETL Newsletter #125, February 6, 1974, Courant Institute of Mathematical Sciences, New York University.

[Kennedy 77] Kennedy, K. and Zucconi, L. Application of a Graph Grammar for Program Control Flow Analysis. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 72-85. Association for Computing Machinery-SIGPLAN, January, 1977.

[Kibler 77] Kibler, D. F., Neighbors, J. M. and Standish, T. A. Program Manipulation Via Efficient Production Systems. *SIGPLAN Notices* 12(8):163-173, August, 1977.

[Knuth 71] Knuth, D. E. An Empirical Study of FORTRAN Programs. *Software Practice and Experience* 1:105-133, 1971.

[Loveman 77] Loveman, D. Program Improvement by Source-to-source Transformation. *Journal of the ACM* 24(1):121-145, January, 1977.

[Paige 77] Paige, R. and Schwartz, J. T. Expression Continuity and the Formal Differentiation of Algorithms. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 58-71. Association for Computing Machinery-SIGPLAN, January, 1977.

[Paull 82] Paull, M. C.
*Design of Algorithms.*
in preparation, 1982.

[Reiser 76] Reiser, J. (editor)
*SAIL.*
Stanford Artificial Intelligence Laboratory Memo AIM-289, Stanford University, August, 1976

[Reps 82] Reps, T.
Optimal-time Incremental Semantic Analysis for Syntax-directed Editors.
In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 169-176 Association for Computing Machinery-SIGPLAN, January, 1982.

[Robinson 76] Robinson, S. K. and Torsun, I. S.
An Empirical Analysis of FORTRAN Programs.
*Computer Journal* 19(1):56-62, 1976

[Rosen 77] Rosen, B. K.
High-level Data Flow Analysis.
*Communications of the ACM* 20(10):712-724, October, 1977.

[Ryder 74] Ryder, B. G.
The PFORT Verifier.
*Software Practice and Experience* 4:359-377, 1974

[Ryder 79] Ryder, B. G.
Constructing the Call Graph of a Program.
*IEEE Transactions on Software Engineering* SE-5(3):216-225, May, 1979.

[Ryder 82a] Ryder, B. G.
*Incremental Data Flow Analysis Based on a Unified Model of Elimination Algorithms.*
PhD thesis, Department of Computer Science, Rutgers University, 1982
also available as Department of Computer Science Technical Report #DCS-TR-117.

[Ryder 82b] Ryder, B. G. and Paull, M. C.
A Unified Model of Elimination Algorithms.
1982
in preparation.

[Ryder 82c] Ryder, B. G. and Paull, M. C.
A Comparison of Incremental Data Flow Analysis Algorithms
1982.
in preparation.

[Sharir 79] Sharir, M.
Interprocedural Analysis of Global Variable Usage.
1979.

[Tarjan 74] Tarjan, R. E.
Testing Flow Graph Reducibility.
*Journal of Computer and System Sciences* 9:355-365, 1974 .

[Tarjan 79] Tarjan, R. E.
*Fast Algorithms for Solving Path Problems*
Computer Science Department Technical Report STAN-CS-79-734, Stanford University, April, 1979

[Ullman 73] Ullman, J. D.
Fast Algorithms for the Elimination of Common Subexpressions
*Acta Informatica* 2(3):191-213, 1973.