Unilateral Graphical Approach for Data Flow Analysis in Programming Structure for Software Reliability

S.Chitra[1], K.Thiagarajan[2], M.Rajaram[3]

Abstract

The ways that the methods of data flow analysis can be applied to improve software reliability are described. There is also a review of the basic terminology from graph theory and from data flow analysis in global program optimization. The notation of regular expressions is used to describe actions on data for sets of paths. These expressions provide the basis of a classification scheme for data flow which represents patterns of data flow along paths within subprograms and along paths which cross subprogram boundaries. Fast algorithms, originally introduced for global optimization, are described and it is shown how they can be used to implement the classification scheme. It is then shown how these same algorithms can also be used to detect the presence of data flow anomalies, which are symptomatic of programming errors.

Keywords : Automatic Documentation, Automatic Error Detection, Data Flow Analysis, Software Reliability, Graph Path Cycles and Distance Graph.

[1] Head, Dept. of Computer Science & Engg., M.Kumarasamy College of Engg., Karur.

[2] Lecturer, Dept. of Mathematics, Rajalakshmi Engg College, Thandalam,chennai.

[3] Professor, Dept. of Electrical & Electronics Engg., Thandhai Perriyar Government Institute of Technology, Vellore.

Introduction

It is believed that a careful analysis of the use of data in a program, such as that done in global optimization, could be a powerful means for detecting errors in software and otherwise improving its quality. Our recent experience [27, 28] with a system constructed for this purpose confirms this belief. As so often happens on such projects, our knowledge and understanding of this approach were deepened considerably by the experience gained in constructing this system, although the pressures of meeting various deadlines made it impossible to incorporate all of our developing ideas into the system. Moreover, during its construction advances were made in global optimization algorithms that are useful to us, which for the same reasons could not be incorporated in the system. Our purpose in writing this paper is to draw these various ideas together and present them for the instruction and stimulation of others who are interested in the problem of software reliability.

The phrase "data flow analysis" became firmly established in the literature of global program optimization several years ago through the work of Cocke and Allen [2,3,4,5,6]. Considerable attention has also been given to data flow by Dennis and his co-workers [9, 29] in a different context, advanced computer architecture. Our own interpretation of data flow analysis is similar to that found in the literature of global program optimization, but our emphasis and objectives are different. Specifically, execution of a computer program normally implies input of data, operations on it, and output of the results of these operations in a sequence determined by the program and

the data. We view this sequence of events as a flow of data from input to output in which input values contribute to intermediate results, these in turn contribute to other intermediate results, and so forth until the final results, which presumably are output, are obtained. It is the ordered use of data implicit in this process that is the central object of study in data flow analysis.

Data flow analysis does not imply execution of the program being analyzed. Instead, the program is scanned in a systematic way and information about the use of variables is collected so that certain inferences can be made about the effect of these uses at other points of the program. An example from the context of global optimization will illustrate the point. This example is known as the live variable problem, which determines whether the value of some variable is to be used in a computation Reliability after some designated computation step.

If it is not to be used, space for that variable may be reallocated or an unnecessary assignment of a value can be deleted. To make this determination it is necessary to look in effect at all possible execution sequences starting at the designated execution step to see if the variable under consideration is ever used again in a computation. This is a difficult problem in any practical situation because of the complexity of execution sequences, the aliasing of variables, the use of external procedures, and other factors. Thus a brute force attack on this problem is doomed to failure. Clever algorithms have been developed for dealing with this and related problems. They do not require explicit consideration of all execution sequences in the program in order to draw correct conclusions about the use of variables. Indeed, the effort expended in scanning through the program to gather information is remarkably small. We discuss some of these algorithms in detail, because they can be adapted to deal with our own set of problems in software reliability, and turn to these problems now.

Data flow in a program is expected to be consistent in various ways. If the value of a variable is needed at some computation step, say the variable "a" in the step

$\gamma \leftarrow \alpha+1$,

Then it is normally assumed that at an earlier computation step a value was assigned to "a". If a value is assigned to a variable in a computation step, for example to ~, then it is normally assumed that that value will be used in a later computation step. When the pattern of use of variables is abnormal, so that our expectations of how variables are to be used in a computation are violated, we say there is an anomaly in the data flow. Examples of data flow anomalies are illustrated in the following programming constructions. The first is

X=A

X=B

The first assignment to X is useless. Why is the statement there at all? Perhaps the author of the program meant to Write

X=A

Y=B

Another data flow anomaly is represented by the other construction

SUBROUTINE SUB(X, Y, Z)

Z=Y+W

Here W is undefined at the point that a value for it is required in the computation. Did the author mean X instead of W, or W instead of X, or was W to be in COMMON? We do not know the answers to these questions, but we do know that there is an anomaly in the data flow.

As these examples suggest, common programming errors cause data flow anomalies. Such errors include misspelling, confusion of names, incorrect parameter

usage in external procedure invocations, omission of statements, and similar errors. The presence of a data flow anomaly does not imply that execution of the program will definitely produce incorrect results; it implies only that execution may produce incorrect results. It may produce incorrect results depending on the input data, the operating system, or other environmental factors. It may always produce incorrect results regardless of these factors, or it may never produce incorrect results. The point is that the presence of a data flow anomaly is at least a cause for concern because it often a symptom of an error. Certainly software containing data flow anomalies is less likely to be reliable than software, which does not contain them.

Our primary goal in using data flow analysis is the detection of data flow anomalies. The examples above hardly require very sophisticated techniques for their detection. However, it can easily be imagined how L. D. Fosdick and L. J. Osterweil[27,28] similar anomalies could be embedded in a large body of code in such a way as to be very obscure. The algorithms, which we describe, make it possible to expose the presence of data flow anomalies in large bodies of code where the patterns of data flow are almost arbitrarily complex. The analysis is not limited to individual procedures, as is often the case in global optimization, but it extends across procedure boundaries to include entire programs composed of many procedures.

The search for data flow anomalies can become expensive to the point of being totally impractical unless careful attention is given to the organization of the search. Our experience shows that a practical approach begins with an initial determination of Whether or not any data flow anomalies are present, leaving aside the question of their specific location. This determination of the presence of data flow anomalies is the main subject of our discussion. We will see that fast and effective algorithms can be constructed for making this determination and that these algorithms identify the variables involved in the data flow anomalies and provide rough information about location. Moreover, these algorithms use as their basic constituents the same algorithms that are employed in global optimization and require the same information, so they could be particularly efficient if included within an optimizing compiler.

Localizing an anomaly consists in finding a path in the program containing the anomaly; this raises the question of whether the path is executable. For example, consider Figure 1 and observe that although there is a path proceeding sequentially through the boxes 1, 2, 3, 4, 5, this path can never be followed in any execution of the program. An anomaly on such a nonexecutable path is of no interest. The determination of whether or not a path is executable is particularly difficult, but often can be made with a technique known as Symbolic Execution [8, 19, 22]. In symbolic execution the value of a variable is represented as a symbolic expression in terms of certain variables designated as inputs,

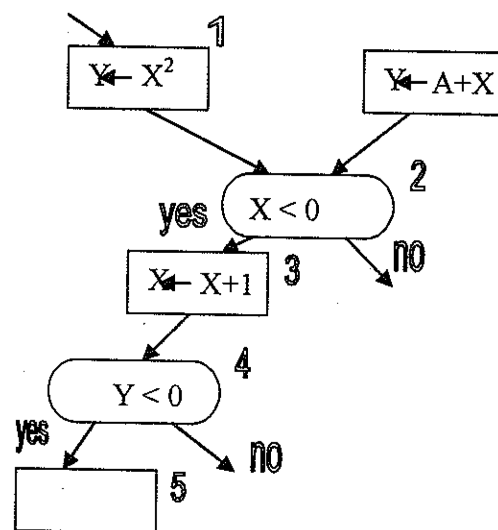


Fig.1

Figure 1: The Path In This Segment Of A Flow Diagram Represented By Visiting The Boxes In The Sequence 1, 2, 3, 4, 5 Is Not Executable. Note That Y= 0 Upon Leaving Box 1 And This Condition Is True Upon Entry To Box 4, Thus The Exit Labeled T Could Not Be Taken.

rather than as a number. The symbolic expression for a variable carries enough information that if numerical values were assigned to the inputs a numerical value could be obtained for the variable. Symbolic execution requires the systematic derivation of these expressions. Symbolic execution is very costly, and although we believe further study will lead to more efficient implementations, it seems certain that this will remain relatively expensive. Therefore a practical approach to anomaly detection should avoid symbolic execution until it is really necessary. In particular, with presently known algorithms the least expensive procedure appears to be: 1) determine whether an anomaly is present, 2) find a path containing this anomaly, and then 3) attempt to determine whether the path is executable.

We show that the algorithms presented here do provide information about the presence of anomalies on executable paths. While they do not identify the paths, the fact that they can report the presence of an anomaly on an executable path without resorting to symbolic execution is of considerable practical importance.

While an anomaly can be detected mechanically by the techniques we describe, the detection of an underlying error requires additional effort. The simple examples of data flow anomalies given earlier make it clear that a knowledge of the intent of the programmer is necessary to identify the error. It is unreasonable to assume that the programmer will provide in advance enough additional information about intent that the errors too can be mechanically detected. We visualize the actual error detection as being done manually by the programmer, provided with information about the anomalies present in his program. Obviously, many tools could be provided to make the task easier, but in the end it must be a human who determines the meaning of an anomaly. We like to think of a system which detects data flow anomalies as a powerful, thorough, tireless critic, which can inspect a program and say to the programmer: "There is something unusual about the way you used the variable a in this statement. Perhaps you should check it." The critic might be even more specific and say, "Surely there is something wrong here. You are trying to use ~ in the evaluation of this expression, but you have not given a value to a".

The data flow analysis required for detection of anomalies also provides routine but valuable information for the documentation of programs. For example, it provides information about which variables receive values as a result of a procedure invocation and which variables must supply values to a procedure. It identifies the aliasing that results from the multiple definitions of COMMON blocks in programs. It identifies regions of the program where some variables are not used at all. It recognizes the order in which procedures may be invoked. This partial list illustrates that the documentation information provided by this mechanism can be useful, not only to the person responsible for its construction, but also to users and maintainers.

We are ready now to enter into the details of this discussion. We begin with a presentation of certain definitions from graph theory. Graphs are an essential tool in data flow analysis, used to represent the execution sequences in a program. We follow this with a discussion of the expressions we use to represent the actions performed on data in a program. The notation introduced here greatly simplifies the later discussion of data flow analysis. Next, we discuss the basic algorithmic tools

required for data flow analysis. Then we describe both a technique for segmenting the data flow analysis and the systematic application of this technique to detect data flow anomalies in a program. We conclude with a discussion of the experience we have had with a prototype system based on these ideas.

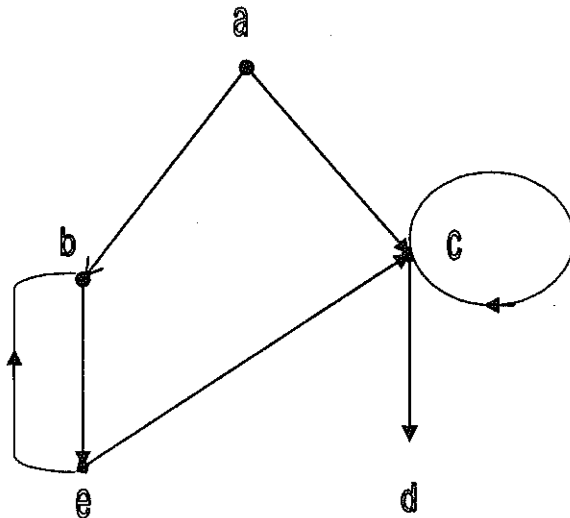
Basic Definitions - Graphs : (35)

Formally a graph is represented by G(N, E) where N is a set of nodes {nl, n2 , . . . , nk}and E is a set of ordered pairs of nodes called the edges, {(n~ , n~), (nj3 , n:4), "'" , (n~_~ , n~m)}, where the n~,s are not necessarily distinct. For example, for the graph in Figure 2,

N -- {a,b,c,d,e},

E = {(a,b), (a,c), (b,d), (d,b), (c,c),(d,c), (c,e)}.

The number of nodes in the graph is represented by]N[and the number of edges by]E[. For the graph in Figure 2, i N_ i1~ ,5 and]E[= 7. For any graph]E[I _ ,, I since



a particular ordered pair of nodes may appear at most once in the set E.

Figure : 2 Pictorial Of A Directed Graph. The Points, Labeled Here As A, B, C, D, E Are Called Nodes, And The Lines Joining Them Are Called Edges.

For the graphs that will be of interest to us it is usually true that |E| is substantially less than $|N|^2$ in fact it is customary to assume that $|E| \leq k|N|$ where k is a small integer constant.

For an edge, say (n, n_j), we say that the edge goes from n, to n~ ; n, is called a Predecessor of n_j, and n_j is called a Successor of n,. The number of Predecessors of a node is called the in-degree of the node, and the number of successors of a node is called the out-degree of the node. For the graph shown in Figure 2, a is the predecessor of b and c, the out-degree of a is two; a is not a successor of any node, it has the indegree zero. In this figure we also see that e is both a successor and a predecessor of b, and c is a successor and predecessor of itself. A node with no predecessors (i.e., in-degree = 0) is called an entry node, and a node with no successors (i.e., out-degree = 0) is called an Exit node; in Figure 2, a is the only entry node and d is the only exit node.

A path in G is a sequence of nodes n_{j1} , n_{j2} ,..., n_{jk} such that every adjacent pair (n_{ji}, n_{ji}+1) is in E. We say that this path goes from n_{j1} to n_{jk}. In Figure 2, a, c, d is a path from a to d; b , e , b, is a path from b to b. There is infinity of paths from b to b: b, e, b; b, e, b, e, b; etc. The length of a path is the number of nodes in the path, less one (equivalently, the number of edges) ; thus the length of the path a, b , e , b, c, d in Figure 2 is six. If n_{j1}, n_{j2}, ... , n_{jk} is a path p, then any subsequence of the form n_{ji}, n_{ji}+1, ... , n_{ji+m} for $1 \leq i < k$ and $1 \leq m \leq k - i$ is also a path, p'; we say that p contains the path p'.

If p is a path from n_i to n_j and i=j, then p is a cycle. In Figure 2 the paths b, e, b;b ,e ,b, e, b and c, c are cycles. The path a, b, e, b, e , c , d contains a cycle. A path, which contains no cycles, is "acyclic", and a graph in which all paths are acyclic is an "acyclic graph".

If every node of a connected graph has indegree one and thus has a unique predecessor, except for one node

which has indegree zero, the graph is a tree T(N, E). The graph in Figure 3 is a tree, and if the

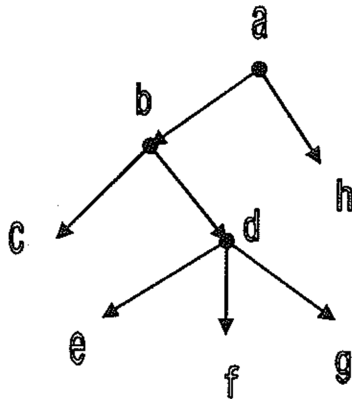


Fig 3

Figure : 3 Pictorial representation of a tree rooted at 0. Each node has a unique predecessor except the root which has no predecessor.

edges (e, b), (e, c), and (c, c) in Figure 2 are deleted, then the resulting graph is also a tree. The unique entry node is called the root of the tree and the exit nodes are called the leafs. It will be recognized that there is exactly one path from the root to each node in a tree; thus we can speak of a partial ordering of the nodes in a tree. In particular, if there is a path from ni to nj in a tree, then ni, comes before nj in the tree; we say that ni , is an ancestor of nj and nj is a descendent of ni . In Figure 3 every node except a is a descendent of a, and a is the ancestor of all of these nodes. Similarly b is an ancestor of the nodes c, d , e, f , g; on the other hand, h is not an ancestor of these nodes. A tree, which has been derived from a directed graph by the deletion of certain edges, but of no nodes, is called a spanning tree of the graph.

These elementary definitions are commonly accepted, but they are not universal. Graph theory seems to be notorious for its nonstandard terminology. Additional information on this subject can be found in various texts such as Knuth [24], and Harary [13].

The following figure 4 explains some assignment of program segment node p represents the statement x = x+1, node p+1 represents the first part of IF statement i.e., IF (X < Y), node p+2 represents the second part of IF statement Big = Y and node p+3 represents the statement of the next line of the program.

Pseudo code :

```
x = x + 1.O
IF(X LT Y) J=j+1
A = X*X
```

Figure 5 explains the programming statement of IF – THEN

```
IF(A .LE. 1.0)J = J + 1,
```

Figure 6 shows the working condition of IF – THEN – ELSE

```
IF(A .LE. 1.0)J = J + 1
    ELSE
      I = I+1
A=X*X
```

Figure 7 depicts the looping statement FOR – TO – NEXT

```
FOR I = 1 TO 10
I = I + 1
NEXT I
```

Figure 8 exhibits pictorial representation of DO – WHILE and UNTIL – DO.

```
DO
J = J * 5
WHILE(J.LT.100)
```

ALGORITHMS TO SOLVE THE LIVE VARIABLE PROBLEM AND THE AVAILABILITY PROBLEM THROUGH DISTANCE GRAPH (NAMELY 2-DISTANCE GRAPH)[13]

In the last section the live variable problem and the availability problem were defined and a simple example was given to show how a solution to the live variable

problem can be used to determine the presence of absence of data flow anomalies. In this section we describe particular algorithms for solving the live variable problem and the availability problem. Several such algorithms have appeared in the literature [6,16, 23, 31, 35]. The pair of algorithms we have chosen for discussion do not have the lowest asymptotic bound on execution time. However, they are simpler and more widely applicable than others and their speed is competitive.

Algorithm Depth First Search:

1. Push the entry node on a stack and mark it (this is the first node visited; nodes are marked to prevent visiting them more than once).

2. While the stack is not empty do the following:

2.1 While there is an unmarked edge from the node at the top of the stack, do the following:

2.1.1 Select an unmarked edge from the node at the top of the stack and mark it (edges are marked to prevent selecting them more than once) ;

2.1.2 If the node at the head of the selected edge is unmarked,then mark it and push it on the stack (this is the next node visited) ;

2.2 Pop the stack;

3. Stop.

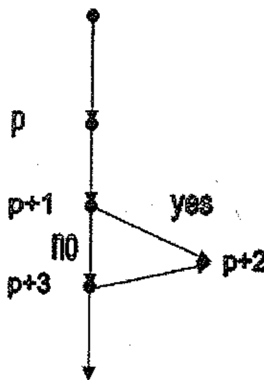


Fig 4

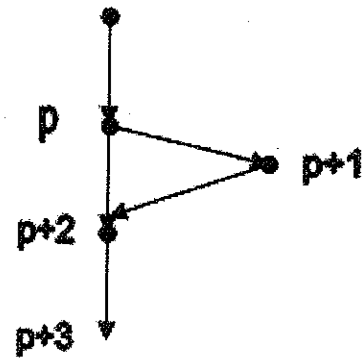


Fig 5

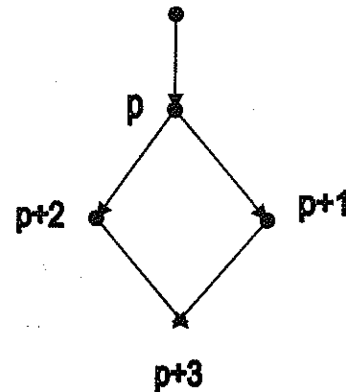


Fig 6

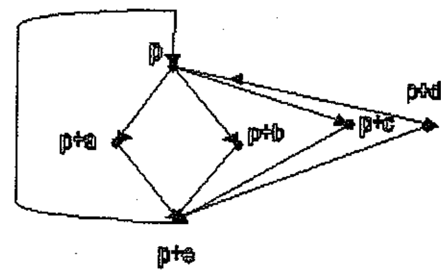


Fig.7

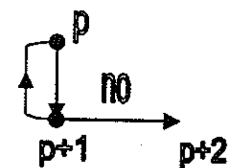


Fig.8

The above mentioned algorithms represented by the respective diagrams involve a search over a flow graph in which the nodes are visited in a specific order derived from a depth first search. This search procedure is defined by the following algorithm, where it is assumed that a flow graph G~,(N, E, no) is given, and a push down stack is available for storage.

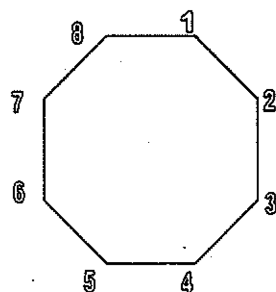


Figure : 9 Numbering of the nodes of a graph in the order in which they are first visited during a depth first search. This numbering is called preorder.

In Figure 9 the nodes of the flow graph are numbered in the order in which they are first visited during the depth first search. We follow the convention that the leftmost edge (as the graph is drawn) which is not yet marked is the next edge selected in step2.1.1; thus the numbering of the successor nodes of a node increases from left to right in the figure. The ordering of the nodes implied by this numbering is called preorder [24]. The order in which the nodes are popped from the stack during the depth first search is called postorder [16, 24].

Definition: Let G be a connected graph with vertex set V and edge set E ,H is the subgraph of G is defined as r-distance graph of G is defined d(u,v) = r in G, where d(u,v) is nothing but the distance between the two vertices u and v in V.

Note: Here 0 is the convenient vertex to make the required distance from the given situation. If necessary introduce else.

2-Distance Graph of the above figure 9:

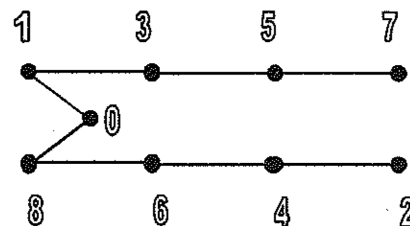


Figure : 10 Illustration of Postorder and R-distance Numbering of the Nodes of a Graph.

In Figure 10 the nodes are numbered in postorder. This numbering could be generated in the following way. Introduce a counter in the depth first search algorithm and initialize it to 0 in step 1. In step 2.2, before popping the stack, number the node at the top of the stack with the counter value and then increment the counter. If each postorder node number, say k, is complemented with respect to |N| , i.e., k' ¢- IN {- k, then the new numbering represents an ordering known as r-postorder |16|. This numbering is shown in parentheses in Figure 10.

The depth first spanning tree |33] of a flow graph is an important construction for the analysis of data flow. This construction can be obtained from the depth first search algorithm in the following way. Add a set E-which is initialized to empty in step 1. In step 2.1.2 put the selected edge in E' if the head of the selected edge is unmarked. After execution of this modified de~.th first search algorithm, the tree T(N, E) is the depth first spanning tree of Gp(N, E, no),the flow graph on which the search was executed. The depth first spanning tree

of the flow graph in Figure 9 is explained as follows, These edges in the set E - E' fall into three distinct groups:

1) forward edges with respect to T: e E E - E', is in this group if this edge goes from an ancestor to a descendant of T;

2) back edges with respect to T: e E E -- E', is in this group if this edge goes from a descendant to an ancestor of T, or if this edge goes from a node to itself;

3) cross edges with respect to T: e E

Conclusion

In this paper, We observed the following

1) They require no human intervention or guidance and

2) They are capable of scanning all paths for possible data flow anomalies.

3) A human tester need not be concerned with designing test cases for this system. This can be assured by the system that no anomalies are present. In case an anomaly is present, the system will advise the tester and further testing or debugging would be necessary. It is advisable to adopt the above proposed graphical model in the early phases of a testing through directed graph. Further fixing hypothesis and testing efforts involves more powerful systems that employ techniques such as symbolic execution. In future, this work can be extended to widen the class of errors detectable by means such as Artificial Immune Recognition System (AIRS).

Acknowledgement

We want to be thankful to Dr.Ponnammal Natarajan, Advisor (R&D), Rajalakshmi Engineering College formally worked as the Director for Research, Anna University, Chennai, India, for her valuable guidance throughout this work. In this moment we also want to express our gratitude to Dr.A.R.Meenakshi, Dean, Department of Computer Applications and Mathematics, Karpagam Arts & Science College, Coimbatore, India for her worthful encouragement.

References

[1] A ullman. J. D, *"Node listings for reducible flow graphs"*, in Proc. Of the 7th Annual ACM Symposium on Theory of Computing, 1975, ACM, New York, PP. 177-185, 1975.

[2] ALLEN. F.E., *"Program optimization"* in Annual Review in Automatic Programming, Pergamon Press, New York, PP. 239-307, 1969.

[3] Allen, F.E, *"A basis for program optimization"*, in Proc. IFIP Congress 1971, North-Holland Publ. Co., Amsterdam, The Netherlands, PP. 385-390, 1972.

[4] Allen. F. E and Cocke. J, *"Graph theoretic constructs for program control flow analyszs"*, IBM Research Report RC3923, T. J.Watson Research Center, Yorktown Heights,New York, 1972.

[5] Allen. F. E, *"Interprocedural data flow analysis"*, in Proc. IFIP Congress 1974, North Holland Publ. Co., Amsterdam, The Netherlands, PP. 398-402, 1974.

[6] Allen. F. E and Cocke J, *"A program Comm"*.

[7] Balzea. R. M, *"EXDAMS: Extendable debugging and monitoring system"*, in Proc. AFIPS 1969 Spring Jr. Computer Conf., Vol. 34, AFIPS Press, Montvale. N.J, PP. 567-580, 1969.

[8] Clarke. L. A, *"System to generate test data and symbolically execute programs"*, Dept. Of Computer Science Technical Report $Cu- CS-060-75, Univ. Of Colorado, Boulder, 1975.

[9] Dennis. J.B, *"First version of a data flow procedure language"*, in Lecture notes ~n computer science 19, G. Goos and J. Hartmanis (Eds.), Springer-Verlag, New York, PP. 241-271, 1974.

[10] famle~C. R.E, *"An experimental program testing facility"*, in Proc. First National Conf. On Software Engineering, 1975, IEEE $75CH0992-8C, IEEE, New York, PP. 47-55, 1975.

[11] Goldstine. H. H and on Neumann. J, *"Planning and coding problems for an electronic computing instrument"*, in John yon Neumann, collected works, A. H. Taub (Ed.), Computing Surveys, V~I. 8, No. 3, September 1976 330 • Data Flow Analysis In Software Reliability Pergamon Press, London, England, PP. 80-235, 1963.

[12] Habermann. A.N. Path expressions, Dept. Of

Computer Science Technical Report, Carnegie-Mellon Univ., Pittsburgh, Pa., 1975.

[13] Harary. F, *"Graph theory"*, Addison-Wesley Publ. Co., Reading, Mass., 1969.

[14] Hecht.M.S and Ullman. J. D, *"Flow graph reducibility"*, SIAM J. Computing 1, 188-202, 1972.

[15] Hecht. M. S, ann Ullman. J.D, *"Characterizations of reducible flow graphs"*, J. ACM 21, 3 367-375, July 1974.

[16] Hecht. M. S. and Ullman, J. D. *"A simple algorithm for global data flow analysis problems"*, SIAM J. Computing 4 PP. 519-532, Dec. 1975.

[17] Hopcroft.J, and Tarjan N. R. E, *"Efficient algorithms for graph manipulation"*, Comm. ACM 16 372-378, June 1973.

[18] Hopcroft. J. E, and Ullman. J.D, *"Formal languages and their relation to automata"*, Addison Wesley Publ. Co., Reading, Mass., 1969.

[19] Howden W.E, "Automatic case analysis of programs", in Proc. Computer Science and Statistics: 8th Annual Symposium on the Interface, PP. 347-352, 1975.

[20] Kallal. V and Osterweil. L. J, *"Constructing flowgraphs for assembly language programs"*, Dept. Of Computer Science Technical Report Univ. Of Colorado, Boulder,(to appear 1976).

[21] Karp. R.M, *"A note on the application of graph theory to digital computer programming"*, Information and Control 3 179-190, 1960.

[22] Kino. J. C, *"A new approach to program testing"*, in Proc. Internatl. Conf. On Reliable Software, 1975, IEEE ~75CH0940-7CSR, IEEE, New York, PP. 228-233, 1975.

[23] Kennedy.K.W, *"Node listings applied to data flow analysis"*, in Proc. Of 2nd ACM Symposium on Pmncipals of Programming Languages, 1975, ACM, New York, PP. 10-21, 1975.

[24] Knuwh. D. E, *"The art of computer programming, Vol. I fundamental algorithms"*, (2d Ed.), Addison Wesley Publ. Co., Reading, Mass., 1973.

[25] Knuth. D. E, *"An empirical study of FORTRAN programs"*, Software—Practice and Experience 1, 2 105-134, 1971.

[26] Miller. E. F. JR. Rxvp, *" FORTRAN automated verification system"*, Program Validation Project, General Research Corp., Santa Barbara, Calif., PP. 4, 1974.

[27] Osterweil. L. J, and Fosdick. L D, *"DAVE--a FORTRAN program analysis system"*, in Proc. Computer Science and Statistics: 8th Annual Symposium on the Interface, PP. 329-335, 1975.

[28] Osterweil, L. J and Fosdick, L. D, *"DAVE--a validation, error detection and documentation system for FORTRAN programs"*, Software--Practice and Experience (to appear 1976).

[29] Rodriguez, J. D, *"A graph model for parallel computation"*, Report MAC-TR-64, Project MAC, MIT, Cambridge, Mass., 1969.

[30] Rosen. B, *"Data flow analysis for recursive PL/I programs"*, IBM Research Report RC5211, T. J. Watson Research Center, Yorktown Heights, New York, 1975.

[31] Schaeffer. M, *"A mathematical theory of global program optimization"*, Prentice-Hall Inc., Englewood Cliffs, N. J., 1973.

[32] Stuckl. L. G, *"Automatic generation of self-metric software,"* in Proc. IEEE Symposium on Computer Software Reliability, 1973, IEEE $73CH0741-9CSR, IEEE, New York, PP. 94-100, 1973.

[33] Tarjan. R. E, *"Depth-first search and linear graph algorithms"*, SIAM J. Computing 146-160, (Sept. 1972,

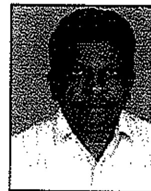
[34] Tarjan.R.E, *"Testing flow graph reducibility,"* J. Computer and System Sciences 9, 3 355-365 Dec. 1974,

[35] Ullman. J. D, *"Fast algorithms for the elimination of common subexpressions"*, Acta Informatiea 2 191-213, 1973.

Author's Biography

S.Chitra, is the Head of the Department of Computer Science and Engineering in M. Kumarasamy College of Engineering, Karur, India. She has 15 years of experience in active teaching. She completed her BE and ME in Computer Science and Engineering and undergoing her research in the area of Software Reliability. She presented more than 15 papers in national, international conferences and journals. Mail id: schitra3@gmail.com

K. Thiagarajan, Lecturer, Department of Mathematics, Rajalakshmi Engineering College, Thandalam, Chennai, India. He has attended and presented 23 papers in national and international conferences. He has published 13 international journals, and one National journal. He has 13 years of teaching experience.



Dr. M. Rajaram, is working as a Professor in Electrical And Electronics Engineering department in Thandhai Perriyar Government Institute of Technology, Vellore, India. He is guiding 12 research scholars and 4 have been awarded doctorate. He presented more than 100 papers including national, international journals and has 20 years of teaching experience.