THE DETECTION OF ANOMALOUS INTERPROCEDURAL DATA FLOW*

by

Lloyd D. Fosdick
Leon J. Osterweil

Department of Computer Science
University of Colorado
Boulder, Colorado 80309

Report #CU-CS-090-76 April 1976

Key Words: data flow analysis, software reliability, automatic error detection

CR Categories: 4.4, 5.24

* This work supported by NSF Grant DCR 75-09972.

ABSTRACT

In an earlier paper, the authors have defined type 1 and type 2 data flow anomalies to be, respectively, the reference to an undefined variable and the definition of a variable without subsequent reference. It is not difficult to devise search techniques to detect such anomalies when the anomalous data flow is contained in a single procedure. When the data flow crosses procedure boundaries, however, many difficulties may arise. In this paper, we carefully define the conditions under which interprocedural anomalies occur. We also show how algorithms currently used in global program optimization can easily be adapted to yield highly efficient algorithms for the detection of such interprocedural anomalies.

I. Introduction

Common programming errors often cause anomalies in the data flow of a program. For example, a confusion of variables or a similar blunder might result in the sequence of FORTRAN statements

```
X = X + Y
X = Z
```

Perhaps the programmer meant to write Z = X on the second line. We are interested in locating these and other data flow anomalies in programs and using the presence of such anomalies as a signal that errors may be present, or the absence of such anomalies to guarantee the absence of certain types of errors.

We focus our attention on three actions which may be performed on data: reference (r), definition (d), undefinition (u). For example, in the statement

```
X = X + Y
```

the action performed on Y is d, the actions performed on X are rd where the left to right order of this pair of actions denotes the order in which the actions are performed. Similarly, for the sequence of three statements

```
X = X + Y
Y = Y + 1
X = Z
```

the actions performed on X are rdd. In FORTRAN the undefinition action happens to a DO index when the DO is satisfied, in block structured languages it happens to local variables of a block when control exits the block.

If we consider a sequence of statements which might be executed in a program, then for each variable we have a corresponding sequence

of actions. Examples of such sequences are:

d r r d r

d r u d r

d u d r

We call these sequences path expressions*. A path expression is associated with a particular variable and a particular sequence of statements in the program. As explained later, it may also be associated with a set of sequences of statements. A path expression is anomalous if it has one of the following three forms:

p d d p'

p d u p'

p u r p'

where p and p' stand for arbitrary path expressions.

There are three major difficulties encountered in any attempt to detect these anomalous path expressions in a large program. First, there is the danger of a combinatorial explosion. The large number of paths in a typical program makes it immediately apparent that a straightforward search of all of the paths will be hopelessly expensive. Second, the large number of variables in a program and the fact that all need to be considered further complicates the detection problem. Finally, there is the problem of finding anomalies on executable paths as distinguished from non-executable paths. The point here is that if one looks at the program as a flow diagram, ignoring the contents of the boxes, then many of the paths that are represented by this structure might never be taken during execution (cf. Figure 1). Only anomalous expressions on executable paths are of interest, but it is

---

* Habermann [1] uses this terminology in a slightly different context.

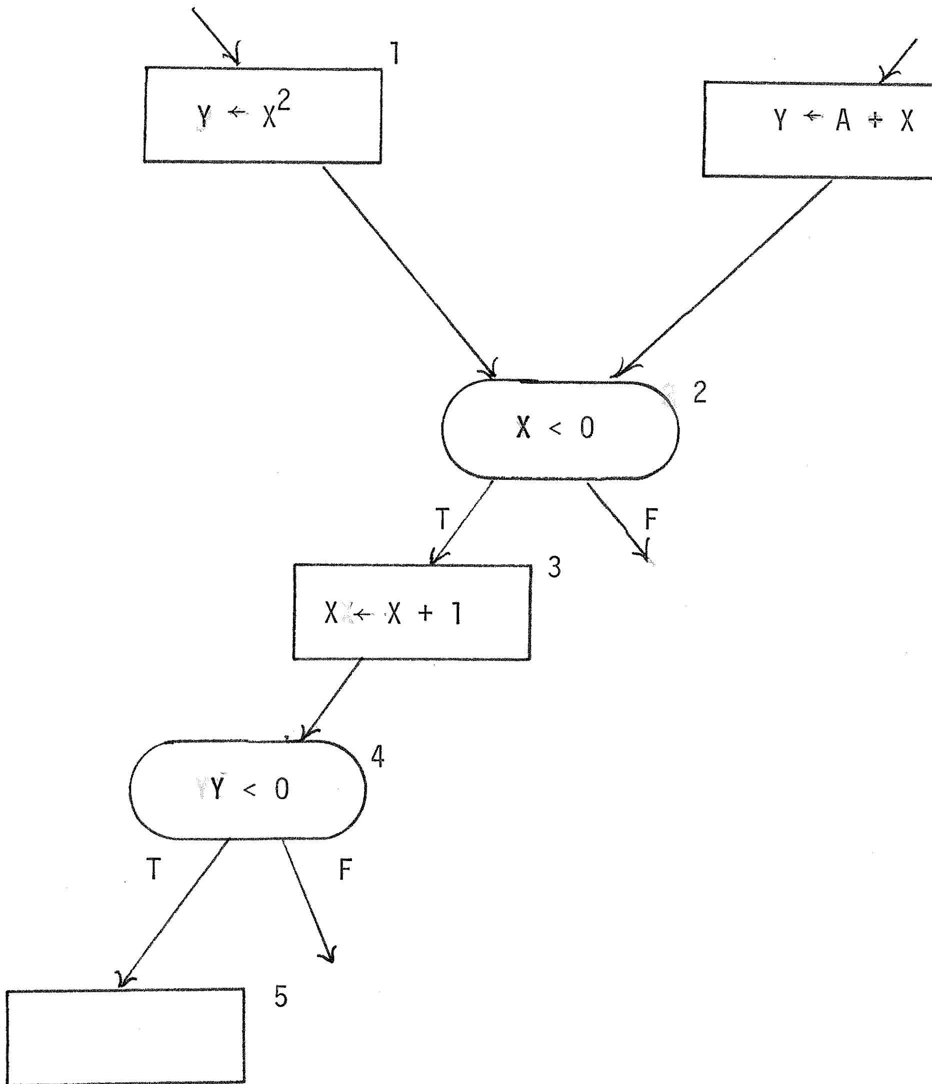


Figure 1: The path in this segment of a flow diagram represented by visiting the boxes in the sequence 1,2,3,4,5 is not executable. Note that $Y \geq 0$ upon leaving box 1 and this condition is true upon entry to box 4, thus the exit labelled T could not be taken.

known that in general one cannot always recognize executable paths*.

Here we describe techniques which permit detection of the presence of anomalous path expressions in large programs. These techniques are very efficient in many practical situations although they do not entirely overcome the difficulties mentioned above. Our principal focus here is on how the problem can be partitioned so that anomalous expressions on paths that cross procedure, or sub-program, boundaries can be detected efficiently. Such partitioning is absolutely essential if a combinatorial explosion in examining all such paths is to be avoided in large programs.

II. Notions from global optimization.

The problem of searching for certain patterns of data actions is common in the field of global program optimization. Recently, good algorithms have been developed to deal with two forms of this problem: the live variable problem and the availability problem**. We will show that these algorithms, given the generic names LIVE and AVAIL, can be used to detect anomalous path expressions, and in particular that they can be used to implement an effective partitioning of path expressions on subprograms.

The live variable problem arises in global optimization when one wishes to determine if a variable will ever be referenced, before being

---

* Such recognition is equivalent to solution of the halting problem which is known to be unsolvable. (cf.pages 108, 109 of Hopcroft and Ullman [2].)

** The subject of global program optimization receives extensive treatment in a recent book by Schaeffer [3]. An extensive discussion of graph theoretic approaches to this problem will be found in papers by Allen and Cocke [4,10]. Simple and efficient algorithms for the live variable problem and the availability problem are discussed by Hecht and Ullman [5].

defined or undefined, after a selected statement is executed. The availability problem arises in global optimization when one seeks to determine if the value of an expression, say $\alpha + \beta$, which may be needed for the execution of a selected statement actually needs to be computed, or may be obtained instead by fetching a previously generated and stored value for it. In dealing with these problems it is customary to represent the program as a directed graph, as illustrated in Figure 2, where each node represents a statement or part of a statement and each edge joins pairs of nodes which can be executed in succession. Directed graphs used for this purpose are generally required to have a unique entry node, and are called flow graphs [3]. A sequence of nodes joined by edges is a path in the flow graph. A generalization and formal specification of these notions is presented in the next section.

III. Definitions.

We use G(N,E) to denote a directed graph consisting of the node set N and edge set E*. A flow graph is a directed graph which has a single node with in-degree zero (i.e. no edges enter it) and it is connected such that there is a path from the entry node to every other node in N. A flow graph may have one or more nodes with out-degree zero (i.e. no edges leaving the node) called exit nodes but in this presentation we will always assume that there is exactly one exit node. We use $G(N,E,n_0,n_x)$ to denote a flow graph with node set N, edge set E, entry node n_0, and exit node n_x.

---

* We assume a familiarity with elementary concepts of graph theory. A brief discussion of basic ideas and concepts will be found on pages 362-375 of Knuth [6], or Chapter 6 of Liu [7].

```
read (n,k)
if (k ≠ o)
   then
      while (n > 0) do  S1
   else
      S2 ;
stop
```

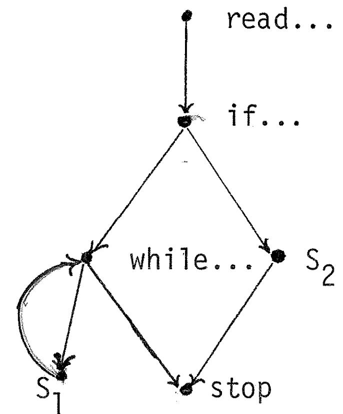


Figure 2: Representation of a program as a directed graph. Nodes are labelled to indicated the part of the program represented.

We associate a token set, _tok_, with a flow graph. The elements of _tok_ are denoted by $\alpha, \beta, \ldots$. With every node $n \in N$ we associate a set _gen_(n), a set _kill_(n), and a set _null_(n) which are subsets of _tok_. This association is illustrated in Figure 3. Informally we think of the tokens as representing variables in a program. The membership of tokens in these sets is determined by actions performed on the tokens according to rules which depend on the problem to be solved. Typically these rules take the following form: if the first action performed on α at node n is d then $\alpha \in$ _gen_(n), if no action is performed on α at node n then $\alpha \in$ _null_(n), etc. These rules for several problems of interest will be explained later, but for the time being let us simply take the sets _gen_(n), _kill_(n), and _null_(n) as given.

Let p be a path in a flow graph and α an element of the associated token set. Traverse the path and as each node n is visited write down a g if $\alpha \in$ _gen_(n), a k if $\alpha \in$ _kill_(n), and 1 if $\alpha \in$ _null_(n). The resulting sequence of g's, k's, and 1's is called a path expression for α on p and it is denoted by $P(p;\alpha)$. Referring to Figure 3, the path expression for α on p = 0,1,2,4,2,5 is

$$P(0,1,2,4,2,5;\alpha) = 1kgkgk;$$

similarly

$$P(0,1,2,5;\beta) = 1k1k \quad .$$

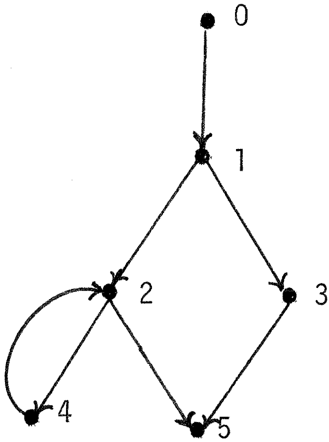
We use the notation of regular expressions* to represent sets of path expressions. For example, the set of path expressions for α on the set of all paths leaving node 1 in Figure 3 is

$$P(1\rightarrow,\alpha) = g(kg)^*k + 1k.$$

It is to be noted that the k associated with node 1 is not included.

---

* See, for example page 39 of Hopcroft and Ullman [2].



(a)

n	gen	kill
0		
1		α,β
2	α	
3	β	
4		α,β
5		α,β

(b)

live	avail
α,β	
α	α

(c)

Figure 3: (a) flow graph with nodes numbered for identification;
(b) the tok set is $\{\alpha,\beta\}$, the gen and kill subsets assigned to the nodes are shown, null = tok - (gen $\cup$ kill);
(c) the derived live and avail sets are shown.

We call such an expression a path expression too, and when a distinction is important we say a path expression is simple if it corresponds to a single path. It should be evident that a simple path expression will not contain the symbols * or +. Path expressions can be concatenated. Thus, referring again to Figure 3,

$$P(1;\alpha)P(1\to;\alpha) = k(g(kg)^*k + 1k) \ .$$

Likewise the path expression for α on the set of all paths entering node 5 is

$$P(\to 5;\alpha) = 1kg(kg)^* + 1k1 \ .$$

Here note that we do not include the k associated with the distinguished node 5. Concatentation with the path expression for α on node 5 yields

$$P(\to 5;\alpha)P(5;\alpha) = (1kg(kg)^* + 1k1)k \ .$$

Two path expressions representing identical sets of simple path expressions are equivalent. Thus, using the last path expression above, it is easily seen that

$$(1kg(kg)^* + 1k1)k \equiv 1kg(kg)^*k + 1k1k \ .$$

Furthermore, two path expressions differing only by transformations of the form

$$1g\to g,\ g1\to g,\ 1k\to k,\ k1\to k,\ 11\to 1$$

are also equivalent. For example,

$$1^*gk + kk1 + 11 \equiv gk + kk + 1.$$

Thus, the form of a path expression which represents a particular set of simple path expressions is not, in general, unique.

We now define the sets live(n) and avail(n), subsets of the token set, in terms of path expressions. In particular, for each $\alpha \in$ tok and

each $n \in N$ of $G(N,E,n_o,n_x)$

$\alpha \in$ live(n) if and only if $P(n\rightarrow,\alpha) \equiv gp + p'$,

and

$\alpha \in$ avail(n) if and only if $P(\rightarrow n,\alpha) \equiv pg$,

where p and p' stand for arbitrary path expressions. In words, $\alpha \in$ live(n) if and only if on some path from n the first action on α, other than null, is g; and $\alpha \in$ avail(n) if and only if the last action on α, other than null, on all paths entering n is g. These definitions are illustrated in Figure 3c where the live and avail sets are shown.

Algorithms, LIVE and AVAIL, have been developed by Hecht and Ullman [5] which permit determination of the live and avail sets, respectively, for all of the nodes of a flow graph in time propotional to $d|N|$ where $|N|$ is the number of nodes in the flow graph and d is a number which depends on the structure of the graph. They note that there is empirical evidence with FORTRAN programs showing that $d \leq 6$ and on the average $d \leq 2.75$. These results rest on the following reasonable assumptions:

1. Bit vector operations requiring unit time are permitted to perform set unions, intersections, and complements.
2. The number of edges in any flow graph is bounded by $k \cdot |N|$ where k is an arbitrary, but fixed, number.

The first assumption permits execution of algorithms LIVE and AVAIL in a time which is independent of the number of tokens. It is evident that one cannot strictly adhere to this but it is a reasonable first approximation. The second assumption is equivalent to putting an upper bound on the number of edges that can be directed away from a node which is perfectly reasonable for flow graphs of programs in most contemporary languages.

It will be noted that we have formulated path expressions in terms of the alphabet {r,d,u} and also in terms of the alphabet {g,k,l}. In applications we define a mapping of the r's, d's, and u's onto the g's, k's, and l's. A corresponding mapping of path expressions is then defined. The mapping used depends on the application as we illustrate below.

IV. An application.

In order to illustrate the application of these ideas let us consider the problem of detecting the presence of path expressions for references, definitions, and undefinitions of the form pddp' or pdup'. We begin by determining membership in the gen and kill sets according to the following rules:

1. $\alpha \in$ kill(n) if α is referenced at n.
2. $\alpha \in$ gen(n) if α is defined or undefined at n and it is not referenced at n.
3. $\alpha \in$ null(n) otherwise.

These rules define a many-to-one mapping of r's, d's, and u's onto g's, k's, and l's. In the application of these rules tokens represent variables of the program. Points of undefinition which usually occur on the transition from one statement to another take place at nodes especially introduced to the flow graph to represent such an event. After the gen, kill, and null sets have been assigned according to these rules the algorithm LIVE is executed. Suppose α is defined at node n and suppose further that execution of LIVE shows that

$\alpha \in$ live(n),

then it follows that there is a path expression of the form pddp' or pdup' in the flow graph. The truth of this conclusion is apparent,

since $\alpha \in$ live(n) implies $P(n\rightarrow,\alpha) \equiv gp + p'$ by definition. According to rule 2 above g stands for a definition or an undefinition; therefore, in terms of r's, d's, and u's one of the following (or both) is true:

$$P(n;\alpha)P(n\rightarrow;\alpha) = ddp + p';$$

$$P(n;\alpha)P(n\rightarrow;\alpha) = dup + p'.$$

Consider the converse, suppose that at every node n at which α is defined we determine that

$$\alpha \notin \text{live}(n).$$

Then we may conclude that no path expression of the form pddp' or pdup' is present; that is, there are no data flow anomalies of this type.

V. External procedures and segmentation.

Here we focus attention on the problem of identifying the critical features of a path expression for a flow graph which represents an external procedure invoked by some other flow graph, or a flow graph which is a single-entry, single-exit subgraph of a flow graph. The situation is depicted in Figure 4. The idea is that we want to determine enough about $P(n',\alpha)$ so that anomolous path expressions contained in $P(\rightarrow n';\alpha)P(n';\alpha)P(n'\rightarrow;\alpha)$ can be detected. Suppose we are interested in detecting the presence of path expressions of the form pddp'. We will assume that $P(n';\alpha)$ does not itself contain a path expression of either of these forms so that

$$P(\rightarrow n';\alpha)P(n';\alpha)P(n'\rightarrow;\alpha) \equiv pddp'$$

implies

$$P(\rightarrow n';\alpha) \equiv pd + p', \quad P(n';\alpha) \equiv dp + p'$$

or

$$P(n';\alpha) \equiv pd + p', \quad P(n'\rightarrow;\alpha) \equiv dp + p'$$

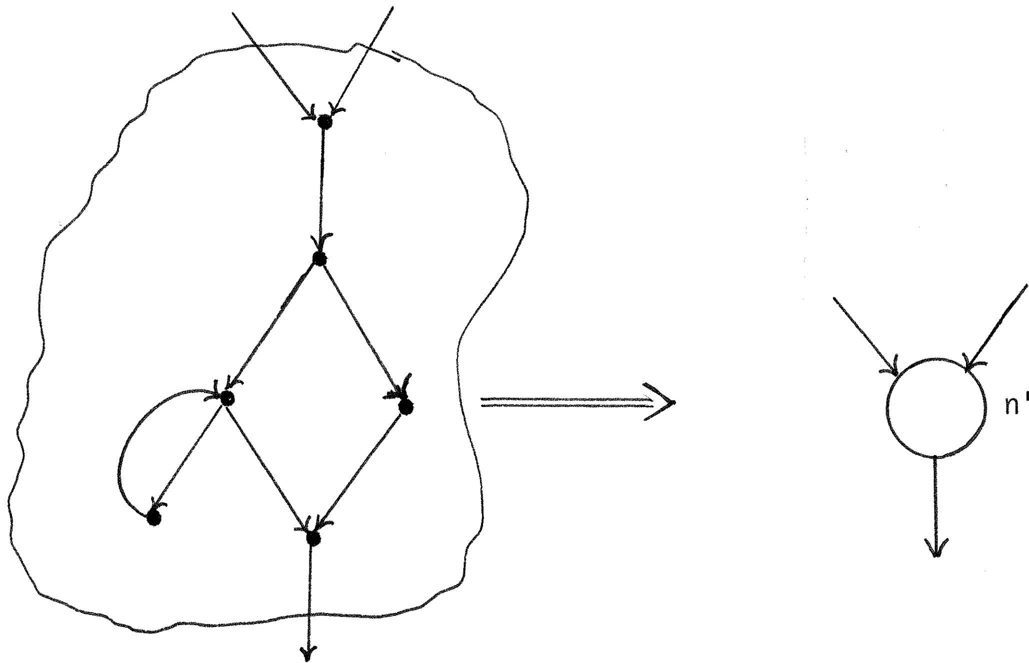


Figure 4: Segmentation of a flow graph. The derived flow graph consists of nodes such as n' which represent a sub-graph of the original graph. Alternatively we may think of n' as representing a node in which an external procedure is referenced.

or

$P(\rightarrow n';\alpha) \equiv pd + p'$, $P(n';\alpha) \equiv 1 + p$, $P(n'\rightarrow;\alpha) \equiv dp + p'$

In this last case we have had to add the element 1 to the alphabet for path expressions previously given in terms of r's, d's, and u's only so that we may explicitly recognize the possibility of paths along which no action takes place. Consistency with the earlier discussion is maintained by adopting the following obvious reduction rules:

$1d \rightarrow d$, $d1 \rightarrow d$, $11 \rightarrow 1$

and similarly for d replaced by r or u.

It is evident from this that we must recognize three types of path expressions: dp+p'; pd+p'; 1+p'. We label these three types X, Y, Ƶ, respectively. If the path expression is not equivalent to one of these types we label it type W. Now consider a flow graph in which the nodes are either simple (i.e. represent a statement or part of a statement), or like n' above, representing a single-entry, single-exit subgraph and focus attention on the actions performed on a variable, α. For each simple node it is trivial to determine the type of the path expression. For example, if the node represents the statement

$\alpha \leftarrow \beta + \gamma$

then the path expression for α is of type X and of type Y for this node; if it represents

$\beta \leftarrow \beta + 1$

then the path expression for α is of type Ƶ. On the other hand, if the node is not simple then determination of the type cannot be done by inspection, as it were, but the algorithm LIVE can be used to determine if the path expression is of type X or type Ƶ, and AVAIL can be used to determine if the path expression is of type Y. The basic ideas of how

to use these algorithms for this purpose have been described earlier [8].

Since LIVE and AVAIL operate in parallel on a set of tokens we can take advantage of this by regarding the above classifications as representing sets. Thus $\underline{X}(n)$ is the set of tokens for which the path expression has the form dp+p'; in particular $\alpha \in \underline{X}(n)$ if and only if $P(n,\alpha) \equiv dp+p'$. Corresponding remarks apply to Y, Ƶ, and W.

Once the sets $\underline{X}(n)$, $\underline{Y}(n)$, $\underline{Ƶ}(n)$, and $\underline{W}(n)$ have been determined for every node of a flow graph, the presence of a path expression of the form pddp' can be detected with the help of LIVE. This requires a mapping onto the sets $\underline{gen}(n)$, $\underline{kill}(n)$, and $\underline{null}(n)$:

$\underline{X}(n) \rightarrow \underline{gen}(n)$;

$\underline{Ƶ}(n) \rightarrow \underline{null}(n)$;

$\underline{tok} - (\underline{X}(n) \cup \underline{Ƶ}(n)) \rightarrow \underline{kill}(n)$;

Suppose, after execution of LIVE, it is found that $\alpha \in \underline{live}(n)$ and suppose $\alpha \in \underline{Y}(n)$, then we may conclude that a path expression of the form pddp' exists for some path in the flow graph. Notice that $\alpha \in \underline{live}(n)$ implies $P(n\rightarrow;\alpha) \equiv gp+p'$ and from the mapping we have used this implies $P(n\rightarrow;\alpha) \equiv dp+p'$. Also $\alpha \in \underline{Y}(n)$ implies $P(n;\alpha) = pd+p'$. Hence $P(n;\alpha)P(n\rightarrow;\alpha) = pddp' + p''$ verifying the foregoing conclusion that a path expression of the form pddp' exists for some path in the flow graph.

Thus, by segmenting the flow graph and the corresponding path expressions it is possible to reduce the size of the combinatorial problem without losing the information we are seeking. It will be noted however that we do lose information about actions on specific paths by this

approach. Such detailed information can be obtained from slower search techniques once it is known there is something to be searched for; that is, once its known there exists a path expression of the form, say, pddp' on some path.

By extending the above classification scheme and applying these ideas one can detect the presence of paths of the form pdup' and purp'. Also, such an extension makes possible determination of the presence of a particular form, say purp' on all paths entering a selected node. With this information it is possible to deal, at least partially, with the problem of determining whether purp' occurs on an executable path: if it occurs on all paths to a node, then it is reasonable to conclude that it occurs on an executable path if we permit the reasonable condition that there is at least one executable path to every node*.

VI. Conclusion.

We have shown how it is possible to detect the presence of anomalous use of data in a program. The process we have described is efficient because it makes use of fast algorithms developed for global optimization problems, and because it employs a path expression classification technique which allows segmentation of a large flow graph in a way which preserves the most important facts about data flow needed for the detection of data flow anomalies. In other work [8], based on an early version of these ideas, we have described a working system which has been successfully used to detect errors in large FORTRAN programs.

---

* An early version of these ideas is described in [8]. A more complete discussion appears in [9].

References

[1] Habermann, A. N. Path expressions. Department of Computer Science Technical Report, Carnegie-Mellon University, June 1975.

[2] Hopcroft, J. E. and Ullman, J. D. Formal Languages and their Relation to Automata. Addison-Wesley (1969)

[3] Schaeffer, M. A Mathematical Theory of Global Program Optimization. Prentice-Hall (1973).

[4] Allen, F. E. and Cocke, J. Graph theoretic constructs for program control flow analysis. IBM Research Report RC 3923, T. J. Watson Research Center, Yorktown Heights, New York (July 1972) pp. 65.

[5] Hecht, M. J. and Ullman, J. D. A simple algorithm for global data flow analysis problems. SIAM J. COMPUTING 4 (December 1974) pp. 519-532.

[6] Knuth, D. E. The Art of Computer Programming; Volume 1, Fundamental Algorithms. Addison-Wesley (1968).

[7] Liu, C. L. Introduction to Combinatorial Mathematics. McGraw-Hill (1968).

[8] Fosdick, L. and Osterweil, L. Validation and global optimization of programs. Proceedings of the Fourth Texas Conference on Computing Systems, Austin, Texas (November 1975).

[9] Fosdick, L. and Osterweil, L. Data flow analysis in software reliability. (Submitted for publication).

[10] Allen, F. E. and Cocke, J. A program data flow analysis procedure. CACM 19,3 (March 1976), pp. 137-147.