Omega -- A DATA FLOW ANALYSIS TOOL FOR THE C PROGRAMMING LANGUAGE

by

Cindy Wilson* and Leon J. Osterweil**

*Bell Laboratories, Denver, Colorado

**Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado 80309

CU-CS-217-82 May 3, 1982

This research was supported, in part, by NSF grant MCS 7702194 and NSF grant MCS 8000017.

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE NATIONAL SCIENCE FOUNDATION.

THIS MATERIAL IS BASED UPON WORK SUPPORTED BY THE NATIONAL SCIENCE FOUNDATION UNDER GRANT NO. MCS 79-Ø3838.

Omega -- A DATA FLOW ANALYSIS TOOL FOR THE C PROGRAMMING LANGUAGE

Cindy Wilson
Bell Laboratories
Denver, Colorado

Leon J. Osterweil
Department of Computer Science
University of Colorado
Boulder, Colorado

1. Abstract

This paper describes Omega, a prototype system designed to analyze data flow in C programs. Omega is capable of detecting certain types of common programming errors, or assuring their absence. Omega also addresses the problems of analyzing pointer variables.

2. Background

Within the past decade there has been considerable activity in the area of developing tools to assist in the process of testing, debugging, analyzing, editing, and documenting programs. Most of the tool development work has been language specific, and aimed primarily at such languages as Fortran [Oste 76], [Stuc 75], [Rama 75] and Jovial [Gann 78]. In fact it has been suggested that the popularity of these languages in the face of "superior" newer languages can be traced, at least in part, to their support by superior tools. C is one such newer language [Kern 78] which has attracted considerable attention and support. It appears that its growing popularity is at least partially due to the ready availability (within C's surrounding UNIX* environment) of superior support tools. Unfortunately few of these tools are C specific, and fewer still (e.g. LINT [John 78]) are designed to support the analysis, documentation and debugging of C programs. This seems unfortunate and paradoxical, as many of the analytic and diagnostic capabilities developed in the last decade are language independent. Hence we set out to develop a C-specific analysis, documentation and verification tool based upon principles first applied in developing a tool for another language.

The analytic technique we chose to exploit is data flow analysis. This term and analytic procedure were first developed in the context of global program optimization [Alle 76]. Later, however, it was shown that these techniques are also valuable in error analysis [Fosd 76]. In particular the data flow analysis approach has been applied to the analysis of Fortran programs by the DAVE tool [Oste 76].

* UNIX is a Trademark of Bell Laboratories.

Data flow analysis is a form of static analysis -- a term generally used to describe techniques used to study a program without having to exercise it on sample test data. Perhaps the most distinguishing and significant feature of static analysis is that it is capable of assuring the absence of certain kinds of errors in a program. Testing approaches, on the other hand, are best at detecting the presence of errors, by noting their occurrence in the course of (a usually heavily monitored) execution of the program with carefully selected test data. In general static analysis entails careful examination of the details of a program for the purpose of inferring its general structure and nature. These inferred analytic results are typically used as the basis for subsequent error checking procedures. They are, however, also usually quite useful as program documentation. Hence static analysis techniques are best thought of as verification and documentation aids. We shall see that the tool described here does in fact provide both of these kinds of benefits.

Data flow analysis is essentially a technique for studying sequences of program events. Data flow analysis algorithms are capable of documenting all sequences of certain types of program events which might possibly occur in a program execution. This is generally quite valuable and informative documentation. In addition, however, when certain sequences of events can be classified as erroneous or suspicious, this analytic technique _de facto_ becomes capable of detecting errors or anomalies [Oste 76]. Because, moreover, all possible execution sequences are inferred by data flow analysis, if no erroneous or anomalous event sequences are detected then the error or anomaly thereby defined has been shown to be absent from the program under study.

The DAVE system focussed on three types of program events:

1. definition of a variable (a "d" event)

2. reference to a variable (an "r" event)

3. undefinition of a variable (a "u" event)*

Three specific sequences of these events were sought -- undefined reference, dead definition and double definition. Undefined reference was defined to be a "u" event followed immediately by an "r" event without an intervening "d" event. Such a sequence usually results when a variable is referenced after program execution begins and before the variable has been defined. This was classified as an error.

---

* This event occurs when execution leaves the scope of definition of a variable, or in certain languages (e.g. Fortran) when a loop iterator is exhausted. Most commonly it occurs implicitly at the beginning of a program or procedure's execution, where it is assumed that no variable is assigned an initial value.

Double definition was defined to be two consecutive definition ("d") events for a variable without any intermediate reference ("r") or undefinition ("u") events. Dead definition was defined to be a definition (d) of a variable followed immediately by an undefinition (u) of that variable. In both cases the first definition is clearly superfluous. Neither was classified an error but rather both are considered anomalies -- causes for concern.

It should be pointed out that papers have been written describing the applicability of data flow analysis to other events and corresponding errors and anomalies. In [Oste 82] data flow analysis is shown to be useful in detecting concurrency errors. In [Oste 82] it is also shown to be applicable to the detection of certain file manipulation errors. In this paper, and the tool described, however, we focus only on undefined reference, dead definition and double definition errors and anomalies.

One important and novel feature of this work is that it addresses the problems of analyzing pointer variables. An important assumption of data flow analysis is that the location of every r, d, and u event in a program be detectable, and the identity of the variable being affected also be determinable. In more straightforward languages (such as Fortran) this is a reasonable assumption. In languages which allow the use of pointer variables (e.g.

PL/I and C) this assumption is questionable. Some analytic procedures for handling pointer variables are known [Aho 79] [Harr 77], but appear to be of only limited applicability to C. One of these has been applied in the work described here, enabling us to begin to evaluate its practicality for C.

3. Definitions and Examples

We now briefly summarize some definitions which are necessary in order to understand the work described here. These definitions are somewhat cursory. A more complete treatment can be found in [Fosd 76].

Of most basic importance is the notion of a graph. A graph G is an ordered pair G = (V, E) where V is said to be the vertex or node set, and E is said to be the edge set. The edge set, E, consists of edges which are ordered pairs (v_i, v_j) of vertices from the vertex set V. One specific type of graph will be of particular interest to us -- the flowgraph. The nodes of the flow graph correspond roughly to the statements of a program. An edge appears in a flowgraph for every ordered pair of vertices (v_i, v_j) for which the statement represented by v_j can be executed immediately after the statement represented by v_i. In order to do data flow analysis, the nodes of the flowgraph must be annotated according to actions which are to be performed on data at that particular node. The three actions which may be performed on data are: reference (r), definition (d), and undefinition (u). When execution of an expression requires that the value of a variable, say alpha, be retrieved, alpha is said to be referenced within the expression. When execution of an expression causes a value to be assigned to alpha, alpha is said to be defined. In C, all local variables are undefined at the point of function invocation and at the function exit node.

Thus the flowgraph of the function of Figure 1 would be as shown in Figure 2.

```
extern int Z;
funcb()
        { int X, Y;

          if (X == Z) Z = X++;
          Y = Y + 1;
          Z = Y;
        }
```

Figure 1

Function Containing Data Flow Anomalies

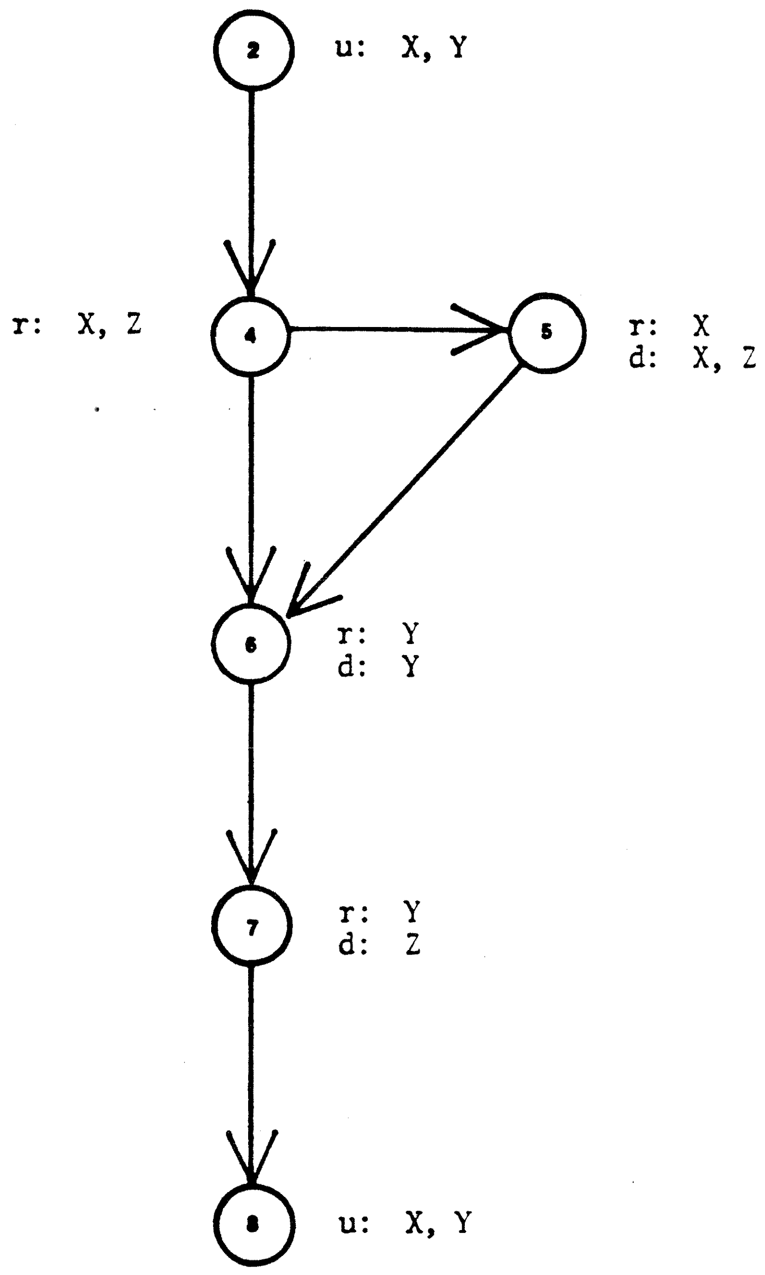


Figure 2

Annotated Flow Graph Corresponding to Figure 1

By inspection it is seen that the actions performed on X are either uru or urrdu (depending upon which execution path is taken); the actions performed on Y are urdru; and the actions performed on Z are either rd or rdd, where the left to right order of the actions denotes the order in which the actions are performed. We see that a dead definition anomaly may occur for X due to the subsequence "du". An undefined reference error occurs for Y due to the subsequence "ur". A double definition anomaly may occur for Z due to the subsequence "dd". Sequences of actions on a variable such as we have just discussed are called "path expressions". In general we say that path expressions of the form purp′, pddp′, pdup′ (where p and p′ stand for arbitrary sequences of r′s, d′s, and u′s) are indicative of the three data flow anomalies we have discussed.

We now discuss the problem of determining the presence or absence of these path expressions by analyzing annotated flowgraphs.

4. Basic Data Flow Analysis Notions

Associated with every flow graph is a token set, <u>tok</u> , the elements of which are those variables which are currently active within the associated function. In C, this set would consists of the local variables, external variables, and those nonlocal variables whose addresses have been passed to the function as parameters. With every node n are associated three sets: gen(n), kill(n), and null(n), where gen(n) U kill(n) U null(n) = tok. Membership of variables in these sets at a particular node is determined by which of the actions r,d, and u are performed on which variables at the statement represented by that node. The rules for determining membership differ for each of the three problems to be solved.

Hence, although the r, d, and u flowgraph annotations are determined and fixed once by textual analysis, the sets gen(n), kill(n), and null(n) will be initialized differently for each error or anomaly to be studied. These initializations are made in such a way as to make the detection of erroneous or anomalous path expressions expedient. The path expressions are not actually created explicitly but are rather inferred implicitly with the aid of two standard algorithms, LIVE and AVAIL [Hecht 75]. These algorithms associate with each node n the sets live(n) and avail(n), subsets of tok. For each variable v in tok, and each node n of the flow graph,

- v is a member of live(n) if and only if on some flowgraph path from n the first non- null set to which v belongs is a gen set.

- v is a member of avail(n) if and only if on all paths entering _n_, the last non- _null set to which v belongs is always a gen_ set.

Efficient LIVE and AVAIL algorithms are described in the literature [e.g. Hech 75]. Our tool uses these algorithms to detect the presence and assure the absence of the three data flow anomalies of interest here.

For example, Figure 3 shows how the flowgraph of Figure 2 must be annotated to facilitate detection of those variable references which always result in the undefined variable reference error. Specifically for a given node _n_ _gen_(_n_) is defined to be the set of all variables which are considered to become undefined at node _n_, and _kill_(_n_) is defined to be the set of all variables which become defined at node _n_. After executing the AVAIL algorithm we see that a variable v is _avail_(_n_) if and only if v has been undefined on all paths leading up to n and has not been defined on any path subsequent to the last undefinition. Hence if $v \in$ _avail_(_n_) it is certain that the path expression for v on any and all paths leading to _n_ is of the form "pu". Therefore if v is also referenced at _n_ (an "r" event for v occurs at _n_) we can be sure that the reference results in the undefined reference error for v at _n_.

In order to detect the impossibility of this error, a different initialization of the _gen_ and _kill_ sets would be necessary. In order to study the other two anomalies still different initializations are necessary.

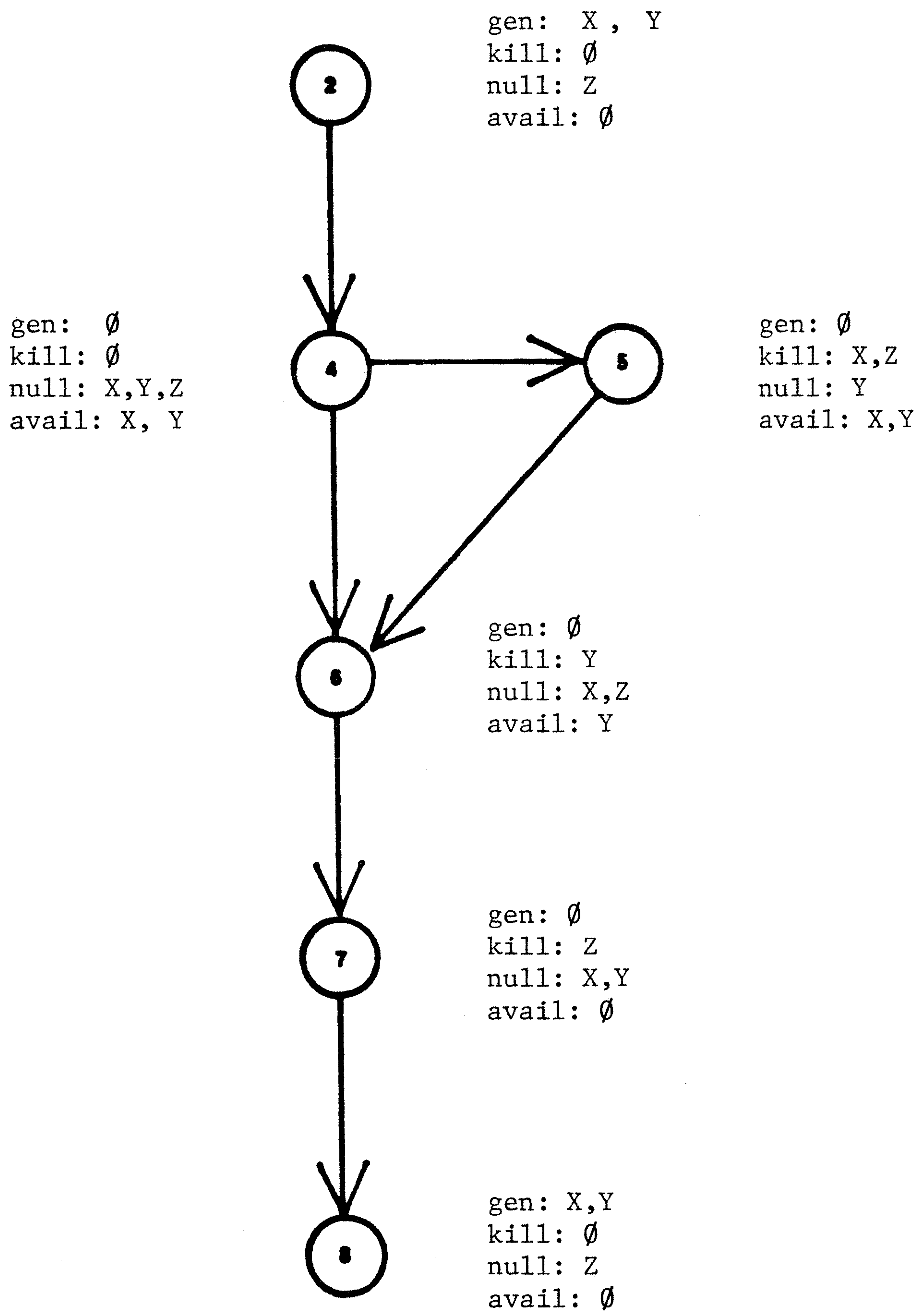


Figure 3

The flow graph of Figure 2 annotated to facilitate detection of data flow anomalies

In order to properly annotate a flow graph, the identity of every variable being affected by an r, d, or u event must be determinable. For nodes at which actions occur on those variables corresponding to a dereferenced pointer variable, the set of variables to which the pointer may point must be determined before we may annotate the flow graph. In order to accomplish this, certain assumptions must be made as to the effects of pointer assignments.

As an example, let PTRX and PTRY be pointer variables, let Z be an elementary data type or an array of elementary data types, and let K be a nonzero integer constant. Omega makes the following assumptions concerning the effects of assignments to PTRX:

An assignment statement of the form PTRX = address(Z) causes PTRX to point to Z. If Z is an array, an assignment of the form PTRX = address(Z) + K is also allowed. PTRX = PTRY causes PTRX to point to anything currently pointed to by PTRY. PTRX = PTRY + K causes PTRX to point to any array currently pointed to by PTRY. All other pointer assignments are deemed semantically meaningless or non-analyzable and are ignored by Omega.

Once the effects of the pointer assignments at a given program node have been established, this information must be propagated to the other nodes in the flow graph. This is done by Omega in the following manner:

> Let x = any pointer variable
>
> S(Kx) = the set of objects to which variable x points at node K
>
> Then the set of objects to which x points at node N may be computed as follows:

$$S(Nx) = \begin{cases} \bigcup\limits_{\substack{\text{all } K, \\ \text{immediate} \\ \text{predecessors} \\ \text{of } N}} S(Kx) & \text{if } S(Nx) = 0 \\ S(Nx) & \text{if } S(Nx) \neq 0 \end{cases}$$

This propagation rule, when applied iteratively, effects the annotation of the flow graph to reflect any indirect references or definitions of variables which may occur through pointer variable manipulation.

Since the scope of a variable may extend across procedure boundaries, Omega must be capable of examining intraprocedural data flow. Fosdick and Osterweil [Fosd 76] have given algorithms to detect anomalous data flow on paths crossing procedure boundaries. These algorithms are used by Omega to provide data flow information for any variables whose addresses are passed as function arguments and for globally defined variables used within functions. These algorithms require that each function be analyzed only once. However, a loss of analytical accuracy results.

The loss of accuracy is compounded when pointers must be analyzed, as is the case in most C programs. For this reason, we have chosen to modify the algorithmic procedures presented in [Fosd 76]. These modifications essentially entail having to re-scan procedures in order to properly and adequately capture the behavior of pointer variables used as arguments.

The following example illustrates that the analytic results obtained by the algorithms in [Fosd 76] can yield unacceptably imprecise analytic results even for a simple sequence of statements dealing with pointer arguments. Because such coding sequences are common in C programs we elected to reanalyze procedures in the context of each invocation.

Consider the function in Figure 4.

In order to derive path set information not bound to particular pointer references, we would have to generate generalized information similar to the following:

- at node 2: reference all variables pointed to by ptrx
  reference all variables pointed to by ptrx
- at node 3: adjust ptry to point to everything pointed to by ptrx
- at node 4: define all variables to which ptry possibly points

Without extensive bookkeeping, we could not derive a generalization such as "the last thing that happens to all

```
node number

     1          func(ptrx, ptry) int *ptrx, *ptry;
                        /*  "ptrx" and "ptry" are pointers to   */
                        /*  variables of type integer           */

                        {
     2                  if (*ptry == *ptrx)
                                /*  if the contents of what     */
                                /*  ptry points to equals the   */
                                /*  contents of what ptrx       */
                                /*  points to . . .             */

                                {
     3                          ptry = ptrx;
                                  /*  Set ptry to point to      */
                                  /*  everything ptrx points to.*/
                                  /*  This statement does not   */
                                  /*  alter the set of possible */
                                  /*  pointer references for    */
                                  /*  the actual parameter     */
                                  /*  corresponding to ptry.   */

     4                          *ptry = 0;
                                  /*  set the contents of what  */
                                  /*  ptry points to equal to 0 */
                                }
     5                  }
```

Figure 4

Sample function which illustrates difficulties involved with pointer variable analysis.

variables pointed to by ptry on path 1-2-3-4-5 is 'd'" due to the fact that the set of variables to which ptry points at node 4 is changed to the set of variables pointed to by ptrx. The proper detection and characterization of this entails considerable data flow analysis. at node 3. Any additional adjustments to ptrx and/or ptry would complicate matters further. Application of the techniques described in [Fosd 76] unfortunately result in obtaining an overly general characterization of what this function does that serves to weaken the analytic results obtainable.

If a function is called from several locations, and at each of them only one variable is bound to each pointer argument, we would like to obtain specific, precise diagnostic information about each single variable. This is the case if the invoked function is reanalyzed in the context of each invocation. If the function is analyzed only once, however, the analytic reports must be weakened by inability to distinguish precisely which data flows must happen. Instead, numerous reports of what may happen are produced. In general, this problem will arise for any function which has a pointer parameter which appears in the function in non-dereferenced form as part of an assignment statement. Requiring that each call to a function of this type invokes the re-computation of path expressions, with the actual parameters substituted for the formal parameters, eliminates these complications at the expense of greatly decreased time efficiency.

In addition, for any function which employs pointer type input parameters, the set of all variables to which pointer parameters may point must be known before the function is processed. Therefore, the calling function must have propagated pointer information throughout the flow graph before the subfunction analysis may begin.

Thus, the overall design of Omega is heavily influenced by pointer variable processing requirements. The algorithm of Figure 5 is used by Omega as the driver routine for processing the functions. Figure 6 illustrates the overall design of Omega.

```
process(this_func, nprime) :

        {
        do until (the set of all possible pointer references
                for all pointers does not change)
                {
                propagate pointer information to all nodes;
                make any necessary adjustments;
                }

        if (this_func calls that_func at node n)
           and (that_func is not a system function)

                {
                substitute the actual parameter references
                for the formal parameter references in
                that_func;

                process(that_func, n);
                }


        determine the path sets for this_func;

        export path information to nprime;

        determine and report local anomalies for this_func;

        }
```

Figure 5

Omega Driver Routine for Analyzing Functions

```
Parse the source program, generating flow
graph and calling graph information;

Construct the flow graphs and call graph;

Starting with the main function, apply
the algorithm of Figure 5.
```

Figure 6

Overall Design of Omega

5. Example

As an example of Omega´s analytic power, consider the program in Figure 7, which is intended to do the following:

- for each row of a given matrix, compute the sum of the absolute values of the elements
- print the largest sum computed

In line 8, "matrixptr" is referenced without being previously defined, a undefined reference. The function "readmatrix" expects a pointer to an N by N matrix. Perhaps the programmer meant to set "matrixptr" equal to the address of "matrix" before the subroutine invocation. Or the programmer may have meant to pass the parameter "matrix," but confused the variable names. Due to this error, the

```
#define N 5
main()
        {
        int matrix[N][N];
        int *matrixptr;
        int rowmax;

        readmatrix(matrixptr);
        rowmax = getmax(matrixptr);

        printf("  Maximum row sum is:  %d0, rowmax);
        }


readmatrix(ptr)     int ptr[N][N];
        {
        int i, j;

        for (i = 0; i < N; i++)
                {
                for (j = 0; j < N; j++)
                        {
                        scanf("%d", &ptr[i][j]);
                        }
                }

        }


getmax(ptr)   int ptr[N][N];
        {
        int row, temp, max;

        for (row = 0; row < N; row++)
                {
                temp = rowsum(ptr[row]);
                if (temp > max) max = temp;
                }
        return(max);
        }

rowsum(vecptr) int *vecptr;
        {
        int column, sum;

        while (column < N)
                {
                sum = sum + abs(*(vecptr++);)
                column++;
                }
        column = sum = 0;
52          return(sum);
53          }
54
55    abs(value) int value;
56          {
57          if (value >= 0) return (value);
58                          else return (-value);
59
60          }
```

Figure 7

Program containing data flow anomalies

function "readmatrix" will silently initialize the (N * N) locations starting at whatever location "matrixptr" is pointing to.

Likewise, in line 37, a undefined reference occurs for the variable "max," which should have been initialized to zero.

In the function "rowsum," the statement "row = sum = 0" was accidentally placed at line 51 when it should have been at line 45. This results in a undefined reference for "column" at line 46, double definitions for "sum" and "column" at line 51, and a dead definition anomaly for "column" at line 53.

The annotated flow graphs and call graph corresponding to this program are given in Figures 8 and 9. Figure 10 demonstrates the diagnostic capabilities of Omega.

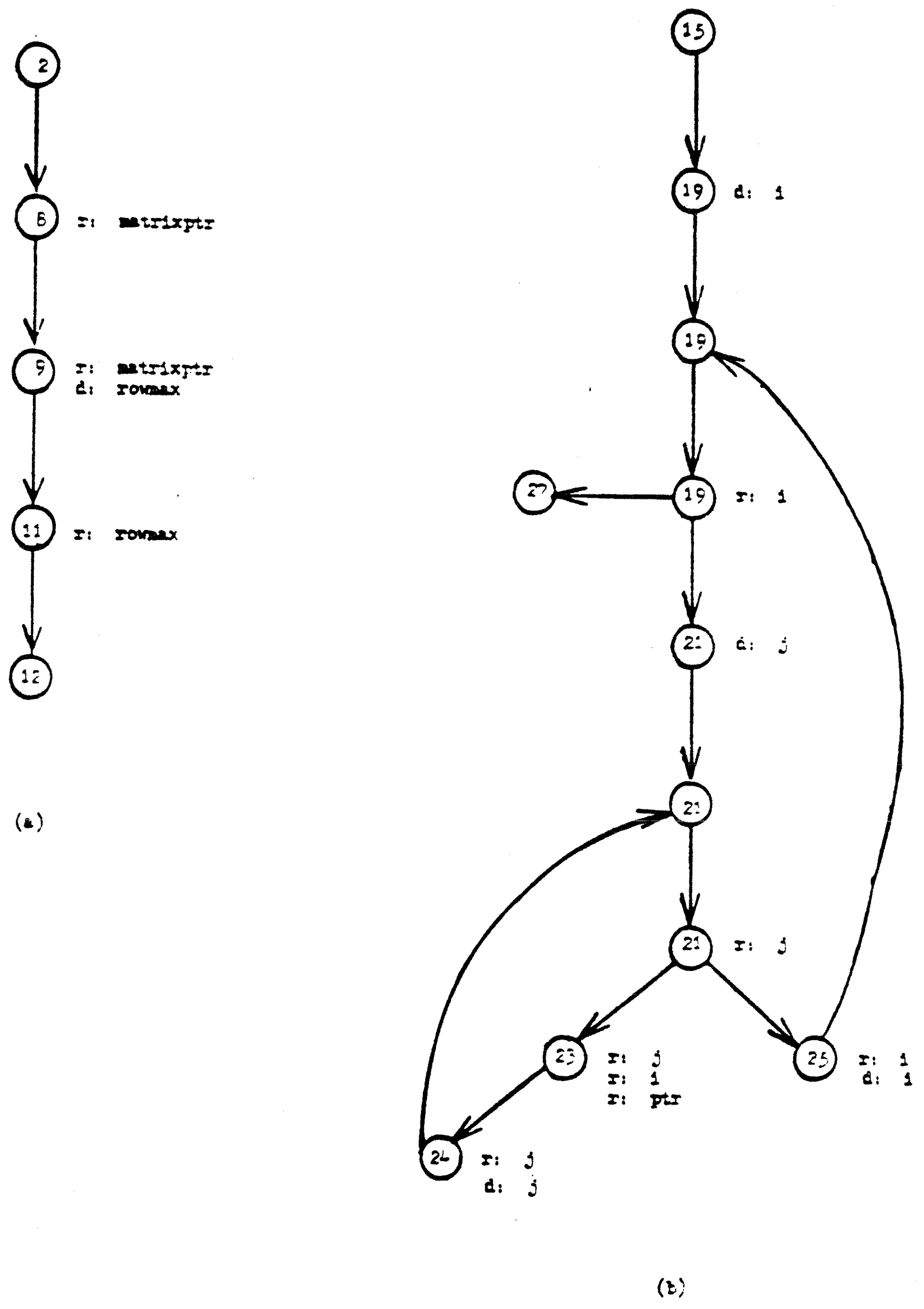
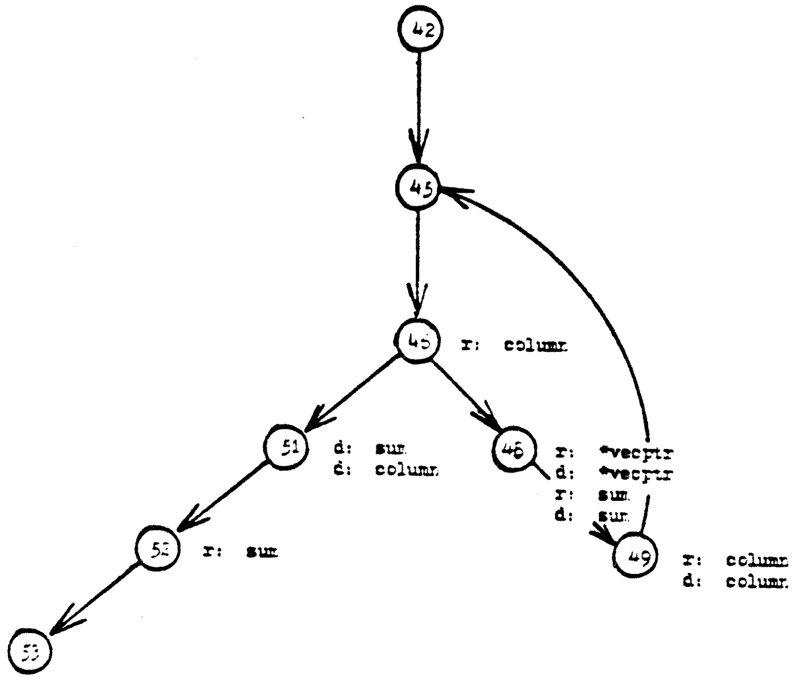
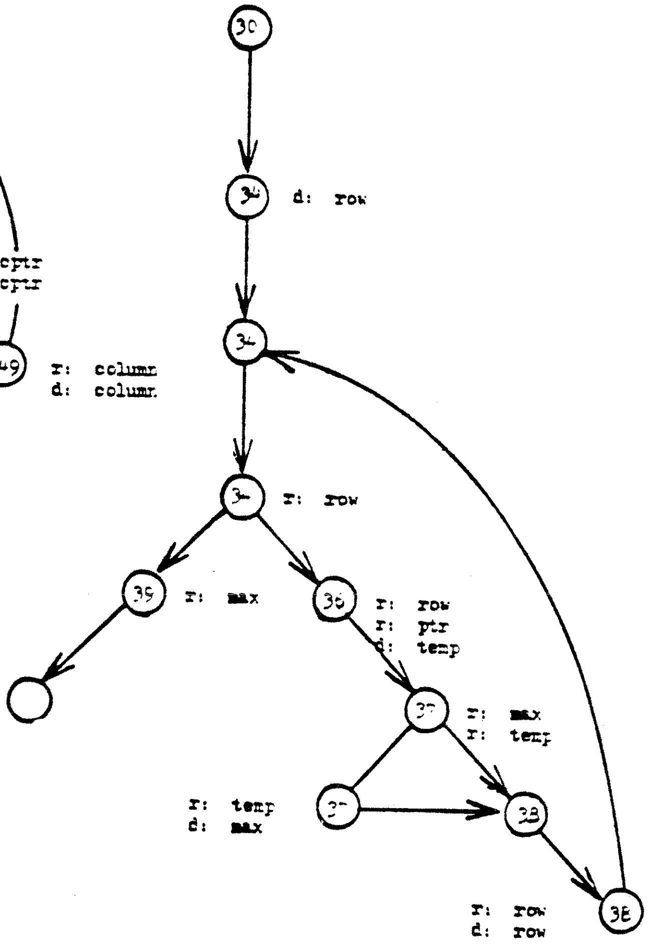


Figure 8

Annotated flow graphs corresponding to
the example program of Figure 7:
(a) main, (b) readmatrix, (c) rowsum, (d) getmax, (e) abs



(c)



(d)

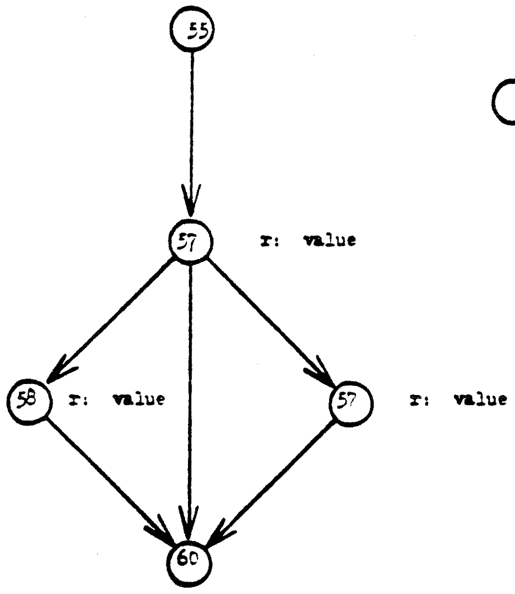


Figure 8, continued

(e)

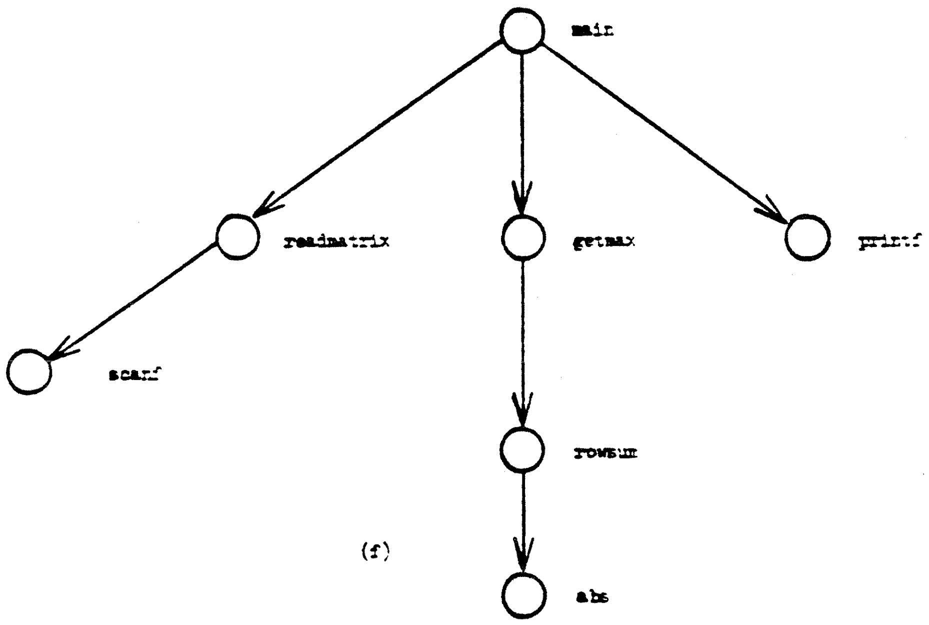


Figure 9

Call graph corresponding to the
example program given in Figure 7

```
*** Variable with name "sum" local to function "rowsum"
*** is assigned a value, then reassigned another value
*** before the first value is used.
*** This occurs on some path(s) emanating from line number: 48
*** Path(s) (indicated by one or more line numbers)
*** in which this anomaly was detected are:
***
***
***     48
***     49
***     46
***     50
***     51
***
***
***
*** Variable with name "column" local to function "rowsum"
*** is assigned a value which is never used.
*** This occurs on all paths emanating from line number: 51
*** Path(s) (indicated by one or more line numbers)
*** in which this anomaly was detected are:
***
***
***     51
***     52
***     53
***
***
***
*** Variable with name "column" local to function "rowsum"
*** is referenced before it is assigned a value.
*** This occurs on all paths emanating from line number: 43
*** Path(s) (indicated by one or more line numbers)
*** in which this anomaly was detected are:
***
***
***     43
***     46
***
***
***
*** Variable with name "sum" local to function "rowsum"
*** is referenced before it is assigned a value.
*** This occurs on some path(s) emanating from line number: 43
*** Path(s) (indicated by one or more line numbers)
*** in which this anomaly was detected are:
***
***
***     43
***     46
***     48
```

```
***
***
***
*** Variable with name "max" local to function "getmax"
*** is referenced before it is assigned a value.
*** This occurs on all paths emanating from line number: 31
*** Path(s) (indicated by one or more line numbers)
*** in which this anomaly was detected are:
***
***
***     31
***     34
***     38
***     39
***
***
***
***     and also the same initial path as above,
***     with the following path after line 34:
***
***     36
***     37
***
***
***
*** Variable with name "matrixpt" local to function "main"
*** is referenced before it is assigned a value.
*** This occurs on all paths emanating from line number: 3
*** Path(s) (indicated by one or more line numbers)
*** in which this anomaly was detected are:
***
***
***     3
***     8
***
***
***
```

Figure 10

Actual Omega Output for the example program
of Figure 7

6. Conclusion

Omega currently is implemented in prototype form and is being used experimentally at Bell Telephone Laboratories, Denver, Colorado. An important goal of this experimental use is to determine whether the decision to perform reanalysis of every function at the point of its invocation was a good one. We have observed earlier that this decision enables us to obtain sharper analytic results at the expense of increased execution time. In the worst case, where a subject program′s procedure invocation structure is sufficiently intricate, our analytic procedure may require an amount of time which is exponential in the number of functions rather than linear, as is the case for the procedure in [Fosd 76].

Our hypothesis is that for most actual programs in need of data flow analysis, the penalty in running time will be far less severe, and will be amply justified by the greatly sharpened analytic results and the thorough documentation of the actual usage of pointers which are obtained. This hypothesis is to be explored by our experimentation.

REFERENCES

[Aho 79] Aho, A. V.; and Ullman, J. D. Principles of compiler design, Addison - Wesley Publishing Company, Reading, Mass., 1979.

[Boll 79] Bollacker, L. A., "Detecting Unexecutable Paths Through Program Flow Graphs," unpublished Master′s thesis, Department of Computer Science, University of Colorado, Boulder, CO, 1979.

[Fosd 76] Fosdick, L. D.; and Osterweil, L. J. "Data flow analysis in software reliability," Computing Surveys Vol. 8, No. 3 [Sept. 1976], 305 - 330.

[Gann 78] Gannon, C., "JAVS: A JOVIAL Automated Verification System," Proc. COMPSAC ′78, November 1978, pp. 539-544.

[Harr 77] Harrison, William H., "Compiler Analysis of the Value Ranges for Variables," IEEE Trans. Software Eng., SE-3, no. 3 (May 1977), 243-250.

[Hech 75] Hecht, M. S.; and Ullman, J. D. "A Simple algorithm for global data flow analysis problems," SIAM J. Computing 4 [Dec. 1975], 519 - 532.

[John 78] Johnson, S. C. "Lint, a C Program Checker," Computer Science Tech. Report, Bell Laboratories, Murray Hill, New Jersey (July 1978).

[Kern 78] Kernighan, Brian W.; and Ritchie, Dennis M. The C Programming Language, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.

[Oste 76] Osterweil, L. J.; and Fosdick, L. D. "DAVE -- a validation, error detection and documentation system for FORTRAN programs," Software Practice and Experience, 6, no. 4 (September 1976), 473-486.

[Oste 82] Osterweil, L. J., Fosdick, L. D., and Taylor, R. N., "Error and Anomaly Diagnosis Through Data Flow Analysis," in Program Test Methods (Chandrasekaran and Radicchi, editors), North Holland, 1982.

[Rama 75] Ramamoorthy, C. V., and Ho, S. B. F., "Testing Large Software With Automated Software Evaluation Systems," IEEE Trans. Software Eng., SE-1, no. 1 (March 1975), 46-58.

[Stuc 75] Stucki, L. G., and Foshee, G. L., "New Assertion Concepts for Self-Metric Software Validation," Proc. 1975 Int. Conf. Reliable Software, Los Angeles, CA (April 1975), pp. 59-71.