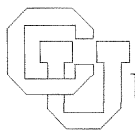
DAVE – A Validation, Error Detection and Documentation System for Fortran Programs *

Leon J. Osterweil
Lloyd D. Fosdick

CU-CS-071-75



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* Supported in part by NSF Grant GJ-36461.

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 6, 473–486 (1976)

DAVE—A Validation Error Detection and Documentation System for Fortran Programs*

LEON J. OSTERWEIL AND LLOYD D. FOSDICK
Department of Computer Science, University of Colorado, Boulder, Colorado, U.S.A.

SUMMARY

This paper describes DAVE, a system for analysing Fortran programs. DAVE is capable of detecting the symptoms of a wide variety of errors in programs, as well as assuring the absence of these errors. In addition, DAVE exposes and documents subtle data relations and flows within programs. The central analytic procedure used is a depth first search. DAVE itself is written in Fortran. Its implementation at the University of Colorado and some early experience are described.

KEY WORDS Program testing Data flow analysis Software validation Automated documentation Debugging

INTRODUCTION

This paper describes an operational system to improve software reliability. It is designed to examine a program and report the presence, possible presence or complete absence of a significant class of programming errors. We call it and other systems designed primarily for this purpose *software validation systems.* Other approaches to software reliability using the computer as a primary instrument are verification or proof of correctness,[1] debugging[2] and dynamic testing.[3] Falling in a different category are those approaches which deal with communication between humans and machines. Here are included language design,[4] top down systems design[5] and step-wise refinement.[6] Finally, there are those approaches which deal with the organization and management of the people who construct software.[7] Each of these approaches is important and most need much more development. Although significant problems remain in designing validation systems, the work described here shows that it is currently possible to build validation systems capable of aiding programmers engaged in actual software construction.

Data flow analysis is the primary methodology in our approach to software validation. We use it to reveal suspicious or erroneous use of data. Others have used data flow analysis to study machine design,[8] and for global optimization of computer programs,[9] but it has seen little use as a tool in software validation. Since it provides information about the use of variables in a program it is also an important tool in automatic program documentation.

Our system, called DAVE, is characterized by the following features.

1. It performs an exhaustive search for data flow anomalies in time which is linearly proportional to the product of the number of edges in the flow graph and the number of program variables.

* Supported in part by NSF Grants GJ-36461 and DCR-75-09972.

Received 2 July 1975
Revised 18 November 1975

2. It applies to programs written in ANSI Fortran, and can be modified to apply to other languages.
3. It employs a static analysis of the program; i.e. it avoids executing the program.
4. It classifies the usage of all local and global variables.
5. It applies to an entire program or to selected parts of a program.
6. It is written in Fortran, and is designed for ease of portability.

DEFINITIONS AND EXAMPLES OF DATA FLOW ANOMALIES

In data flow analysis attention is directed at the sequential pattern of definitions, references and undefinitions of values for variables. The actual values assigned or referenced are ignored, only the fact that an assignment, reference or undefinition was made is used. Two rules concerning the sequence of these events along each path from the start of a program to a stop are expected to be obeyed.

1. A reference must be preceded by an assignment, without an intervening undefinition.
2. A definition must be followed by a reference, before another definition or undefinition.

Violation of the first rule should cause an erroneous result during program execution; moreover, in the case of Fortran it is a violation of the ANSI Standard[10] (Section 10.3). Violation of the second rule should result in a waste of time, but not an erroneous result.

Many things can cause a violation of either or both rules. Forgetting to initialize a variable is the most obvious cause of a violation of the first rule. However, spelling errors, confusion of names, misplaced statements and faulty subprogram references also cause violations of this rule. The second rule may be violated when a programmer forgets that a variable is already defined or that it will not be used later. Many optimizing compilers remove this 'dead' variable assignment, assuming these to be the only causes. However, many common errors also cause violations of the second rule. We call violations of the two rules anomalies 'type 1' and 'type 2' respectively.

Figure 1 is a simple illustration of how anomalies can arise in a program segment. A linear scan of the program text will detect the type 1 anomaly and some compilers (e.g. MNF[11], WATFOR[12] and G. E. Time Sharing[13]) detect this condition by performing such a scan. Many compilers enable detection of this condition during execution by initializing each variable to some distinguished value. In more complex cases both approaches prove unsatisfactory. Consider the example in Figure 2. Here, as in Figure 1, a misspelling of PI (or P) is present, but now it is far less apparent. A simple linear scan stands no chance of detecting the anomaly, and any execution of DOLS for which the second argument is not equal to 2 will not cause it to be detected during execution.

```
      PI = 3.1416
      READ(5, 100) X
      AREA = P*X**2
      WRITE(6, 100) AREA
      STOP
100   FORMAT(E12.4)
      END
```

Figure 1. A program containing a type 1 anomaly (the variable P is referenced before definition) and a type 2 anomaly (the variable PI is defined but never referenced afterwards). The programmer probably intended P to be PI, or conversely

```
      SUBROUTINE DOLS(PSF, LCRT, D1, D2, COST)
      PI = 3.1416
      IF(LCRT.NE.1) GO TO 10
      COST = PSF*AREAR(D1, D2)
      RETURN
10    IF(LCRT.NE.2) GO TO 20
      COST = PSF*AREAC(P, D1)
      RETURN
20    COST = PSF*AREAT(D1, D2)
      RETURN
      END
      FUNCTION AREAR(X, Y)
      AREAR = X*Y
      RETURN
      END
      FUNCTION AREAC(PI, R)
      AREAC = PI*R**2
      RETURN
      END
      FUNCTION AREAT(B, H)
      AREAT = 0.5*B*H
      RETURN
      END
```

Figure 2. A more complicated example of a program having a type 1 anomaly and a type 2 anomaly. Notice that the function subprogram AREAC is executed without initialization of the first argument (dummy argument PI) and that there is no use of PI in the subroutine DOLS after it is assigned a value in the first statement; evidently PI should be P or vice versa

Our system, DAVE, is designed to locate anomalies in these and other far more complex situations. It prints a description of each anomaly located which is designed to simplify the difficult task of identifying the cause. For example, in the case of the program displayed in Figure 2 DAVE would report that in the program unit DOLS

> THE LOCAL VARIABLE NAMED P IS REFERENCED BEFORE BEING DEFINED ON SOME PATHS

and following this message one such path would be printed. DAVE would also report that in the program unit DOLS

> THE VARIABLE NAMED PI IS ASSIGNED A VALUE IN ITS LAST USAGE ON ALL PATHS

and following this message one such path would be printed. A complete list of messages produced by DAVE is available.[14]

THE DEPTH FIRST SEARCH PROCEDURE

The heart of the DAVE system is a depth first search procedure which determines the input and output usage of a variable, α, within a program unit.

A variable α is strict input to a statement if the statement cannot be executed successfully until the value of α is obtained. For example, α is a strict input to each of the following statements: $X = A + \alpha$; ARRAY($\alpha + 2$) = 2.9; DO 10 I = 1, α, 3.

A variable α is strict output from a statement if the value which α had before execution of the statement is replaced by a value generated during execution of the statement. For example, α is strict output from each of the following statements: $\alpha = 0$; DO 100 $\alpha = 1$, 10; READ (5, 25)α.

We extend these concepts to a leaf subprogram; i.e. a subprogram which does not reference another subprogram. Let S be a Fortran subprogram consisting of the statements $s_1, s_2, \ldots, s_n$.* Form G_S, the program flow graph of S, by taking the vertex set of G_S to be $\{s_1, \ldots, s_n\}$, where it is assumed that s_1 is the statement containing the unique entry point of S, and the edge set of G_S to be the set of all ordered pairs of vertices (s_x, s_y) for which it is possible to execute s_y immediately after s_x. Hence the vertices of G_S are the statements of S and the edges of G_S are the possible transfers of control between statements. For simplicity we have used s_x to stand for a statement in S and a vertex representing that statement in G_S; similarly in discussing the flow graphs we will use the terms 'statement' and 'vertex' interchangeably. For the definitions of other graph theoretic terms used in this paper, see Harary.[15]

A *path* through G_S is a sequence of vertices $s_{i_1}, s_{i_2}, \ldots, s_{i_x}$ where $(s_{i_j}, s_{i_{j+1}})$ is an edge of G_S for $j = 1, 2, \ldots, x-1$, $i_1 = 1$ and s_{i_x} is a STOP or RETURN statement. It is to be noted that a path is determined entirely by the incidence relations which define the graph and does not take into account any constraints that might be implied by execution of the statements s_{i_j}. Thus a path does not necessarily represent a sequence of statements that could actually be executed; the latter we call an *execution path.*

A minimum function, min(X), and a maximum function, max(X), on a finite set of integers, X, are defined as follows:

$$\min(X) = \begin{cases} \infty & \text{if } X = \phi, \\ \text{smallest integer in} \\ \quad X \text{ if } X \neq \phi \end{cases} \qquad \max(X) = \begin{cases} -\infty & \text{if } X = \phi, \\ \text{largest integer in} \\ \quad X \text{ if } X \neq \phi \end{cases}$$

Sets $I_{p,\alpha}$, $O_{p,\alpha}$ and $U_{p,\alpha}$ for a path, $p = s_{i_1}, s_{i_2}, \ldots, s_{i_x}$, in S are defined as follows:

$I_{p,\alpha} = \{j \mid \alpha \text{ is a strict input variable for } s_{i_j}\}$
$O_{p,\alpha} = \{j \mid \alpha \text{ is a strict output variable for } s_{i_j}\}$
$U_{p,\alpha} = \{j \mid \alpha \text{ becomes undefined after execution of } s_{i_j}\}$

If the condition

$$(\min(I_{p,\alpha}) \neq \infty \wedge \min(I_{p,\alpha}) \leqslant \min(O_{p,\alpha}) \wedge \min(I_{p,\alpha}) \leqslant \min(U_{p,\alpha}))$$

is true for no p in G_S, then α is a *non-input variable for S*, if it is true for some p in G_S then α is an *input variable for S*, if it is true for all p in G_S, then α is a *strict input variable for S.* If the condition

$$\max(U_{p,\alpha}) < \max(O_{p,\alpha})$$

is true for no p in G_S then α is a *non-output variable for S*, if it is true for some p in G_S then α is an *output variable for* S, if it is true for all p in G_S then α is a *strict output variable for S.*

Hence we recognize three input classes and three output classes for the variables of a subprogram.

Two algorithms, *inpvar* and *outpvar*, are presented below to illustrate the depth first search we employ to make the input/output classification of a variable in a subprogram represented as a directed graph. It is convenient to present them in Algol though their implementation in DAVE is in Fortran. Each of these algorithms operates on a subprogram S, where S must be a leaf subprogram. The first algorithm, *inpvar*, determines for a given

* In the following discussion it is convenient to think of a logical IF statement as being not one, but rather two separate consecutively numbered statements, the first consisting of the letters IF and the parenthesized logical expression which follows, and the second consisting of the subsequent executable statement.

variable *v* its input type: strict input, input or non-input. The second, *outpvar*, determines the output type for *v*.

Program notes for the procedure *inpvar*

Global quantities

Procedures.

bboutpv (*a*, *b*)—Boolean procedure, with *bboutpv* assigned the value **true** if the variable *b* is a strict output variable for statement *a*; otherwise it is assigned the value **false.**
bbinpv (*a*, *b*)—Boolean procedure, with *bbinpv* assigned the value **true** if the variable *b* is a strict input variable for statement *a*; otherwise it is assigned the value **false.**
undef (*a*, *b*, *c*)—Boolean procedure, with *undef* assigned the value **true** if the variable *c* becomes undefined upon traversing the edge from statement *a* to statement *b*; otherwise it is assigned the value **false.**

Arrays.

outdegree [*a*]—the outdegree of the statement *a.*
head [*a*, *b*]—the statement at the head of the edge *b* from the statement *a.*

Variables.

n—The number of statements, an integer.

Formal parameters

v—integer identification of variable being classified.
input—a Boolean variable which is assigned a value by *inpvar*: **true** if *v* is an input, or a strict input variable for S, **false** otherwise.
strict—a Boolean variable which is assigned a value by *inpvar*: **true** if *v* is strict input, **false** otherwise.

Thus upon exit from *inpvar*:

input = **false** implies *v* is non-input;
strict = **false** $\wedge$ *input* = **true** implies *v* is input;
strict = **true** $\wedge$ *input* = **true** implies *v* is strict input;

Numbering conventions

Statements of a subprogram are assumed to be numbered 1, 2, ..., n where n is the number of statements in the subprogram. The unique entry vertex of the subprogram graph represents statement 1.
The edges from a vertex are numbered 1 ,2, ..., n_e where n_e is the outdegree of the vertex.

```
procedure inpvar (v, input, strict);
  integer v; Boolean input, strict;
  begin
    Boolean array visited [1:n]; integer j;
    comment statements are numbered 1, 2, ..., n;
    procedure dfsrchi (statement);
      integer statement
    begin
      integer edge;
      visited [statement] := true;
      if bbinpv (statement, v) then input := true
```

```
        else
          if (outdegree [statement] = 0 ∨ bboutpv (statement, v))
            then strict := false
          else
            for edge := 1, edge + 1 while
            ((edge ⩽ outdegree [statement]) ∧ (strictly ∨ ¬ input))
          do
          begin
          if undef (statement, head [statement, edge], v)
            then strict := false
            else
            if ¬ visited [head [statement, edge]] then
              dfsrchi (head [statement, edge]);
          end
      end dfsrchi;
          strict := true; input := false;
          for j := 1 step 1 until n do visited [j] := false;
          dfsrchi (1)
    end inpvar
```

Program notes for procedure *outpvar*

Global quantities

Procedures.

bboutpv (*a*, *b*)—Boolean procedure, with *bboutpv* assigned the value **true** if the variable *b* is a strict output variable for statement *a*; otherwise it is assigned the value **false.**

undef (*a*, *b*, *c*)—Boolean procedure, with *undef* assigned the value **true** if the variable *c* becomes undefined upon traversing the edge from statement *a* to statement *b*; otherwise it is assigned the value **false.**

Arrays.

indegree [*a*]—the indegree of statement *a* in the subprogram graph.

tail [*a*, *b*]—the statement at the tail of the edge *b* to statement *a.*

leaf [*j*]—the statement which is the jth leaf (i.e. a vertex with outdegree = 0) of the subprogram graph.

Variables.

n—the number of statements in the subprogram graph.

number of leaves—the number of leaves in the subprogram graph.

Formal parameters

v—integer identification of variable being classified.

output—a Boolean variable which is assigned a value by *outpvar*: **true** if *v* is an output or strict output variable for S, **false** otherwise.

strict—a Boolean variable which is assigned a value by *outpvar*: **true** if *v* is strict output, **false** otherwise.

Thus upon exit from *outpvar*:

output = **false** implies *v* is non-output

strict = **false** ∧ *output* = **true** implies *v* is output;

```
  strict = true ∧ output = true implies v is strict output;
procedure outpvar (v, output, strict);
  integer v; Boolean output, strict;
  begin
    Boolean array visited [1:n]; integer j;
    comment statements are numbered 1, 2, ..., n;
    procedure dfsrcho (statement);
      integer statement;
    begin
      integer edge;
      visited [statement] := true;
      if bboutpv (statement, v) then output := true
      else
        if indegree [statement] = 0 then strict := false
        else
        for edge := 1, edge + 1 while
          ((edge ≤ indegree [statement] ∧ (strict ∨ ¬ output))
        do
        begin
          if undef (tail [statement, edge], statement, v)
            then strict := false
            else
              if ¬ visited [tail [statement, edge]]
                then dfsrcho (tail [statement, edge])
        end
    end dfsrcho;
    strict := true; output := false;
    for j := 1 step 1 until n do visited [j] := false;
    for j := 1 step 1 until number of leaves do
      dfsrcho (leaf [j])
  end outpvar
```

The execution time of both procedures for a given variable is linearly proportional to the number of edges in G_S, because the algorithms guarantee that no edge of G_S is examined more than once in either procedure. Moreover, because the procedures operate on G_S, and not a program source text, the two procedures could be applied to subprograms written in languages other than Fortran.

It should be noted that if there are vertices of G_S which cannot be reached from the start vertex *outpvar*'s analysis may be incorrect. Such source routines have an obvious flaw, detected by most compilers, which should be fixed before attempting the more complex analysis discussed here.

DAVE SYSTEM DESCRIPTION

The structure of DAVE is indicated in Figure 3. The subject program consisting of a main program and all subprograms referenced either directly or indirectly is assumed to have been found syntactically correct either by a compiler or a verifier such as PFORT.[16] DAVE begins by dividing the program into program units. These are then divided into

statements and statement type determination is made. Next, the subject program is passed to a lexical analysis routine which creates a token list to represent each of the program's source statements.

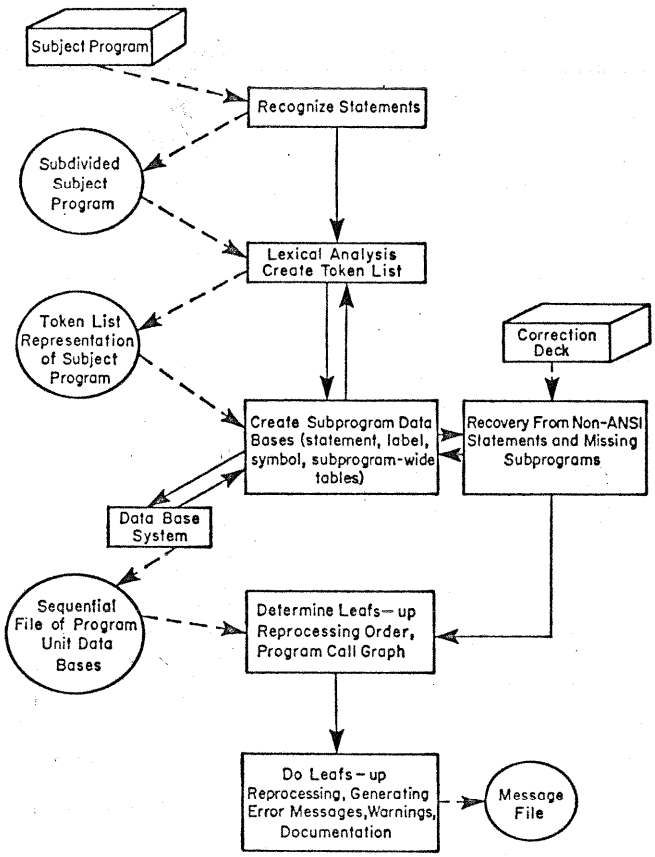


Figure 3. Structure of DAVE

As the token lists are created, comprehensive data bases for each of the program units are also created. Each of these data bases contains a symbol table, label table, statement table and table of subprogramwide data. The symbol and label tables contain the same kind of information found in most compiler symbol and label tables, listing symbol and label attributes as well as the locations of all references to the symbols and labels. The data bases are accessed using a data base creation and accessing package, designed to facilitate data base restructuring.[17]

During this lexical scan phase DAVE determines the input/output classifications of all variables used in each statement, except variables used as actual arguments in subprogram invocations, and loads this information into the statement table. DAVE contains the input/output classifications for all ANSI Fortran intrinsic functions and basic external functions. the input/output classes of variables used as actual arguments in these function invocations are also loaded into the statement table during this phase, otherwise blanks are placed in the statement table for the input/output classifications of variables used as actual arguments in subprogram invocations, and these will be filled in during a later phase of processing.

The table of subprogramwide data for a program unit contains an external reference list containing all subprograms referenced by the program unit, as well as representations of all non-local variable lists; i.e. the program unit's dummy argument list and COMMON block lists. Ultimately these lists will contain the information necessary to establish the input/output classification within the invoking program unit of all variables used as actual arguments in invocations of this program unit. The external reference lists are used to construct the program call graph, a structure that indicates which subprograms invoke which other subprograms.

DAVE is capable of pausing after this phase of processing and accepting a correction deck containing replacements for statements which are not syntactically defined under the ANSI Standard. DAVE also examines the external reference lists to determine whether all referenced subprograms have been submitted. If not, new symbolic decks may also be incorporated in the correction deck.

After all the program units of the subject program have been processed in this way, DAVE enters the main phase of *d*ocumentation, *a*nalysis, *v*alidation and *e*rror detection. The program call graph is examined, and a leaf subprogram is selected for processing. Because this subprogram is a leaf, the input/output search procedures described above are immediately usable.

The local variables of the subprogram are analysed first. An error message is generated for all local variables which are found to be strict input for the subprogram, since this situation implies a type 1 anomaly is certain. Correspondingly, local variables found to be of type input cause the generation of a warning message. The last usage of all local variables is also determined by means of an algorithm which is similar to the output category classification algorithm, but searches only as deeply as the last usage of the variable. If a local variable is used last as an output, a type 2 anomaly is present and a warning is issued.

The input/output classifications of the non-local variables of the subprogram are then determined. These classifications are printed, and also stored in the subprogramwide table of the subprogram under study. Warning messages are also printed for all dummy arguments which are found to be non-input and non-output. This table is then copied into a master data base, so that all invoking program units will be able to easily access the data needed to classify the input/output categories of variables used as actual arguments in invocations of this subprogram

The system makes a special check of the usage of all DO-loop index variables following satisfaction of their DO's. If the first use of a DO index following DO satisfaction is input or strict input, a type 1 anomaly is indicated and a warning or error message is produced. These situations are detected by initiating an input category determination trace for the DO index where the trace is begun with the flow graph edge which represents the DO satisfaction branch.

The analysis of a non-leaf program unit is more complicated. Such a program unit will, of course, not be analysed until all subprograms which it calls have been analysed. Then DAVE can fill in all entries which had been left blank during the creation of the calling unit's statement table. Certain Fortran errors are detected as this proceeds. For example, mismatches between either the types or numbers of actual arguments in an invocation and the members of the corresponding dummy argument list are detected here. The use of an expression or function name as an argument to a subprogram whose corresponding dummy argument is either an output or strict output variable is also detected here.

Illegal side effects, as defined by the ANSI Standard[10] (Section 8), are also detected. It is easily seen that an illegal side effect is certain to occur if a single variable is used within a single

statement, but not within the same function invocation, once as a strict input variable, and a second time as a strict output variable (other than on the left of an assignment statement). For example, in the statement X = F1(A, B) + F2(A) an illegal side effect occurs if the first dummy argument of F1 is classified strict input, and the dummy argument of F2 is classified as strict output (or vice versa). DAVE prints an error message in such cases. A warning message is issued if either classification is non-strict.

DAVE also exposes concealed data flows through subprogram invocations. Concealed data flows result from the use of COMMON variables as inputs (or outputs) to (from) an invoked subprogram. Such situations are easily exposed by examination of the COMMON block variable lists in the subprogramwide table of the invoked subprogram. Because data flows through such COMMON variables just as surely as through explicitly referenced parameters, the statement table entry of such an invocation statement is augmented by the input/output classifications of such variables. This assures that the results of global input/output category determination within the invoking program unit will be correct for these variables. DAVE can also print the names (those by which they are referenced in the invoked subprogram) and usages of all the variables which are used as inputs or outputs for a statement but are not explicitly referenced. This information seems useful as a form of automated documentation. It also seems to be useful as a debugging aid in that it alerts a programmer to data flows which are hidden, perhaps forgotten, and hence more prone to error.

The omission of a COMMON block declaration in an invoking program unit presents a difficult problem. If the COMMON block is referenced in the invoked subprogram, then the variables named in the COMMON block may or may not become undefined upon return to the calling program unit. Undefinition will not occur provided that the COMMON block is defined in some program unit currently invoking the program unit which omits the COMMON declaration. In the absence of such a reference by a higher level program unit, errors are possible. In particular, variables in such a COMMON block which are strict output or output from the invoked subprogram will become undefined—a type 2 anomaly—and a warning is issued. Variables in such a COMMON block which are strict input or input can receive values only through BLOCK DATA subprograms. Hence a check of the subprogramwide tables of such subprograms is made. If no data initialization is found, a warning is issued.

If a COMMON block, B, is declared by a high level program unit which invokes a subprogram, S, in which the block is not declared, then the ANSI Standard[10] (Section 10.2.5) specifies that B must still be regarded as implicitly defined in S provided that some subprogram directly or indirectly invoked by S does declare B. Hence data referenced by the variables in B may flow freely through routines which do not even make reference to B. As already observed, such data flows are noted and monitored by DAVE. In addition, DAVE is capable of printing the names and descriptions of all COMMON blocks whose declarations are implicit in a given subprogram. This, too, seems to be useful program documentation. The algorithm for determining which blocks are implicitly defined in which routines involves a preliminary leafs-up pass through the program call graph and then a final root-to-leafs pass.

After all of the above described checking and insertion of input/output data into the statement table has been done, DAVE proceeds with the analysis of the variables, explicit and implicit, local and non-local, as described in the case of a leaf subprogram. The algorithms used here are generalizations of those described in the previous section. Allowance is made in these algorithms for the possibility that variables may be input (but not strict input) or output (but not strict output) to or from a statement.

Subprograms are processed in this way until the main program is reached. Processing of non-COMMON variables in the main program is the same as the processing of such variables in any non-leaf, but COMMON variables must be treated differently. Any COMMON variable which has an input or strict input classification for the main program must be initialized in a BLOCK DATA subprogram. If not, a warning message (if the classification is input) or an error message (if the classification is strict input) is issued. Similarly, if a COMMON variable's last use was as an output from a main program a warning message is issued.

As implied by the foregoing discussion the messages issued by DAVE are divided into three categories: error, warning and general information. An error message is issued whenever DAVE is certain that a type 1 anomaly is present on an execution path. A warning message is issued whenever a type 1 anomaly might be present on an execution path. In particular, if a variable is strict input for some statement and is not defined on all paths leading to the statement an error message is issued, but if there might be at least one path on which the variable is defined then a warning message is issued. A warning message is issued if a type 2 anomaly is detected, but error messages are never issued for type 2 anomalies. The most common type of general information message which is issued consists of an input/output classification of all variables in a program unit.

LIMITATIONS, EXPERIENCE AND FUTURE PLANS

DAVE is written in ANSI Fortran except for a small number of non-ANSI, or machine dependent subprograms. It contains approximately 25,000 source statements, and executes in four overlaid phases, the largest of which requires approximately 50,000 decimal words of central memory on the CDC 6400. Analysis of a program consisting of approximately 2,000 source statements can be expected to require several minutes of CPU time on the CDC 6400.

DAVE is designed to allow the analysis of programs consisting of large numbers of program units of modest size. After each program unit is processed during the data base creation phase of DAVE, both the source text and partially completed data base are written out to a mass storage file. A copy of the subprogram's subprogramwide table is retained in a master data base which remains central memory resident. In the next processing phase, the data bases are randomly accessed from mass storage in the leafs-up order dictated by this phase. The recalled data base and central memory master data base are all that are necessary for the final analysis of the recalled subprogram. After this analysis, the completed data base is again written out to mass storage. Thus no more than the master data base and one subprogram data base need be central memory resident at any time, enabling DAVE to process a program consisting of a large number of program units. This is because the master data base entry for a subprogram is usually small. On the other hand, DAVE is more limited in analysis of a single large program. With a 6,500 word data base area on the CDC 6400, subprograms consisting of up to approximately 200 source statements can be analysed. This maximum program unit size is determined by array sizes which are easily changed, and we have observed that the cost of running DAVE does not change noticeably as these are altered.

The current version of DAVE is an experimental prototype. Some inefficient structures and algorithms have been employed in order to gain flexibility. It is expected that a future version of DAVE, redesigned for efficiency, will be capable of analysing larger program

units in a smaller data base area, and will execute faster. We now feel that we understand the kinds of data structures that are required so they can be frozen, losing flexibility but gaining speed and space. We also see a way[18] to use faster algorithms developed for global optimization.

We have characterized DAVE by calling it a validation system because it can determine the presence or absence of type 1 and 2 data flow anomalies. It might also be regarded as a debugging tool since it exposes bugs and as a documentation tool since it provides information about the input/output use of variables. We do not regard these matters of terminology as being important. Time and experience will determine the utility and appropriate characterization for such systems.

We have had some experience in using DAVE. One of the first programs analysed by DAVE was found to have a misspelling error despite the fact that the program had been in use for several months and was thought to be error free. DAVE detected that the misspelled variable was referenceable before definition along all paths, and that the correctly spelled variable was, on some paths, defined but not subsequently referenced (because the succeeding reference was misspelled). Another program comprising part of a Master's thesis in Computer Science was analysed by DAVE and found to contain a variety of errors. Most were ANSI Standard violations (e.g. mismatched actual argument and formal argument list lengths, use of exhausted DO indices and failure to reference COMMON blocks, either directly or indirectly, in program units which invoked subprograms attempting to share the COMMON variables) and did not prevent the program from running correctly on the permissive compiler used in developing the program. Such errors might be obstacles in transferring the program to another compiler. At least one such error, referencing an exhausted DO-loop index, is not to our knowledge infallibly detected by any other diagnostic system.

In the experience we have had using DAVE we find that interesting and useful messages are produced, but many uninteresting messages are also produced. We expect to focus future effort on the problem of reducing unwanted output or at least permitting the user to suppress or de-emphasize certain specifiable classes of diagnostic output.

Some of DAVE's current limitations are due to fundamental problems of decidability. In particular, the question of whether an arbitrarily selected path is executable is not decidable. Thus we can never be sure, in general, that anomalies detected by DAVE are significant in the sense that they lie on executable paths. However, one should not be too discouraged by this. We have already shown one situation where it can be determined that an anomaly is present on an executable path, and there are clearly others in the case of Fortran programs. Furthermore, recent results by Clarke[19] suggest a technique that may prove useful for dealing with this path separation problem. Closely related to this is the problem of proper identification of array elements when the subscript is a variable. Since it is not decidable whether an arbitrarily chosen variable has a particular value at an arbitrary point in the execution it follows that one cannot in general know which array element is being referenced or defined in a statement. Again, there are many situations in Fortran where one can resolve this identification problem with some effort. The present DAVE system treats all elements of the same array as a single variable. Slightly different are the problems which arise in attempting to analyse a large program by analysing its segments. The segmentation we use is natural for Fortran, our segments being the program units. The path information we pass across these boundaries is incomplete. The result is that we sometimes report that an anomaly is present only on some paths when in fact it is present on all paths.

Other limitations are not so fundamental. The restriction to ANSI Fortran can be easily overcome and a modification currently nearing completion will permit many non-ANSI constructions. The restriction to syntactically correct programs is not fundamental. It arose because of a desire to simplify the lexical analysis and it did not appear to us to be an important restriction considering the way in which we expected DAVE to be used. DAVE also cannot currently handle subprogram names passed as actual parameters. An algorithm for analysing such programs has been produced[20] but has not been incorporated into DAVE. Finally, DAVE does not detect all type 2 anomalies. When this project began we focused on type 1 anomalies and somewhat ignored the significance of type 2 anomalies. The present algorithms used in DAVE miss type 2 anomalies that arise on paths across subprogram boundaries. This will be corrected in later versions of DAVE.

While it is evident that the data flow analysis currently employed could be improved to provide sharper results, it is also evident that there are practical limits to improvements that depend only on more exhaustive analysis of the code. We believe that these practical limits as well as the theoretical limits mentioned earlier can be greatly relaxed by providing for the possibility of an interchange of allegations between the user and an analytic package such as DAVE. Others[21] have suggested interactive systems for proving programs correct, and of course interactive debugging systems are rather well-known.[22] The kind of interactive system we are suggesting here would lie somewhere in between; falling short of proof of correctness but being far more sophisticated than the usual debugging system. Moreover, such a system would provide a powerful documentation and testing tool.

We expect that the direction of our future work with DAVE will be moulded by continuing experience with the system, results of evaluation experiments currently in progress, as well as the growing literature on observed distributions of types of errors in programs.[23]

REFERENCES

1. B. Elspas, K. N. Levitt, R. J. Waldinger and A. Waksman, 'An assessment of techniques for proving program correctness', *ACM Computing Surveys*, **4**, 97–147 (1972).
2. R. Grishman, 'The debugging system AIDS', *AFIPS 1970 SJCC*, **36**, 59–64, AFIPS Press, Montvale, N.J.
3. R. E. Fairley, 'Introduction to the Interactive Semantic Modelling System', University of Colorado, Department of Computer Science, *No. CU–CS–038–74.*
4. J. D. Gannon and J. J. Horning, 'The impact of language design on the production of reliable software', *Proceedings of 1975 International Conference on Reliable Software*, IEEE Cat. No. 75CHO 940–7CSR, pp. 10–23, New York.
5. H. D. Mills, 'Top-down programming in large systems', in *Debugging Techniques in Large Systems* (Ed. R. Rustin), Prentice-Hall Inc., Englewood Cliffs, N.J., 1971, pp. 41–45.
6. N. Wirth, 'Program development by stepwise refinement', *CACM*, **14**, 221–227 (1974).
7. F. T. Baker, 'Chief programmer team management of production programming', *IBM Systems Journal*, **11**, 56–73 (1972).
8. D. J. Kuck *et al.*, 'Measurements of parallelism in ordinary FORTRAN programs', *Computer*, **11**, No. 1, 37–45 (1974).
9. F. E. Allen, 'A basis for program optimization', *Proceedings of IFIP Conference, 1971*, North Holland Publishing Company, Amsterdam, 1971, pp. 385–390.
10. American National Standards Institute, *FORTRAN*, ANSI X3.9 (1966).
11. *MNF Reference Manual for CDC 6000/7000/Cyber Series Computers* (Ed. M. Frisch and L. A. Liddiard), University of Minnesota Univ. Computer Center, Revision 3, November 1974, Minneapolis, Minn.
12. P. Cress, P. Dirksen and J. W. Graham, *Fortran IV with WATFOR and WATFIV*, Prentice-Hall Inc., Englewood Cliffs, N.J., (1970).

13. *Fortran IV Reference Manual 3102.01A, Mark III Foreground* (March 1973), General Electric, Information Services Business Division.
14. L. Osterweil and L. D. Fosdick, 'DAVE—A Validation, Error Detection and Documentation System for Fortran Programs', University of Colorado, Department of Computer Science, *TR No. CU-CS-071-75.*
15. F. Harary, *Graph Theory*, Addison-Wesley, Reading, Mass., 1969.
16. B. G. Ryder, 'The PFORT verifier', *Software–Practice and Experience*, **4**, 359–378 (1974).
17. L. Osterweil, L. Clarke and D. W. Smith, 'A Fortran System for Flexible Creation and Accessing of Data Bases', University of Colorado, Department of Computer Science, *TR No. CU-CS-057-74.*
18. L. D. Fosdick and L. Osterweil, 'Validation and global optimization of programs', *Fourth Texas Conference on Computing Systems (1975).*
19. L. Clarke, 'A System to Generate Test Data and Symbolically Execute Programs', University of Colorado, Department of Computer Science, *TR No. CU-CS-060-75.*
20. V. Kallal, *An Algorithm for Constructing the Flowgraph of an Assembly Language Program*, Master's thesis, University of Colorado, Department of Computer Science (to appear).
21. D. I. Good, R. L. London and W. W. Bledsoe, 'An interactive program verification system', *Proceedings of 1975 International Conference on Reliable Software*, IEEE Cat. No. 75CHO 940-7CSR, pp. 482–492, New York.
22. R. M. Balzer, 'EXDAMS: extensible debugging and monitoring system', *AFIPS 1969 SJCC*, **34**, 567–580, AFIPS Press, Montvale, N.J.
23. R. Rubey, 'Quantitative aspects of software validation', *Proceedings of 1975 International Conference on Reliable Software*, IEEE Cat. No. 75CHO 940-7CSR, pp. 246–251, New York.