Data Flow Analysis as an Aid in Documentation, Assertion Generation, Validation, and Error Detection *

Leon J. Osterweil
Lloyd D. Fosdick

CU-CS-055-74

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* Supported in part by NSF Grant GJ-36461.

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR(S) AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE AGENCIES NAMED IN THE ACKNOWLEDGMENTS SECTION.

Data Flow Analysis as an Aid in Documentation, Assertion Generation, Validation, and Error Detection*

by

Leon J. Osterweil
and
Lloyd D. Fosdick

Department of Computer Science
University of Colorado
Boulder, Colorado 80302

Report #CU-CS-055-74 September 1974

Key Words: program testing, data flow analysis, software validation, automated documentation, debugging

* Supported in part by NSF Grant GJ-36461.

ABSTRACT

Data flow analysis has already been applied to problems in code optimization and hardware design. It now appears that data flow analysis can also be a useful tool in validation and verification of software. By examining the data flow in a program much can be learned about its quality and correctness, for the pattern of data accesses and data assignments within a program is often diagnostic of sloppy or incorrect code. In an earlier paper by the authors, an algorithm for analyzing data flows in programs was presented. The problems of applying that algorithm to ANSI Fortran programs are described here. It is shown that Fortran programming errors and abuses such as the use of uninitialized variables, the generation of unneeded values, improper subprogram invocations and the creation of illegal side effects are all examples of data flow anomalies and can all be detected through the use of a system based upon this algorithm. The implementation of such a system, called DAVE, has been carried out and is described here. Results of applying this analytic system are presented.

Key Words: program testing, data flow analysis, software validation, automated documentation, debugging

Introduction

Recently much work has been focused on the analysis of computer programs. Active research is being carried out in such areas as proof of program correctness, detection of program errors, symbolic execution of programs, design of error-resistant languages and coding techniques, and automatic generation of assertions and documentation. The success of these endeavors seems to us to depend upon the extent to which investigators are able to devise methods of modelling and understanding the behavior of programs.

We too have devoted considerable effort to the analysis of program behavior. We are convinced that the study of data flows within a program provides a key to an understanding of program behavior. By application of data flow analysis to programs we feel that we have made significant attacks on such problems as detection of semantic program errors, automated creation of assertions and automated production of program documentation.

In this paper we describe a data flow analysis system which has been built. The way in which our system tracks the flow of data from statement to statement, block to block, and subprogram to subprogram, is described, and we point out how anomalous data flow is detected by this system, observing that it is quite often symptomatic of program errors. We show how hidden data flows can be exposed, and data flow assertions generated by the system. Finally, we observe that subtle--even invisible--data flows are all too easily created. We note that the difficulty of detection and analysis of such flows, both for analytic systems and human programmers, seems to increase the likelihood of error.

Our present work uses ANSI Fortran as the language of the subject program, however the ideas and algorithms we use have a much wider application. The analysis used in the verification deals with a data structure for variables and a directed graph for the control paths which could have been derived from a program written in a language other than Fortran. The implementation itself is written in ANSI Fortran and considerable attention has been given to making it portable. It is called DAVE (Documentation, Assertion generating, Validation, Error detection). Our work began in the spring of 1973 and at the time of this writing a limited version of the system is completed and being tested. This version contains about 8K source statements and occupies about 50K words of memory. Examples of some output from DAVE appear later in this paper. We have been assisted in this work by a number of part-time graduate research assistants, Richard Maguire, Lamar Ledbetter, Lori Clarke, David Smith, and a full-time computer scientist, Carol Miesse.

Definitions

We represent the flow of control in a program by a directed graph called the control flow graph. We use Γ to denote the successor operator; thus, Γx denotes the set of nodes joined to the node x by edges directed from x and $\Gamma^{-1}x$ denotes the set of nodes joined to the node x by edges directed toward x. The nodes in this graph are sequences of statements called basic blocks. A basic block is a maximal sequence of statements having the property that whenever any one of the statements in the basic block is executed, every statement in the basic block is executed. This coincides with the definition often used in code optimization ([1], p. 12).

We represent the linkages between program units by a directed graph called the calling graph. The calling graph of a program has as its nodes the program's constituent program units, and has an edge from node A to node B if the subprogram represented by B is invoked from the program unit represented by A.

The control flow graph is assumed to have one entry point (a node with indegree zero) and one or more exit points (nodes with outdegree zero). A control path is a path in this graph; i.e. a list of nodes $x_{i_1}, x_{i_2}, \ldots,$ where $x_{i_{n+1}} \varepsilon \Gamma x_{i_n}$. Not every control path is a sequence of basic blocks which could be executed. An execution path is a control path which could be executed.

The role that a variable, V, plays in the data flow for execution of a statement, a basic block, and a subprogram is identified by assigning an input-output classification to it for each of these structures. In a statement such as

$$A = B + C$$

the variables B and C are referenced to define a value for A. To identify this role of the variables B and C we say that B and C are strict input variables for this statement and we say that A is a strict output variable for this statement. In a statement a variable may be strict input and strict output; this is the case for X in the statement

$$X = X + Y.$$

For completeness we classify the input role and the output role of a variable in a statement. In the first statement above A is non-input and strict output (NI, SO), while B and C are non-output and strict input (SI, NO). This classification is extended to basic blocks and subprograms. Thus a basic block or subprogram strict input variable is one which is referenced by the block or subprogram before all definitions of the variable. A strict output is one which is defined for all control paths within the block or subprogram. Since there is usually more than one possible sequence of statements which may be executed in a subprogram the modifier strict is used only when the indicated role applies for every control path. Consider, for example, the subprogram in Fig. 1. Focusing attention on the formal parameters only, the classification of these variables for the subprogram would be: A - (SI,NO) ; SUM - (NI,O) ; N - (I,NO) ; FLAG - (NI,SO).

It should be noted that the input-output classification for a variable in a statement does not always contain the modifier strict. For example, in the statement

```
CALL SERIES(X,Y,K,L)
```

analysis of the subroutine SERIES would produce the classification: X - (SI,NO) ; Y - (NI,O) ; K - (I,NO) ; L - (NI - SO).

When a DO in Fortran is satisfied the index becomes undefined. This situation is recognized by assigning a special output class, undefined (U). For example, in the statement

```
READ (5,100)(X(K),K=1,5)
```

we have the classification: X - (NI,SO) ; K - (NI,U). This example illustrates another aspect of the classification, namely a single classification is used for all of the elements in an array. This has obvious weaknesses which we will discuss later.

```
      SUBROUTINE SERIES (A,SUM,N, FLAG)
      LOGICAL FLAG
      IF(A.GE.0..AND. A.LT.1.0) GO TO 10
      FLAG=.FALSE.
      RETURN
10    SUM=0
      DO 20 K=1,N
      X=FLOAT(K)
      SUM=SUM + 1.0/SQRT(A+X)
20    CONTINUE
      FLAG=.TRUE.
      RETURN
      END
```

Figure 1: Example for classification of global parameters

Data Flow Anomalies Detected

Execution of a program involves input of data, the generation and use of intermediate results, and output of data. This process takes place as program control proceeds along an execution path. We are concerned with the determination that the flow of data from input to output during this process is consistent and meaningful. Accordingly, we focus special attention on two types of events in the data flow:

Event 1: Referring to a variable which has not been assigned a value on a path leading to this reference.

Event 2: Assigning a value to a variable which is not referenced on a path leading from this assignment.

The presence of either of these events on an execution path is an _anomaly_ in the data flow, and is symptomatic of an error. An anomaly associated with event 1 is called a _type 1 anomaly_ and an anomaly associated with event 2 is called a _type 2 anomaly_. A type 1 anomaly violates the principle that a value must flow into a variable before it can flow out, and a type 2 anomaly violates the principle that data which flows into a variable should flow out. The viewpoint here is that there is a _conservation principle_ to be applied to the data flow: it should be free of sources and sinks, excepting data boundary points (READ's and WRITE's) and violation of this principle is likely to be symptomatic of errors in the program. It is evident that an observed violation of the conservation principle may be traceable to any of a wide variety of common programming errors: keypunch error, misspelling, statements out of order, failure to initialize, incorrect label, incorrect use of parameters in a subprogram reference, abuse of concealed data flows through subprogram invocations, etc.

It is to be carefully noted that we have defined anomalous data flow in terms of events which take place along an execution path of the program. The recognition of execution paths is exceedingly difficult, in fact for arbitrary programs this problem is not decidable. We look therefore at the control paths, which of course contain the execution paths as a subset. Detection of either event on a control path may mean that an anomaly is present but does not assure it. However, if an event at some statement S is observed for all control paths leading to S, in the case of type 1, or for all control paths leading from S in the case of type 2, then one can be certain that the event is present on an execution path and certainty of an anomaly is established.

DAVE issues messages where the presence of data flow anomalies is detected or suspected. These messages are in the form of warnings and errors. We have put into the category of errors all those situations which are certain to yield an illegal computation, while warning messages are issued where only the possibility of an illegal computation is established. In particular, observation of event 1 on all control paths leading to a statement will cause an error message to be issued, while a warning message is issued if the event is present on some, but not all, control paths. We have decided to issue warning messages when an event 2 is detected regardless of whether or not it is present on all control paths. The reason for this is that event 2 does not seem to imply an erroneous computation in the same way event 1 does. An event 2 might just mean a superfluous computation has taken place.

When one considers the complexity of the data flow which exists in a program consisting of a number of subprograms, all using such features as COMMON and EQUIVALENCE declarations, it is evident that the detection of all events is a difficult matter. The present implementation misses a small class of events 1 and 2 which will be discussed later.

To illustrate these ideas and some of the difficulties, we look at some examples. Consider first the input-output classification of local variables for the statements in a subprogram. If on one of the control paths from the entry node, the first use of a local variable is classified (SI,•) or (I,•), then we have an instance of event 1. This is the case for the variable K in Fig. 2 and Fig. 3. Moreover a type 1 anomaly is certain in Fig. 2. In Fig. 3, however, a type 1 event occurs only along one of the two control paths. In this case DAVE issues a warning of a possible type 1 anomaly. This is done because DAVE does not attempt to determine which control paths are actually execution paths. Hence, allowing for the possibility that the path on which the event occurs is not an execution path, DAVE *warns* that a type 1 anomaly is possible, but not certain. We observe that if we make the not unreasonable assumption that every block of a program is executable, then it is possible to devise an analytic procedure which would find both control paths of Fig. 3 to be execution paths. In this case an error message - not a warning - could be generated. In DAVE, however, we have not made that assumption and have not produced the needed analytic procedures. If on one of the control paths to an exit node, the last use of a local variable is classified (•, SO) or (•, O) then we have an instance of event 2. This is the case for the variable L in Fig. 2 and Fig. 3. Again we have a certain anomaly (type 2) in Fig. 2, and we have an event (type 2) only on one of the two control paths in Fig. 3. For the same reasons as discussed above DAVE can be certain of a type 2 anomaly in Fig. 2 and

```
SUBROUTINE X(I,J)
J=2*K+I
L=I**2
RETURN
END
```

Figure 2. Type 1 and type 2 anomalies on all control paths

```
SUBROUTINE X1(I,J)
IF(I.LT.0) RETURN
J=2*K+I
L=I**2
RETURN
END
```

Figure 3. Type 1 and type 2 anomalies on some control paths.

can only suspect the possibility of a type 2 anomaly in Fig. 3. As noted earlier, a warning message is produced in either case.

Type 1 and type 2 anomalies arising in the data flow between program units are more difficult to detect. In Fortran this flow takes place through argument lists and common lists. Consider once again the subprogram SERIES in Fig. 1. Recall that A was classified as (SI, NO). If there is no definition of the actual parameter, A', used in place of A on any path leading to an invocation of SERIES then a type 1 anomaly is present. DAVE will detect this situation and issue an error message. If there are some paths (but not all) for which A' is not defined then a type 1 event is present, a type 1 anomaly may be present, and DAVE issues a warning. The parameter N was classified (I, NO): if there are any control paths leading to an invocation of SERIES on which N' is not defined a warning is issued. Variables SUM and FLAG were classified (NI, O) and (NI, SO), respectively. If the last use of SUM' or FLAG' on a path beginning with the invocation and going to a stop node was a definition then a warning is issued.

The Structure of the Analysis Program

The structure of DAVE is indicated in Fig. 4. The subject program, consisting of a main program and all subprograms referenced either directly or indirectly is first preprocessed by an adapted version of BRNANL, a program analysis and instrumentation package [2]. It is assumed that the subject program is a syntactically correct ANSI Fortran program, however as noted below recovery procedures are possible when illegal statements are encountered.

During this pass, the program is divided into program units and these are divided into basic blocks and statements. Statement type determination is also made here. The preprocessed program is then passed to a lexical analysis routine. This routine creates a token list to represent each of the program's source statements. Clearly knowledge of the statement type makes the job of the token list generator easier.

As the token lists are created, comprehensive data bases of information about the various program units are also created. The data bases are accessed using a data base creation and accessing package, designed to facilitate data base restructuring [3]. Each subprogram data base contains a symbol table, label table, statement table, and table of subprogramwide data. The symbol and label tables contain much the same type 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 primary purpose of the statement table is to hold the input-output classification for every variable referenced or defined in each executable or DATA statement. During this lexical scan phase it is possible to determine the input-output classes of all variable references

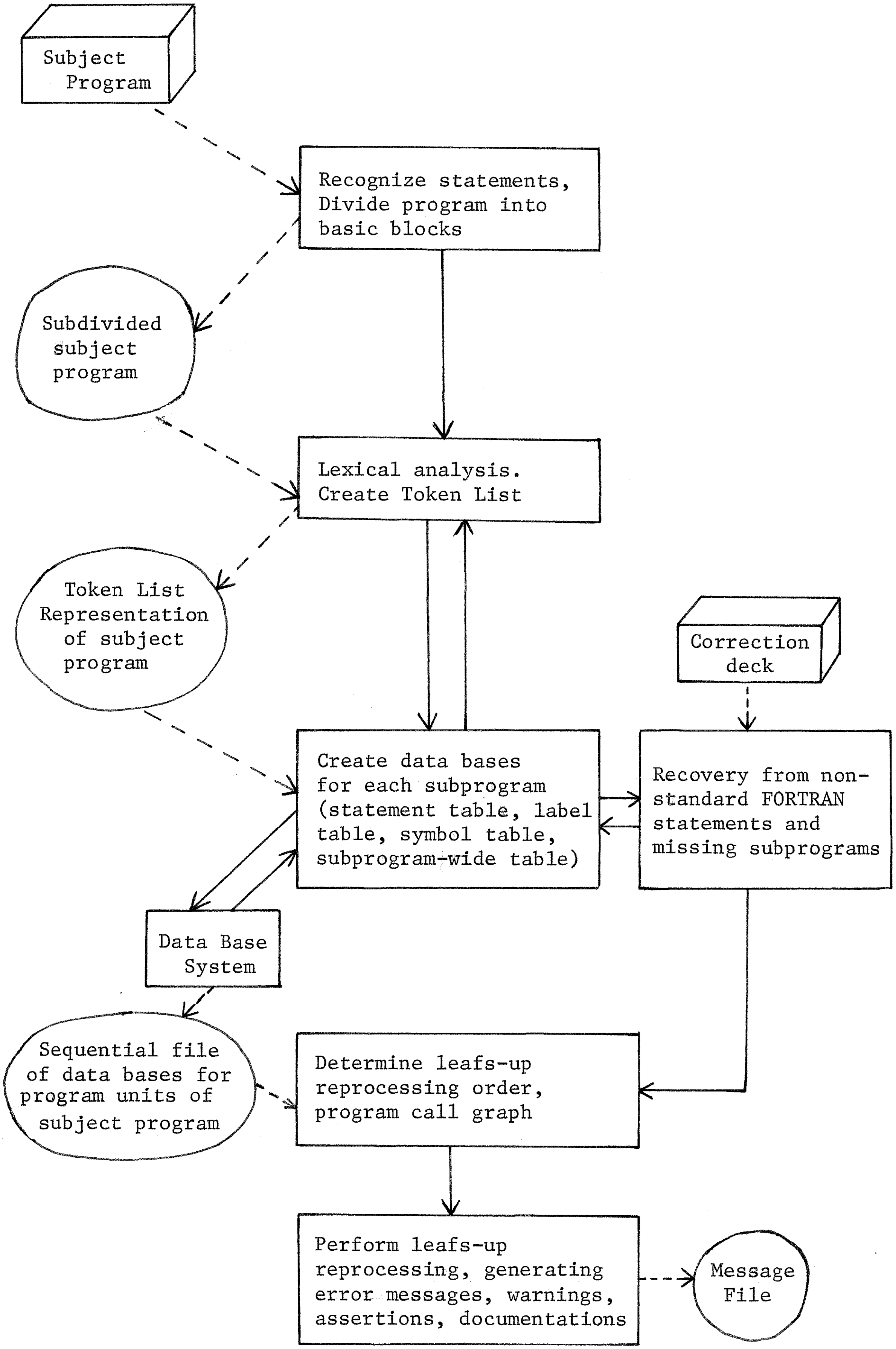

Subject Program
Recognize statements, Divide program into basic blocks
Subdivided subject program
Lexical analysis. Create Token List
Token List Representation of subject program
Correction deck
Create data bases for each subprogram (statement table, label table, symbol table, subprogram-wide table)
Recovery from non-standard FORTRAN statements and missing subprograms
Data Base System
Sequential file of data bases for program units of subject program
Determine leafs-up reprocessing order, program call graph
Perform leafs-up reprocessing, generating error messages, warnings, assertions, documentations
Message File

Figure 4. Structure of DAVE.

and definitions except those in which variables are used as arguments to subprogram invocations. DAVE contains the input-output classifications for ANSI Fortran intrinsic functions and basic external functions so the input-output classes of variables used as arguments in these functions are also determined during this phase. This determinable input-output data is stored in the statement table. Blanks are placed in the statement table for the input-output classifications of variables used as actual parameters in subprogram (excepting ANSI functions) invocations; these blanks will be filled in during a later phase of processing.

The table of subprogramwide data for a given program unit contains a list of all subprograms referenced by the program unit, as well as representations of all non-local variable lists; i.e. the program unit's parameter list and COMMON block lists. Ultimately the non-local variable lists will be used to hold data about the subprogramwide input-output behavior of these non-local variables. The external reference lists of the various program units will be used to construct the program call graph.

During this phase of processing statements which are syntactically illegal under the ANSI standard may be encountered. Our system is capable of pausing at this point and accepting a correction deck containing replacements for the offending statements. In addition, our system will examine the external reference lists to determine whether all referenced subprograms have been submitted. If not, DAVE will, at this time, also accept new symbolic decks in order to satisfy such unsatisfied external references.

In the next phase, DAVE builds and examines the program call graph. Using the call graph, leaf subprograms (those with no external references) are identified. For such subprograms, the input-output classifications of all variable references are already known, hence the input-output classifications of all non-local variables can be made [4]. Hence the input-output behavior of all variables used in all invocations of such subprograms can now be filled in, enabling in turn determinations of input-output classifications of non-local variables in other subprograms. Using this scheme, all input-output classifications can eventually be entered for all variables in all statement table entries in all program unit data bases. This leafs-up subprogram reprocessing order (also referred to as inverse invocation order by Allen [5]) is determined in the next phase through analysis of the program call graph.

The final phase of processing is the most interesting. During this phase, the program units are reprocessed in the above-mentioned leafs-up order. Missing input-out-put information is supplied, and global data flow analysis is performed. It is at this time that events of types 1 and 2 in the data flow are identified, and data flow assertions are made.

The Data Flow Analysis Phase of Processing

As already noted, the final phase of processing begins with the analysis of leaf subprograms. The analysis begins with the construction of a basic block table for the subprogram. This table holds input-output information about all variables referenced in each of the basic blocks. It is constructed from data in the subprogram's statement table.

Once the basic block table has been constructed, the input-output classification of program variables can be determined through the use of the algorithms already described by us [4]. The local variables are analyzed 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 adapted output category classification algorithm. If a local variable is used last as an output, an event of type 2 is present and a warning is issued.

The input-output classifications of the non-local variables are then determined. These classifications are printed out, and also stored in the subprogramwide table of the subprogram under study. Warning messages are also printed for all parameters which are found to be non-input and non-output. Clearly each of these items of data in the subprogramwide table can be viewed as being an automatically generated assertion about the subprogram. These assertions are useful moreover in producing documentation about the subprogram. 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 arguments in invocations of this subprogram.

In addition the system makes a special check of the usage of all DO-loop index variables following satisfaction of their DO's. The ANSI standard specifies that the value of the DO index becomes undefined upon satisfaction of the DO (whether the DO is explicit or implicit). Hence if the first use of a DO index following DO satisfaction is input or strict input, a type 1 event is indicated and a warning 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 analyzed until all subprograms which it calls have been analyzed. At such a time, however, it is possible to fill in all entries which had to be left blank during the creation of the calling unit's statement table. Hence such blanks are filled in. Certain Fortran semantic errors are also detected as this proceeds. For example, a mismatch between the length of a calling sequence and the length of the corresponding parameter list is detected here. Likewise, at this point we detect the use of an expression or function name as an argument to a subprogram whose corresponding parameter is either an output or strict output variable. Both of these semantic errors can also be viewed as data flow anomalies.

A mismatch in parameter list lengths may be either a type 1 or type 2 anomaly or both; erroneous use of an expression or function name is an anomaly of type 2.

At this point in the processing illegal side effects, as defined by the ANSI standard ([6], 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 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

```
CALL SIDFKT (A+5.0, A, X)
```

where the first formal parameter in SIDFKT is classified as strict input and the second formal parameter is classified as strict output. A warning message is issued if either classification is non-strict.

Our system 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 out 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 to a statement but are not explicitly referenced. Such information seems most useful as a form of automated documentation. It also seems to be useful as a debugging aid in that it may alert 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 tricky 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 event - 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 [6], sec. 10) 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 out the names and descriptions of all COMMON blocks whose declarations are implicit in a given subprogram. This, too, seems to us 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. It is described in detail in [7].

Only after all of the above described checking and insertion of input-output data into the statement table has been done, does the system proceed to the creation of the basic block table. As might be expected the creation of the basic block table entry for a basic block containing subprogram invocations is rather complicated. The algorithm must contend with such problems as non-strict usage of variables, and references to variables not explicitly named. This algorithm also is described in detail in [7].

Once the basic block table is constructed, analysis of the variables, explicit and implicit, proceeds as described in the case of a leaf subprogram.

Subprograms are processed in this way until the main program is reached. Processing of the main program is the same as the processing of any non-leaf, except that 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 (class is input) or an error message (class 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.

Conclusions

We have described the workings of DAVE, a data flow analysis system, and have shown that this system can be used to detect errors, produce assertions, and generate documentation about a program. We feel that DAVE has demonstrated the usefulness of data flow analysis in studying programs, and regard this as a first effort rather than a finished product.

In DAVE, certain simplifying assumptions have been made, facilitating our analysis, but often weakening our results. For example, DAVE recognizes that variables may become undefined (e.g., upon satisfaction of a DO loop, the DO parameter becomes undefined), but considers undefined to be an output class. However in the subprogram of Fig. 5, I becomes undefined only on some control paths. It is strict output on the others. We choose to consider I to be output (not undefined) from subprogram SEARCH. Hence, even the possibility of a type 1 anomaly in subprogram INSERT (see Fig. 6) is undetected. This is because no input analysis of I is initiated starting after the invocation of SEARCH in subroutine INSERT. No such search is indicated because no variables in the invocation of SEARCH appear to have output class undefined. Choosing to consider I to be undefined in the above case leads to other undesirable imprecise analysis. Future revisions of DAVE will allow a variable to be classified as any of non-undefined, undefined or strict undefined independent of being classified as non-output, output or strict output.

It should also be noted that DAVE does not detect all events of type 2 in a program. DAVE's analysis focuses on the first and last uses of program variables, but does not examine the pattern of

```
      SUBROUTINE SEARCH(VECT,I)
      DIMENSION VECT(100)
      DO 10 I=1,100
      IF(VECT(I).EQ.0.0)RETURN
   10 CONTINUE
      RETURN
      END
```

Figure 5. A subprogram in which I becomes undefined only on some control paths and is output on others.

```
      SUBROUTINE INSERT(VECT,DATA)
      DIMENSION VECT(100)
      DO 10 J=1,100
   10 VECT(J)=J
      I=100
      CALL SEARCH(VECT,I)
      IF(I.NE.100)VECT(I)=DATA
      RETURN
      END
```

Figure 6. A subprogram expecting I to be a strict output from SEARCH

intermediate uses. Hence no message would be generated in response to a sequence of code like

```
 .
 .
 .
J=1
J=2
 .
 .
 .
```

even though there is a type 2 event along every control path containing this sequence. We acknowledge that it is sometimes not unreasonable to write code containing such type 2 events (see for example Fig. 7). For completeness, however, we feel that all such events should be detected. As already observed, an anomaly is to be regarded as a symptom of error. Hence the final determination of the meaningfulness and disposition of all messages produced by DAVE must be made by the user. The user should therefore have messages about all detectable anomalies.

We have already observed that we have simplified our work by making no attempt at determining which control paths through a program are execution paths. Consequently, our analysis of Figure 3 resulted in the detection of a type 1 event on one of two control paths. We reasoned that that path could be unexecutable, and hence issued only a warning. On the other hand, if we had made the assumption that every block of the program is executable, we would have been lead to the conclusion that the path must be an execution path. This follows from the observation that no other control path includes the last sequential block of the subprogram. Thus we could have been sure of the existence of a type 1 anomaly. As noted earlier, the discovery of all execution

```
      SUBROUTINE TEMPLT(A,B,C,D,E,F,SWITCH,X)
      LOGICAL SWITCH
      SWITCH=.FALSE.
      IF(X.GE.A.AND.X.LT.B)GO TO 10
      IF(X.GE.C.AND.X.LE.D)GO TO 10
      IF(X.LT.E.OR.X.GT.F)GO TO 20
   10 SWITCH=.TRUE.
   20 RETURN
      END
```

Figure 7. A not unreasonable subprogram which, nevertheless, displays a type 2 event on some control paths.

paths of a program is impossible. Yet this simple example illustrates that it is possible and worthwhile to identify some execution paths. The identification described here was based partially upon the assumption that all basic blocks of a subprogram are executable. This assumption may be questionable, but we have already made the (questionable?) assumption that all subprograms of a program are executable, for we perform analysis and produce messages about all subprograms. Probably the best solution to this dilemma is to produce a system which can accept assertions from the user about whether or not all subprograms and blocks are executable. The program analysis would then be guided by these assertions.

DAVE also makes a simplifying assumption about the use of subscripted variables. The analytic routines treat a subscripted variable in the same way as a simple variable. Thus any reference to or definition of a single element of a subscripted variable is tantamount to a reference to or definition of all elements. Because of this assumption differences in input-output behavior between elements of a subscripted variable will be lost, perhaps resulting in a blurred picture of the use of the variable. For example in Fig. 8 the array A is perhaps best thought of as consisting of two column vectors. The second column is used as a strict output non-input vector. The first column is used as a strict input non-output vector. In order for an invocation of TBLMKR to be free of anomalies, column 1 of A should be initialized before invocation of TBLMKR and column 2 of A should be referenced after invocation of TBLMKR. DAVE currently, however, would treat A as a simple variable, and assert that it is a strict input,strict output variable for TBLMKR. Hence, for example, DAVE would detect nothing even suspicious about the subprogram of Fig. 9.

Here too we observe it is impossible to always correctly analyze subscripted variable usage. In the case illustrated, however, the difference in usages of the two columns of A is detectable because the column subscript expression is always a constant. This does not strike us as being an unusual situation. Although it is probably asking too much to expect an analytic system to detect such situations, it seems reasonable to produce a system capable of sharpened analysis of subscripted variables based upon externally supplied assertions about such things as subarray autonomy.

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 can be greatly extended by providing for the possibility of an interchange of assertions between the user and an analytic package such as DAVE. Others [8] have suggested interactive systems for proving programs correct, and of course interactive debugging systems are reasonably well known [9]. The kind of interactive system we are suggesting here would lie somewhere in between; falling short of a 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.

```
      SUBROUTINE TBLMKR(A)
      DIMENSION A(100,2)
      DO 10 I=1,100
   10 A(I,2)=A(I,1)**2
      RETURN
      END
```

Figure 8. A subprogram in which different array elements have different input-output uses.

```
      SUBROUTINE SUMSQS(A,SUM)
      DIMENSION A(100,2)
      DO 10 I=1,100
   10 A(I,2)=I
      CALL TBLMKR(A)
      SUM=0.0
      DO 20 I=1,100
   20 SUM=SUM+A(I,1)
      RETURN
      END
```

Figure 9. An anomalous invocation of TBLMKR which is not detected by DAVE.

References

1. Schaeffer, Marvin. A Mathematical Theory of Global Program Optimization. Prentice-Hall (1973).

2. Fosdick, L. D. BRNANL, A Fortran program to identify basic blocks in Fortran programs. Report 40 (March 1974) Dept. of Computer Science, University of Colorado, Boulder, CO.

3. Osterweil, Leon; Clarke, Lori; and Smith, David. A Fortran system for flexible creation and accessing of data bases. Report 52 (August 1974) Dept. of Computer Science, University of Colorado, Boulder, CO.

4. Osterweil, Leon and Fosdick, L. D. Automated input/output variable classification as an aid to validation of Fortran programs. Report 37 (January 1974) Dept. of Computer Science, University of Colorado, Boulder, CO. (Presented at Software II, Purdue University, May 1974).

5. Allen, Francis E. Interprocedural data flow analysis. Report RC4633(#20545) (November 1973) Computer Sciences, IBM Research, Yorktown Heights, N. Y. (Presented at IFIP 1974).

6. Fortran, ANSI X3.9-1966. American National Standards Institute, Inc. 1430 Broadway, New York, N. Y. 10018.

7. Osterweil, Leon and Fosdick, L. D. DAVE, A comprehensive data flow analysis system for Fortran programs (in preparation).

8. Elspas, B.; Levitt, K. N.; Waldinger, R. I.; and Waksman, A. An assessment of techniques for proving program correctness. ACM Compt. Surv. 4,2 (June 1972), 97-147.

9. Rustin, Randall, ed. Debugging Techniques in Large Systems. Prentice-Hall (1971).

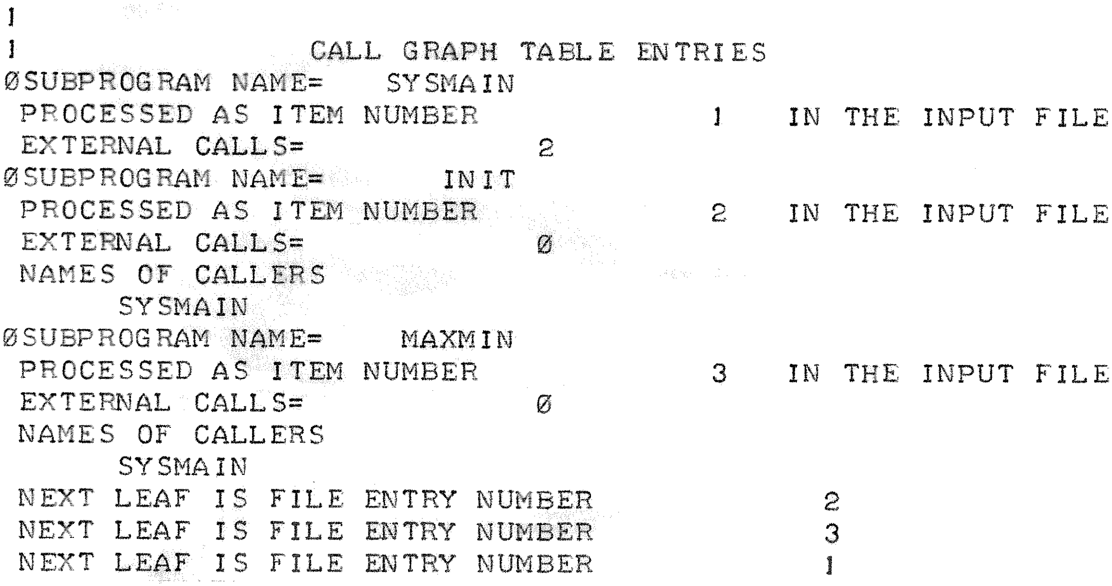
Appendix

At the time of this writing DAVE is being tested. We have constructed an example which illustrates some of the operating features and messages generated.

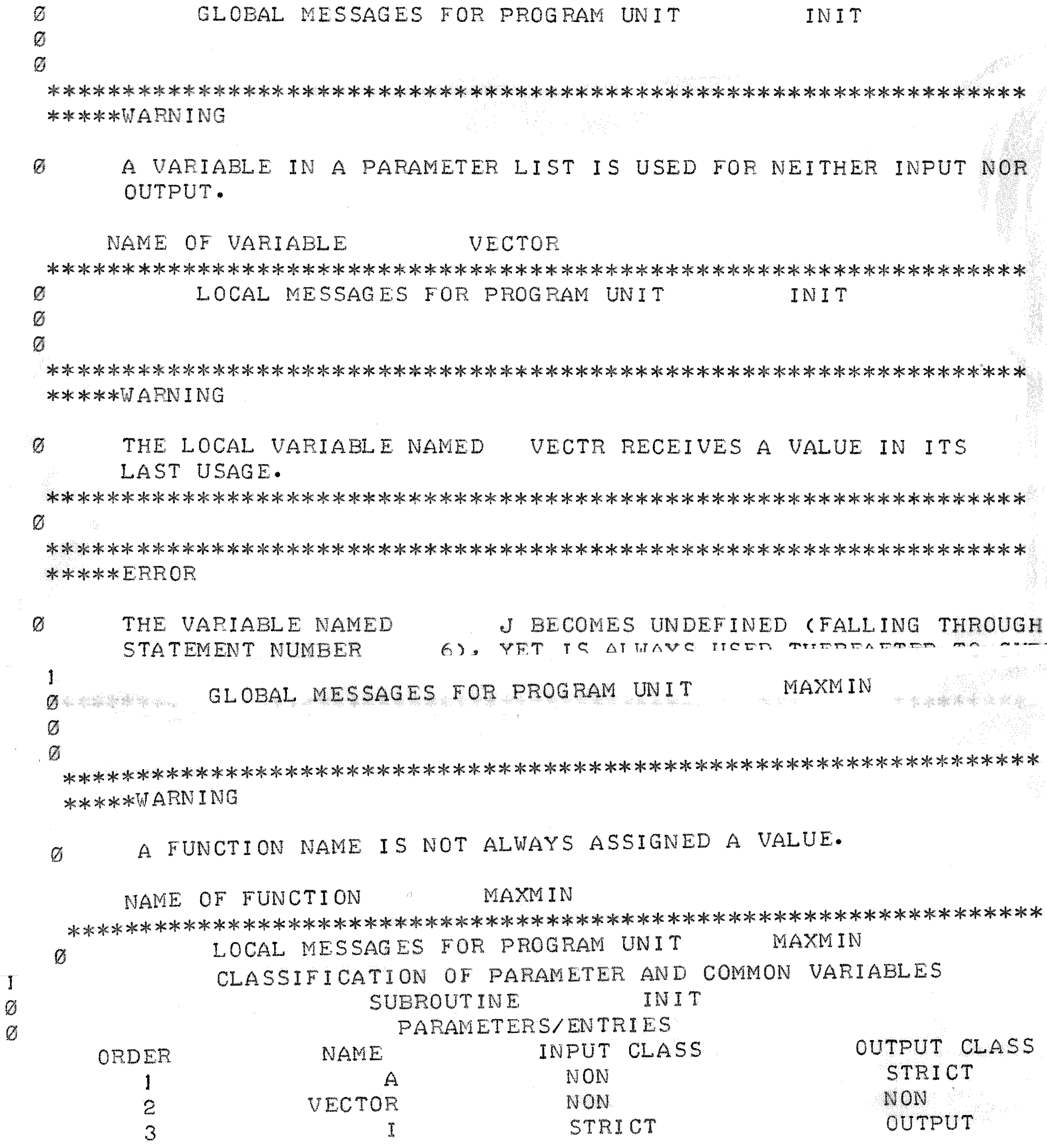
The first part of the material below is a copy of the program being analyzed with line numbers, statement numbers and block numbers shown at the left. This material is output by DAVE for reference purposes. The second part of the output is produced after the first phase of processing is done. In this part of the output the processing order of the calling graph is displayed. The third part of the output contains messages, warning statements, and error statements resulting from the data flow analysis described in this report.

```
 LINE  STMT  BLOCK        SOURCE

1
    1     1     Ø         COMMON/BLK/S,R,XMAX,XMIN
    2     2     Ø         DIMENSION Q(1ØØ), R(1ØØ,2)
    3     3     1         READ(5,1Ø)I,S
    4     4     Ø     1Ø  FORMAT(I1Ø,F6.2)
    5     5     1         CALL INIT(R,Q,I)
    6     6     1         WRITE(6,2Ø)(Q(J),J=1,1ØØ)
    7     7     Ø     2Ø  FORMAT(1ØF8.2)
    8     8     1         INTS=S
    9     9     1         IF(INTS.LE.Ø)
    9    1Ø     2        $INS=1
   1Ø    11     3         M=MAXMIN(R)*INTS
   11    12     3         STOP
   12    13     Ø         END
1
   13     1     Ø         SUBROUTINE INIT(A,VECTOR,I)
   14     2     Ø         DIMENSION A(1ØØ),VECTR(1ØØ)
   15     3     1         IF(I.LT.1.OR.I.GT.98)
   15     4     2        $I=1
   16     5     3         DO1Ø J=1,I
   17     6     4     1Ø  A(J)=J*J-1Ø*(J*J/1Ø)
   18     7     5         IP1=I+1
   19     8     5         IF(I.GT.99)
   19     9     6        $I=99
   2Ø    1Ø     7         DO2Ø K=IP1,1ØØ
   21    11     8     2Ø  A(J)=Ø
   22    12     9         READ(5,3Ø)(VECTR(J),J=1,1ØØ)
   23    13     Ø     3Ø  FORMAT(1ØF8.2)
   24    14     9         RETURN
   25    15     Ø         END
1
   26     1     Ø         FUNCTION MAXMIN(R)
   27     2     Ø         DIMENSION R(1ØØ)
   28     3     Ø         COMMON/BLK/RMAX,RMIN,DUMMY(2Ø1)
   29     4     1         RMAX=R(1)
   3Ø     5     1         RMIN=R(1)
   31     6     1         DO 1Ø I = 1,1ØØ
   32     7     2         IF(RMAX.LT.R(I))
   32     8     3        $RMAX=R(I)
   33     9     4         IF(RMIN.GT.R(I))
   33    1Ø     5        $RMIN=R(I)
   34    11     6     1Ø  CONTINUE
   35    12     7         IF(RMAX.NE.RMIN)
   35    13     8        $MAXMIN=RMAX-RMIN
   36    14     9         RETURN
   37    15     Ø         END
```



```
1
1               CALL GRAPH TABLE ENTRIES
ØSUBPROGRAM NAME=   SYSMAIN
 PROCESSED AS ITEM NUMBER             1   IN THE INPUT FILE
 EXTERNAL CALLS=            2
ØSUBPROGRAM NAME=      INIT
 PROCESSED AS ITEM NUMBER             2   IN THE INPUT FILE
 EXTERNAL CALLS=            Ø
 NAMES OF CALLERS
      SYSMAIN
ØSUBPROGRAM NAME=    MAXMIN
 PROCESSED AS ITEM NUMBER             3   IN THE INPUT FILE
 EXTERNAL CALLS=            Ø
 NAMES OF CALLERS
      SYSMAIN
 NEXT LEAF IS FILE ENTRY NUMBER             2
 NEXT LEAF IS FILE ENTRY NUMBER             3
 NEXT LEAF IS FILE ENTRY NUMBER             1
```



```
Ø          GLOBAL MESSAGES FOR PROGRAM UNIT        INIT
Ø
Ø
 **************************************************************
 *****WARNING

Ø     A VARIABLE IN A PARAMETER LIST IS USED FOR NEITHER INPUT NOR
      OUTPUT.

    NAME OF VARIABLE        VECTOR
 **************************************************************
Ø          LOCAL MESSAGES FOR PROGRAM UNIT        INIT
Ø
Ø
 **************************************************************
 *****WARNING

Ø     THE LOCAL VARIABLE NAMED   VECTR RECEIVES A VALUE IN ITS
      LAST USAGE.
 **************************************************************
Ø
 **************************************************************
 *****ERROR

Ø     THE VARIABLE NAMED      J BECOMES UNDEFINED (FALLING THROUGH
      STATEMENT NUMBER      6), YET IS ALWAYS USED THEREAFTER TO
1
Ø          GLOBAL MESSAGES FOR PROGRAM UNIT      MAXMIN
Ø
Ø
 **************************************************************
 *****WARNING

Ø     A FUNCTION NAME IS NOT ALWAYS ASSIGNED A VALUE.

    NAME OF FUNCTION        MAXMIN
 **************************************************************
Ø          LOCAL MESSAGES FOR PROGRAM UNIT       MAXMIN
           CLASSIFICATION OF PARAMETER AND COMMON VARIABLES
I                     SUBROUTINE        INIT
Ø                      PARAMETERS/ENTRIES
Ø
    ORDER         NAME        INPUT CLASS          OUTPUT CLASS
      1              A         NON                   STRICT
      2         VECTOR         NON                   NON
      3              I         STRICT                OUTPUT
```

```
Ø
1             CLASSIFICATION OF PARAMETER AND COMMON VARIABLES
Ø                       FUNCTION        MAXMIN
Ø                         PARAMETERS/ENTRIES
     ORDER             NAME          INPUT CLASS           OUTPUT CLASS
       1              MAXMIN          NON                   OUTPUT
       2                   R          STRICT                NON
Ø                       COMMON                BLK
Ø                         PARAMETERS/ENTRIES
     ORDER             NAME          INPUT CLASS           OUTPUT CLASS
       1                RMAX          NON                   STRICT
       2                RMIN          NON                   STRICT
       3               DUMMY          NON                   NON
```

```
1
Ø          GLOBAL MESSAGES FOR PROGRAM UNIT
Ø
Ø
 ****************************************************************
 *****WARNING        STATEMENT NO.    5  BASIC BLOCK NO.    1

Ø     CORRESPONDING ARGUMENTS IN THE PARAMETER LISTS ARE OF
      DIFFERENT DIMENSIONALITY.

                     CALLING SUBPROGRAM  CALLED SUBPROGRAM
                            SYSMAIN               INIT
    ARGUMENT POSITION          1                  1
 NUMBER OF DIMENSIONS          2                  1
    NAME OF ARGUMENT              R                  A
    KIND OF ARGUMENT         IDENTIFIER
          INPUT CLASS                           NON-INPUT
         OUTPUT CLASS                           STR.OUTPUT
 ****************************************************************
Ø
 ****************************************************************
 *****WARNING        STATEMENT NO.    5  BASIC BLOCK NO.    1

Ø     CORRESPONDING ARGUMENTS IN THE PARAMETER LISTS ARE OF
      DIFFERENT DIMENSIONALITY.

                     CALLING SUBPROGRAM  CALLED SUBPROGRAM
                            SYSMAIN               INIT
    ARGUMENT POSITION          2                  2
 NUMBER OF DIMENSIONS          1                  Ø
    NAME OF ARGUMENT              Q            VECTOR
    KIND OF ARGUMENT         IDENTIFIER
          INPUT CLASS                           NON-INPUT
         OUTPUT CLASS                           NON-OUTPUT
 ****************************************************************
Ø
 ****************************************************************
 *****WARNING        STATEMENT NO.   11  BASIC BLOCK NO.    3

Ø     CORRESPONDING ARGUMENTS IN THE PARAMETER LISTS ARE OF
      DIFFERENT DIMENSIONALITY.

                     CALLING SUBPROGRAM  CALLED SUBPROGRAM
                            SYSMAIN             MAXMIN
    ARGUMENT POSITION          1                  1
 NUMBER OF DIMENSIONS          2                  1
    NAME OF ARGUMENT              R                  R
    KIND OF ARGUMENT         IDENTIFIER
          INPUT CLASS                           STR.INPUT
         OUTPUT CLASS                           NON-OUTPUT
 ****************************************************************
Ø
 ****************************************************************
```

```
*****WARNING          STATEMENT NO.    11  BASIC BLOCK NO.    3

Ø      A POSSIBLE ILLEGAL SIDE EFFECT HAS BEEN DETECTED. AN
       ARGUMENT PASSED BY THE CALLING SUBPROGRAM (VIA PARAMETER
       LIST OR COMMON) IS ALTERED BY THE CALLED SUBPROGRAM AS AN
       INDIRECT CONSEQUENCE OF THE CALL.

                         CALLING SUBPROGRAM  CALLED SUBPROGRAM
                              SYSMAIN              MAXMIN
   COMMON BLOCK NAME              BLK                 BLK
   NAME OF ARGUMENT                R
         INPUT CLASS        STR.INPUT
        OUTPUT CLASS       NON-OUTPUT           STR.OUTPUT
*****************************************************************


 Ø          LOCAL MESSAGES FOR PROGRAM UNIT
 Ø
 Ø
 *****************************************************************
 *****WARNING

 Ø     THE LOCAL VARIABLE NAMED       I RECEIVES A VALUE IN ITS
       LAST USAGE.
 *****************************************************************
 Ø
 *****************************************************************
 *****WARNING

 Ø     THE LOCAL VARIABLE NAMED     INS RECEIVES A VALUE IN ITS
       LAST USAGE.
 *****************************************************************
 Ø
 *****************************************************************
 *****WARNING

 Ø     THE LOCAL VARIABLE NAMED       M RECEIVES A VALUE IN ITS
       LAST USAGE.
 *****************************************************************



  1          CLASSIFICATION OF PARAMETER AND COMMON VARIABLES
  Ø                   COMMON                BLK
  Ø                     PARAMETERS/ENTRIES
       ORDER          NAME        INPUT CLASS          OUTPUT CLASS
         1               S          NON                   STRICT
         2               R          NON                   STRICT
         3            XMAX          NON                   NON
         4            XMIN          NON                   NON
```