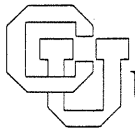
Some Experience with DAVE—A Frotran Program Analyzer

Leon J. Ostterweil and Lloyd D. Fosdick*

CU-CS-088-76 March 1976



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* This work supported by NSF grants GJ-36461 and DCR75-90072

ABSTRACT

This paper describes DAVE, an automatic program testing aid which performs a static analysis of Fortran programs. DAVE analyzes the data flows both within and across subprogram boundaries of Fortran programs, and is able to detect occurrences of uninitialized and dead variables in such programs. The paper shows how this capability facilitates the detection of a wide variety of errors, many of which are often quite subtle. The central analytic mechanism in DAVE is a depth-first search procedure which enables DAVE to execute efficiently. Some experiences with DAVE are described and evaluated and some future work is projected.

INDEX TERMS: Testing, Validation, Data Flow Analysis

Introduction

There is currently a great deal of interest in creating systems capable of assisting in the development of error-free programs. This interest results both from an awareness that erroneous programs are expensive and potentially lethal and from the fact that the problems involved in producing error-free programs are challenging and stimulating. As might be expected in the case of such a problem, which has enormous economic significance and high intellectual appeal, the approaches to its solution are numerous and diverse. This diversity is shown by the following list of approaches, which is intended to be indicative and not exhaustive:

* Devise error resistant design and coding practices: The terms Structured Programming [1], Stepwise Refinement [2], and Top-Down Design [3] are often associated with work in this area.
* Create error resistant languages: Such investigators as Wirth [4] and Gannon and Horning [5] have identified error-prone language features and proposed languages which avoid them.
* Devise better organizational strategies for programming: The Chief Programmer Team strategy of Mills and Baker [6,7] is notable in this area.
* Prove the correctness of programs: This is a relatively difficult and time consuming process, which has been successful largely for relatively small programs. Current work [8], however, offers hope that machine aids may eventually facilitate program proving for large programs

as well.

* Build automated program testing aids: These aids can do such things as monitor program execution [9, 10, 11], perform static diagnostic scans [12,13], and help generate test data [13, 14].

It seems clear that in the future the results of work in several of these areas will be coordinated in any effort to produce high quality, error resistant programs. We feel certain, however, that because humans will always have faulty memories, be prone to commit keyboard errors, and will inject various other errors into their programs, that any such coordinated attack will surely include a testing activity. This activity should rely heavily upon automated program test aids. In addition, we feel that automated test aids are of particular importance at present, because they, unlike most of the other current approaches, offer some hope of helping determine the validity and worth of some of the enormous body of programs already in existence.

For these reasons, we created DAVE, an automated program testing aid which, we believe, embodies important new diagnostic capabilities.

DAVE As An Automated Testing Aid

DAVE performs a diagnostic scan of an ANSI Standard [15] Fortran program for the purpose of detecting erroneous or suspicious situations. Systems such as this are often referred to as static analysis systems, in that they do not require that the program be executed. As a con-

sequence, their analytic results are not restricted in their applicability to a single execution. On the contrary, in DAVE's case it is possible to simulate in a limited way the effect of executing all sequences of statements in a program. Hence DAVE is able not only to detect errors, but, more important, it is also able to determine the absence of certain types of errors or suspicious situations for all possible executions of the program. Because of this latter capability DAVE is a valuable tool in examining existing programs for the purpose of validating them.

Should DAVE detect an error or suspicious situation along some execution sequence through the program an error or warning message describing the situation is produced. A human analyst must then determine the true importance of the message. At this point a dynamic analysis system might be used to instrument the program and gather detailed information about the progress of an actual execution of the sequence of statements which DAVE had pinpointed. Hence in this way DAVE is also useful as a debugging aid during program development.

Errors Detected by DAVE

All program testing aids are incapable of determining whether or not a program is completely correct. A program testing aid can at best determine whether or not a program adheres to some specified standards. A violation of such a standard may be taken to be an a priori error or a suspicious condition, symptomatic of some other error. Hence in all program testing aids there must be an initial

understanding of the standards against which programs are to be measured. In DAVE these standards all relate to the correct flow of data through a program. It is our contention that in a correctly executing program two rules should always be obeyed:

1. No variable will be used in a computation (referenced) until it has previously been assigned a value (been defined).
2. A variable, once defined, will subsequently be referenced before the variable is redefined or the program terminates.

DAVE's diagnostic scan determines whether either of these two rules can be violated for any sequence of statement executions. A violation of the first, called a type 1 anomaly, is a violation of the ANSI Fortran Standard [15] and is considered to be an a priori error. A violation of the second, called a type 2 anomaly, is considered to be a symptom of some other error. DAVE is able to detect a type 1 anomaly for any possible execution sequence. Thus if DAVE does not detect such an anomaly then none exists within the program. Hence DAVE is able to both detect the presence, and assure the absence of data flow anomalies. The former capability we refer to as error detection and the latter we refer to as validation. Clearly the foregoing implies that DAVE is able to validate programs for the absence of uninitialized variables.

In practice we have found, however, that anomalies of both types are usually symptoms of other errors. We have been gratified to find that the range of errors symptomatized by type 1 and type 2

anomalies is quite large, extending from misspellings to subprogram invocation errors. Because of this phenomenon of anomalies occurring as symptoms of other errors, it has turned out that DAVE has been most useful in indirectly detecting errors other than uninitialized and dead variables (in the sense of [16]).

More specifically, a large measure of DAVE's indirect error detection capability arises from the fact that DAVE performs its data flow analysis across subprogram boundaries. This data flow from one program unit to another must be completely determined if all possible anomalies are to be detected. Having made this complete determination, however, DAVE is in a position to also detect a variety of program unit communication errors such as illegal side effects and inconsistent COMMON declarations. Because this interprocedural data flow is often quite subtle, errors involving it are likewise often subtle and difficult for a human to identify. Hence it is not surprising that DAVE's error detection capabilities in this area have proven to be among its most useful features.

An Example

Figure 1 shows a somewhat contrived Fortran program which is designed to illustrate some of the error detection capabilities referred to in the previous section. The purpose of the program is to compute and print out the cost of covering an area with some covering material. The program reads in PSF, the cost per square foot of the material; LCRT, an integer used to denote whether the area is a rectangle (if LCRT is 1), a circle (if LCRT is 2), or a

triangle (if LCRT is 3); and D1 and D2, the two dimensions of the area (D2 is unused if LCRT is 2). The program then branches on LCRT to three different subprograms, AREAR, AREAC, and AREAT, which are supposed to compute the area of the rectangle, circle or triangle (respectively), and place the value of this area in the variable AREA. Subroutine DOLS is then called to compute COST, the product of AREA and PSF. Finally COST, the desired result,is printed out.

Close inspection of the program reveals that it contains errors, some of which are not very obvious. Perhaps the most obvious error is that the value of pi is set into the variable PI, but the variable P is used to pass this value into AREAC, the subprogram which requires it. A second error is that there is a misspelling in the subroutine AREAT. The third parameter is named AREA, but the body of the subroutine defines a value for the variable AREAT instead. Hence upon return there is no value given to the main program variable AREA, which is referenced in a subsequent computation. A third error involves the COMMON block B, which is used for communication between the main program and DOLS. B contains the variables AREA and COST. DOLS, which expects AREA to contain the computed area, uses it to compute the value of COST, which is then passed through B back to the main program. Unfortunately, the order of declaration of AREA and COST in the main program is the reverse of the order of declaration in DOLS.

Detection of these three errors would most likely be at least tedious using conventional debugging methods. The third error would cause any execution to be erroneous, but each of the first two would

```
      COMMON /B/ AREA, COST
      READ (5, 1) PSF, LCRT, D1, D2
      PI = 3.1416
      IF (LCRT .NE. 1) GO TO 10
      AREA = AREAR (D1, D2)
      GO TO 100
10    IF (LCRT .NE. 2) GO TO 20
      AREA = AREAC (P, D1)
      GO TO 100
20    CALL AREAT (D1, D2, AREA)
100   CALL DOLS (PSF)
      WRITE (6, 2) COST
      STOP
1     FORMAT (F6.2, I2, 2F10.4)
2     FORMAT (1H , F8.2)
      END
      FUNCTION AREAR (A, B)
      AREAR = A * B
      RETURN
      END
      FUNCTION AREAC (PI, RAD)
      AREAC = PI * RAD ** 2
      RETURN
      END
      SUBROUTINE AREAT (B, H, AREA)
      AREAT = 0.5 * B * H
      RETURN
      END
      SUBROUTINE DOLS (PSF)
      COMMON /B/ COST, AREA
      COST = PSF * AREA
      RETURN
      END
```

Figure 1: A program illustrating some of the error detection capabilities of DAVE.

cause an erroneous execution only for a single specific value of LCRT. Hence it is reasonable to expect that the three errors would be ferreted out one at a time, perhaps with some difficulty, if the usual procedure of running test cases was followed.

DAVE would facilitate the detection of all three errors in only one diagnostic scan because each one causes data flow anomalies. In the case of the first error, DAVE would identify the definition of PI without subsequent reference as a type 2 anomaly. DAVE would also determine that the first argument in any invocation of AREAC must carry in a value. Hence in analyzing the main program DAVE would conclude that the invocation of AREAC would cause a type 1 anomaly, and would print an appropriate message. DAVE is unable to state directly the true error--namely a misspelling. The two anomaly messages, however, point strongly to the true error.

The second error, also a misspelling, is likewise strongly indicated by anomaly messages. In analyzing AREAT, DAVE would discover that the local variable AREAT is never referenced after definition, and print a message describing this type 2 anomaly. DAVE would also determine that the parameter AREA is neither referenced nor defined in the subroutine. This is regarded as a suspicious situation, and DAVE would produce a message describing it. Finally, in analyzing the main program, DAVE would discover that there is a sequence of statements leading up to the invocation of DOLS which does not cause the variable AREA to be defined (namely the one which includes the invocation of AREAT). No anomaly message will be printed because, as shall be seen, the third error causes DOLS to make no use of AREA. Had the third error not been present, however, a type 1

anomaly message would have been printed. In this case the interaction of two errors causes the suppression of one anomaly message. DAVE, nevertheless, produces two other messages in response to the second error.

The third error is a transposition of variables in a COMMON statement. DAVE would analyze DOLS and determine that it requires a value to be passed in through the second variable in COMMON block B, and that it passes out a value through the first variable in B. Upon analyzing the main program DAVE would find that COST, the second variable in COMMON block B, is never initialized before the invocation of DOLS--hence a type 1 anomaly message would be printed. DAVE would also discover that AREA,the first variable in block B, generally has a freshly computed value when DOLS is invoked. DAVE would observe the DOLS resets this value before it is ever referenced and print a type 2 anomaly message. Finally, DAVE would observe that AREA, the first variable in block B, is never referenced after its definition in DOLS and print another type 2 anomaly message. Here too, it is clear that these three messages strongly illuminate the transposition error, although it is never explicitly identified.

This brief example is intended to give an impression of how DAVE's analysis can assist in isolating subtle errors. We expect that the reader can see how DAVE is also useful in detecting other errors such as transposed statements, illegal side effects, and mismatched argument and parameter lists. Likewise the reader should be able to see that the use of an automatic aid such as DAVE is far more necessary in analyzing a large, complex, real-world program than in detecting the errors in this small, simple, contrived example.

The Design of the DAVE System

DAVE performs its analysis by passing over the program units of a program, from the lowest level subprograms upwards to the main program, analyzing each program unit exactly once, employing a depth-first search of a labelled flow graph of the program unit. Details of the system's design and implementation can be found in [17, 18], and hence are omitted here. For completeness, however, a brief simplified overview shall now be given.

DAVE's analysis is performed on labelled flow graphs, where a different graph represents each of the program units of the program. The nodes of a flow graph represent the program unit's statements and the edges represent intra-program-unit control transfers. Each node's label describes which variables are defined and referenced during the execution of the code corresponding to the node. These graphs are constructed at the start of DAVE's analysis. The graphs are easily constructed, but they cannot immediately be completely labelled, due to the undeterminable status of variables which are used as arguments to subprograms. Hence the graphs are left only partially labelled until a later phase of the analysis. All invocations of subprograms which are made by a program unit are carefully noted, however. After the last program unit flow graph has been created and partially labelled, the totality of these invocations is used to construct the program call graph, a graph whose nodes represent the program units and whose edges represent the subprogram invocations. Due to the impossibility of recursive calling chains

in ANSI Standard Fortran, it is expected (although not always true) that the call graph will be acyclic. Hence there will be leaf nodes (nodes without any outedges) in the graph. These represent program units which do not invoke any subprograms. Hence the flow graphs for these program units are known to be completely labelled. DAVE now continues by carefully analyzing these program units' flow graphs.

Once a program unit's flow graph is completely labelled it is possible to determine the pattern of references to and definitions of each of the program unit's variables for each of the program unit's execution sequences. Uninitialized and dead variables are found by examining these patterns. In DAVE a variable's pattern of references and definitions is determined and examined by a depth-first search procedure (described in detail in [18]) which executes in time proportional to the number of edges in the flow graph. The search procedure is repeated for each variable in the program unit. It classifies each variable as either non-input, input or strict input and either non-output, output or strict output. A variable is classified input if along some, but not all, execution sequences through the program unit the variable is referenced before it is defined. If there is no such execution sequence, the variable is classified non-input. If the variable is referenced before definition along all execution sequences, the variable is classified strict input. Similarly, the variable is classified output if along some, but not all, execution sequences in the program unit the variable is defined. If there is no such execution sequence the variable is classified non-output. If it is defined along all execution sequences, it is classified strict output.

These classifications having been made, DAVE begins its search for anomalies. If a local variable is classified strict input, it is clear that a type 1 anomaly will occur, and an error message is produced. If a local variable is classified input, then a type 1 anomaly exists for some, but not all, sequences of statements. In recognition of the fact that these sequences may not actually be executable in response to any input data, DAVE produces a warning message describing the possibility of executing an anomaly bearing sequence of statements. DAVE performs similar scans for type 2 anomalies by executing searches from a definition of a local variable to determine whether the subprogram terminates or redefines the variable before referencing it.

The determination of the input/output status of non-local variables (ie. parameters and COMMON variables) of leaf subprograms is not used immediately in the detection of anomalies, but rather is used to enable DAVE's analysis to continue for higher level program units. The program call graph is used to locate all invocations of the leaf subprograms, and now the nodes corresponding to these invocations are labelled. At the end of this process, some non-leaf subprograms have become completely labelled and the depth-first search procedure can be applied to them. This process continues until eventually the main program itself is searched.

The process of using the input/output classification of a non-local variable of an invoked subprogram to label an invoking node is worthy of some elaboration here as it incorporates a number of useful error checks. DAVE first compares argument and parameter lists for agreement in length and type. Lack of agreement is considered an error. Next, parameter output classifications are compared to arguments. If

a parameter is classified as output or strict output and the corresponding argument is a constant, expression or subprogram name, DAVE produces a message. COMMON variables which carry data into or out of the invoked subprogram are identified at this time and messages describing them are made available for use as documentation. Finally, the variables in the invoking statement are examined to see whether any of them is used both as an input and an output in separate subprogram invocations. If so, DAVE has detected an illegal side effect, and produces a message identifying it.

After DAVE has searched the main program, it examines the input classifications of its COMMON variables. Error or warning messages are generated for each COMMON variable which is typed strict input or input but is not initialized in a BLOCK DATA subprogram.

Implementation Data

DAVE is implemented as a Fortran program consisting of approximately 25,000 source statements. It operates in four overlaid phases, the largest of which occupies 50,000 decimal words of central memory on the CDC 6400. DAVE is written almost entirely in machine independent ANSI Standard Fortran. Some non-Standard and machine dependent coding practices seemed expedient, however, and they are quarantined to a small number of small subprograms. DAVE was developed on the CDC 6400 at the University of Colorado, but has been successfully moved to a CDC 7600 and two machines in the IBM 360/370 series. Installations on a Univac 1100 series machine and a Honeywell 6000 series machine are planned for the near future.

Under its current configuration DAVE is able to process a program

consisting of a few dozen subprograms, each of which may contain no more than 200 - 250 source statements. These limits depend entirely upon internal table and scratch array sizes, and have been quickly altered to produce different experimental configurations. At this writing, the largest body of code which DAVE has processed has been a 2700 source statement subprogram library. DAVE is currently operational, however, on a machine with sufficient central memory to enable it to process its own source code, and this should be accomplished in the near future. The analysis of a source program by DAVE has been observed to require an average of 0.3 seconds of central processor time per source statement on the CDC 6400 and to cost approximately six to eight cents per source statement under the University of Colorado Computing Center charge algorithm.

Summary of Experience to Date

DAVE has been operational on an experimental basis for a few months to date. During this time we have seen evidence that it can be a valuable tool in the production of high quality, error-free programs. Most of our experience has come from using DAVE in validating completed programs. These included a highly respected matrix manipulation system, several recent algorithms taken from the ACM Transactions on Mathematical Software, and a program submitted as a part of a Master's Thesis in Computer Science. Errors were detected in some of the algorithms, and the Master's Thesis was found to have numerous errors. In most cases the errors were of the type that would hamper program portability, such as reference to uninitialized variables which should have been initialized to zero, reference to exhausted DO loop indices

and subprogram invocations with mismatched argument and parameter lists. In each case, the errors did not seem to prevent successful execution on the author's computer, but seemed likely to cause trouble if executed elsewhere. (There was some suspicion, however, that some of the erroneous subprogram invocations were imbedded in program segments which had never been tested or were unexecutable.)

Perhaps the most surprising observation was that DAVE's messages often gave unexpected insight into the author's coding style. For example, a program for which DAVE produced numerous type 2 anomaly warning messages did not prove to be incorrect, but rather it contained numerous loops in which indices and counters were updated immediately before DO loop endings. It was discovered that the author tended to favor WHILE loop constructions which are often awkward in Fortran. This was observed by DAVE. As another example, some programs contained subroutine definitions which did not use some parameters either as input or output. This was observed to be a symptom of the fact that the code had evolved, but not been carefully polished. We acknowledge that in the first case the author should probably have coded in a more comfortable language, and in the second the program was probably not thoroughly designed before coding began. DAVE can do nothing to prevent these serious breaches of good programming practice. It was surprising and gratifying, however, to discover that DAVE could often strongly indicate their presence--a capability which we believe is quite useful.

We have had less experience in using DAVE as an error detection aid during program development. This seems paradoxical because we feel that DAVE is very well suited to aiding the detection of subtle

errors, thereby speeding program development. The high cost of using and the awkwardness in accessing the current version of DAVE, however, forestalled its use in many cases. DAVE's accessing procedures have recently been streamlined, but the high cost of using the system is attributable to a decision made during development of the prototype system to favor flexibility over efficiency. Hence high costs are likely to remain for the forseeable future. As a consequence of this, the few programs which DAVE helped to debug all had subtle errors which had defied earlier concerted efforts at detection. DAVE was usually able to point rather directly at these. Such errors as camouflaged misspellings (eg. CARD instead of CARDS) and mismatched argument/ parameter lists were discovered in this way.

Our experience has not been entirely positive. An obvious and troublesome difficulty is DAVE's copious output. As already illustrated, a single error often generates numerous messages. Moreover, we have observed that some messages are rarely symptoms of errors. The net effect is that human analysts are often reluctant to pursue all of DAVE's messages, thereby raising the possibility that errors whose symptoms have been detected will go unnoticed. DAVE users have also complained about the unclear wording of many messages. All of these human interface problems must be solved lest DAVE's useful capabilities be buried under an avalanche of opaque verbiage.

Problem Areas and Future Work

We consider the current DAVE system to be a working prototype. Consequently, as might be expected, it has neither the speed nor complete processing capabilities which might be expected of a polished

system. The purpose of this section is to describe the areas in which we feel DAVE is deficient and to indicate where and how improvements might be made.

One of the most immediate problems is that DAVE was designed to analyze only programs written in ANSI Standard Fortran. DAVE has since been liberalized to accept most of the Fortran dialects available on CDC equipment. Little effort, however, has been devoted to the problems of accepting other dialects. Many of the changes required in order to accept such dialects appear to be straightforward, but it is worthwhile to note that some features of some dialects (eg. the ENTRY feature found in FORTRAN V [19]) can not be properly analyzed by DAVE without substantial alterations.

More serious is the fact that there are a number of features of ANSI Standard Fortran which are currently incorrectly or inadequately handled by DAVE. A notable and discouraging example of this is the treatment of arrays. Currently DAVE treats all arrays as simple variables, thereby blurring all distinctions between array elements and eliminating the possibility of detecting certain anomalous uses of the individual elements. Unfortunately, there are fundamental theoretical reasons why patterns of array references in an arbitrary program can never be completely analyzed by a static analysis system such as DAVE.

As already noted, the call graph of a Fortran program may not be acyclic even though the program is incapable of ever executing a recursive calling chain. Such a program can not be analyzed by DAVE. The most promising solution to this problem seems to be to adapt DAVE so that it is able to analyze recursive programs. This is

an interesting and worthwhile problem which seems solvable, and would move DAVE in the direction of being able to analyze programs written in other languages such as ALGOL and PL/I.

DAVE is also currently unable to build the complete call graph for programs in which subprogram names are passed as parameters. Hence DAVE cannot analyze such programs. An algorithm due to Kallal and Osterweil [20] is capable of building the call graph of such a program. This algorithm will probably be incorporated into future versions of DAVE.

Other problems are encountered by DAVE in trying to analyze programs containing extensive or tricky uses of aliasing constructs such as EQUIVALENCE statements and restructured COMMON lists. Most of these will be overcome in future versions of DAVE by using well known compiling techniques. Others,such as using two EQUIVALENCE'd variables as arguments to the same subprogram, challenge some of DAVE's basic assumptions, and may never be satisfactorally solved.

Programs in which variables become undefined (eg. the exhaustion of a DO loop causes the undefinition of the DO index) may, under certain unusual circumstances, be incorrectly analyzed. This results from our tardy recognition that variables must be typed with respect to undefinition just as they are typed with respect to input and output (ie. they must be typed as non-undefined, undefined, or strict undefined). We have developed algorithms for performing and correctly employing this typing of undefinition, but have not yet incorporated them into DAVE.

Finally, we are actively exploring the relationship between static testing aids and global program optimization. Our investigation

[21] has shown that existing algorithms in global optimization can readily be harnessed to do much of the analysis done by DAVE. Hence we foresee the incorporation of systems such as DAVE into a future generation of compilers.

References

1. E. W. Dijkstra, "Notes on Structured Programming", in Structured Programming, by O. - J. Dahl, E. W. Dijkstra and C. A. R. Hoare, Academic Press, London and New York, 1972.

2. N. Wirth, "Program Development by Stepwise Refinement", CACM 14 pp. 221 - 227 (April 1974).

3. H. D. Mills, "Top -Down Programming in Large Systems", in Debugging Techniques in Large Systems, R. Rustin (ed.) Prentice-Hall, Englewood Cliffs, N.J. (1971) pp.41 - 45.

4. N. Wirth, " An Assessment of the Programming Language PASCAL", IEEE Transactions on Software Engineering, SE-1, pp. 192 - 198 (June 1975).

5. J. D. Gannon and J. J. Horning, "Language Design for Program Reliability," IEEE Transactions on Software Engineering SE - 1, pp. 179 - 191, (June 1975).

6. H. D. Mills, "How to Write Correct Programs and Know It", Proceedings of the 1975 International Conference on Reliable Software, IEEE Cat. No. 75CH0940-7CSR, pp.363 - 370.

7. F. T. Baker, "Chief Programmer Team Management of Production Programming," IBM Systems Journal, 11 pp. 56 - 73 (1972).

8. D. I. Good, R. L. London, and W. W. Bledsoe, "An Interactive Program Verification System", IEEE Transactions on Software Engineering SE-1 pp. 59 - 67 (March 1975).

9. R. M. Balzer, "EXDAMS: Extendable Debugging and Monitoring System", AFIPS 1969 SJCC, 34 AFIPS Press, Montvale N.J., pp. 567 - 580.

10. R. E. Fairley, "An Experimental Program Testing Facility," Proceedings of the First National Conference on Software Engineering, IEEE Cat. No. 75CH0992-8C, pp. 47 - 52.

11. L. G. Stucki, "Automatic Generation of Self-Metric Software", Proceedings of the 1973 IEEE Symposium on Computer Software Reliability, IEEE Cat. No. 73C40741-9CSR, pp . 94 - 100.

12. C. V. Ramamoorthy and S-B. F. Ho, "Testing Large Software with Automated Software Evaluation Systems", IEEE Transactions on Software Engineering SE-1 pp. 46 - 58 (March 1975).

13. E. F. Miller, Jr. and R. A. Melton, "Automated Generation of Testcase Datasets", Proceedings of the 1975 International Conference on Reliable Software, IEEE Cat. No. 75CH0940-7CSR, pp. 51 - 58.

14. K. A. Krause, R. W. Smith, and M. A. Goodwin, "Optimal Software Test Planning Through Automated Network Analysis", Proceedings of the 1973 IEEE Symposium on Computer Software Reliability, IEEE Cat. No. 73C40741-9CSR, pp. 18 - 22.

15. American National Standards Institute, FORTRAN, ANSI X3.9 (1966).

16. M. Schaefer, A Mathematical Theory of Global Program Optimization, Prentice-Hall, Englewood Cliffs, N.J., 1973.

17. L. J. Osterweil and L. D. Fosdick, "Data Flow Analysis as an Aid in Documentation, Assertion Generation, Validation, and Error Detection", University of Colorado Department of Computer Science Technical Report No. CU-CS-055-74.

18. L. J. Osterweil and L. D. Fosdick, "DAVE - A Validation, Error Detection, and Documentation System for Fortran Programs", Software Practice & Experience (to appear).

19. Univac Division of Sperry Rand Corporation, UNIVAC 1108 FORTRAN V, Cat. No. UP-4060 Rev. 1, Sperry Rand Corporation, pp. 4-18 to 4-19.

20. V. Kallal and L. J. Osterweil, (to appear as University of Colorado Department of Computer Science Technical Report).

21. L. D. Fosdick and L. J. Osterweil, "Validation and Global Optimization of Programs", Proceedings of the Fourth Texas Conference on Computing Systems (1975).

Some experience with DAVE — A Fortran program analyzer*

by LEON J. OSTERWEIL and LLOYD D. FOSDICK
University of Colorado
Boulder, Colorado

ABSTRACT

This paper describes DAVE, an automatic program testing aid which performs a static analysis of Fortran programs. DAVE analyzes the data flows both within and across subprogram boundaries of Fortran programs, and is able to detect occurrences of uninitialized and dead variables in such programs. The paper shows how this capability facilitates the detection of a wide variety of errors, many of which are often quite subtle. The central analytic mechanism in DAVE is a depth-first search procedure which enables DAVE to execute efficiently. Some experiences with DAVE are described and evaluated and some future work is projected.

INTRODUCTION

There is currently a great deal of interest in creating systems capable of assisting in the development of error-free programs. This interest results both from an awareness that erroneous programs are expensive and potentially lethal and from the fact that the problems involved in producing error-free programs are challenging and stimulating. As might be expected in the case of such a problem, which has enormous economic significance and high intellectual appeal, the approaches to its solution are numerous and diverse. This diversity is shown by the following list of approaches, which is intended to be indicative and not exhaustive:

- Devise error resistant design and coding practices: The terms Structured Programming,[1] Stepwise Refinement,[2] and Top-Down Design[3] are often associated with work in this area.
- Create error resistant languages: Such investigators as Wirth[4] and Gannon and Horning[5] have identified error-prone language features and proposed languages which avoid them.
- Devise better organizational strategies for programming: The Chief Programmer Team strategy of Mills and Baker[6,7] is notable in this area.
- Prove the correctness of programs: This is a relatively difficult and time consuming process, which has been successful largely for relatively small programs. Current work,[8] however, offers hope that machine aids may eventually facilitate program proving for large programs as well.
- Build automated program testing aids: These aids can do such things as monitor program execution,[9,10,11] perform static diagnostic scans,[12,13] and help generate test data.[13,14]

It seems clear that in the future the results of work in several of these areas will be coordinated in any effort to produce high quality, error resistant programs. We feel certain, however, that because humans will always have faulty memories, be prone to commit keyboard errors, and will inject various other errors into their programs, that any such coordinated attack will surely include a testing activity. This activity should rely heavily upon automated program test aids. In addition, we feel that automated test aids are of particular importance at present, because they, unlike most of the other current approaches, offer some hope of helping determine the validity and worth of some of the enormous body of programs already in existence.

For these reasons, we created DAVE, an automated program testing aid which, we believe, embodies important new diagnostic capabilities.

DAVE AS AN AUTOMATED TESTING AID

DAVE performs a diagnostic scan of an ANSI Standard[15] Fortran program for the purpose of detecting erroneous or suspicious situations. Systems such as this are often referred to as static analysis systems, in that they do not require that the program be executed. As a consequence, their analytic results are not restricted in their applicability to a single execution. On the contrary, in DAVE's case it is possible to simulate in a limited way the effect of executing all sequences of statements in a program. Hence DAVE is able not only to detect errors, but, more important, it is also able to determine the absence of certain types of errors or suspicious situations for all possible executions of the

* This work supported by NSF Grants GJ-36461 and DCR75-90072.

program. Because of this latter capability DAVE is a valuable tool in examining existing programs for the purpose of validating them.

Should DAVE detect an error or suspicious situation along some execution sequence through the program an error or warning message describing the situation is produced. A human analyst must then determine the true importance of the message. At this point a dynamic analysis system might be used to instrument the program and gather detailed information about the progress of an actual execution of the sequence of statements which DAVE had pinpointed. Hence in this way DAVE is also useful as a debugging aid during program development.

ERRORS DETECTED BY DAVE

All program testing aids are incapable of determining whether or not a program is completely correct. A program testing aid can at best determine whether or not a program adheres to some specified standards. A violation of such a standard may be taken to be an a priori error or a suspicious condition, symptomatic of some other error. Hence in all program testing aids there must be an initial understanding of the standards against which programs are to be measured. In DAVE these standards all relate to the correct flow of data through a program. It is our contention that in a correctly executing program two rules should always be obeyed:

1. No variable will be used in a computation (referenced) until it has previously been assigned a value (been defined).
2. A variable, once defined, will subsequently be referenced before the variable is redefined or the program terminates.

DAVE's diagnostic scan determines whether either of these two rules can be violated for any sequence of statement executions. A violation of the first, called a type 1 anomaly, is a violation of the ANSI Fortran Standard[15] and is considered to be an a priori error. A violation of the second, called a type 2 anomaly, is considered to be a symptom of some other error. DAVE is able to detect a type 1 anomaly for any possible execution sequence. Thus if DAVE does not detect such an anomaly then none exists within the program. Hence DAVE is able to both detect the presence, and assure the absence of data flow anomalies. The former capability we refer to as error detection and the latter we refer to as validation. Clearly the foregoing implies that DAVE is able to validate programs for the absence of uninitialized variables.

In practice we have found, however, that anomalies of both types are usually symptoms of other errors. We have been gratified to find that the range of errors symptomatized by type 1 and type 2 anomalies is quite large, extending from misspellings to subprogram invocation errors. Because of this phenomenon of anomalies occurring as symptoms of other errors, it has turned out that DAVE has been most useful in indirectly detecting errors other than uninitialized and dead variables (in the sense of Reference 16).

More specifically, a large measure of DAVE's indirect error detection capability arises from the fact that DAVE performs its data flow analysis across subprogram boundaries. This data flow from one program unit to another must be completely determined if all possible anomalies are to be detected. Having made this complete determination, however, DAVE is in a position to also detect a variety of program unit communication errors such as illegal side effects and inconsistent COMMON declarations. Because this interprocedural data flow is often quite subtle, errors involving it are likewise often subtle and difficult for a human to identify. Hence it is not surprising that DAVE's error detection capabilities in this area have proven to be among its most useful features.

AN EXAMPLE

Figure 1 shows a somewhat contrived Fortran program which is designed to illustrate some of the error detection capabilities referred to in the previous section. The purpose of the program is to compute and

```
      COMMON /B/ AREA, COST
      READ (5,1) PSF, LCRT, D1, D2
      PI=3.1416
      IF (LCRT .NE. 1) GO TO 10
      AREA=AREAR (D1, D2)
      GO TO 100
10    IF (LCRT .NE. 2) GO TO 20
      AREA=AREAC (P, D1)
      GO TO 100
20    CALL AREAT (D1, D2, AREA)
100   CALL DOLS (PSF)
      WRITE (6, 2) COST
      STOP
1     FORMAT (F6.2, I2, 2F10.4)
2     FORMAT (1H, F8.2)
      END
      FUNCTION AREAR (A, B)
      AREAR=A * B
      RETURN
      END
      FUNCTION AREAC (PI, RAD)
      AREAC=PI * RAD ** 2
      RETURN
      END
      SUBROUTINE AREAT (B, H, AREA)
      AREAT=0.5 * B * H
      RETURN
      END
      SUBROUTINE DOLS (PSF)
      COMMON /B/ COST, AREA
      COST=PSF * AREA
      RETURN
      END
```

Figure 1—A program illustrating some of the error detection capabilities of DAVE

print out the cost of covering an area with some covering material. The program reads in PSF, the cost per square foot of the material; LCRT, an integer used to denote whether the area is a rectangle (if LCRT is 1), a circle (if LCRT is 2), or a triangle (if LCRT is 3); and D1 and D2, the two dimensions of the area (D2 is unused if LCRT is 2). The program then branches on LCRT to three different subprograms, AREAR, AREAC, and AREAT, which are supposed to compute the area of the rectangle, circle or triangle (respectively), and place the value of this area in the variable AREA. Subroutine DOLS is then called to compute COST, the product of AREA and PSF. Finally COST, the desired result, is printed out.

Close inspection of the program reveals that it contains errors, some of which are not very obvious. Perhaps the most obvious error is that the value of pi is set into the variable PI, but the variable P is used to pass this value into AREAC, the subprogram which requires it. A second error is that there is a misspelling in the subroutine AREAT. The third parameter is named AREA, but the body of the subroutine defines a value for the variable AREAT instead. Hence upon return there is no value given to the main program variable AREA, which is referenced in a subsequent computation. A third error involves the COMMON block B, which is used for communication between the main program and DOLS. B contains the variables AREA and COST. DOLS, which expects AREA to contain the computed area, uses it to compute the value of COST, which is then passed through B back to the main program. Unfortunately, the order of declaration of AREA and COST in the main program is the reverse of the order of declaration in DOLS.

Detection of these three errors would most likely be at least tedious using conventional debugging methods. The third error would cause any execution to be erroneous, but each of the first two would cause an erroneous execution only for a single specific value of LCRT. Hence it is reasonable to expect that the three errors would be ferreted out one at a time, perhaps with some difficulty, if the usual procedure of running test cases was followed.

DAVE would facilitate the detection of all three errors in only one diagnostic scan because each one causes data flow anomalies. In the case of the first error, DAVE would identify the definition of PI without subsequent reference as a type 2 anomaly. DAVE would also determine that the first argument in any invocation of AREAC must carry in a value. Hence in analyzing the main program DAVE would conclude that the invocation of AREAC would cause a type 1 anomaly, and would print an appropriate message. DAVE is unable to state directly the true error—namely a misspelling. The two anomaly messages, however, point strongly to the true error.

The second error, also a misspelling, is likewise strongly indicated by anomaly messages. In analyzing AREAT, DAVE would discover that the local variable AREAT is never referenced after definition, and print a message describing this type 2 anomaly. DAVE would also determine that the parameter AREA is neither referenced nor defined in the subroutine. This is regarded as a suspicious situation, and DAVE would produce a message describing it. Finally, in analyzing the main program, DAVE would discover that there is a sequence of statements leading up to the invocation of DOLS which does not cause the variable AREA to be defined (namely the one which includes the invocation of AREAT). No anomaly message will be printed because, as shall be seen, the third error causes DOLS to make no use of AREA. Had the third error not been present, however, a type 1 anomaly message would have been printed. In this case the interaction of two errors causes the suppression of one anomaly message. DAVE, nevertheless, produces two other messages in reponse to the second error.

The third error is a transposition of variables in a COMMON statement. DAVE would analyze DOLS and determine that it requires a value to be passed in through the second variable in COMMON block B, and that it passes out a value through the first variable in B. Upon analyzing the main program DAVE would find that COST, the second variable in COMMON block B, is never initialized before the invocation of DOLS—hence a type 1 anomaly message would be printed. DAVE would also discover that AREA, the first variable in block B, generally has a freshly computed value when DOLS is invoked. DAVE would observe the DOLS resets this value before it is ever referenced and print a type 2 anomaly message. Finally, DAVE would observe that AREA, the first variable in block B, is never referenced after its definition in DOLS and print another type 2 anomaly message. Here too, it is clear that these three messages strongly illuminate the transposition error, although it is never explicitly identified.

This brief example is intended to give an impression of how DAVE's analysis can assist in isolating subtle errors. We expect that the reader can see how DAVE is also useful in detecting other errors such as transposed statements, illegal side effects, and mismatched argument and parameter lists. Likewise the reader should be able to see that the use of an automatic aid such as DAVE is far more necessary in analyzing a large, complex, real-world program than in detecting the errors in this small, simple, contrived example.

THE DESIGN OF THE DAVE SYSTEM

DAVE performs its analysis by passing over the program units of a program, from the lowest level subprograms upward to the main program, analyzing each program unit exactly once, employing a depth-first search of a labelled flow graph of the program unit. Details of the system's design and implementation can be found in References 17 and 18, and hence are

omitted here. For completeness, however, a brief simplified overview shall now be given.

DAVE's analysis is performed on labelled flow graphs, where a different graph represents each of the program units of the program. The nodes of a flow graph represent the program unit's statements and the edges represent intra-program-unit control transfers. Each node's label describes which variables are defined and referenced during the execution of the code corresponding to the node. These graphs are constructed at the start of DAVE's analysis. The graphs are easily constructed, but they cannot immediately be completely labelled, due to the undeterminable status of variables which are used as arguments to subprograms. Hence the graphs are left only partially labelled until a later phase of the analysis. All invocations of subprograms which are made by a program unit are carefully noted, however. After the last program unit flow graph has been created and partially labelled, the totality of these invocations is used to construct the program call graph, a graph whose nodes represent the program units and whose edges represent the subprogram invocations. Due to the impossibility of recursive calling chains in ANSI Standard Fortran, it is expected (although not always true) that the call graph will be acyclic. Hence there will be leaf nodes (nodes without any outedges) in the graph. These represent program units which do not invoke any subprograms. Hence the flow graphs for these program units are known to be completely labelled. DAVE now continues by carefully analyzing these program units' flow graphs.

Once a program unit's flow graph is completely labelled it is possible to determine the pattern of references to and definitions of each of the program unit's variables for each of the program unit's execution sequences. Uninitialized and dead variables are found by examining these patterns. In DAVE a variable's pattern of references and definitions is determined and examined by a depth-first search procedure (described in detail in Reference 18) which executes in time proportional to the number of edges in the flow graph. The search procedure is repeated for each variable in the program unit. It classifies each variable as either non-input, input or strict input and either non-output, output or strict output. A variable is classified input if along some, but not all, execution sequences through the program unit the variable is referenced before it is defined. If there is no such execution sequence, the variable is classified non-input. If the variable is referenced before definition along all execution sequences, the variable is classified strict input. Similarly, the variable is classified output if along some, but not all, execution sequences in the program unit the variable is defined. If there is no such execution sequence the variable is classified non-output. If it is defined along all execution sequences, it is classified strict output.

These classifications having been made, DAVE begins its search for anomalies. If a local variable is classified strict input, it is clear that a type 1 anomaly will occur, and an error message is produced. If a local variable is classified input, then a type 1 anomaly exists for some, but not all, sequences of statements. In recognition of the fact that these sequences may not actually be executable in response to any input data, DAVE produces a warning message describing the possibility of executing an anomaly bearing sequence of statements. DAVE performs similar scans for type 2 anomalies by executing searches from a definition of a local variable to determine whether the subprogram terminates or redefines the variable before referencing it.

The determination of the input/output status of non-local variables (i.e., parameters and COMMON variables) of leaf subprograms is not used immediately in the detection of anomalies, but rather is used to enable DAVE's analysis to continue for higher level program units. The program call graph is used to locate all invocations of the leaf subprograms, and now the nodes corresponding to these invocations are labelled. At the end of this process, some non-leaf subprograms have become completely labelled and the depth-first search procedure can be applied to them. This process continues until eventually the main program itself is searched.

The process of using the input/output classification of a non-local variable of an invoked subprogram to label an invoking node is worthy of some elaboration here as it incorporates a number of useful error checks. DAVE first compares argument and parameter lists for agreement in length and type. Lack of agreement is considered an error. Next, parameter output classifications are compared to arguments. If a parameter is classified as output or strict output and the corresponding argument is a constant, expression or subprogram name, DAVE produces a message. COMMON variables which carry data into or out of the invoked subprogram are identified at this time and messages describing them are made available for use as documentation. Finally, the variables in the invoking statement are examined to see whether any of them is used both as an input and an output in separate subprogram invocations. If so, DAVE has detected an illegal side effect, and produces a message identifying it.

After DAVE has searched the main program, it examines the input classifications of its COMMON variables. Error or warning messages are generated for each COMMON variable which is typed strict input or input but is not initialized in a BLOCK DATA subprogram.

IMPLEMENTATION DATA

DAVE is implemented as a Fortran program consisting of approximately 25,000 source statements. It operates in four overlaid phases, the largest of which occupies 50,000 decimal words of central memory on the CDC 6400. DAVE is written almost entirely in

machine independent ANSI Standard Fortran. Some non-Standard and machine dependent coding practices seemed expedient, however, and they are quarantined to a small number of small subprograms. DAVE was developed on the CDC 6400 at the University of Colorado, but has been successfully moved to a CDC 7600 and two machines in the IBM 360/370 series. Installations on a Univac 1100 series machine and a Honeywell 6000 series machine are planned for the near future.

Under its current configuration DAVE is able to process a program consisting of a few dozen subprograms, each of which may contain no more than 200-250 source statements. These limits depend entirely upon internal table and scratch array sizes, and have been quickly altered to produce different experimental configurations. At this writing, the largest body of code which DAVE has processed has been a 2700 source statement subprogram library. DAVE is currently operational, however, on a machine with sufficient central memory to enable it to process its own source code, and this should be accomplished in the near future. The analysis of a source program by DAVE has been observed to require an average of 0.3 seconds of central processor time per source statement on the CDC 6400 and to cost approximately six to eight cents per source statement under the University of Colorado Computing Center charge algorithm.

SUMMARY OF EXPERIENCE TO DATE

DAVE has been operational on an experimental basis for a few months to date. During this time we have seen evidence that it can be a valuable tool in the production of high quality, error-free programs. Most of our experience has come from using DAVE in validating completed programs. These included a highly respected matrix manipulation system, several recent algorithms taken from the *ACM Transactions on Mathematical Software*, and a program submitted as a part of a Master's Thesis in Computer Science. Errors were detected in some of the algorithms, and the Master's Thesis was found to have numerous errors. In most cases the errors were of the type that would hamper program portability, such as reference to uninitialized variables which should have been initialized to zero, reference to exhausted DO loop indices and subprogram invocations with mismatched argument and parameter lists. In each case, the errors did not seem to prevent successful execution on the author's computer, but seemed likely to cause trouble if executed elsewhere. (There was some suspicion, however, that some of the erroneous subprogram invocations were imbedded in program segments which had never been tested or were unexecutable.)

Perhaps the most surprising observation was that DAVE's messages often gave unexpected insight into the author's coding style. For example, a program for which DAVE produced numerous type 2 anomaly warning messages did not prove to be incorrect, but rather it contained numerous loops in which indices and counters were updated immediately before DO loop endings. It was discovered that the author tended to favor WHILE loop constructions which are often awkward in Fortran. This was observed by DAVE. As another example, some programs contained subroutine definitions which did not use some parameters either as input or output. This was observed to be a symptom of the fact that the code had evolved, but not been carefully polished. We acknowledge that in the first case the author should probably have coded in a more comfortable language, and in the second the program was probably not thoroughly designed before coding began. DAVE can do nothing to prevent these serious breaches of good programming practice. It was surprising and gratifying, however, to discover that DAVE could often strongly indicate their presence—a capability which we believe is quite useful.

We have had less experience in using DAVE as an error detection aid during program development. This seems paradoxical because we feel that DAVE is very well suited to aiding the detection of subtle errors, thereby speeding program development. The high cost of using and the awkwardness in accessing the current version of DAVE, however, forestalled its use in many cases. DAVE's accessing procedures have recently been streamlined, but the high cost of using the system is attributable to a decision made during development of the prototype system to favor flexibility over efficiency. Hence high costs are likely to remain for the foreseeable future. As a consequence of this, the few programs which DAVE helped to debug all had subtle errors which had defied earlier concerted efforts at detection. DAVE was usually able to point rather directly at these. Such errors as camouflaged misspellings (e.g., CARD instead of CARDS) and mismatched argument/parameter lists were discovered in this way.

Our experience has not been entirely positive. An obvious and troublesome difficulty is DAVE's copious output. As already illustrated, a single error often generates numerous messages. Moreover, we have observed that some messages are rarely symptoms of errors. The net effect is that human analysts are often reluctant to pursue all of DAVE's messages, thereby raising the possibility that errors whose symptoms have been detected will go unnoticed. DAVE users have also complained about the unclear wording of many messages. All of these human interface problems must be solved lest DAVE's useful capabilities be buried under an avalanche of opaque verbiage.

PROBLEM AREAS AND FUTURE WORK

We consider the current DAVE system to be a working prototype. Consequently, as might be expected, it has neither the speed nor complete processing capabilities which might be expected of a polished system.

The purpose of this section is to describe the areas in which we feel DAVE is deficient and to indicate where and how improvements might be made.

One of the most immediate problems is that DAVE was designed to analyze only programs written in ANSI Standard Fortran. DAVE has since been liberalized to accept most of the Fortran dialects available on CDC equipment. Little effort, however, has been devoted to the problems of accepting other dialects. Many of the changes required in order to accept such dialects appear to be straightforward, but it is worthwhile to note that some features of some dialects (e.g., the ENTRY feature found in FORTRAN V[19]) cannot be properly analyzed by DAVE without substantial alterations.

More serious is the fact that there are a number of features of ANSI Standard Fortran which are currently incorrectly or inadequately handled by DAVE. A notable and discouraging example of this is the treatment of arrays. Currently DAVE treats all arrays as simple variables, thereby blurring all distinctions between array elements and eliminating the possibility of detecting certain anomalous uses of the individual elements. Unfortunately, there are fundamental theoretical reasons why patterns of array references in an arbitrary program can never be completely analyzed by a static analysis system such as DAVE.

As already noted, the call graph of a Fortran program may not be acyclic even though the program is incapable of ever executing a recursive calling chain. Such a program cannot be analyzed by DAVE. The most promising solution to this problem seems to be to adapt DAVE so that it is able to analyze recursive programs. This is an interesting and worthwhile problem which seems solvable, and would move DAVE in the direction of being able to analyze programs written in other languages such as ALGOL and PL/I.

DAVE is also currently unable to build the complete call graph for programs in which subprogram names are passed as parameters. Hence DAVE cannot analyze such programs. An algorithm due to Kallal and Osterweil[20] is capable of building the call graph of such a program. This algorithm will probably be incorporated into future versions of DAVE.

Other problems are encountered by DAVE in trying to analyze programs containing extensive or tricky uses of aliasing constructs such as EQUIVALENCE statements and restructured COMMON lists. Most of these will be overcome in future versions of DAVE by using well-known compiling techniques. Others, such as using two EQUIVALENCE'd variables as arguments to the same subprogram, challenge some of DAVE's basic assumptions, and may never be satisfactorally solved.

Programs in which variables become undefined (e.g., the exhaustion of a DO loop causes the undefinition of the DO index) may, under certain unusual circumstances, be incorrectly analyzed. This results from our tardy recognition that variables must be typed with respect to undefinition just as they are typed with respect to input and output (i.e., they must be typed as non-undefined, undefined, or strict undefined). We have developed algorithms for performing and correctly employing this typing of undefinition, but have not yet incorporated them into DAVE.

Finally, we are actively exploring the relationship between static testing aids and global program optimization. Our investigation[21] has shown that existing algorithms in global optimization can readily be harnessed to do much of the analysis done by DAVE. Hence we foresee the incorporation of systems such as DAVE into a future generation of compilers.

REFERENCES

1. Dijkstra, E. W., "Notes on Structured Programming," in *Structured Programming*, by O. J. Dahl, E. W. Dijkstra and C. A. R. Hoare, Academic Press, London and New York, 1972.
2. Wirth, N., "Program Development by Stepwise Refinement," *CACM* 14, pp. 221-227, April 1974.
3. Mills, H. D., "Top-Down Programming in Large Systems," in *Debugging Techniques in Large Systems*, R. Rustin (ed.), Prentice-Hall, Englewood Cliffs, N.J., 1971, pp. 41-45.
4. Wirth, N., "An Assessment of the Programming Language PASCAL," *IEEE Transactions on Software Engineering*, SE-1, pp. 192-198, June 1975.
5. Gannon, J. D. and J. J. Horning, "Language Design for Program Reliability," *IEEE Transactions on Software Engineering*, SE-1, pp. 179-191, June 1975.
6. Mills, H. D., "How to Write Correct Programs and Know It," *Proceedings of the 1975 International Conference on Reliable Software*, IEEE Cat. No. 75CH0940-7CSR, pp. 363-370.
7. Baker, F. T., "Chief Programmer Team Management of Production Programming," *IBM Systems Journal*, 11, pp. 56-73, 1972.
8. Good, D. I., R. L. London and W. W. Bledsoe, "An Interactive Program Verification System," *IEEE Transactions on Software Engineering*, SE-1, pp. 59-67, March 1975.
9. Balzer, R. M., "EXDAMS: Extendable Debugging and Monitoring System," *AFIPS 1969 SJCC*, 34 AFIPS Press, Montvale, N.J., pp. 567-580.
10. Fairley, R. E., "An Experimental Program Testing Facility," *Proceedings of the First National Conference on Software Engineering*, IEEE Cat. No. 75CH0992-8C, pp. 47-52.
11. Stucki, L. G., "Automatic Generation of Self-Metric Software," *Proceedings of the 1973 IEEE Symposium on Computer Software Reliability*, IEEE Cat. No. 73C40741-9CSR, pp. 94-100.
12. Ramamoorthy, C. V. and S-B. F. Ho, "Testing Large Software with Automated Software Evaluation Systems," *IEEE Transactions on Software Engineering*, SE-1, pp. 46-58, March 1975.
13. Miller, E. F., Jr. and R. A. Melton, "Automated Generation of Testcase Datasets," *Proceedings of the 1975 International Conference on Reliable Software*, IEEE Cat. No. 75CH0940-7CSR, pp. 51-58.
14. Krause, K. A., R. W. Smith and M. A. Goodwin, "Optimal Software Test Planning Through Automated Network Analysis," *Proceedings of the 1973 IEEE Symposium on Computer Software Reliability*, IEEE Cat. No. 73C40741-9CSR, pp. 18-22.
15. American National Standards Institute, *FORTRAN*, ANSI X3.9, 1966.

16. Schaefer, M., *A Mathematical Theory of Global Program Optimization*, Prentice-Hall, Englewood Cliffs, N.J., 1973.
17. Osterweil, L. J. and L. D. Fosdick, *Data Flow Analysis as an Aid in Documentation, Assertion Generation, Validation and Error Detection*, University of Colorado Department of Computer Science Technical Report No. CU-CS-055-74.
18. Osterweil, L. J. and L. D. Fosdick, "DAVE—A Validation, Error Detection, and Documentation System for Fortran Programs," *Software Practice & Experience* (to appear).
19. Univac Division of Sperry Rand Corporation, *UNIVAC 1108 FORTRAN V*, Cat. No. UP-4060 Rev. 1, Sperry Rand Corporation, pp. 4-18 to 4-19.
20. Kallal, V. and L. J. Osterweil, (to appear as University of Colorado Department of Computer Science Technical Report).
21. Fosdick, L. D. and L. J. Osterweil, "Validation and Global Optimization of Programs," *Proceedings of the Fourth Texas Conference on Computing Systems*, 1975.