Experiments with a symbolic evaluation system

by WILLIAM E. HOWDEN
University of California at San Diego
La Jolla, California

ABSTRACT

Symbolic evaluation techniques can be used to determine the cumulative effects of a program's calculations on the branching predicates and output variables in the program. If the evaluation techniques are carefully and selectively applied, they can be used to generate revealing symbolic representations of the computations carried out by the paths in a program, and of the systems of predicates that describe the input data that causes program paths to be executed. A symbolic evaluation system called DISSECT is described which can be used to analyze FORTRAN programs. The system includes a sophisticated command language that allows the user to selectively apply symbolic evaluation techniques to different program paths and subpaths. The command language allows the user to carry out different levels of symbolic testing of a program and to construct systems of predicates that can be used to automate the generation of numeric test data. Experiments with the system which illustrate its advantages and limitations are included. DISSECT can be used to carry out a systematic, documented reliability analysis of a program. The paper concludes with a discussion of the potential use of systems like DISSECT as the basic software certification tool in the software development process.

INTRODUCTION

When a program path is executed by running the program on a given input, the correctness of the path for that input can be determined by examining the effects of the calculations carried out by the path. If the path is executed "symbolically" rather than with actual data, it may be possible to use a single execution to illustrate its correctness on a large subset of the input domain rather than on just a single value. Symbolic execution of a program is carried out by giving dummy symbolic values rather than actual numeric (string, logical etc.) values to all or some of the input variables of the program. An expression in the program involving variables with symbolic values is evaluated by substituting the current symbolic values of the varia-

bles into the expression. The resulting expression is then simplified algebraically. All operators having only actual as opposed to symbolic operands are evaluated in the normal way. The resulting expression is the symbolic value of the original expression.

Figure 1 contains a program for carrying out polynomial interpolation.[1] The documentation for the program describes it as consisting of four segments, each of which computes part of the interpolation process. Lines 34 through 48 are supposed to compute a set of coefficients a_i which are given in terms of y_i and Δ_i by the formula:

$$a_k = \frac{y_k}{\prod\limits_{i=1}^{k-1} (\Delta_k - \Delta_i)} - \sum\limits_{j=1}^{k-1} \frac{a_j}{\prod\limits_{i=j}^{k-1} (\Delta_k - \Delta_i)}$$

The program is written so that a_k corresponds to $A(k)$, Δ_i to DELTA(i) and y_i to $Y(I1+i-1)$. Symbolic values are designated by alphanumeric strings surrounded by quotes. Suppose "I1," "DELTA (I)" I=1, 10 and 2 are assigned to I1, DELTA(I) I=1,10 and NTERMS at statement 36 and "A(1)" to A(1) at statement 37. Then the effect of the calculations carried out in a single iteration of the loop in lines 34 through 47 can be determined by symbolically executing the code and printing out the values of A(1) at statement 36 and A(k) at statement 46. Figure 2 contains these symbolic values. The quotes designating symbolic values are deleted from the output to increase readability whenever this is unambiguous. Inspection of the values reveals that they agree with the formula and that this part of the program is correct for all input data having NTERMS=2.

Symbolic evaluation can be used to generate symbolic representation of the effects of the calculations carried out by paths in a program. It can also be used to generate sets of predicates in input variables which describe the input data that causes different program paths to be executed. A "complete" symbolic evaluation analysis of a program can be used as the validation documentation for the program.

Certain features must be present in an automated symbolic evaluation system in order for the system to be useful in analyzing realistic programs. The user

```
        SUBROUTINE INTERP(X, Y, NPTS, NTERMS, XIN, YOUT) #0
        DOUBLE PRECISION DELTAX, DELTA, A, PROD, SUM #1
        DIMENSION X(1), Y(1)                           #2
        DIMENSION DELTA(10), A(10)                     #3
C                                                      #4
C       SEARCH FOR APPROPRIATE VALUE OF X(1)           #5
C                                                      #6
     11 DO 19 I=1, NPTS                                #7
        IF (XIN—X(I)) 13, 17, 19                       #8
     13 I1=I—NTERMS/2                                  #9
        IF (I1) 15, 15, 21                             #10
     15 I1=1                                           #11
        GO TO 21                                       #12
     17 YOUT=Y(I)                                      #13
     18 GO TO 61                                       #14
     19 CONTINUE                                       #15
        I1=NPTS—NTERMS+1                               #16
     21 I2=I1+NTERMS—1                                 #18
        IF (NPTS—I2) 23, 31, 31                        #19
     23 I2=NPTS                                        #20
        I1=I2—NTERMS+1                                 #21
     25 IF (I1) 26, 26, 31                             #22
     26 I1=1                                           #23
     27 NTERMS=I2—I1+1                                 #24
C
C       EVALUATE DEVIATIONS DELTA                      #26
C                                                      #27
     31 DENOM=X(I1+1)—X(I1)                            #28
        DELTAX=(XIN—X(I1))/DENOM                       #29
        DO 35 I=1, NTERMS                              #30
        IX=I1+I—1                                      #31
        DELTA(I)=(X(IX)—X(I1))/DENOM                   #32
     35 CONTINUE                                       #32.1
C                                                      #33
C       ACCUMULATE COEFFICIENTS A                      #34
C                                                      #35
     40 A(1)=Y(I1)                                     #36
     41 DO 50 K=2, NTERMS                              #37
        PROD=1.                                        #38
        SUM=0.                                         #39
        IMAX=K—1                                       #40
        IXMAX=I1+IMAX                                  #41
        DO 49 I=1, IMAX                                #42
        J=K—I                                          #43
        PROD=PROD * (DELTA(K)—DELTA(J))                #44
     49 SUM=SUM—A(J)/PROD                              #45
        A(K)=SUM+Y(IMAX)/PROD                          #46
     50 CONTINUE                                       #47
C                                                      #48
C       ACCUMULATE SUM OF EXPANSION                    #49
C                                                      #50
     51 SUM=A(1)                                       #51
        DO 57 J=2, NTERMS                              #52
        PROD=1.                                        #53
        IMAX=J—1                                       #54
        DO 56 I=1, IMAX                                #55
     56 PROD=PROD * (DELTAX—DELTA(I))                  #56
     57 SUM=SUM+A(J)*PROD                              #57
     60 YOUT=SUM                                       #58
     61 RETURN                                         #59
        END                                            #60
```

Figure 1—Interpolation subroutine

A(1) = Y(I1)
A(2) = (Y(I1+1)/(DELTA(2) —DELTA (1)))
    — (A(1)/(DELTA(2) —DELTA (1)))

Figure 2—Symbolic values for A(1) and A(2)

must be able to easily set up and carry out a number of different analyses. He must be able to select sub-segments of a program and individual paths or classes of paths. The system must contain facilities for assigning actual or symbolic values to variables and for printing out values of variables at different points in a program. The user may also wish to print out systems of predicates formed by symbolically evaluating the branch conditions in a path and to check the consistency of these systems.

The remaining sections of the paper describe a sophisticated symbolic evaluation system called DISSECT and a number of experiments using the system. DISSECT can be used to analyze programs written in ANSI Standard FORTRAN. In the experiments the system is used to analyze two complex statistical routines taken from Reference 1.

THE DISSECT SYSTEM

(a) Structure of DISSECT—The DISSECT system operates on two input files and produces an output file. One of the input files contains a FORTRAN program to be analyzed and the other contains the DISSECT commands which tell the system what kind of symbolic evaluation analysis to carry out. The output file contains the results of the analysis.

The command file for a DISSECT analysis is divided into a number of *cases*. The program in the input file is processed completely for each case. Each case has a section for a text description of the part of the program to be analyzed by the case. This text is not processed by the system and is considered to be the specification for the case, The rest of a case contains commands which identify the part of the program which corresponds to the case, commands which assign (symbolic) values to variables and commands which specify what output is to be generated.

The output file which is generated by a DISSECT analysis is also divided into cases. Each output case contains the case specifications and commands as well as the output generated by the system for that case. The user can check the validity of his program by comparing case specifications with system output for cases.

A program *path* is a possible flow of control through the program. A path is *feasible* if at least one element of the program's input domain causes that path to be executed. In general, a complete set of DISSECT cases for a program should "cover" the program in some sense. One approach is to analyze each feasible path (up to some number of iterations of loops). Complex

programs having many paths can be divided into segments and analyzed using separate cases.

(b) DISSECT commands—The DISSECT commands can be divided into three groups: path selection, output and value commands. The commands can either be given in the *global commands section* of a DISSECT command file or can be given as part of a case. If they appear in the global commands section they apply to all cases. The commands can be used with or without a statement sequence number. In general, when used with a sequence number they are executed only when the system is evaluating that statement. Otherwise they are used for every statement or, in some cases, are only applied at the end of the DISSECT analysis of a program path.

(c) Path Selection Commands—DISSECT processes a case by symbolically evaluating one or more paths in a program. The path selection commands cause DISSECT to select part of a program for analysis by directing it to follow certain paths through a program. The "SELECT" command is used for directing DISSECT to follow a specified branch or branches from a conditional branching statement. The "LOOP" command directs DISSECT to carry out a given number of iterations of a loop. Loops are specified by naming the statement number of the first statement in the loop.

Examples

(i) n SELECT .GT. will cause DISSECT to select the .GT. branch from an arithmetic conditional statement n.
(ii) n SELECT ALL will cause DISSECT to set up a subpath for each branch from conditional statement n.
(iii) n LOOP k will cause DISSECT to iterate loop n k times.
(iv) n SKIP m will cause DISSECT to skip from statement n to m during processing of a program. SKIP can be used to set up cases which only deal with paths through segments of a program rather than the whole program.

(d) Output commands—The user of DISSECT can generate several different kinds of output. The PATH command causes the system to print out the sequence numbers of the statements in the paths in a case. PATH DESCRIPTION will cause the output of all of the statements in the paths. PREDICATES will result in the construction of symbolically evaluated systems of predicates which describe the input that causes paths to be followed. The OUTPUT command can be used to print out the symbolic values of variables, symbolically evaluated subroutine calls and symbolically evaluated program output statements. If an output command is preceded by a statement sequence number then the command is invoked when the system encounters that statement during its symbolic evalua-

tion of the program. If the command is not preceded by a statement number then, in general, it is invoked after the completion of any path belonging to the case containing the output command.

(e) Value commands—The most important value assigning command in DISSECT is the ASSIGN command. ASSIGN can be used to assign either actual or symbolic values to any variable at any point in the processing of a program. The DISSECT system is designed so that symbolic values are automatically assigned to variables whenever the variables appear in SUBROUTINE headers or COMMON or input statements. The automatically assigned symbolic values are text strings formed from the variable names. ASSIGN can be used to override these default symbolic value assignments or to assign actual values.

ASSIGN is used in basically two different ways. In many situations, the user will want to carry out a symbolic evaluation of a path in which some of the variables are given actual values. He can use ASSIGN to give these values to the variables at the beginning of the path. In other situations a complicated segment of calculations on several variables divides naturally into several segments and a user will want to print out the symbolic values of one or more variables at the end of one segment and then reset the values of the variables to simple symbolic values before continuing with the processing of the next segment. The ability to do this avoids the necessity of having to analyze complicated symbolic expressions resulting from the symbolic execution of several conceptually distinct sequences of operations.

(f) Conditional Execution of commands—There are two ways in which a user can specify that a DISSECT command is to only be applied under certain conditions. The first involves the use of a *conditional form*. If a command appears in the form "IF condition THEN command 1 [ELSE command 2]" then command 1 is only carried out if the condition holds (and command 2 if the condition does not hold). Conditional forms are not commands and cannot be nested.

Conditions are constructed using three types of expressions. The first consists of ordinary program expressions in program variables. The second consists of the special variable ATTRIBUTE and the third the special function LOOPCOUNT(n). DISSECT contains a number of commands for attaching *attributes* to paths during their symbolic evaluation. It is possible to specify that the execution of a DISSECT command is conditional upon association of a given attribute with a path. LOOPCOUNT(n) returns an integer giving the number of times that loop n has been iterated by the path currently being traversed. When the function is called it is assumed that the statement currently being processed by DISSECT is in the loop and that LOOPCOUNT(n) is the number of iterations that have been completed during the current traversal of the loop.

The second way of conditionally carrying out a DISSECT command involves the use of the CONSISTENCY option. Many of the DISSECT commands can be used with several flags. When the CONSISTENCY flag is attached to a command the system constructs the predicate that would be added to the system of predicates for the current path if the command were to be executed. If the addition of this predicate to the system would cause the system to be inconsistent then the command is not executed. The CONSISTENCY option can be used to stop DISSECT from traversing program paths that would result in the generation and analysis of infeasible program paths. The consistency routines which are currently implemented are very simple. Although they will catch only certain kinds of inconsistencies they have proved to be very powerful.

A related feature in DISSECT is the DEFAULT option. DISSECT expects that certain kinds of program statements will always have a command associated with them. It expects the command file to contain, for example, path selection commands for each conditional branching statement which it encounters when it is processing a program path. In general, a user may only want to construct commands for a fraction of the branching statements in a program. He can "cover" the remaining branching statements by constructing selection commands which have no statement number and which include the DEFAULT option. When DISSECT reaches a conditional statement it first looks to see if there are any selection commands for the statement which do not have the DEFAULT option set. If there are none it then tries to find a selection command that it can apply which has the DEFAULT option set. Recall that DISSECT commands which do not have statement numbers are applied by DISSECT to all appropriate statements during the processing of the program.

EXAMPLES

(a) Interpolation Example—The first example describes the use of DISSECT in analyzing the interpolation program in Figure 1. In both this and the next example DISSECT was used to confirm that the program agreed with its specifications. The INTERP routine is written so that the number of points NTERMS used in the interpolation process may be less than the number of points available (NPTS). The first segment of the routine, lines 1-25, decides which points to use. It involves choosing a value for I1. The points $X(I1)$, $X(I1+1)$, $\ldots$, $X(I1+n-1)$ in the program correspond to the points x_1, x_2, $\ldots$, x_n in the documented formulae ($n=\min\{NPTS, NTERMS\}$). The documentation states that x_1 (i.e. $X(I1)$) "is chosen so that x_1 and x_n straddle x". "If the value of x is too close to the lower or upper limit of the values of x_i, the corresponding value of x_1 or x_n is set equal to the limiting value."

The documentation for this segment of the program is quite vague. A casual reading of the program reveals that the segment is of some complexity and has a number of paths. The process that is to be carried out by the part of the program appears to be typical of the types of processes that may not work for limiting values in the input. It was decided to examine the paths through this section of code for NPTS=1 and 2 (the limiting cases) and also for NPTS=3. The DISSECT command file for those three cases is given in Figure 3.

These three cases cause DISSECT to analyze all paths in the program up to statement 28 for NPTS= 1, 2 and 3. Case 1 has 10 paths, case 2 17 paths and case 3 24 paths. This is a large number of paths but it was found that the output for each was easy to read and that it was easy to determine if the program was correct for the case. The output for each case is divided into subcases, several of which are reproduced in Figure 4. Each subcase corresponds to a path.

The structure of the code in the first segment of the program is such that the complete set of paths for cases one, two and three indicates that the segment is correct. Note that we have not formally proved that the segment is correct. The symbolic predicates and values which are produced assist the user in reading the code and in carrying out a proof which is partly formal and partly informal.

The second segment of DISSECT, lines 26-33 is supposed to compute the following values of Δ and Δ_i.

$$\Delta = \frac{X - X_1}{X_2 - X_1} \qquad \Delta_i = \frac{X_i - X_1}{X_2 - X_1}$$

The correctness of this segment is easily confirmed with two or three simple cases. The commands for the case where NPTS=2 are given in Figure 5.

The output for this case is given in Figure 6. In the program Δ is represented by DELTAX. In reading the output recall that x_i is represented by $X(I1+i-1)$.

The Case 6 command ASSIGN I1="I1" ensures that the output for Case 6 will use the symbol I1 to stand for I1 rather than the value it may have as the result of earlier calculations.

Symbolic output for the code in the third segment, lines 34 through 48, appears in Figure 2. The segment was thoroughly examined by looking at the output for the cases where NPTS=3 and 4.

The final segment of the program, lines 49 through 60, is supposed to compute the formula:

$$y(x) = a_1 + \sum_{j=2}^{n} \left[a_j \prod_{i=1}^{j-1} (\Delta - \Delta_i) \right].$$

This segment can be checked by constructing cases corresponding to $n=1$, 2 and 3. The case containing the commands for $n=2$ appears in Figure 7. The program uses the variable YOUT to represent the value $y(x)$. The output for the case appears in Figure 8.

(b) Correlation Example—The complexity of the INTERP routine is due both to its control logic and its array manipulation operations. DISSECT was useful in analyzing INTERP by allowing the user to look at how the program operated for arrays of given dimensions.

In the correlation example, DISSECT was used to

```
TITLE: ANALYSIS OF INTERP
GLOBAL COMMANDS:
   MAXPATHS 300;
   MAXLENGTH 200;
   DEFAULT LOOP ANY CONSISTENT (1-10);
   DEFAULT SELECT ANY CONSISTENT;
   PATH; PREDICATES;

CASE 1: ANALYSIS OF SEGMENT OF CODE THAT DETERMINES I1.
   SET NPTS=1
CASE COMMANDS:
   OUTPUT I1, NTERMS;
   SELECT ALL;
 7 ASSIGN NPTS=1;
28 HALT;

CASE 2; SET NPTS=2.
CASE COMMANDS:
   OUTPUT I1, NTERMS;
   SELECT ALL;
 7 ASSIGN NPTS=2;
28 HALT;

CASE 3: SET NPTS=3.
CASE COMMANDS:
   OUTPUT I1, NTERMS;
   SELECT ALL;
 7 ASSIGN NPTS=3;
28 HALT;
```

Figure 3—DISSECT commands for first segment of INTERP

CASE 1.1
PATH: 0-12 18-24

PREDICATES:
: 1   0   SUBROUTINE INTERP(X,Y,NPTS, NTERMS, XIN, YOUT)
: 7   8   XIN−X(1) .LT. 0
: 9   10   1−(NTERMS/2) .LE. 0
:13   19   1−NTERMS .LT. 0
:16   22   2−NTERMS .LE. 0

OUTPUT:
:18   28   I1=1
             NTERMS=1

CASE 2.3
PATH: 0-12 18-19

PREDICATES:
: 1   0   SUBROUTINE INTERP(X,Y,NPTS, NTERMS, XIN, YOUT)
: 7   8   XIN−X(1) .LT. 0
: 9   10   1−(NTERMS/2) .LE. 0
:13   19   2—NTERMS .GE. 0

OUTPUT:
:13   28   I1=1
             NTERMS=NTERMS

CASE 3.11
PATH: 0-8 15 7-10 18-24

PREDICATES:
: 1   0   SUBROUTINE INTERP(X,Y,NPTS, NTERMS, XIN, YOUT)
: 7   8   XIN−X(1) .GT. 0
:11   8   XIN−X(2) .LT. 0
:13   10   2−(NTERMS/2) .GT. 0
:15   19   2+NTERMS/2−NTERMS .LT. 0
:18   22   4−NTERMS .LE. 0

OUTPUT:
:20   28   I1=1
             NTERMS=3

Figure 4—DISSECT output for first segment of INTERP

CASE 6: TEST SECOND SEGMENT WITH NPTS=4.
CASE COMMANDS:
     OUTPUT, DELTAX, DELTA;
  7 SKIP 28;
 28 ASSIGN NTERMS=4;
 28 ASSIGN I1="I1";
 36 HALT

Figure 5—Case commands for symbolic evaluation of second
segment

CASE 6.1
PATH 0-3 28-32.1 30-32.1 30-32.1 30-32.1 30

PREDICATES:

OUTPUT:
: 1   0   SUBROUTINE INTERP(X,Y, NPTS, NTERMS, XIN, YOUT)
: 6   28   **ASSIGN I1=I1
:29   36   DELTAX=(XIN−X(I1))/(X(I1+1)−X(I1))

             DELTA(4) = (X(I1+3)−X(I1))/(X(I1+1)−X(I1))
             DELTA(3) = (X(I1+2)−X(I1))/(X(I1+1)−X(I1))
             DELTA(2) = (X(I1+1)−X(I1))/(X(I1+1)−X(I1))
             DELTA(1) = (X(I1)−X(I1))/(X(I1+1)−X(I1))

Figure 6—Symbolic output for analysis of second segment

CASE 9: ANALYSIS OF SEGMENT FOR COMPUTING FINAL VALUE OF Y.
    NTERMS=2.

CASE COMMANDS:
    OUTPUT YOUT;
  7 SKIP 51;
 51 ASSIGN NTERMS=2, A="A", DELTAX="DELTAX";
 51 ASSIGN DELTA="DELTA";

Figure 7—Case commands for analyzing last segment of program

analyze a program whose complexity is entirely due to its control logic and to the computations it carries out. The program has no arrays and is not complicated by looping mechanisms for carrying out array operations.

The PCORRE routine is used to determine the probability $P_c(r,N)$ that N random data points would yield a linear-correlation coefficient as large or larger than an observed correlation value $|r|$. The documentation for PCORRE lists two formulae which are supposed to be computed by the routine. For $v=N-2$, one of the formula is for v even and the other for v odd. The formula for v even is

$$P_c(r,N) = 1 - \frac{2}{\sqrt{\pi}} \frac{\Gamma[(v+1)/2]}{\Gamma(v/2)}$$

$$\left\{ \sum_{i=0}^{I} \left[(-1)^i \frac{I!}{(I-i)!i!} \frac{|r|^{2i+1}}{2i+1} \right] \right\}$$

where $I = \frac{v-2}{2}$ and Γ is the gamma function. The formula for v odd is equally complex.

The PCORRE routine divides naturally into three segments. The first determines the quantity v and whether it is even or odd. The second and third compute $P_c(r,N)$ for even and odd v. The part of the code that computes $P_c(r,N)$ for v even is reproduced in Figure 10.

A command file containing the case in Figure 11 was constructed for analyzing PCORRE. The output for one of the subcases which analyzes the first two segments of the routine is reproduced in Figure 12.

The text strings in quotes after some of the commands in Figure 11 are attributes. Each time a path follows a branch associated with a particular SELECT command, the path is assigned any attributes listed for that branch. The complete collection of attributes for a path is printed along with the output for the path. The use of attributes makes it easy to identify paths in terms of particular properties associated with the branches in the paths.

CASE 9
PATH: 0-3 51-56 55 57 52 58-59
PREDICATES:
OUTPUT:
  : 6 51 **ASSIGN A=A
  : 7 51 **ASSIGN DELTAX=DELTAX
  : 8 51 **ASSIGN DELTA=DELTA
  :21 59 YOUT=A(1)+A(2) * (DELTAX-DELTA(1))

Figure 8—Output for Case 9

It is evident from this output that the formula is correct for the case where v is even and $(v-2)/2=1$ DISSECT was also used to generate symbolic out for the cases where $(v-2)/2=2$ and $(v-2)/2=3$. This output taken together with the pattern of the code in the program, indicates that the program computes the correct formula for v even. DISSECT was also used to generate symbolic output for the code which calculates $P_c(v,N)$ for v odd.

RELATED WORK

The DISSECT system is built on an earlier path analysis system[2] which can be used to generate descriptions of the sets of input data that cause classes of program paths to be executed. The major improvement is the addition of a command language which allows the user to selectively control the application of the analysis routines. In addition, DISSECT has many features not present in the previous system.

Several systems have been constructed which can be used to carry out program analysis similar to those which can be carried out with DISSECT. The EFFIGY,[3] SELECT,[4] RXVP[5] and the system described by Clarke in Reference 6 all allow the selection and evaluation of paths in computer programs. The DISSECT system is closest to the EFFIGY and SELECT systems. Symbolic evaluation can also be used in constructing program proofs.[7,8]

DISSECT is unique in its use of cases to structure the validation analysis of a program. It is also the only system that includes a language that allows a user to write simple analysis procedures (the command language). The EFFIGY and SELECT systems are path-following procedures which the user directs by selecting conditional statement branches interactively.

CONCLUSIONS

In the experiments with DISSECT which are described above, the user was faced with analyzing and determining the validity of a program he had not seen before. He was provided with the program and documentation and no outside help. The pattern of usage of DISSECT was one in which the user alternated between reading a program and running DISSECT. First the program and the documentation were studied and a preliminary command file constructed. The

```
     FUNCTION PCORRE (R,NPTS)                                   #1
     DOUBLE PRECISION R2, TERM, SUM, FI, FNUM, DENOM            #2
C                                                               #3
C        EVALUATE NUMBER OF DEGREES OF FREEDOM                  #4
C                                                               #5
  11 NFREE=NPTS-2                                               #6
     IF (NFREE) 13, 13, 15                                      #7
  13 PCORRE=0.                                                  #8
     GO TO 60                                                   #9
  15 R2=R**2                                                    #10
     IF (1.-R2) 13, 13, 17                                      #11
  17 NEVEN=2*(NFREE/2)                                          #12
     IF (NFREE-NEVEN) 21, 21, 41                                #13
C                                                               #14
C        NUMBER OF DEGREES OF FREEDOM IS EVEN                   #15
C                                                               #16
  21 IMAX=(NFREE-2)/2                                           #17
     FREE=NFREE                                                 #18
  23 TERM=ABS (R)                                               #19
     SUM=TERM                                                   #20
     IF (IMAX) 60, 26, 31                                       #21
  26 PCORRE=1.-TERM                                             #22
     GO TO 60                                                   #23
  31 DO 36 I=1, IMAX                                            #24
     FI=I                                                       #25
     FNUM=IMAX-I+1                                              #26
     DENOM=2*I+1                                                #27
     TERM=-TERM * R2 * FNUM/FI                                  #28
  36 SUM=SUM+TERM/DENOM                                         #29
     PCORRE=1.128379167 * (GAMMA((FREE+1.)/2.)/GAMMA(FREE/2.))  #30
     PCORRE=1.-PCORRE*SUM                                       #31
     GO TO 60                                                   #32
```

Figure 10—First two segments of PCORRE

CASE 1: MINIMUM CASE TO ILLUSTRATE THE PCORRE FORMULAE.
CASE COMMANDS:

```
        OUTPUT PCORRE
7       ASSIGN NFREE="V";
7,11    SELECT .GT.;
13      SELECT .GT. "DEGREES OF FREEDOM ODD.";
13      SELECT .LE. "DEGREES OF FREEDOM EVEN";
21,39   SELECT .LT. "SUMMATION MAX NEGATIVE—NO CALCULATION";
21,39   SELECT .EQ. "SUMMATION MAX ZERO—NO SUMMATION";
21,39   SELECT .GT. "SUMMATION MAX POSITIVE";
```

Figure 11—Case commands for analyzing PCORRE

CASE 1.1
ATTRIBUTES: SUMMATION MAX POSITIVE. DEGREE OF FREEDOM EVEN

PATH: 1-7 7 10-21 24-29 24 30-32 49

PREDICATES:

```
: 1    1   FUNCTION PCORRE(R,NPTS)
: 5    7   V .GT. 0
: 7   11   1.0-R**2 .GT. 0
: 9   13   V-2*(V/2) .LE.0
:14   21   ((V-2)/2) .GT. 0
:21   24   ((V-2)/2).GE.1
```

OUTPUT:

```
: 1    1   FUNCTION PCORRE (R,NPTS)
:26   49   PCORRE=1.0-1.128379*(GAMMA((1.0+V)/2.0)/
                   GAMMA(V/2.0))*ABS(R)+
                   1.128379*(GAMMA((1.0+V)/2.0)/
                   GAMMA (V/2.0)) * ((ABS(R)*R**2*((V-2)/2))/3)
```

Figure 12—Symbolic output for case 1.1 of analysis of PCORRE

command file was based on an initial understanding of the structure of the program. The output which was generated by DISSECT from the preliminary command file was often inadequate. In some cases, too many paths were generated for a piece of code. In other cases, more analysis of a piece of code, or analysis with different actual or symbolic assignments of variables, was needed. A second DISSECT analysis resulted in a better understanding of the program which sometimes prompted the user to carry out a third analysis.

The readability and usefulness of DISSECT output can be judged from the above examples. The output was generated from either the first or second command files which were constructed in the process of analyzing the programs. One or two additional versions of the command file would produce what would probably be the final version.

It is our experience that systems like DISSECT can be useful in two ways: The first is the help that the system can give the user in carrying out a validation analysis of a program. The output from a symbolic evaluation of a piece of code is often much more revealing than output from an execution with actual data. Similarly, the symbolic evaluation of a system of predicates associated with a path provides documentation describing the input associated with the path.

DISSECT can be useful in helping to "unravel" the computations carried out by different types of programs. Some programs are complicated to read because they are cluttered with the control structure needed to carry out an operation iteratively over the elements of a data structure. The validity of these programs can often be checked by looking at the manipulations that are carried out for structures of fixed sizes. DISSECT can be used to generate descriptions of these manipulations. Other programs are complicated to read because they contain control structure for constructing, and at the same time computing a value for an iterative formula. The number of iterations used in constructing the formula is often dependent on an input variable. DISSECT can be used to generate the instances of this formula that are computed by the program for different numbers of iterations. These instances, together with a knowledge of the structure of the iteration used by the program are often enough to convince the user that the program is correct. Examples 1 and 2 illustrate this use of DISSECT for programs of this type.

The second way in which DISSECT can be useful is in forcing a systematic, intuitively meaningful validation discipline on the programmer. DISSECT can be used as the basis of a validation methodology for a verification group. In verifying a program, the user must first break the specifications for the program down into cases. Each case is then described in terms of a case specification. The parts of the program which are supposed to take care of these cases are identified

using the command language. The system carries out the specified analysis and the user compares the output with the case specifications. After several rounds of improving the cases and possibly correcting program errors, a full validation document is produced. Experience may indicate that other types of validation analysis facilities should be added to the DISSECT without changing the basic structure of the system.

There are several situations in which systems like DISSECT may fail or not be useful. If a program is in error because some case was not coded into it, and no DISSECT case is constructed to correspond to this forgotten case, then the error may not be discovered. It is suspected that DISSECT may not be as useful for assembly language programs as it is for high-level language numeric programs. Since, in theory, a well-structured program is self-documenting and it is possible to understand the code by reading it, it might be argued that DISSECT will only be useful for programs written in languages like FORTRAN. This is only partly true. The observations made above about program complexity due to the mixing together of control structure for manipulating data structures and the computations to be carried out on the structures are true for most programming languages. The same thing is true of complexity due to mixing together the code which constructs an iteratively defined formula together with the computations for evaluating the formula.

Further experiments will be carried out with DISSECT and improvements to the system are planned. The system will be used to analyze programs known to contain errors in order to determine its usefulness in detecting bugs. Use of the system in a full-scale software development project is also being planned.

ACKNOWLEDGMENTS

REFERENCES

1. Benington, Philip R., *Data Reduction and Error Analysis for the Physical Sciences*, McGraw-Hill, 1969.

2. Howden, William E. and Jeffrey Laub, "Automatic Case Analysis of Programs," *Proceedings of Computer Science and Statistics: 8th Annual Symposium on the Interface*, Los Angeles, February, 1975.

3. King, James C., "A New Approach to Program Testing," *Proceedings of the International Conference on Reliable Software*, Los Angeles, April 1975.

4. Boyer, Robert S., Bernard Elspas and Kan N. Levitt, "SELECT—A Formal System for Testing and Debugging Programs by Symbolic Execution," *Proceedings of the International Conference on Reliable Software*, Los Angeles, April 1975.

5. Miller, E. F., "RXVP: An Automated Verification System for FORTRAN," *Proceedings of Computer Science and Statistics: 8th Annual Symposium on the Interface*, Los Angeles, February 1975.

6. Clarke, Lori, *A System to Generate Test Data and Symbolically Execute Programs*, Department of Comp. Science, University of Colorado, CU-CS-060-75, February 1975.

7. Deutsch, L. P., *An Interactive Program Verifier*, Ph.D. dissertation, University of California, Berkeley, May 1973.

8. Burstall, R. M., "Proving Correctness as Hand Simulation with a Little Induction," *Proceedings of IFIPS 74*, North Holland Publishing Company, 1974.