SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 8, 381–397 (1978)

An Evaluation of the Effectiveness of Symbolic Testing

WILLIAM E. HOWDEN
University of California, San Diego, California, U.S.A.

SUMMARY

The effectiveness in discovering errors of symbolic evaluation and of testing and static program analysis are studied. The three techniques are applied to a diverse collection of programs and the results compared. Symbolic evaluation is used to carry out symbolic testing and to generate symbolic systems of path predicates. The use of the predicates for automated test data selection is analysed. Several conventional types of program testing strategies are evaluated. The strategies include branch testing, structured testing and testing on input values having special properties. The static source analysis techniques that are studied include anomaly analysis and interface analysis.

Examples are included which describe typical situations in which one technique is reliable but another unreliable. The effectiveness of symbolic testing is compared with testing on actual data and with the use of an integrated methodology that includes both testing and static source analysis. Situations in which symbolic testing is difficult to apply or not effective are discussed. Different ways in which symbolic evaluation can be used for generating test data are described. Those ways for which it is most effective are isolated. The paper concludes with a discussion of the most effective uses to which symbolic evaluation can be put in an integrated system which contains all three of the validation techniques that are studied.

KEY WORDS Testing Symbolic testing Symbolic evaluation Test data generation Anomalies Specifications

INTRODUCTION

There has been an increasing interest in the past few years in the use of symbolic evaluation. Symbolic evaluation is useful for several different kinds of program analysis. It can be used to automate test data generation, to assist in the proof of correctness of programs and program paths and to carry out symbolic program testing.[1–10] A number of symbolic evaluation systems have been built which have a variety of capabilities.[2–5]

Research on symbolic evaluation, like that on other types of program testing and analysis methods, has concentrated on the design and implementation of tools which can be used to assist the user in the application of the methods. There has been relatively little research on the effectiveness of program testing and program analysis techniques. Exceptions include Gerhart and Goodenough's study of the errors in a string processing program,[11] Gerhart and Yelowitz's survey of errors in programs and programming methodologies,[12] analyses carried out by the author,[5, 13] and the series of experiments described by Hetzel in his PhD. thesis.[16] This paper describes the results of an analysis of the effectiveness of symbolic testing.

If the use of a program testing technique is guaranteed to always reveal the presence of a particular error in a program, then the technique is said to be reliable for the error. The project described in this paper involved an investigation of the reliability of symbolic testing for the errors in six selected programs. Although the number of programs which were

0038–6644/78/0408–0381$01.00 *Received 24 March 1977*

analysed is small the analysis which was carried out was thorough and the results are of general interest. The reliability of several techniques which involve testing on actual data and of several program analysis techniques which do not involve testing was also investigated. One of the goals of the project was to determine if the use of symbolic testing increases the reliability of the testing process in the sense that if symbolic testing is used in addition to testing on actual data then more errors are discovered than if testing is carried out on actual data alone. The use of symbolic testing in automatic test data generation was also investigated.

RESEARCH METHODOLOGY

Programs

Six programs were selected for analysis. All six programs have been previously studied in articles and books on testing, proving correctness and program debugging. Five of the programs contain naturally occurring errors. The sixth contains a seeded error. The six programs, together with the details of the results of the analyses, are contained in a more detailed report.[14]

The programs can be divided into three categories, each containing two programs.

Data processing

The first program is a COBOL validation program. It contains about 450 lines of code. The program appears in the book on debugging by Brown and Sampson.[15] The program validates transaction cards that record stock update information. The validated cards are used by another program to update the stock inventory file. There are a number of different possible transactions, each requiring a different validation subroutine. The program contains three errors.

The second program is a PLI updates program and is one of the examples used by Hetzel in his thesis.[16] It contains about 175 lines of code. The program reads in student exam scores from an updates file and uses the information to update a history file. It also prints out a number of reports during the updating process. The program contains 20 errors.

String processing

Two of the six programs are written in ALGOL and are string editing programs. They are typical of a class of programs which has been used in the literature to discuss structured programming.

The first ALGOL program appears in the paper 'An experiment in structured programming.[17] The program has 25 lines of code. The program reads a string of characters that form a sequence of telegrams. A telegram is a string that terminates with a special substring. The program isolates and prints out each telegram along with a word count. It contains one error.

The second ALGOL program appears in 'Programming by action clusters'[18] and is the program analysed by Goodenough and Gerhart.[11] It has 21 lines of code. The program is supposed to read a string of characters consisting of words separated by blanks and line break (NL) characters, and output as many words as possible on each line (in the same order) so that there is a separator between every two words, no word is broken between two lines, and each line has at most MAXPOS characters. The program contains two errors.

Utility routines

The remaining two programs will be classified as utility routines. The first of the two is a PL-360 sort program which has 13 lines of code. The program uses the familiar 'find the next largest' algorithm for sorting. The source of the program is an article on PL-360 by Wirth[19] and is one of the programs described by Gerhart and Yelowitz.[12] The program contains one error.

The second of the two utility routines is a program for computing the total, mean, standard deviation, minimum and maximum for each variable in an observation matrix. The program has 28 lines of code and appears in a study of program errors by Gould and Drongowski.[20] The program contains one error.

Reliability of testing on actual data

The first part of the analysis which was carried out for each program was designed to determine which of the errors in the program would be reliably discovered using one of several conventional types of testing strategies. Each of the conventional testing strategies requires the execution of programs on actual data. The conventional testing strategies whose reliability was analysed are described briefly below.

Path testing[13]

Path testing involves the testing of every path through a program at least once. Since most programs will have a very large, if not infinite, number of paths, the method is of theoretical rather than practical importance. It is useful to analyse how reliable path testing would be if it could be implemented since it provides an upper bound on the reliability of test techniques which involve the selection of a subset of the set of all program paths.

Branch testing[21–23]

The branch testing technique requires that each 'branch' in a program be tested at least once. It is one of the simplest and best known approaches to systematic testing.

Structured testing[24, 25]

The structured testing approach is an attempt to approximate path testing. It requires that the program be decomposed into a hierarchy of functional modules. Modules at the lowest level of abstraction are segments of code. Modules at the higher levels are mixed sequences of code and functional modules from the next lower level of abstraction.

In the structured testing approach all paths through a functional module which require less than or equal to k (usually $k = 2$) iterations of loops are tested at least once. Variations in this rule are necessary in special cases where this would leave parts of a module untested because of complicated loop indexing operations and dependencies between loop bounds.

It is possible to distinguish between two kinds of structured testing: *integrated* and *functional*. In functional testing individual functional modules are tested like separate programs. Input is constructed for each module and the final values of selected variables in the module are examined when the module terminates. In integrated testing functional modules are tested within the context of the entire program. For each functional module subpath that needs to be tested, a test for the entire program is constructed which causes that subpath to be traversed.

Special values testing

Special values testing is a test strategy which requires the testing of a program over input values having special properties which, experience indicates, are important to the testing process. Several useful special values rules were discovered during the project. Some of these are listed below. A more extensive research project might have the expansion and completion of this list as one of its goals.

(i) *Distinct values.* The distinct values rule requires that when a program contains a number of different input variables or arrays which are used for holding different kinds of, but related, data that the input values for the variables and arrays should be distinct (i.e. not equal). Elements of individual arrays should be distinct if possible and if two arrays store values from the same set, elements in corresponding array element positions should be distinct.

(ii) *Zero values.* The zero values rule requires that tests should be carried out which result in the assignment of zero values to variables which occur in arithmetic expressions.

(iii) *Input classes.* For many programs, the input divides naturally into several classes. String processing programs, for example, may distinguish between blanks, certain special words and non-blanks. Data processing programs may distinguish between several kinds of input records based on the values of one or more keys. The input classes rule requires that a program be tested on each of the different possible general classes of input to a program.

Special values testing can be used to refine the structured testing approach. In this combined method, special values are constructed for testing the paths which are generated by the structured testing approach.

Reliability of other program analysis techniques

Several of the errors in the sample programs either will not be discovered by testing on actual or symbolic data or their discovery by these methods appears to be unnatural and by chance. The discovery of some of these errors is more likely when program analysis techniques other than testing are used. The second part of the analysis which was carried out for each program involved an examination of the reliability of these other program analysis techniques for each of the sample programs. Several of these program analysis techniques are described below.

Anomaly analysis[26-28]

This static analysis technique involves the examination of a program for suspicious looking constructs. Two examples of the types of constructs it might report on are:

(i) *Size condition anomaly.* Suppose that a fixed point (PL1) variable appears on the left-hand side of an assignment inside a loop with a variable loop bound. There may be input data which would cause a size condition to occur for that occurrence of the variable.

(ii) *Array bounds anomaly.* Suppose that an array reference with a variable index is contained inside a loop and that the variable changes values in the loop (e.g. the loop index variable). If the loop bounds are variable and are different from the array bounds it may be possible for an out of bounds array reference to occur. The situation is particularly dangerous if the loop bounds are variable and the array bounds constant.

Specifications requirements

Some errors can be avoided if certain kinds of information are required to be included in program specifications. The 'variable ranges and types rule' requires that for each input variable the type of the variable and the allowable range of its values must be specified. The action to be taken by a program for values of the wrong type and outside the expected range must also be specified. If a program carries out conceptually different functions for different value subranges these must be specified.

Interface analysis

This term is used to denote rules for checking the interface between two routines or between a routine and the external environment. One example is a rule for checking the consistency of the format of structured input records with the format of the variables used to read in the structured records. The data base verifier described by Hodges and Ryan[29] is an example of an interface analysis tool.

Effectiveness of symbolic evaluation

The final part of the analysis which was carried out for each program was designed to investigate the effectiveness of symbolic testing for discovering errors and the use of symbolic evaluation in the automated generation of test data.

Symbolic testing[1–5]

A program is symbolically tested by symbolically evaluating selected program paths. The path selection strategy that was used in the analysis of symbolic evaluation is the same as that which is used in structured testing. It is possible to carry out functional and integrated symbolic testing using this path selection strategy. Throughout the rest of the paper 'functional testing' and 'integrated testing' will refer to functional and integrated structured testing on actual data. When symbolic testing is being referred to the terms 'symbolic functional testing' and 'symbolic integrated testing' will be used.

Some programming errors are due to faulty statements or combinations of statements which give incorrect answers whenever they are executed. Other errors are more subtle: the same sequence of statements may compute correct answers for some data but not for others. Testing rules that involve the execution of programs on actual data are reliable for catching errors of the first type. These rules force the testing of all program constructs of some kind. In theory, symbolic evaluation should extend the reliability of these rules to errors of the second type. When a construct is tested on actual data the output is a single value which indicates the effect of the construct on a single piece of data. When a construct is symbolically tested the output is a symbolic expression which indicates the computational effect of the construct for all data.

The output from a symbolic test of a path consists of the symbolic values of a set of selected variables together with the symbolic system of predicates that describes the data causing that path to be followed. Symbolic testing is reliable for an error occurring in some path if the symbolic output for the path reveals the presence of the error. If a set of output assertions is provided along with the program then the correctness of a program path can be evaluated by determining if the output is consistent with the assertions. In general, there will not be any assertions provided along with a program and the specifications for the program will be informal and incomplete. When there are no assertions the process of determining whether the symbolic output for a path is correct is informal. In order to be

reliable, symbolic testing must not only reveal errors but must reveal them in a way that will, in some ill defined sense, catch the attention of the programmer. In the reliability analysis, symbolic testing was said to be reliable for an error if the symbolic output for a selected path revealed the error in an obvious way. If the error in a path occurred in the symbolic output in the same symbolic form as it appeared in the path, then symbolic testing was not considered to be reliable for the error. There is no reason for assuming that a programmer would notice the error in the symbolic output if the error was represented in the same way in the program. An example of this kind of error is a missing pair of parentheses that occurs in an expression in the program which also occurs in the symbolic output.

Automated test data generation[1-8]

Symbolic evaluation can be used to automate the generation of test data in different ways. The three possibilities that were analysed are:

(i) *Feasible path selection.* When functional or integrated testing is being carried out it is necessary to consider all program subpaths through a module which cause less than or are equal to two iterations of a loop. In some cases there will be an impracticably large number of paths which satisfy this criterion of which only a small number are feasible. (A path is feasible if input data exists which causes that path to be executed.) When a path is symbolically evaluated the conditional branching statement predicates along the path can be evaluated to form a system of predicates that describes the set of all data causing that path to be followed. In many examples it is possible to use systems of predicates to weed out the infeasible paths in a collection of paths. A path is feasible if and only if its symbolic system of predicates has a solution. Infeasible paths have inconsistent systems of predicates.

(ii) *Assisted test data generation.* In many examples the symbolic systems of predicates for feasible paths are concise and easy to read. The predicate systems for these examples can be used to assist the programmer in generating test data.

(iii) *Automatic test data generation.* Data for testing a program path can be automatically generated if the system of predicates for the path is automatically generated and then automatically solved. In some cases it may not be possible to solve automatically the systems of predicates due to their complexity, (e.g. non-linear system of algebraic inequalities).

RESEARCH RESULTS

Reliability of actual data testing and other program analysis methods

The six programs which were analysed contain a total of 28 errors. Table I indicates the reliability of different testing techniques which involve testing on actual data. The table also describes the reliability of program analysis techniques which do not involve testing and which were useful for discovering errors in the sample programs.

The reliability statistics in Table I indicate that path testing is reliable for a significant number of the errors which were analysed (18/28). The low reliability of branches testing (6/28) indicates that the detection of a significant number of the errors that will be discovered by path testing depends on the testing of combinations of program branches rather than single branches. Structured testing increases the reliability of branch testing (to 12/28) by forcing the testing of some of the relevant combinations of program branches. For three of the six programs, structured testing is sufficient to reveal all errors. Branch testing is reliable for only one of the programs.

Table I indicates that structured testing is not powerful enough to reveal a large number of the errors in the PL1-TEST program. TEST is larger than all but one of the other programs (the COBOL program) and it contains a much wider variety of errors than the other programs. The reliability statistics for TEST indicate the importance of program analysis and testing procedures other than those which are defined in terms of the control structure of a program. The most reliable testing procedure for TEST is the special values

Table I. Reliability statistics for testing strategies that use actual data and for other types of program analysis techniques

	COBOL A	PL1 (TEST) B	Telegram C	Line editor D	Sort E	Statistics F	Total
1. Paths	2/3	12/20	1/1	2/2	1/1	0/1	18/28
2. Branches	3/3	3/20	0/1	0/2	0/1	0/1	6/28
3. Structured (functional)	3/3	4/20	1/1	2/2	0/1	0/1	10/28
4. Structured (integrated)	3/3	6/20	1/1	2/2	0/1	0/1	12/28
5. Structured (combined)	3/3	6/20	1/1	2/2	0/1	0/1	12/28
6. Special values	3/3	10/20	1/1	2/2	1/1	0/1	17/28
7. Anomaly analysis	2/3	2/20	0/1	0/2	0/1	0/1	4/28
8. Special requirements	0/3	4/20	1/1	2/2	0/1	0/1	7/28
9. Interface analysis	0/3	2/20	0/1	0/2	0/1	0/1	2/28
10. Combined (5–9)	3/3	18/20	1/1	2/2	1/1	0/1	25/28

method. The reliability figures for special values may be abnormally high since most of the special values rules were constructed by analysing TEST. It is important to note, however, that these rules are general and can be used with a wide range of programs.

Testing strategies such as branches and structured testing force the testing of a program over classes of program input which are defined by the control structure of the program. In some cases a control structure testing strategy will fail to be reliable because it does not force the testing of the relevant paths through the program. In other cases it will fail because some of the input data causing a control path to be followed results in incorrect output and other data causing the same path to be followed results in correct output. Control structure testing strategies may also fail for certain kinds of errors which have no relationship with the control structure of a program.

The following examples describe some typical situations in which structured testing and other actual data testing and program analysis techniques are reliable or unreliable.

Example 1.—Structured testing reliable, branch testing not reliable. The loop in Figure 1 is taken from the third example, the ALGOL telegram program. The program incorrectly

```
word := empty string;
      ⋮
repeat
   if input = empty string then input := read + ' ';
   letter := first (input); input := rest (input);
   word := word + letter;
   until letter = ' ';
```

Figure 1. Fragment of ALGOL string processing program

fails to delete leading blanks whenever a new buffer-full of text is read in with the 'input : = read + ' ' ' statement. This can result in the generation of a word consisting of a single blank. In order to force the error in the fragment it is necessary to follow the true branch of the conditional statement and exit from the loop on some first iteration of the loop. Structured testing but not branch testing will force the testing of a path with this property.

Example 2.—Special values reliable, structured testing not reliable. The code fragment in Figure 2 is taken from the fifth sample program, the PL-360 sort routine. The entire sort loop of the program has been reproduced. The inner loop of the program finds the largest element in the list a(R1), a(R2), ..., a(RN). R3 indicates the position of the element and R0 its value. The outer loop is supposed to interchange a(R3) and a(R1) and increment R1. The error results from the failure of the outer loop to reinitialize R3 to R1 before beginning the execution of the inner loop. If the value of R3 is not set during an execution of the inner loop then it will retain the value set on the previous execution of the inner

```
for R1 = 0 by 1 to N begin
  R0 ← a(R1)
  for R2 = R1 + 1 by 1 to N begin
    if a(R2) > R0 then begin
        R0 ← a(R2)
        R3 ← R2
    end
  end
  R2 ← a(R1)
  a(R1) ← R0
  a(R3) ← R2
end
```

Figure 2. Sort program

loop. This can result in a program state where R0 = a(R1) and R3 ≠ R1. When the interchange code at the end of the outer loop is executed and the program is in this state, the array element currently stored in a(R1) will be duplicated. It will be recopied into a(R1) and also copied into a(R3).

If uninitialized variables are detected during program execution then structured testing will reveal the error in this example. If, as is more likely, all variables are automatically initialized to zero then structured testing will not reveal the error if certain elements of the array have the same value. In order to ensure that the error will be discovered it is necessary to apply the 'distinct values rule'. This rule requires that input values for different variables and for different elements of arrays be distinct. The rule is particularly applicable to data processing programs which move data around from one data structure to another and which carry out relatively simple calculations on the data. The rule helps detect the use of the wrong variable or array element in an assignment statement.

Example 3.—Anomaly analysis reliable, structured testing not reliable. The code fragment in Figure 3 is taken from the second example, the PLI-TEST program. The innermost loop in the fragment has a loop bound of #EXAMS which is an input variable of precision binary (15, 0). The array PERCENTS is declared to be of size 10. The expected range of

```
DO I = 1 TO #STUD
     ⋮
  DO J = 1 TO #EXAMS
    CUMAV(I) = OLDSCORES(I, J) * PERCENTS(J) + CUMAVE(I);
  END;
     ⋮
END;
```

Figure 3. Fragment of PLI-TEST program

values for #EXAMS at this point in the program is $1 \leqslant \#EXAMS \leqslant 11$. If #EXAMS = 11 an out of bounds array reference will occur. The error will not be discovered by a structured testing technique that involves iterating loops less than or equal to two times. The error will be discovered by the array bounds anomaly check described earlier in the paper.

Example 4.—Specifications requirements reliable, all other methods unreliable. The second sample program, the PL1-TEST program, contains a section of code that reads in student answers to exam questions. The answers appear in single card columns and are expected to be integers in the range 0–9. The PL1-coversion routines will convert a blank answer to a 0. There is no control structure, special values or anomalies technique that will reliably reveal this error. Of the different reliability techniques that were studied the only one which addresses itself to this kind of error is a requirement that the user list the acceptable ranges and types for each input variable. In addition, he should list the expected actions to be taken by the program for input which is out of range or of a different type. This will not ensure the reliable discovery of this error but it will focus the programmer's attention on potential problem areas.

Effectiveness of symbolic evaluation

Table II describes the reliability of symbolic testing and the effectiveness of automated test data generation.

Symbolic testing

The results in Table II indicate that symbolic testing which is based on structured testing path traversal will result in the discovery of about 10–20 per cent more of the errors in the programs than structured testing on actual data. The rows in the table which show the increase in reliability for symbolic testing contain two figures. The first figure, not in parentheses, shows the number of errors that would be discovered by symbolic testing that would not be discovered by the corresponding actual data testing procedure. Symbolic functional testing, for example, will discover a total of five more errors than functional structured testing on actual data. The figure in parentheses indicates the number of errors

Table II. Effectiveness of symbolic evaluation for program analysis

	COBOL A	PL1 (TEST) B	Telegram C	Line editor D	Sort E	Statistics F	Total
Functional	2/3	8/20	1/1	2/2	1/1	0/1	14/28
Increase for functional	−1	4 (1)	0	0	1	0	5 (1)
Integrated	2/3	9/20	1/1	2/2	1/1	0/1	15/28
Increase for integrated	−1	3 (0)	0	0	1	0	4 (0)
Combined symbolic	2/3	11/20	1/1	2/2	1/1	0/1	17/28
Feasible paths		*		*	*	*	4/6
Assisted test selection		*		*	*	*	4/6
Automatic test selection							0/6

that would be found by symbolic testing that would not be reliably discovered by any of the actual data testing and program analysis techniques in Table I.

The five errors that would be discovered by symbolic functional testing but not functional testing on actual data all involve the use of an incorrect variable. There are four kinds of errors: use of wrong array variable; incorrect construct resulting in wrong array reference index; reference to the wrong fields in an input card record; and a missing array reference construct surrounding an array reference indexing expression. In all five errors it is possible for the incorrect variable to take on the values of the correct variable during testing on actual data, thus hiding the presence of the error.

Three of the five errors that would be discovered by symbolic functional testing but not functional testing would 'probably' be discovered by functional testing. This is because there has to be an odd pattern of input data in order for the incorrect variables to take on the same values as the correct variables. Two of these three errors would be revealed by structured-testing/special-values, the third by interface analysis.

One of the five errors that would be discovered by symbolic functional testing would not be discovered by any of the testing and program analysis techniques in Table I. This is the error involving the missing array reference construct: an expression incorrectly returns the index of an array element rather than the referenced element of the array. In general, these values will not be the same but none of the testing rules that were studied, except symbolic functional testing, will be certain of revealing the error to the user. The error is described below in Example 6.

The four errors that would be revealed by symbolic integrated testing but not integrated testing on actual data are of the same type as the five errors that would be discovered by symbolic functional testing but not functional testing. All four involve the referencing of incorrect variables. Three of these four errors would also be discovered by symbolic functional testing. The fourth error involves the use of an incorrect array whose name and function are similar to that of the correct array. It is unlikely that the error will be revealed if the arrays are assigned symbolic values that are the same as their names since the error will then be present in both the program and the symbolic output in the same form. If the functional module which contains the error is tested using symbolic functional testing the natural thing to do would be to assign symbolic values to the arrays which correspond to their array names. If the module is tested using symbolic integrated testing then the arrays will have symbolic values which are assigned earlier in the program and which are distinctively different. The error will be highly visible in the symbolic output which is generated during symbolic integrated testing but it is not unlikely that it will go unnoticed in the symbolic output generated during symbolic functional testing.

Two of the five errors that would be discovered by symbolic functional testing but not functional testing on actual data would also not be discovered by symbolic integrated testing. One of the errors is the error that would not be discovered by any of the other techniques. The distinctive property of the two errors is that they both involve references to incorrect variables and that both the correct and incorrect variables that are involved in the errors are assigned values that are generated internally during the execution of the program. The values of the variables are not input values nor are they directly related to input values. If the programs in which the errors occur are tested using symbolic integrated testing both the correct and incorrect variables will take on numeric rather than symbolic values. The errors cannot be reliably discovered by examining output expressions involving the variables since the correct and incorrect variables can, under some circumstances, have the same numeric values. The reason why the two errors will be discovered by symbolic functional testing is that the numeric values which are generated internally for the variables involved in the error are not generated in the same functional module in which the incorrect variable references occur. The values for the variables are generated in one module and then the variables are incorrectly referenced in a second module. In order to carry out symbolic functional testing of the second module it is necessary to assign initial symbolic values to the variables that are involved in the incorrect variable reference errors. The errors will be reliably discovered by examining the output for the module since the incorrect variables will have symbolic rather than numeric values. The incorrect variable references are clearly revealed in the symbolic output expressions which involve the variables.

Although symbolic testing is only reliable for one error which would not be reliably revealed by other testing techniques, there are six errors for which symbolic testing is a natural error discovery tool. Special values testing will discover most of the errors. This is because all six errors involve incorrect variable usage. If input arrays and variables are assigned distinct values the incorrect variable usage will be reliably discovered in five of the six cases.

The following examples describe typical situations in which symbolic testing is reliable or unreliable.

Example 5.—Symbolic testing reliable, structured testing on actual data unreliable, special values reliable. The code fragment in Figure 2 that was used in Example 2 contains an error for which symbolic testing is the appropriate testing method but for which distinct values are also reliable. Figure 4 contains sample symbolic output for a test in which the inner loop in the program is iterated twice. It is assumed that uninitialized variables are initialized to zero. The error is clearly revealed in the symbolic output for the fragment.

Predicates	Output
$a(1) > a(0)$	a(1)
$a(2) \leqslant a(1)$	a(2)
$a(2) \leqslant a(0)$	a(2)

Figure 4. Symbolic output for sort program

Example 6.—Symbolic functional testing reliable, no other techniques reliable. The code fragment in Figure 5 is from the PL1-TEST program. The correct statements which should be substituted for the corresponding incorrect statements are included in the code and are in italics. The code fragment is from the section of the program that computes the quantities Q1 and Q3 from an ordered list of student scores in the vector GRADE( ). The size of the vector is #STUD. The error causes incorrect output to be generated for Q1 and Q3 if

GRADE(CEIL(#STUD/4.0)) ≠ CEIL(#STUD/4.0)

or

GRADE(CEIL(#STUD*3./4.0)) ≠ CEIL(#STUD*3./4.0).

In general, it can be expected that these quantities will not be equal and the error would be discovered by testing on actual data. However, structured testing on actual data does not force the quantities to be unequal and it is therefore only 'probably' reliable rather than reliable for the error.

```
      ⋮
IF DIVBY4 THEN DO;
  Q1 = (GRADE(#STUD/4)+GRADE(1+#STUD/4))/2.0;
  Q3 = (GRADE(3*#STUD/4)+GRADE(1+3*#STUD/4))/2.0;
END; ELSE DO;  Q1 = CEIL(#STUD/4.0);
               Q1 = GRADE(CEIL(#STUD/4.0));
               Q3 = CEIL(#STUD*3./4.0);
               Q3 = GRADE(CEIL(#STUD*3./4.0));
  END;
```

Figure 5. Fragment of PL1-TEST program with corrections

The variable #STUD in the code fragment in Figure 5 is a loop upper bound for one of the loops in the functional module containing the fragment. When the module is tested using symbolic functional structured testing, #STUD will be assigned a small value (1 or 2) in order to cause the iteration of the loop the appropriate number of times during the symbolic evaluation of the module. Assume that the elements of the array GRADE( ) are

assigned symbolic values constructed from the name of the array (i.e. 'GRADE(1)', 'GRADE(2)', etc.). Figure 8 contains the symbolic functional output for Q1 and Q3 which is generated by one of the paths through the module when the module is tested using symbolic functional testing. The output for both the correct and incorrect versions of the program in included.

Q1 = 1	Q1 = GRADE(1)
Q3 = 2	Q3 = GRADE(2)
(a) Incorrect program	(b) Correct program

Figure 6. Symbolic functional output for correct and incorrect versions of TEST program fragment

Example 7.—Specifications requirements reliable, all other methods, including symbolic testing, unreliable. The error which is described above in Example 4 is not reliably discovered by symbolic testing or by any of the other program analysis techniques except specifications requirements.

Example 8.—Symbolic testing reliable, interface analysis reliable, structured testing on actual data unreliable. In this example an error is described for which symbolic testing is reliable but for which some other kind of technique is perhaps more natural or appropriate. The specifications for the PL1-TEST program specify that the PERCENTS weightings for past exams are to appear in columns 11–15, 16–20, ..., 56–60 on a special test weights input card record. The code which is supposed to read in these percentages appears in Figure 7. It is in error because it starts reading from column 6 instead of 11.

```
GET SKIP FILE(SYSIN) EDIT(#EXAMS, (PERCENTS(I) DO I = 1 TO 10))
  (F(5), (10)F(5, 2));
```

Figure 7. Incorrect input code fragment

The program containing this error prints out, as part of a report, the percentages which it has read as input. When the program is symbolically evaluated the output values will be symbolic descriptions of card record fields which, presumably, can be compared with the specifications and noticed to be in error by the programmer. The difficulty with this approach is that when card column numbers are used as symbolic input values it is very difficult to read symbolic output, which will consist of expressions in card column numbers. This may cause errors to go unnoticed which would be easily detected if some more mnemonic type of symbolic input was used. A more appropriate method for revealing errors of this type would be to use a special interface analysis tool which compares input specifications with input statements. This will allow the programmer to assign arbitrary symbolic values to input variables without accidentally hiding the presence of interface errors.

Automated test data generation

(i) *Feasible path selection.* The results in Table II indicate that symbolic evaluation would be useful for eliminating infeasible paths during actual data testing for four of the six programs. In all four of these programs the functional modules at the lowest level of abstraction have a large number of paths which cause loops to be iterated less than or equal to two times and only a small number of these paths is feasible. Symbolic evaluation can be conveniently used for eliminating the infeasible paths through the low level functional modules.

In the four examples for which symbolic evaluation would be useful for feasible path selection, very simple methods are sufficient for determining the feasibility of

the associated systems of predicates. In all four cases it is sufficient to check the consistency of pairs of predicates which involve only constants and the same, single variable. A symbolic evaluation system like DISSECT[5] is ideally suited to the task of determining the feasible paths through a functional module and of generating descriptions of the paths.

In two of the six examples symbolic evaluation is not required for eliminating infeasible paths during structured testing. In both cases the functional modules at the lowest level of abstraction have very few infeasible paths and there is no advantage in having an automatic system to eliminate them.

In all six examples symbolic evaluation is either difficult to use or not required for determining the feasible paths through functional modules which are not at the lowest levels of abstraction. Suppose that a path P through some module M contains an abstract single statement S which consists of a functional module at a lower level of abstraction. In order to determine the feasibility of P it is necessary to determine if there is a subpath through S which makes P feasible. If S contains loops it may be only partially decidable whether such a subpath exists. If the search through the subpaths of S is limited to subpaths which iterate loops less than or equal to two times then the whole process of carrying out the structured testing of M degenerates to carrying out the structured testing of M and S taken together as a single module.

(ii) *Assisted test data generation.* In all four of the examples for which symbolic evaluation would be useful for determining the feasible subpaths through a functional module, the symbolically evaluated systems of predicates for the subpaths would be useful to the programmer for generating test data for functional structured testing. In one of the four examples the control structure of the functional modules is simple enough that it would be easy for the programmer to generate test data for a feasible subpath from a simple listing of the conditional statement branches in the subpath; symbolic systems of predicates are useful but not necessary. If symbolic evaluation is used for feasible subpath selection then the systems of predicates for the feasible subpaths are available as a side-product of the feasibility analysis. There is no extra cost involved in using systems of predicates to assist the user in test data selection rather than some simpler kind of path information.

When a program is tested using integrated structured testing it is necessary for the programmer to construct tests that cause selected subpaths through functional modules to be traversed when the program is executed on those tests. There are two possible ways of using symbolic evaluation to assist the user in generating test data for integrated testing. The first is to use it to generate symbolic systems of predicates for complete program paths. In order to do this it is necessary to construct feasible partial paths which go from the beginning of the program up to the beginning of the selected subpaths through functional module and other partial paths which go from the ends of the subpaths to the end of the program. There is no obvious way of automatically selecting feasible partial paths that would be consistent with selected subpaths through functional modules. It is likely that the task of selecting the partial paths that are needed to test selected subpaths would be left to the programmer.

The second way in which symbolic evaluation could be used to assist the programmer during integrated structured testing is to restrict its use to the generation of symbolic systems of predicates for selected subpaths through functional modules and not to use it to generate predicates for complete paths. In order for the second

way of using symbolic evaluation to be useful it would have to be possible for the programmer to generate program test data directly from subpath systems of predicates which would cause the subpaths to be traversed when the program was executed.

In the examples that were studied it would be easy for the programmer to generate test data by hand, given the systems of predicates for module subpaths. It would be tedious and time consuming for him to have to construct the partial paths needed to complete a module subpath. In all four of the examples for which symbolic evaluation would be useful for generating test data for functional structured testing it would also be useful for generating data for integrated structured testing. For both types of testing it would be sufficient to generate systems of predicates for subpaths through functional modules rather than predicates for complete program paths.

(iii) *Automatic test data generation.* The use of symbolic evaluation for automatic generation of test data was not found to be useful for any of the sample programs. There were several different reasons for this. In some cases, the symbolic output that would be generated during feasible path selection is more revealing than output from actual data and there is no point in carrying out the execution of the program on actual data. In other cases, it is important to consider the use of special values rules during test data generation and it is more convenient to let the user do this, assisted by the systems of predicates for program paths and subpaths. In one program (COBOL) it is trivial for the user to generate test data for a path given the sequence of branches along the path and there is no need for the use of symbolic evaluation.

CONCLUSIONS

The results described in this report are for a small sample of programs. Different patterns may be demonstrated by other sets of examples although it is likely that this will only be different for different types of programs. Continuing research will include the analysis of additional programs. Plans have been completed to carry out a study of numerical analysis programs in order to determine if this type of program generates different kinds of results from those described above.

The basic research methodology used in the project involved the evaluation of testing strategies by determining which errors would always be discovered by the strategies when they were applied to a program containing known, naturally occurring errors. Experience with the project indicates that this is a sound way to evaluate test strategies. An alternative approach is described by Hetzel.[16] In his thesis research Hetzel carried out a number of statistical experiments in which different groups of programmers used three different testing and program analysis techniques to debug three programs containing known errors. Hetzel's approach is suitable for the analysis of ill defined techniques like 'code-reading' and 'specifications testing'. The approach which is described in this paper is more applicable to well defined rules like structured testing and interface analysis. These well defined techniques do not require any selective judgement on the part of the programmer and their reliability can be analysed without the need for statistical experiments. The approach which was used in this paper is heavily oriented towards the discovery of new testing rules like the special values and anomaly analysis rules described in the Research methodology section.

The most important result of the project was the clear indication that no one program analysis technique or program testing strategy should be used to the exclusion of all others. Some types of errors are more readily discovered by one technique, such as anomaly analysis, and others by different techniques such as structured testing, refined by special

values. The project can be viewed as a pilot study of the effectiveness of program testing and program analysis methodologies. The results of the project are encouraging and imply that a larger, more comprehensive project should be carried out, modelled along the lines of this study. The goal of the larger project would be to provide extensive guidelines for the design of a comprehensive set of testing techniques and program analysis tools.

In the experiments with the six sample programs the use of symbolic testing resulted in an increase in reliability of 10–20 per cent over the use of structured testing on actual data. These figures agree with those reported in Reference 5. This increase is reduced to 3–4 per cent if structured testing is augmented with other program analysis and testing techniques such as special values and interface analysis. In an integrated system of analysis and testing techniques, symbolic testing would be a 'natural' error discovery method for 10–20 per cent of the errors in the sample programs even though some of these errors would also be reliably discovered by other methods.

In addition to the small absolute increase in reliability which is obtained when symbolic testing us used, symbolic evaluation is useful in other ways in the testing process. Symbolic evaluation was very useful in four of the six sample programs for eliminating infeasible subpaths from the sets of subpaths through functional modules and for generating descriptions of the data causing the subpaths to be followed.

Experience with the examples that were studied indicates that symbolic integrated testing is more difficult to use than symbolic functional testing. Symbolic integrated testing requires that the user carry out symbolic tests of complete program paths which contain selected subpaths through individual functional modules. This requires that the user construct partial paths which lead up to and are consistent with the selected subpaths, and partial paths which lead from the functional module subpaths to the end of the program. The difficulties are similar to those that are encountered when symbolic evaluation is used to generate test data descriptions for integrated testing on actual data. The problems are compounded when symbolic integrated testing is applied to functional modules which are not at the lowest level of abstraction.

In general, it was found to be difficult to apply symbolic evaluation to all but the functional modules at the lowest level of abstraction. The results of the experiments with the six examples indicate that in an integrated testing system the application of symbolic evaluation should be limited to low level modules and can be used to carry out the following types of analysis: symbolic functional testing, elimination of infeasible subpaths through functional modules and generation of symbolic descriptions of the tests that will cause selected module subpaths to be traversed. The descriptions of the tests that cause selected subpaths to be traversed can be used to assist the programmer in generating test data both for functional and integrated structured testing on actual data. In the six examples that were studied there were three errors that would be discovered by symbolic integrated testing but not symbolic functional testing. Two of the three errors would also be discovered by integrated testing on actual data and the third by special values testing so that, at least for those examples, there would be no loss in reliability in a system which restricted symbolic evaluation to the three types of analysis listed above.

Continuing research on symbolic evaluation will include a survey of symbolic evaluation techniques and an analysis of the cost of symbolically evaluating functional modules. These results will be used, together with the results of the research project described in this report, to help determine if the increase in reliability and the increased ease in carrying out structured testing that results from the use of symbolic evaluation is worth the extra cost that would be incurred in including a symbolic evaluation capability in an integrated testing system.

ACKNOWLEDGEMENTS

The author wishes to thank Jeffrey Laub for his assistance in carrying out the research described in this paper. Mr. Laub carried out a preliminary analysis of two of the six examples. The research project was funded by the National Bureau of Standards under the supervision of Dr. Dennis Fife.

REFERENCES

1. W. E. Howden, 'Methodology for the generation of program test data', *IEEE Trans. on Computers*, **C-24**, 554–559 (1975).
2. L. Clarke, 'A system to generate test data and symbolically execute programs', *IEEE Trans. on Software Engineering*, **SE-2**, 215–222 (1976).
3. J. C. King, 'Symbolic execution and program testing', *CACM*, **19**, 385–394 (1976).
4. R. S. Boyer, B. Elspas and K. N. Lewitt, 'SELECT—A formal system for testing and debugging programs by symbolic execution', *Proc. 1975 Int. Conf. on Reliable Software, IEEE Cat. No. 75CH094-7CSR*, IEEE Computer Society, Long Beach, Ca., 234–245 (1975).
5. W. E. Howden, 'Symbolic testing and the DISSECT symbolic evaluation system', *IEEE Trans. on Software Engineering*, **SE-3**, 266–278 (1977).
6. E. F. Miller and R. A. Melton, 'Automated generation of test case data sets', *Proc. 1975 Int. Conf. on Reliable Software, IEEE Cat. No. 75CH094-7CSR*, IEEE Computer Society, Long Beach, Ca., 51–58 (1975).
7. J. C. Huang, 'An approach to program testing', *Comput. Surv.*, **7**, 113–128 (1975).
8. C. V. Ramamoorthy, S. F. Ho and W. T. Chen, 'On the automated generation of program test data', *IEEE Trans. on Software Engineering*, **SE-2**, 293–300 (1976).
9. R. M. Burstall, 'Proving correctness as hand simulation with a little induction', *Proc. IFIPS 74*, North Holland Publishing Co., Amsterdam, 308–312 (1974).
10. L. P. Deutsch, 'An interactive program verifier', *Ph.D. thesis*, University of California, Berkeley (1973).
11. J. B. Goodenough and S. L. Gerhart, 'Toward a theory of test data selection', *IEEE Trans. on Software Engineering*, **SE-1**, 156–173 (1975).
12. S. L. Gerhart and L. Yelowitz, 'Observations of fallibility in applications of modern programming methodologies', *IEEE Trans. on Software Engineering*, **SE-2**, 195–207 (1976).
13. W. E. Howden, 'Reliability of the path analysis testing strategy', *IEEE Trans. on Software Engineering*, **SE-2**, 208–214 (1976).
14. W. E. Howden, 'An evaluation of symbolic testing', *University of California, San Diego, Computer Science Technical Report No. 15, APIS* (1977).
15. A. R. Brown and W. A. Sampson, *Program Debugging*, Macdonald and Co., London, 1973.
16. W. Hetzel, 'An experimental analysis of program verification methods', *Ph.D. thesis*, University of North Carolina (1976).
17. P. Henderson and R. Snowden, 'An experiment in structured programming', *BIT*, **12**, 38–53 (1972).
18. P. Naur, 'Programming by action clusters', *BIT*, **9**, 250–258 (1969).
19. N. Wirth, 'PL360, a programming language for the 360 computer', *JACM*, **15**, 37–74 (1968).
20. J. Gould and P. Drongowski, 'A controlled psychological study of program debugging', *IBM Report RC4083*, Yorktown Heights, N.Y. (1972).
21. K. W. Krouse, R. W. Smith and M. A. Goodwin, 'Optimal software test planning through automated network analysis', *Proc. 1973 IEEE Symp. on Computer Software Reliability*, IEEE Computer Society, Long Beach, Ca., 18–22 (1973).
22. L. G. Stucki, 'Automatic generation of self-metric software', *Proc. 1973 IEEE Symp. on Computer Software Reliability*, IEEE Computer Society, Long Beach, Ca., 94–100 (1973).
23. R. E. Fairley, 'An experimental program testing facility', *IEEE Trans. on Software Engineering*, **SE-1**, 350–357 (1975).
24. H. D. Mills, 'Top down programming in large systems', in *Debugging Techniques in Large Systems* (Ed. R. Rustin), Prentice-Hall Inc., Englewood Cliffs, N.J., 1971, pp. 41–55.
25. C. V. Ramamoorthy, K. H. Kim and W. T. Chen, 'Optimal placement of software monitors aiding systematic testing', *IEEE Trans. on Software Engineering*, **SE-1**, 403–410 (1975).
26. C. V. Ramamoorthy and S. F. Ho, 'Testing large software with automated software evaluation systems', *IEEE Trans. on Software Engineering*, **SE-1**, 46–58 (1975).

27. L. J. Osterweil and L. D. Fosdick, 'DAVE—A validation error detection and documentation system for Fortran programs', *Software—Practice and Experience*, **6**, 473–486 (1976).
28. L. D. Fosdick and L. J. Osterweil, 'The detection of anomalous interprocedural data flow', *Proc. 2nd Int. Conf. on Software Engineering, IEEE Cat. No. 76CH1125–4C*, IEEE Computer Society, Long Beach, Ca., 624–629 (1976).
29. B. C. Hodges and J. P. Ryan, 'A system for automatic software evaluation', *Proc. 2nd Int. Conf. on Software Engineering, IEEE Cat. No. 76CH1125–4C*, IEEE Computer Society, Long Beach, Ca., 617–623 (1976).