A PROGRAM TESTING SYSTEM*

Lori A. Clarke
Computer and Information Science Dept.
University of Massachusetts
Amherst, Massachusetts 01002

A system that aids in testing programs is described. This system symbolically executes program paths and creates symbolic representations of the output variables that aid in verifying a path's computations. The conditional statements that affect the flow of control are also symbolically represented by a set of inequalities. The inequalities are then evaluated to determine input data that would cause execution of the path. The system also does extensive error checking by simulating possible data dependent errors and then attempting to detect data sets that would cause execution errors.

1. INTRODUCTION

Typically programmers are very lax about testing their code. One study found that programmers tend to test less than half of the source statements and only about one third of the possible branch conditions [1]. On the other hand, exhaustively testing a program is generally impractical and undesirable. Testing methods that are more rigorous than the current haphazard approach yet less comprehensive than exhaustive testing need to be explored. In addition selecting test data that will exercise a program is a complex and tedious task, especially for a large program. Therefore, automated tools are needed to alleviate the problems of program testing. This paper describes an automated system that aids in testing programs by generating test data, symbolically representing the program's output, and extensively checking for errors. This system analyzes programs written in ANSI FORTRAN and is also written in ANSI FORTRAN.

Automatic test data generation is a powerful tool that can assure that the code is adequately tested. It can be used to select test data that executes every statement, executes all possible branch conditions, or satisfies any desired testing criteria.

Executing a particular program path will not guarantee the path is correct for all possible input data. Therefore, to aid program testing further, the system displays symbolic representations of the path's output variables. The symbolic representations are similar to algebraic expressions and aid in verifying a path's computations. The symbolic representations are useful not only in detecting errors but also in detecting the source of the errors.

Though a path's symbolic representation may appear to be correct, errors may still exist. Perhaps for some input data a subscript bounds error could occur or perhaps a divisor could evaluate to zero. It is often difficult to detect these types of error conditions by examining the symbolic representations and even more difficult to detect them by examining the source code. Therefore, the system also attempts to detect errors.

2. SYMBOLIC REPRESENTATION GENERATION

The system analyzes selected program paths in three phases; the symbolic execution phase, the inequality simplification phase, and the inequality solver phase. During the symbolic execution phase the symbolic representations are generated. In this phase the system executes a path's instructions. Instead of using input data, however, the system symbolically represents the input data with system generated variables. Since it is later necessary to distinguish real and integer input values, the system uses two arrays, I and X, to represent the input data. If the nth input datum is integer and the next input datum is real then I(n) and X(n+1) would be used to represent these items respectively. Thus, a path's computations are represented symbolically by algebraic expressions in terms of the array elements of I and X.

Whenever an output statement is encountered during the symbolic execution of a path, the symbolic representations of the output variables are displayed. These symbolic representations

*Supported in part by the National Science Foundation under grant GJ-36461

are often more useful in testing a program than a computed value. The symbolic representations contain information about the evolution of a variable and represent the variable for all input values that would cause the selected path to be executed. Thus, the symbolic representations represent the results for a class of input data instead of just one set of input data.

3. TEST DATA GENERATION

To generate test data that will cause the execution of a particular program path requires that the conditional statements on the path be satisfied. Therefore, the system first determines the conditional statements and then attempts to find a set of test data satisfying these relationships.

The conditional statements are represented during the symbolic execution phase. For example, the true branch of the FORTRAN statement:

```
IF  (A.GT.B)  GO TO  10
```

is represented by the inequality S(A) > S(B), where S(A) and S(B) represent the symbolic representations of variables A and B at the given point in the path. If the false branch of this conditional statement is chosen then the expression is represented by (S(A) > S(B)). Each conditional branch in the path is represented by an inequality. The path, therefore, is represented by a set of inequalities where the unknown variables represent input data. A solution to the system of inequalities is test data that will cause execution of the path. If the set of inequalities is inconsistent then the selected path is nonexecutable.

During the symbolic execution phase, conditional statements often can be evaluated to a true or false value. If an inequality evaluates to the value true, then no further analysis of that inequality is necessary. If an inequality evaluates to the value false, then the path is nonexecutable.

Usually the values of all the inequalities can not be computed and therefore, the set of inequalities must be evaluated to determine if the path is executable or not. The inequalities however may be long and complex. Thus, it is necessary to simplify these expressions before a solution can be attempted. The second phase of the system, inequality simplification, is accomplished by an ALTRAN program. ALTRAN is a portable system that does symbolic algebraic manipulations [2]. Then the third phase of the system, the inequality solver, attempts to find a solution to the set of inequalities or to determine if the inequalities are inconsistent. In general, evaluating an arbitrary set of inequalities is an unsolvable problem [3]. Experience with the system has demonstrated, however, that the inequalities are usually linear [4]. A linear programming system [5] has successfully been used to evaluate the linear inequalities. Only when the inequalities are nonlinear is it necessary to manually attempt a solution.

In order to determine the subject program's nonexecutable subpaths, the inequalities are passed to the inequality solver one at a time. The inequality solver first checks to see if the previous solution satisfies the new inequality. If so, the next inequality is attempted. If not, the inequality solver attempts a new solution. If the set of inequalities is found to be inconsistent then the subpath from the start of the path to the statement where the current inequality was generated is nonexecutable. This subpath should not be included in future test paths.

Some language dependent properties, in addition to the condition statement, must be represented by inequalities when generating test data. One such property in FORTRAN is integer truncation. for example, in the statement J=Y where J is integer and Y is real, the value of Y is truncated and assigned to J. If the value of Y can be computed for the path then the actual value of J is also computed and assigned to J. If the value of J depends on an input variable, however, the truncation is represented symbolically. To represent the truncation, a new symbolic variable is introduced and constrained to be the truncated value. The integer variable is then assigned this new symbolic variable. For example, let I(n) be the symbolic value assigned to represent the truncated value of Y in the above expression. Then J = I(n) and I(n) is constrained by the following inequality where S(Y) is the symbolic representation of variable Y.

$$((S(Y)\geq 0) \wedge (S(Y)\geq I(n)) \wedge (S(Y)-1<I(n))) \vee$$
$$((S(Y)<0) \wedge (S(Y)\leq I(n)) \wedge (S(Y)+1>I(n)))$$

Figures 1-3 demonstrate a path where an additional constraint representing integer truncation is necessary. Assume the statements in Figure 1 are statements on a path and that variable Z depends on an input variable. Figure 2 displays the generated constraints when truncation is not considered. From these constraints it appears that the GO TO 10 statement is executable on this path. Figure 3 displays the generated constraints when truncation is considered. I(n) represents the new variable that is introduced to represent the truncated value of VAL + Z. S(VAL) and S(Z) represent the symbolic representations of variables VAL and Z. The constraints in Figure 3 are inconsistent and correctly represent the path as nonexecutable.

There are several other properties of FORTRAN that require the assertion of additional constraints in order for the generated test data to be valid.

```
IF  (0..GT.VAL)  STOP
IF  (Z.LT.0..OR. Z.GE.1.)  STOP
K = VAL + Z
  :
  :
Y = K
IF  (Y.GT.VAL)  GO TO  10
```

Figure 1.

$$0. \leq S(VAL)$$

$$0. \leq S(Z) < 1.$$

$$S(VAL) + S(Z) > S(VAL)$$

Figure 2.

$$0. \leq S(VAL)$$

$$0. \leq S(Z) < 1.$$

$$((S(VAL) + S(Z) \geq 0) \wedge (S(VAL) + S(Z) \geq I(n)) \wedge (S(VAL) + S(Z) - 1 < I(n))) \vee ((S(VAL) + S(Z) < 0) \wedge (S(VAL) + S(Z) \leq I(n)) \wedge (S(VAL) + S(Z) + 1 > I(n)))$$

$$I(n) > S(VAL)$$

Figure 3.

4. ERROR DETECTION

Generating data to force execution down a path can assure that the code has been tested but cannot assure that all errors have been detected. Some errors will occur only with a specific set of input data. The system, therefore, tries to detect data dependent errors by temporarily generating artificial inequalities that simulate these error conditions. The system then attempts to solve the set of inequalities with each additional artificial inequality. If there exits a solution to any of the augmented sets of inequalities then data exists that would cause an error while executing the code and a message is issued.

Division by zero is one of the errors that the system attempts to detect and will be used to illustrate the error detection process. Assume that during the symbolic execution phase the assignment statement W = P/R is encountered on the path. If the divisor R evaluates to a constant value then a check is made to assure the value is not zero. If the divisor does not evaluate to a constant value then an artificial inequality is created that sets the symbolic representation of the divisor to zero (e.g. S(R) = 0.). This inequality is simplified and then during the inequality solver phase temporarily added to the set of previously generated constraints. If the system can solve the new set of constraints then test data exists that would cause a division by zero error. If a solution exists or not, the artificial inequality is removed from the set of constraints and the analysis of the path continues.

The system currently checks for the following errors:

1) Division by zero
2) Subscripts out of bounds
3) Illegal variable dimensions
4) Illegal DO parameters
5) Illegal mixed mode expressions
6) References to undefined variables

The error checking capabilities have proven to be quite useful and may be extended to include other error conditions in the future.

5. RELATED WORK

A technique similar to symbolic execution was proposed by Balzer in the EXDAMS system [6]. More recently others have also independently developed symbolic execution [7,8]. There are several systems that detect the path constraints [7,8,9,10,11]. The EFFIGY [7] and SELECT [8] systems analyze programs written in a subset of PL/1 and LISP, respectively. It is felt that a standard user language poses a wider range of problems and is a more realistic test for a testing system. The system developed by Howden [9], Miller [10], and Huang [11] do not attempt to simplify the constraints or generate test data. Experience with this testing system has demonstrated that often manually analyzing the inequalities is a difficult process and automated assistance is desired.

An interactive system developed at TRW recognizes some nonexecutable paths and aids the user in selecting test data [12]. Goodenough and Gerhart have proposed a method of selecting test data using decision tables [13]. Both these methods require more user participation than the other systems mentioned here.

The error checking capability is similar to the technique proposed by Sites [14]. To the author's knowledge, none of the other testing systems do extensive error checking of this type.

6. CONCLUSION

It is felt that the testing tools of this system, automatic test data generation, symbolic representation of the output variables, and extensive error checking, provide the user with powerful tools to aid in program testing. Initial tests of the system have been quite promising.

The symbolic execution phase has proved to be quite useful. It supplies the user with valuable information about the variable relationships that evolve executing a program path. This information is used to describe the flow of control relationships as well as the computational relationships. During this phase the system detects some infeasible paths as well as program errors, such as illegal mixed mode expressions and references to undefined variables.

The simplification phase has been reasonably reliable. However, the simplification process is relatively expensive and since this phase of the system is written in ALTRAN it is not as portable as the other phases.

The inequality solver phase has been very satisfactory. Though it cannot handle nonlinear constraints this problem has occurred relatively infrequently. Even when the inequality solver cannot analyze the constraints the system provides the user with a wealth of information.

REFERENCES

[1] L. G. Stucki, "Automatic generation of self-metric software," in Rec. 1973 IEEE Symp. Software Reliability, pp. 94-100.

[2] W. S. Brown, Altran User's Manual, Bell Telephone Lab., vol. 1, 1973.

[3] M. Davis, "Hilbert's tenth problem is unsolvable," Amer. Math. Mon., vol. 80, pp. 233-269, Mar. 1973.

[4] L. A. Clarke, "Test Data Generation and Symbolic Execution of Programs as an Aid to Program Validation," Ph.D. Thesis, Univ. of Colorado, 1976.

[5] F. Glover, private communications.

[6] R. M. Balzer, "EXDAMS--Extendable debugging and monitoring system," in 1969 Spring Joint Computer Conf., AFIPS Conf. Proc. vol. 34. Monvale, NJ: AFIPS Press, 1969, pp. 567-580.

[7] J. C. King, "A new approach to program testing," in Proc. Int. Conf. Reliable Software, Apr. 1975, pp. 228-233.

[8] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT--A formal system for testing and debugging programs by symbolic execution," in Proc. Int. Conf. Reliable Software, Apr. 1975, pp. 234-244.

[9] W. E. Howden, "Methodology for the generation of program test data," IEEE Trans. Comput., vol. C-24, pp. 554-559, May 1975.

[10] E. F. Miller, and R. A. Melton, "Automated generation of test case datasets," in Proc. Int. Conf. Reliable Software, Apr. 1975, pp. 51-58.

[11] J. C. Huang, "Program testing," Dep. Comput. Sci., Univ. Houston, Houston, TX, May 1974.

[12] K. W. Krause, R. W. Smith, and M. A. Goodwin, "Optimal software test planning through automated network analysis," in Rec. 1973 IEEE Symp. Software Reliability, pp. 18-22.

[13] J. B. Goodenough, and S. L. Gerhart, "Toward a theory of test data selection," Proc. Int. Conf. Reliable Software, Apr. 1975, pp. 493-510.

[14] S. L. Sites, "Proving that computer programs terminate clearly," Dept. Comput. Sci., Stanford Univ., Stanford, CA.