DATA SPACE TESTING

Michael Paige
Manager Product Quality Assurance
Wang Laboratories
Lowell, Mass. 01851

ABSTRACT

A complete software testing process must concentrate on examination of the software characteristics as they may impact reliability. Software testing has largely been concerned with structural tests, that is, test of program logic flow. In this paper, a companion software test technique for the program data called data space testing is described.

An approach to data space analysis is introduced with an associated notation. The concept is to identify the sensitivity of the software to a change in a specific data item. The collective information on the sensitivity of the program to all data items is used as a basis for test selection and generation of input values.

INTRODUCTION

The initial emphasis in software development was on efficiency. Hardware costs were high so there was pressure to fully utilize available resources. Software, on the other hand, was relatively cheap and could be made to mask out hardware deficiencies. The situation has changed and software effeiciency has given way to effectiveness (reliability) as the cornerstone of software development.

The problems in developing reliable software systems are becoming increasingly critical, and the need for integrated, practical methodologies for the design, implementation, and testing of such systems has been widely recognized. In the short term, software reliability can be enhanced through use of systematic program testing techniques.

The central concept in software is that of a program and the most frequently used definition of a program is a sequence of instructions. This approach tends to ignore the role of data in the program; hence, an alternative definition is that a program is a series of transformations and other relationships over sets of data. The purpose of this paper is to examine this data and how it can be tested in conjunction with the program.

Data testing is of key concern in software checkout. The current literature concerning data testing has largely centered on "data flow analysis", that is, the use of the set/use pattern for a given variable[1] within the software. Anomalies in the usage of given variables can be detected by this approach, for example, if a variable is referenced (used) before it is defined (set). This approach, however, does not shed much light on the problem of actually creating software tests.

The current emphasis on software test data generation is divided into two modes: control path related and data definition related. The first mode uses the control flow (or program graph) representation of the software as a basis for testing[2,3]. The concept is to select a possible test path through the code, determine the variables which control the execution of that path, determine the constraining conditions on those variables, and then select a specific set of values for those variables which satisfy those conditions. The result is test data for the given test path. (Computational accuracy may be checked as a by-product of this approach.)

The alternative test data generation approach is to concentrate on the user supplied data declarations[4]. The concept is to fabricate an input value which satisfies all the declared (or implied) size, type, etc. characteristics. This test can then be executed and the corresponding test path determined. Techniques for random number generation can be used to determine how the software will handle variable conditions.

A third approach for the use in test data generation is considered in this paper. This approach represents a synthesis of the two preceding techniques. It also introduces a more tractable basis for systematic test data generation than either paths which in most cases are extremely large in number, or static data declaration manipulation, which tends to ignore the actual software algorithms. This approach

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

©1981 ACM 0-89791-038-9 /81/0003/0117 $00.75

is based on the usage patterns and the physical characteristics of the data, jointly considered as data space.

In the next section the concept of data space is considered and an analysis model is described. An approach to data space testing is discussed in the following section.

DATA SPACE AND DATA FLOW ANALYSIS

The first step in this discussion must be to separate the data space from the program.

As described earlier, a program can be regarded as a transformation which converts input data to output data. In this sence, the data space consists of a set of containers, that is, forms to contain the information to be processed. Each container has a size and shape, and other attributes which relate to form. Information within the containers is identified by types and values and other attributes which relate to content.

The containers for the sample program shown in Figure 1 are listed in Figure 2. Note the containers are either constant or identifiers. Subroutine calls are treated like arrays, for example, CALL PE(12) is considered to be data items 12 and PE.

Implicit in the program is an order of the effect of one data item on another. True to the nature of programming languages, this order of precedence is in the form of a modifier, that is, an item which restricts or limits another. A subscript expression, for example, modifies the variable to which it applies. Consider the assignment statement shown below:

```
R(N) = T(M+L,K)**2 + R(S(K))
```

We will let the notation (X,Y) imply that X modifies Y; the parsing of this assignment statement would yield:

(N,R)	(2,R)
(K,T)	(K,S)
(M,T)	(S,R)
(L,T)	(R,R)
(T,R)	

The list above of formulae can also be presented in a matrix format where the rows and columns are the container names and the entries.

A container which appears only as a row but not as a column is said to be prime. In Figure 4 L2, L4, 1, 600, 2, 12 are primes. All constants must be prime.

A container X's reach, $\rho(X)$, is determined by iteratively forming the union of all the rows which correpond to columns in its modification matrix. For example the reach for J would be formed as indicated in Figure 5. A reach is the set of all data items which could be effected by a change in a given data item. A reach matrix can be formed by collecting all the individual row's reaches as shown in Figure 6.

The modification matrix represents the effect of one data item on the next and the complete effect of a single data item. In a more global basis the reach of a container X represents the sensitivity of the program to that item.

The sensitivity of program P to item X will be referred to as dP/dX, that is, the change in P due to a change in X. This sensitivity is indicated by an item's reach, e.g.,

$$\frac{dP}{dX} = p(X)$$

Hence,

$$dP = p(X)dX$$

which implies that the change in X, or dX, is propagated through p(X) to its global program effect. This effect is felt through one or two routes; it is either

1) computation (R)
2) control (C)
3) both

SOFTWARE TESTING APPROACH

The most practical means to demonstrate software reliability is by testing. Testing is the process of evaluating the appropriateness of the program results and the robustness of the code in practical machine environments. In a large sense, testing is equivalent to exercising the system. A test corresponds to input stimulus or event, and a corresponding system response. In production environments, it is far too often the case that unexpected events prove to be the major obstacle in demonstrating software reliability. Programmers, and the program designers before them, often overlook or discount some of the potentialities; this is understandable since there is usually too many events to consider at one time.

For purposes of this discussion, a software test will be defined not only in the terms of the input and the corresponding output but the data items which are modified by that particular input.

```
1.    SUBROUTINE SAMPLE (L1,L2,L3,L4)
      DIMENSION L3(1,2)
      INTEGER L1,L2,L3,L4,P,J,E
2.    P = MD(L2,L4) +1
3.    IF (L2.EQ.600) RETURN
4.    IF (L3(P,1).EQ. L2)
5.      L1 = L3(P,2)
6.    ELSE
7.       J = P
8.       WHILE (L3(J,1).NE.L2.AND.J.NE.P-1)
9.         IF (J.LT.L4)
10.           J = J+1
11.        ELSE
12.           J = 1
13.        ENDIF
14.      ENDWHILE
15.      IF (L3(J,1).EQ.L2)
16.        L1 = L3(J,2)
17.      ELSE
18.        CALL PE(12)
19.      ENDIF
20.    ENDIF
21.    RETURN
22.    END
```

Figure 1. Sample Program

CONTAINER	TYPE	SHAPE	SIZE
L1	INTEGER	Word	1
L2	INTEGER	Word	1
L3	INTEGER	Array	(1,2)
L4	INTEGER	Word	1
P	INTEGER	Word	1
J	INTEGER	Word	1
E	INTEGER	Word	1
1	CONSTANT		
2	CONSTANT		
600	CONSTANT		
PE	SUBROUTINE		
12	CONSTANT		

Figure 2. Containers for SAMPLE

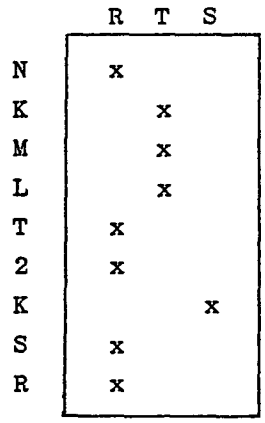

Figure 3. Modification Matrix

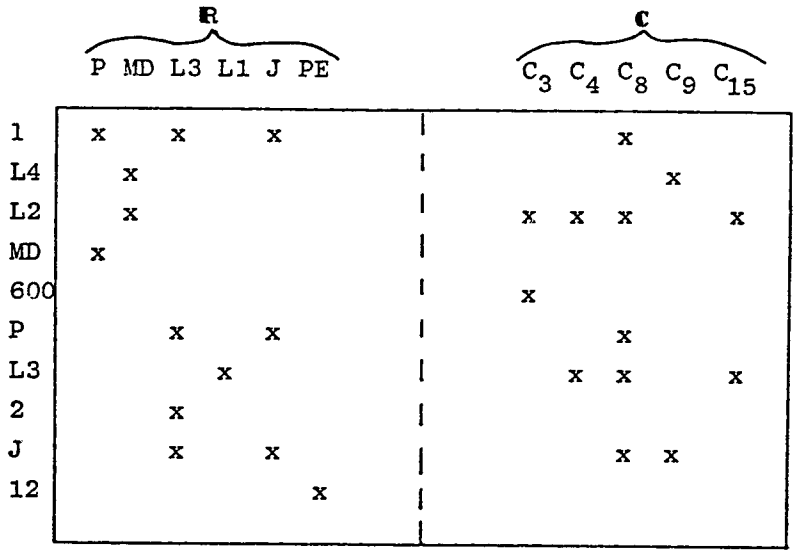

Figure 4. Modification Matrix for SAMPLE

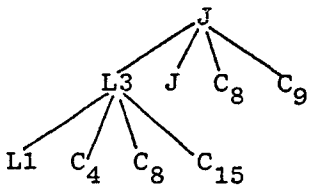

Figure 5. J's Reach

	R						C				
	P	MD	L3	L1	J	PE	C_3	C_4	C_8	C_9	C_{15}
1	x		x	x	x			x	x	x	x
L4	x	x	x	x	x			x	x	x	x
L2	x	x	x	x	x		x	x	x	x	x
MD	x		x	x	x			x	x	x	x
600							x				
P			x	x	x			x	x	x	x
L3				x				x	x		x
2			x	x				x	x		x
J			x	x	x			x	x	x	x
12						x					

Figure 6. Reach Matrix

The modification matrix (such as that shown in Figure 4) presents two types of test information: data items which affect the control of the program (C_i), and data items which affect the computation of the program. These are not necessarily disjoint classes; for example, the item J belongs to both classes.

A test corresponds to an input value, a corresponding series of modified items, and an output. During the test design process there are three questions that need to be addressed:

1) What is a "good" test set?
2) What is a test for a given data item?
3) During the test for a given data item what else is affected?

Before the formation of tests is addressed an observation concerning the modification and reaching matrices is necessary. The set of prime items may not include all input data items; however, the set of prime items is a ready vehicle for manipulation of all program data items. The prime items are, in essence, a baseline set of test controls.

The answers to the three questions can now be addressed as follows:

A "good" test set should exercise each data item in the program at least once. The set of prime items represents an ideal series of test controls, however, it is often the case that these items do not cover all other data items. If a program has no prime items or the primes do not form a complete cover then a cover has to be selected; in this case, a cover is a set of data items which have within their reach all other items.

A test is an input, the data items modified, and the output; hence a test T is represented as:

$$(dX, p(X), dP)$$

or simply $T = dP = p(X)dX$. In words, a test is in input change, the resulting set of modified items, and their effect on the program.

The answer to the second question involves the determination of which data item, Y, is to be tested and the selection of all primes which reach Y. In short, select all primes such that $Y \in \rho(X)$.

The thrid question simply involves the determination of the reach of the data item.

The remaining issue to address is the formulation of dX.

The data items can be defined as constants or parameters. The testing with these two types differ significantly. The test for a constant requires a check to insure the equivalance of the item name, e.g., 0, 1, 201, and the actual item value. Often constants can be "clobbered" and hence must be checked for value. Hence, dX corresponds to creating an entity whose name and value differ, although Y is a constant.

The test for parameters requires that a sample item be generated. The container information now becomes extremely important. To test L3, for example, (from Figure 2) we known that L3 is an array of size (1,2) consisting of integers. The nature of these numbers, that is, their properties would be of help in generating a test, however, for the first level of testing it is important to merely establish the response of the program to item X.

Much of the process of software testing invloves rediscovery of software intent, that is, the testing process should re-establish the general software characteristics. These discovered features are then compared with the original specifaction to validate the software design. By focusing attention on the effect of data characteristics, e.g., shape, size, type, and not on actual values, questions can be resolved on the ability of the program to process that particular data item. Using the information in Figure 2 a set of possible input values can be selected by fabricating a sample which lies inside the acceptable definition and then outside by virtue of the wrong type, shape, or size as shown in Figure 7.

A "good" test set for the example program (Figure 1) is (1, 2, 12, 600, L2, L4), the primes from the program. One set of test cases is shown in Figure 8. If further information is made available concerning the actual output data items or the ranges on the values of L2 and L4 further dX situations could be developed.

FINAL NOTE

At the present time there is no comprehensive strategy for software testing. Much of the literature has concentrated on structural or flow tests and very little on the testing of data. The technique described in this paper provides an analytic basis for data testing. The concept of data space is intended to include the relationships induced by the program, since the idea of data base seems to often exclude the actual program and its characteristics.

A program is often best understood as a sum of its parts. A program should be tested in parts as well as a whole in order to avoid the myopia which often accompanies dealing with an integrated product.

The role of testing is to rediscover the nature of the software, in essence a second opinion on its operational capabilities. (The first opinion, that of the designer-developer is too often biased.)

Figure 7. TEST DATA GENERATION

DEFINITION	TEST	EXAMPLE
dX	CHANGE SHAPE	ARRAY ⟷ WORD
	CHANGE SIZE	SMALL ⟷ BIG
	CHANGE TYPE	INTEGER ⟷ FLOATING ALPHA ⟷ NUMERIC
	CHANGE RANGE(S)	EXTREMES MEAN
	. . . . .	
$\frac{dR}{dX}$	CHANGE ON RESULTS DUE TO dX	
	. . . . .	
$\frac{dS}{dX}$	CHANGE ON STRUCTURE DUE TO dX	
	. . . . .	

TASC
THE ANALYTIC SCIENCES CORPORATION

Figure 8. TEST DATA FOR SAMPLE

X	dX	ρ(X)
1	"1" = 1	{R; C} = {P,L3,L1,J; C}
2	"2" = 2	{R; C} = {L3,L1; C}
12	"12" = 12	{R} = {J}
600	"600" = 600	{C}
L2	L2 FLOATING L2 NEGATIVE L2 ZERO L2 LARGE	{R; C} = {P,MD,L3,L1,J; C}
L4	L4 FLOATING L4 NEGATIVE L4 ZERO L4 LARGE	{R; C} = {P,MD,L3,L1,J; C}

TASC
THE ANALYTIC SCIENCES CORPORATION

The work described here is one step in an extended effort at The Analytic Sciences Corporation (TASC) to develop a complete software test strategy. Such a strategy would address the different aspects of a program: control, data, operations, timing, etc. The approach discussed in this paper is intended as a way to get a handle on the testing of the data aspect of a program. This technique admittedly treats the program as a static entity and hence misses the execution relationships between data elements. The techniques described in this paper are intended to serve as much as a format of addressing this problem as a technique for actually deriving tests.

REFERENCES

1) L.D. Fosdick and L.J. Osterweil, "Data Flow Analysis in Software Reliability", ACM Computing Surveys, September 1976, pp. 305-330

2) W.E. Howden, "Methodology for the Generation of Program Test Data", IEEE Transactions on Computers, May 1975, pp. 554-560.

3) C.V. Ramamoorthy et al, "On the Automated Generation of Program Test Data", IEEE Transactions on Software Engineering, December 1976, pp. 293-300.

4) "MIS Using Test Data Generators to Reduce Software Development Costs", Dept. of the Army Technical Bulletin, T.B. 18-22, 1974.