A Methodology for Test Selection

J.A.N. Lee
Xudong He

TR 88-29

A METHODOLOGY FOR TEST SELECTION

September 2, 1988

J.A.N. Lee
Xudong He

Department of Computer Science
Virginia Tech
Blacksburg VA 24061
(703) 961-5780

Abstract[1]

JAN Lee and Xudong He

Department of Computer Science
Virginia Tech
Blacksburg, VA 24060

Software creation requires not only testing during the development cycle by the development staff, but also independent testing following the completion of the implementation. However in the latter case, the amount of testing that can be carried out is often limited by time and resources. At the very most, independent testing can be expected to provide 100% test coverage of the test requirements (or specifications) associated with the software element with the minimum of effort. This paper describes a methodology employing *Integer Programming* by which the amount of testing required to provide the maximum possible test coverage of the test requirements (for the given test set) is assured while at the same time minimizing the total number of tests to be included in a test suite. A collateral procedure provides recommendations on which tests might be eliminated if less than 100% test coverage of the test requirements is permitted. This latter procedure will be useful in determining the risk of not running the minimum set of tests for 100% test coverage. A third process selects from the test matrix the set of tests to be applied to the system following maintenance modification of any test requirements -- that is, to provide a submatrix for regression testing. The potential benefits for applying the integer programming technique in test data selection is also discussed.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging. D.2.10 [**Software Engineering**]: Design - Methodologies. G.1.6 [**Numerical Analysis**]: Optimization - Integer Programming.

General Terms: Testing, Regression Testing, Test Data Selection, Tests Selection, Test Requirements, Test Matrix, Test Coverage.

---

[1] The research described herein was partially developed under contract for the Naval Surface Weapons Center, Dahlgren VA, Division N44 under the supervision of Mrs. Penny Selph. Contract No. N60921-84-D-A127, Southeastern Center for Electrical Engineering Education (SCEEE), St. Cloud, FL.

A Methodology for Test Selection

1. Introduction

Testing is a fundamental part of the development and maintenance phases in the life cycle of any software system from the very small to the enormous, although the intensity of testing probably increases exponentially with the size of the intended product and its required reliability level. Unfortunately although the degree of testing needs to increase exponentially, the time provided for *acceptance testing* [3] is commonly limited to linear increases. Many of the projects with which we have been involved had the time period available for acceptance testing set up to 5-8 years previously. The cumulative slips in the timing of the delivery of components or integrated systems were subtracted from the final testing period so as to maintain the promised delivery date. While we do not condone such poor management, it is important to develop a test plan which minimizes the number of tests which need to be applied while still satisfying the project requirements by examining every *test requirement* [10]. A methodology employing integer programming formulation is developed herein by which a minimal set of tests can be selected.

The acceptance testing of a software system proceeds in three steps:

1. The development of a test suite of activities composed of programs and/or procedures (which may be manual or administrative), with the objective that every test requirement in the system is to be examined,

2. The verification of the adequacy[2] of that collection of activities with respect to the *test coverage* of the test requirements, and the reduction of the test suite to remove any redundancies, and

3. The application of these activities to the targeted system and the formal documentation of the comparison of the expected versus the actual results.

This paper deals with the second of these activities, i.e. for a given test suite and a set of test requirements developed in the first activity , to find a minimum set of tests to be conducted while maintaining the maximum possible test coverage.

In some instances we have also been faced with the possibility of not being able to conduct a complete set of acceptance tests in the remaining time available and thus were required to provide a recommendation as to which subset of tests would provide the greatest test coverage of the test requirements in the time allotted. A methodology for accomplishing this test selection is presented herein.

Complete test requirements coverage is not required when testing a system during the maintenance period and following the modification of certain functionalities. Two conditions can exist in this situation:

1. The modification corrects an error and the original test requirements have not been altered, or

2. New features have been added to the system together with new test requirements which now require examination.

In either situation it is not necessary that the complete test suite be applied to the system; instead, only those test requirements which are affected by the modification need to be tested. The methodology developed herein also selects the appropriate tests for such regressive testing.

While *test selection* tells how to use existing tests during the acceptance testing phase, *test data selection* deals with how to construct tests at design, implementation and testing phases. Integer programming formulation has the potential to reduce test data redundancies, to minimize test costs

---

[2] We will assume that adequacy has different meanings depending on the context -- 100% (but minimal) test coverage during acceptance testing, less than 100% in the maintenance phase or in special cases of "random" independent testing.

and to avoid exhaustive testing. The implication for applying integer programming techniques to test data selection is discussed later.

2. The Method for Test Selection

2.1. The Construction of the Test Matrix

There are a number of different methods for acquiring an adequate test suite. Clearly the suite should contain tests which, for each requirement, (1) are typical of the applications of the system, and (2) test the limits of the domain of application.[3] These tests may be developed as a part of the implementation process itself by the system development group or, more appropriately, are created independently by the testing organization. Where a system is to be placed in an already existing environment with an existing library of applications (data sets or procedures), that library is a candidate for the basis for the test suite. For example, in the case of the introduction of a new compiler into an environment which already contains a library of programs written in that language, or a library that can be readily converted to that language by mechanical means, there is a ready-made test suite of typical programs.

Whatever the source of the test suite, the individual tests need to be placed under configuration management in the same manner as the modules of the system under test are controlled. In fact, links between the modules and their tests are highly appropriate to resolving the test requirement. As the system is maintained, any modifications to the modules can be validated by the linked tests, or if the test requirements are changed, the tests must be updated to accommodate the functionalities of the new module. It is possible to construct a binary *test matrix* in which the entries cross-reference test requirements and tests, in much the same manner as a similar matrix can cross-reference test requirements and fulfilling modules. Let us propose that the test matrix is row indexed by the test requirements, and column indexed by the identifiers of each test. A test requirement is *test covered* by some test if and only if the corresponding row is not a zero row. The complete or 100% test coverage of the test requirements is defined as the state in which every test requirement is test covered.

Prather [12] provides an example of a program which classifies triangles, which is required to conform to the following test requirements:

> **Input:** Three positive integers, $a \leq b \leq c$.
> **Output:** An indication as to whether the input are:
>
> 1. not the sides of a triangle,
> 2. the sides of an equilateral triangle,
> 3. the sides of an isosceles triangle,
> 4. the sides of a scalene right triangle,
> 5. the sides of a scalene obtuse triangle,
> 6. the sides of a scalene acute triangle.

From these initial test requirements Prather distinguishes eight functionalities to be tested; dividing the first indicator into two cases representing the invalid and valid cases (which we shall designate as 1i and 1v respectively), and by noting that there are two distinct conditions for identifying isosceles triangles (3a and 3b) provide two additional cases. Prather then provides eight data triples t1 ... t8 each of which examine an individual test requirement output, resulting in the following test matrix:

---

[3] Goodenough and Gerhart [4] suggest that there are five conditions which must be met by a test suite; in this study we consider that these conditions are part of the test requirements to be met.

	Test Identifier							
Test Requirements Identifier	t1	t2	t3	t4	t5	t6	t7	t8
1i	1	0	0	0	0	0	0	0
1v	0	1	1	1	1	1	1	1
2	0	0	1	0	0	0	0	0
3a	0	0	0	1	0	0	0	0
3b	0	0	0	0	1	0	0	0
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	0	1	0
6	0	0	0	0	0	0	0	1

From this matrix it is obvious that a special test for test requirement 1v (that is, test t2) is unnecessary since that same test requirement is incidentally examined in other tests.

A second (and third) level of test development can be envisaged in which pairs (and triples) of test requirements are examined in combination. Pairings would result in squaring the size of the matrix [9], and sequences of pairs (A before B, A after B) would result in a further doubling of the matrix dimensions. Many combinations and sequences are likely to be impossible or highly unlikely, and thus may be eliminated from consideration. However any test suite which considers pairings of test requirements, also provides for tests of individual test requirements. Consequently, it is possible that tests related to individual test requirements may be eliminated from the test suite, provided that those same test requirements are "covered" by the pairings related tests.

During the construction of the test suite, each test should be analyzed to determine the test requirements which it covers. That is, while a test may be designed to cover a specific test requirement, other test requirements may be covered incidentally. For example, most tests will involve input or output, even though the test does not have the specific objective of testing those facilities. The results of this analysis should be incorporated into the test matrix. Following such an analysis, new tests should be acquired (or planned) to remove any deficiencies in the test coverage of the test requirements. At this stage it is not necessary to eliminate any tests from the test suite as the result of any recognizable redundancy of test coverage. At the unit (or module) level, it is better to utilize a test specifically designed to cover the corresponding test requirement(s) than a test which incidentally provides the same test coverage. In some cases it may not be possible to test the module using a more comprehensive test since the matching modules are not yet available and the provision of drivers or stubs is inappropriate to the stage of integration.

At the stage of acceptance testing, the test matrix for the complete suite of tests should be created and analyzed for redundancy. The coverage method described in the next section performs such an analysis and produces a reduced test suite which still provides 100% test coverage of the test requirements. The choice of tests is determined on the basis of the cumulative priorities of the test requirements as defined by the user and the complexity of the tests as defined by the test designers. That is, the priority or importance of a test is the sum of the priorities of the test requirements it covers. In terms of the algorithms for test requirements coverage, the product of the complexity and the inverse of this cumulative priority is used as the *cost* of the test. Thus the coverage algorithm selects that set of tests which has the minimum cost.

In our experience in working with several mission critical systems, the consumers were able to provide a priority ordering to the test requirement and some quantitative measure of the importance of each requirement. We were able to easily use this information in preparing test matrix data. Similarly, the cost of testing was readily available in terms of time and manpower required to accomplish similar tests at the unit level or in prior implementations.

2.2. Formalizing The Problem

Let us construct a binary test matrix in which test identifiers are the column indices and the identifications of the test requirements (natural numbers in this program) are the row indices, entry [i,j] is 1 if and only if test requirement i is to be examined by test j. The test selection problem is to find a subset of the tests which covers the given set of test requirements of a software system with minimum cost.

The set covering problem is not uncommon, having been applied previously to such problems as scheduling (air-line crew, truck), political redistricting, switching theory, line balancing, information retrieval, and capital investment [11]. Mathematically the set covering problem can be expressed as:

$$\begin{array}{ll} \text{minimize} & \vec{c}\,\vec{x} \\ \text{subject to} & E\vec{x} \geq \vec{e} \\ \text{and} & x_j = 0 \text{ or } 1 \quad (j = 1,..,n) \end{array}$$

where $E = (e_{ij})$ is an m by n matrix, row indexed by test requirements and column indexed by tests, the entries e_{ij} (1 or 0) reflect whether test requirement i is being examined by test j or not, $\vec{c}$ is a cost vector representing the cost of conducting each test and which is to be minimized.[4] $\vec{x}$ is a binary vector which identifies the tests to be conducted, and $\vec{e}$ is a unit vector.

The method for solving this type of problem is *integer programming* [5], but, to our knowledge it has not been applied to the problem of choosing the set of tests to be conducted in the software development process though *linear programming* technique was mentioned in [13] to be used in selecting an optimal set of test tools for implementation.

2.3. Integer Programming -- Implicit Enumeration Method

There are three major methods in solving 0-1 integer programming problems, especially the set covering problem. They are *Gomory Cutting Plane* [2], *Branch and Bound* [7], and *Implicit Enumeration* [1].

The Gomory cutting plane method is not very reliable in solving 0-1 integer programming problems due to machine round-off errors, and a random change of the constraints may increase the number of iterations required significantly. Only the first few cuts progress towards the optimal solution rapidly, then the solution process tends to remain constant for many iterations. The branch and bound method is too general so that large problems would require unacceptably long execution times. The implicit enumeration method is the best out of the three major ones in solving 0-1 integer programming problems due to its simple principles and freedom from machine round-off errors; so far it is most widely applied in solving 0-1 integer programming problems [11].

The implicit enumeration method was discovered by Balas [1] in solving 0-1 integer programming problems. Later, improvements were made by others [14]. In implicit enumeration method, only a small subset of all possible combinations of search space is explicitly enumerated. Though the method can be considered as a special case of branch and bound method, a different approach is taken to obtain efficient solutions to 0-1 integer programming problems.

The heart of the implicit enumeration algorithm is a Point Algorithm which keeps track of the variables already fixed at 0 or 1 and the remaining free variables. The remaining free variables and the associated constraints constitute subproblems which are of the same type as the original 0-1 integer problem. Variables are fixed at 1 at a forward step and are cancelled to 0 when they are revisited in a backward step. Many techniques have been developed to speed up fixing variables, such as the ceiling test, the nonnegative cost test, the infeasibility test, the cancellation zero test, the cancellation one test, linear programming, post optimization, and surrogate constraints [14].

---

[4] The cost vector was computed in the implementation associated with the sponsoring project, as the product of the complexity of each test and the inverse of the sum of the priorities of the test requirements covered by each test.

A test requirements coverage analysis algorithm using implicit enumeration method and linear programming technique is developed and implemented on both IBM 3090 and IBM PC (in PASCAL). A sketch of the basic implicit enumeration algorithm is presented in the appendix.

The results of the analysis are simply a listing of the tests which should be applied to the system under test in order to assure that 100% of the test requirements specified in the system documentation are covered. Where there is a choice between subsets of tests which would satisfy the test requirements coverage, the subset which provides the same test coverage with the minimum (cumulative) cost is selected. The minimal test set is that set which, while providing complete test coverage of the test requirements, also minimizes the cost of testing.

2.4. The Application of the Coverage Algorithm

The following example (modified from [8]) illustrates the application of the coverage algorithm:

Suppose a software system has 15 test requirements to be examined and 32 tests that have been designed independently to test the system and with the following complexities[5] :
(1, 1, 1, 1, 1, 1, 1, 3, 4, 4, 4, 4, 2, 4, 4, 6, 6, 9, 15, 12, 12, 12, 20, 30, 25, 25, 24, 24, 18, 49, 64, 90);
the relationship between the test requirements and tests is represented in the following matrix E:

Test Reqts	Tests (32 total)																													
					5					10					15					20					25					30		
1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	1
2	0	1	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	1	0	1	0	1	0	0	0	1	0	1	1	1	1
3	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	1	0	1
4	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	0	0	0	1
5	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	1	1	0	0	0	1	0	0	0	1	0
6	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	1	0	1
7	0	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	1	1	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	1	0	0	1	0	0	0	0	1
9	0	0	0	0	0	0	0	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	0
10	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1	1
11	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	1	0	1
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	1	0	1	0	0	1	0	1	0
13	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	1	0	1
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0	0	0	0	0	1	0
15	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	1

Assume the priority for each test requirement is uniformly 1, the cost vector $\vec{c}$ associated with the 32 tests is calculated (cost[i] = complexity[i] / ($\sum_{j=1}^{15} priority[j] \times E[i,j]$)) as:

(1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 3 3 3 3 4 4 4 4 5 5 5 6 6 6 7 8 9)

(1) The optimal linear programing solution representing the vector which defines the choice of tests to be utilized is:

---

[5] The values are chosen in order to develop a simple solution and as an illustration of the application of the coverage algorithm. The values stand for the relative complexity of conducting each of the tests; in an actual application, the complexity of each test could be calculated based on several criteria: the length of a test (space), the running time of a test and the priority of a test.

x = (0, 0, 0, 0.6, 0, 1, 0.4, 0.8, 0, 0.4, 0.6, 0, 0, 0, 0, 0, 0, 0.8, 0.6, 0, 0, 0, 0.2, 0, 0.4, 0, 0, 0, 0, 0, 0.2, 0)

with a minimum cost $\sum_{i=1}^{32} c[i] \times x[i] = 13.4$; obviously, the only appropriate values for selection are 0's and 1's, thus

(2) an integer solution y is obtained by setting the non-zero entries of the optimal solution x to 1's:

y = (0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0)

The cost associated with the solution y is $\sum_{i=1}^{32} c[i] \times y[i] = 31$ which is much worse than the optimal cost 13.4;

(3) The sub-subproblem is then formed by retaining exactly the columns corresponding to the unit entries in the solution y, except those already fixed at 1 in the optimal solution x (here test 6, which has been selected and will not be considered anymore) and retaining all the rows except those corresponding to the 1's in the optimal solution x (here test requirement 6):

Test Reqts	Test Numbers									
	4	7	8	10	11	18	19	23	25	31
1	0	0	0	0	0	0	1	0	1	0
2	0	0	0	0	0	0	1	1	0	1
3	0	0	0	1	0	0	1	0	0	0
4	1	0	0	0	0	0	0	0	1	0
5	0	0	0	0	1	0	0	1	0	1
7	0	1	0	0	0	0	0	0	1	1
8	0	0	1	0	0	0	0	1	0	0
9	0	0	1	0	0	0	0	0	0	1
10	0	0	0	1	0	0	0	0	1	1
11	0	0	0	0	0	1	0	1	0	0
12	0	0	0	0	0	1	1	0	0	1
13	0	0	0	0	1	0	0	0	1	0
14	0	0	0	0	0	0	1	1	0	1
15	0	0	0	0	0	1	0	0	0	1

(4) The overall optimal integer solution $y_{optimal}$ is acquired by merging the optimal integer solution of this sub-subproblem to the rest already fixed selections:

$y_{optimal}$ = (0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

with the cost $\sum_{i=1}^{32} c[i] \times y_{optimal}[i] = 14$ which is the closest integer value to the optimal cost 13.4.

That is, the tests 4, 6, 7, 8, 10, 11, 18 and 19 have been selected for examining the test requirements.

3. Subsequent Processing

In many cases, the time available for acceptance testing is not sufficient to allow for 100% test coverage of all the test requirements. On the assumption that unit and pre-formal testing has provided 100% test coverage of the first-level test requirements, an acceptance test may need to cover only a sample of the test requirements. The methodology described in the succeeding section provides an ordering of tests in the reduced test suite, ordered by increasing cost with the implication that the tests to be omitted would be those with the highest cost. The technique is then extended to create a reduced submatrix of the tests to be conducted during regression testing

following the installation of changes to the system which affect only certain test requirements. This new test matrix may be further reduced using the coverage algorithm, and thus may be generated initially either from the original (unreduced) test matrix, or that matrix generated from the test coverage analysis procedure.

The reduced test matrix is most useful in acceptance stage of testing since it relies to a great extent on the authenticity of the tests which are applied incidently during a test associated with some other primary objective. It is possible to alter the weighting of an element of a test vector in the original matrix by using other values than zero (0) or one (1) thus influencing the cost of the test. Provided that the implementation of the coverage algorithm uses only the existance of a non-zero value (as contrasted with the use of the actual value), emphasis can be given to the objective of the test by increasing its associated value in the test vector, while maintaining the values associated with the incidental tests at a lower value.

3.1. Regression Testing

The selection of a submatrix for regression testing is accomplished in two stages:

1. Given the set of test requirements to be examined that is, the test requirements which are affected by the the changes which have been implemented), select the set of tests from the (original or reduced) test matrix which cover these test requirements. This set of tests to be administered is simply the **or** of the rows of E corresponding to the modified test requirements. This process reduces the columns in the matrix to those applicable. For example, in the above case, if test requirement 5 were modified, tests 11, 23, and 31 would be selected.

2. From this column-reduced matrix, the incidently tested requirements can be deduced, and if necessary the matrix can be further reduced to include only those rows which apply to the test requirements covered.

Consider the following example and sequence of reductions from an original test matrix:

Test Requirements	Test Identifier							
	t1	t2	t3	t4	t5	t6	t7	t8
1	1	1	0	0	0	0	0	1
2	0	1	0	1	0	1	0	0
3	1	0	1	0	1	0	0	0
4	0	0	0	1	0	0	1	0
5	0	0	1	0	1	0	0	0
6	0	0	0	1	0	1	0	0
7	0	1	0	0	0	0	1	0
8	1	0	0	0	1	0	0	1

Assuming that the priorities of the test requirements and the complexities of tests are uniform , the coverage algorithm reduces the test suite to three tests -- test numbers 2, 4 and 5. The resultant test matrix is:

Test Requirements	Test Identifier		
	t2	t4	t5
1	1	0	0
2	1	1	0
3	0	0	1
4	0	1	0
5	0	0	1
6	0	1	0
7	1	0	0
8	0	0	1

This reduced test matrix, containing only three tests, obviously has the same complete test coverage as the original matrix. The costs of all the tests are equal, each test covering three test requirements. However, if test t2 were omitted from the suite, then two test requirements would not be covered by the remaining tests (test requirements 1 and 7). Similarly the omission of test t4 would leave test requirements 4 and 6 uncovered. The omission of test t5 however would not provide test coverage for three test requirements (3, 5 and 8). Thus the preferred order of omission should place test t5 as the least likely to be omitted; the other two having equal cost, they could be omitted in any order.

If regression testing were to be conducted from this test suite with the need to examine (say) test requirements 2 and 6 then only tests t2 and t4 need to be used, since the corresponding rows used to select the tests show that test t5 is not applicable to this requirement:

Test Requirements	Test Identifier		
	t2	t4	t5
2	1	1	0
6	0	1	0

However, during the regression test, test requirements 1, 4, and 7 are examined incidently, which probably implies an interdependence of test requirements, and resulting in the test matrix:

Test Requirements	Test Identifier	
	t2	t4
1	1	0
2	1	1
4	0	1
6	0	1
7	1	0

This regression test matrix already provides minimal test coverage and does not need to be further analyzed for test coverage. The same regression test matrix would have been achieved by selecting the regression test matrix from the original test suite before test coverage analysis and reduction, that is reversing the order of the test coverage analysis and the regression analysis phases.

3.2. Recommendations for Less than 100% Test Coverage

In order to provide guidance on the set of tests which should be applied when less than 100% test coverage of the test requirements is permissible, a listing of the tests in ascending cost (or descending

priority) order can be created. The tests in the first part of this list are then those with the lowest cost and which should be included in a less than complete test coverage test suite. The portion of the list to be used should be the subvector of tests in cost order which includes the test with the lowest cost. If it is necessary to ascertain which test requirements are not covered in this subvector, then the regression analysis method can be used to create a reduced test matrix and consequently a reduced test requirements coverage list.

3.3. Test Matrix Partitions

In a very large system, the test matrix may be too large to be processed in a single pass. In other situations, it would be preferable to partition the test suite into segments and to concentrate the tests for specific purposes such as the set of tests relevant to a module or subsystem. For example, while it to be expected that most system tests will involve input and output as incidental activities, it may be preferable to partition the test suite so that tests with the specific objectives of testing the input/output requirements are individualized and are not pre-empted by tests which cover the same test requirements incidently, as was seen in a prior example.

Where such partitioning is preferred, or where the test matrix must be partitioned for the purposes of processing, the matrix should be partitioned by columns (tests) which are interrelated. It is unlikely that any of these partitions will have 100% test coverage of the test requirements in the whole system, and thus to analyze the partition for test coverage will require the elimination of the non-applicable test requirements. Once the partitions have been reduced individually, each partition may then be treated as a single test, the test condensed vector being the sum of the individual test coverages. The partitions may then be reassembled into a new test matrix in which each column represents a partition. This matrix may then be further reduced if necessary.

Consider the following matrix, which has been divided into three partitions (columns 1-4, 5-10, and 11-15):

```
Partition Number

 1       2       3

1010 000000 00000
0011 000110 01000
0000 100100 00111
1001 000001 00000
0000 100101 10011
1100 010101 01010
1010 101101 00000
```

which collapses to the matrix below by **OR**ing the columns in each partition so that each new column represents the test coverage of the partition of the original test suite:

```
Partition Number

 1 2 3

1 0 0
1 1 1
0 1 1
1 1 0
0 1 1
1 1 1
1 1 0
```

which can be reduced to include only partitions 1 and 3 while still providing the necessary test coverage. Thus independent of the possible reduced form of partition 2, all of its test coverage can be provided in the other two partitions.

3.4. The Implications of the Method for Test Data Selection

The method developed for tests selection has the potential to help designing better test data (test predicates) in the following aspects:

1. to reduce redundancy of test data (overlap of test predicates),
2. to minimize the test costs, and
3. to avoid exhaustive testing.

The adaption of the method to test data selection can be achieved in the following way:

1. construct a functionality - test data matrix where functionalities of a program (module) are regarded as row indices and test data are treated as column indices. Techniques of [6] can be used for identifying functionalities, and
2. build a cost vector by calculating the cost of each test datum in terms of its complexity and significance

However, further study is needed to make this application feasible and practical.

4. Discussion

The design of a test set for a software system can evolve in a number of manners. On the one hand an in-house development activity can merely collect the tests used by individual programmers and assimilate these into a collection which is later used in regression testing and during maintenance. Alternatively a test plan may be developed which matches tests to test requirements independently of the top level development design specifications. In either case there is a need of ensure 100% test coverage of some chosen set of facilities. In the former case this may be based on test coverage of system features (culled from the design specification) while in the latter the system characteristics are best exemplified by the original system requirements document. A test plan should then require the management of a test matrix which records the incidence of tests and facilities tested.

However as a part of the development of individual tests it is often necessary to utilize elementary features (and thus examine certain test requirements) in order to initialize the system in preparation for examining a specific feature or test requirement. In fact, most tests are designed with specific objectives in mind, though other facilities are utilized. It is most likely that given a set of tests and their primary objective features or test requirements that close to 100% test coverage (that is, not involving a high degree of redundancy) is achieved. However, if the incidental uses of features or test requirements is taken into account considerable overlap and redundancy is present in the test set, this is especially true when the test matrix is dense.

This test coverage analysis package will be most useful in the latter case, enabling the test manager to reduce the test set to only that set of tests which are essential to provide the desirable test coverage. Initially the system can be used to confirm 100% test coverage, secondly to reduce the number of tests to a minimum and thirdly to provide direction on providing a test set which provides less than 100% test coverage.

It must be expected that pre-formal testing of any system provides 100% test coverage, but that acceptance testing may be limited (both by time and resources) to some smaller percentage. By providing realistic priorities to test requirements (for example giving higher priorities to test requirements which are close to the highest level of the system) the most important test requirements can be tested in an acceptance test. The same strategy can be applied to regression testing and maintenance activities. In either of the latter cases the adjustment of the priorities of the test requirements to emphasize those which are under suspicion (such as by the installation of

new versions of specific modules) a test order can be chosen to provide less than 100% test coverage while still concentrating on the locale of the changes.

This set of procedures may be considered to be the initial entries into a test environment which eventually would include other tools. Candidates for inclusion in such an environment might include a test configuration management system and test requirements tracking tables.

References

[1] Balas, E.: "An Additive Algorithm for Solving Linear Programs with Zero-One Variables", Operations Research, Vol.13, 1965, pp.517-546.

[2] Gomory, R.E.: "Outline of An Algorithm for Integer Solution to Linear Programs", Bulletin of the American Mathematical Society, Vol.64, 1958, pp.275-278.

[3] Goodenough, J.B.: "A Survey of Program Testing Issues", in *Research Directions in Software Technology* (ed. P. Wegner), The MIT Press, 1979, pp.316-342.

[4] Goodenough, J.B. and S.L. Gerhart: "Toward a Theory of Test Set Selection", IEEE Trans. on Soft. Eng., SE-1, No.2, June 1975, pp.157-173.

[5] Holzman, A.G.: "Mathematical Programming - for Operations Researchers and Computer Scientists", Marcel Dekker, 1981.

[6] Howden, W.E.: "A Functional Approach to Program Testing and Analysis", IEEE Trans. on SE, Vol.SE-12, No.10, Oct. 1986, pp.997-1005.

[7] Land, A.H., and A. Doig: "An Automatic Method of Solving Discrete Programming Problems", Econometrica, Vol.28, 1960, pp.497-520.

[8] Lemke, C.E., H.M. Salkin and K. Spielberg: "Set Covering by Single Branch Enumeration with Linear-Programming Subproblems", Operations Research 19(4), 1971, pp.998-1022.

[9] Mandl, R.: "Orthogonal Latin Squares: An Application of Experiment Design to Compiler Testing", Comm. of ACM, Vol.28, No.10, Oct. 1985, pp.1054-1058.

[10] McGowan, C.L. and R.C. McHenry: "Software Management", in *Research Directions in Software Technology* (ed. P. Wegner), The MIT Press, 1979, pp.207-253.

[11] Ozan, T.M.: "Applied Mathematical Programming for Production and Engineering Management", Prentice-Hall, 1986.

[12] Prather, R.E., "Theory of Program Testing -- An Overview", The Bell System Tech. Jour., Vol.62, No.10, Part 2, December 1983, pp.3073-3105.

[13] Ramamoorthy, C.V. and S.F. Ho: "Testing Large Software with Automated Software Evaluation Systems", IEEE Trans. on SE, Vol.SE-1, No.1, March 1975, pp.46-58.

[14] Salkin, H.M.: "Integer Programming", Addison-Wesley Publishing, 1975.

Appendix

The Coverage Algorithm

The test requirements coverage analysis algorithm is based on the work of Lemke et al [8] which uses implicit enumeration with linear programming method. The heart of the algorithm is the Point Algorithm which keeps track of the variables already fixed at 1, those cancelled (fixed at 0), and the remaining free variables which form the subproblem at next level.

Several techniques are used in the algorithm to speed up the searching process:

1. Linear programming:

   The Dual Simplex method is repeatedly used in solving the subproblems at different level, starting with the original problem, so that several variables may be fixed at the same time instead of one at a time. Hence searching time is considerably reduced.

2. Ceiling Test:

The ceiling test is used in several places in the algorithm. Two ceilings are kept throughout the searching process; one is the optimal (minimal) cost for the corresponding linear programming problem of the initial coverage problem. The optimal cost is fixed once it is obtained from solving the linear programming problem and is a lower bound of the original integer programming problem. This ceiling is used to judge whether the current integer solution is optimal or not. The second ceiling is the cost associated with the best integer solution found so far; it may be replaced once a better integer solution is found and is used to fathom (terminate) those branches of the searching tree which yield costs worse than it. Thus, the search space is greatly reduced.

3. Extreme point checking:

   When a feasible integer solution y is not an extreme point (i.e. the columns of E corresponding to $y_i = 1$ and the extended columns corresponding to positive slack variables[6] form a linear dependent set) the solution can be reduced to an extreme point of the corresponding linear problem with a better cost so that a tighter ceiling can be obtained and thus speed up the fathoming process.

4. Extracting and solving the sub-subproblem:

   The sub-subproblem consists of only those columns of the binary matrix E associated with the positive fractional variables and the rows corresponding to the constraints they satisfy. By rounding an extreme point to an integer solution, some variables may violate the constraints. A better integer solution can be obtained by deleting those variables which violate the constraints.

---

[6] Slack variables are temporary variables introduced to transform the initial inequality problem: $E\vec{x} \geq \vec{e}$ into equality problem $E\vec{x} - I\vec{s} = \vec{e}$, where I is the unit matrix, they do not affect the optimal solution of the initial problem and will be eliminated in the final optimal solution.

Dr. "JAN" Lee is currently Professor of Computer Science in the Department of Computer Science at Virginia Tech, on release to serve as the Director of the Institute of Information Technology for the Commonwealth of Virginia.

He has been involved with the administration of Computer related activities since 1959 when he was appointed the Director of Computing at Queen's University at Kingston Ontario. Subsequently he served as the Associate Director of Computing and Head of the Computer Science Program at the University of Massachusetts, and as Deputy Head and then Head of the Department of Computer Science at Virginia Tech. Since 1960 Dr. Lee has been principal or co-principal investigator on grants and contracts totalling more than $16,500,000.

Since 1966, Dr. Lee has served in several capacities within the Accredited Standards Committee X3 (Information Processing Systems), receiving a Certificate of Appreciation for his significant services to the industry from the Business Equipment Manufacturers Association in 1971. Dr. Lee served as the Secretary of the US Delegation to the International Standards Organization 1981-87.

As a member of the Association of Computing Machinery, Dr. Lee has represented the interests of the Association in the Standards Development processes since 1968 and testified on behalf of the Association in Federal Trade Commission hearings on proposed rules on Standards and Certification. In 1980, 1985 and 1986 he received the ACM Certificates of Recognition and was given the ACM Outstanding Contribution Award for 1981. He was elected as a Member-at-Large to the ACM Council in 1982, and as Vice President in 1984.

Dr. Lee has served on the Editorial Board of the Annals of the History of Computing since 1980, being appointed as the Editor of the Meetings in Retrospect department in 1984 and as Editor-in-Chief in 1987.

Dr. Lee is the author of The Anatomy of a Compiler, Numerical Analysis for Computers, Computer Semantics, and many other technical papers including four American National Standards.

Mr. Xudong He received his B.S. and M.S. degrees in Computer Science from Nanjing University, China, in 1982 and 1984 respectively. He is now a Ph.D candidate in the Department of Computer Science at Virginia Tech. His current research interests are Programming Languages, Formal Semantics, Software Engineering and AI. He is a co-author for several papers in these fields.

Mr. He is a student member of ACM and IEEE Computer Society.