Coverage Analysis Methods for Formal Software Testing

J.A.N. Lee
Xudong He

TR 87-4

COVERAGE ANALYSIS METHODS FOR FORMAL SOFTWARE TESTING

January 7, 1987

J.A.N. Lee
Xudong He

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

Abstract

Software creation requires not only testing during the development cycle by the development staff, but also independent validation 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% coverage of the requirements (or specifications) associated with the software element. This report describes a methodology by which the amount of testing required to provide 100% coverage of the requirements is assured while at the same time minimizing the total number of tests included in a test suite. A collateral procedure provides recommendations on which tests which might be eliminated if less than 100% coverage of the requirements is permitted. This latter process will be useful in determining the risk of not running the minimum set of tests for 100% coverage. A second process selects from the test matrix the set of tests to be applied to the system following maintenance modification of any module -- that is, to provide a submatrix for regression testing.[1]

Keywords: Testing, Regression Testing, System Requirements, 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.

Table of Contents

Introduction

Testing is a fundamental part of the development and maintenance life cycles of any software system from the very small to the enormous, though 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 formal, or acceptance, testing is often limited to linear increases. Many of the projects with which we have been involved in the Acceptance Testing phase had the time period available for testing set up to 5-8 years previously. The cumulative slips in the timing of the delivery of components or integrated systems were taken from the final testing period so as to maintain the promised delivery point. 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 run while still satisfying the project requirements of testing each and every specified element in the system specifications.

Complete 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 system specifications have not been altered, or

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

In either situation it is not necessary that the complete test suite be applied to the system; instead, only those requirements which are affected by the modification need to be tested. A methodology is proposed by which the appropriate tests can be selected.

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 time available and thus were required to provide a recommendation as to which subset of tests would provide the greatest coverage of the requirements in the time allotted. A methodology for this test selection is accomplished is presented.

The Problem

The formal 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 requirement in the system is to be examined,

2. The verification of the adequacy[2] of that collection of activities with respect to the "coverage" of the 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, the third being assumed to be a normal outcome of the previous two and not to be considered here. However before turning to the consideration of the second problem, let us review the common methods for developing an initial test suite.

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 to be used as the basis for the test suite. 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 a configuration management system in the same manner as the modules of the system under test is being controlled. In fact, links between the modules and their tests are highly appropriate. As the system is maintained, any modifications to the modules can be validated by the linked tests, or if the requirements are changed the tests must be updated to accommodate the new module functionalities. From the management point of view, it should be possible to construct a binary test matrix in which the entries cross-reference requirements and tests, in much the same manner as a similar matrix cross-reference requirements and fulfilling modules. Let us propose that the test matrix is row indexed by the system requirements, and column indexed by the test identifiers. Idealistically, such a test matrix will be square, there being exactly one test for each requirement. If there are more tests than requirements then it is likely that some requirements need subdivision. For example, it is likely that

---

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

[3] Goodenough and Gerhart [9] 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 requirements specifications to be met.

each requirement is matched with tests using typical data and tests which probe the boundaries of the data domain.[4] Truly these represent two different requirements.

Prather [17] provides an example of a program which classifies triangles, which is required to conform to the following specifications:

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

From these initial specifications Prather distinguishes eight functionalities to be tested by dividing the first 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). Prather then provides eight data triples t1 ... t8 each of which tests an individual requirement, resulting in the following test matrix:

Test Identifier

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

From this matrix it is obvious that a special test for requirement 1v (that is, test t2) is unnecessary since that same requirement is incidentally validated in other tests. That is, 100% of the requirements could be covered with a less than square test matrix.

A second (and third) level of test development can be envisaged in which pairs (and triples) of requirements are tested in combination. Pairings would result in squaring the size of the matrix, 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 requirements, also provides for tests of individual requirements. Consequently, one may suggest that

---

[4] We consider a set of tests each of which have the same objective but which differ only by the data used to be a (large) single test.

tests on only individual requirements may be eliminated from the test suite, provided that those same requirements are "covered" by the pairings related tests.

An alternative approach to acquiring tests is to use existing scenarios (programs or procedures). Similarly, performance requirements may involve benchmark tests which not only provide for resource quantifications but also exercise certain other requirements.

During the construction of the test suite, each test should be analyzed to determine the requirements which it covers. That is, while a test may be designed to cover a specific requirement, other 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 coverage of the requirements. At this stage it is not necessary to eliminate any tests from the test suite as the result of recognizable redundancy of coverage. At the unit (or module) level, it is better to utilize a test specifically designed to cover the corresponding requirement(s) than a test which incidentally provides the same 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 formal 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% coverage of the requirements. The choice of tests is determined on the basis of the cumulative priorities of the requirements as defined by the user. That is, the priority or importance of a test is the sum of the priorities of the requirements it covers. In terms of the algorithms for requirements coverage, 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 many cases, the time available for formal testing is not sufficient to provide for 100% coverage of all the requirements. On the assumption that unit and pre-formal testing has provided 100% coverage of the first-level requirements, a formal or acceptance test may cover only a sample of the requirements. One of the methodologies described in the succeeding section provides an ordering of tests in the reduced test suite. The methodology provides a list of the tests ordered by priority, with the implication that the tests to be omitted would be those with the lowest priority. The second method creates a reduced submatrix of the tests to be conducted during regression testing following the installation of changes to the system which affect only certain 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 generated from the coverage analysis procedure.

The Coverage Algorithm

The 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. Mathematically the coverage problem can be expressed as:

minimize $\vec{c}\ \vec{x}$

subject to $E\vec{x} \geq \vec{e}$

and $x_j = 0$ or 1 (j = 1,..,n)

where $E = (e_{ij})$ is an m by n matrix whose entries e_{ij} are 0 or 1, c is a cost row (or vector), x is a binary vector which identifies the tests to be conducted, and e is a unit vector.[5] The solution to this problem is well known, but, to our knowledge has not been applied to the problem of choosing the set of tests to be conducted in the software development process.

There are three methods in solving 0-1 integer programming problems, especially the covering problem. They are Gomory Cutting Plane, Branch and Bound, and Implicit Enumeration. In the following sections, these methods are briefly introduced and compared.

Gomory Cutting Plane Method

The cutting plane method was first used by Dantzig in solving integer linear programming problems, but the first convergent algorithm was developed by Gomory [8]. In this method, the original integer programming problem is initially solved as a contiguous linear programming problem by the primal or dual simplex method, then a systematic cutting process starts in order to find the optimal integer solution.

In every step (cut), a new constraint is added to the previous set of constraints so that a portion of the original solution space is cut away and all feasible integer solutions are preserved. During the cutting process, the optimality is maintained. Theoretically, the algorithm will find a feasible optimal solution in finite steps if such a solution exists. The key point of the method is the generation of "Gomory cuts" or "Gomory constraints".

Let the original problem be:

Minimize: $Z = \vec{c}\ \vec{x}$

Subject to:

---

[5] In the test matrix, the cost vector represents the cost of conducting each test, which is to be minimized. The cost vector is computed in this program as the inverse of the sum of the priorities of the requirements covered by each test.

$$a_{11}x_1 + a_{12}x_{2+} \cdots + a_{1n}x_n \geq b_1$$

$$a_{21}x_1 + a_{22}x_{2+} \cdots + a_{2n}x_n \geq b_2$$

...

$$a_{m1}x_1 + a_{m2}x_{2+} \cdots + a_{mn}x_n \geq b_m$$

where $x_i = 0$ or 1

By introducing m surplus variables y_{i+n} (1 f.le i $\leq$ m), the original problem can be transformed into an equality problem:

Minimize: $Z = \vec{c}' \vec{y}$

Subject to:

$$a_{11}y_1 + a_{12}y_{2+} \cdots + a_{1n}y_n - y_{1+n} = b_1$$

$$a_{21}y_1 + a_{22}y_{2+} \cdots + a_{2n}y_n - y_{2+n} = b_2$$

...

$$a_{m1}y_1 + a_{m2}y_{2+} \cdots + a_{mn}y_n - y_{m+n} = b_m$$

where $y_i \geq 0$ $(1 \leq i \leq m+n)$

Let v_i (i = 1,...,m) designate a basic variable which is in the initial linear solution set, and w_j (j = 1,...,n) designate a nonbasic variable. Then we have the following equation:

$$v_i = d_i - \sum_{i=1}^{n} e_{ij}w_j \quad (1 \leq i \leq n) \qquad (1)$$

Assuming that d_i is non-integer, then d_i can be expressed as an integer part $[d_i]$ plus a fraction part $\{d_i\}$ so that:

$$d_i = [d_i] + \{d_i\}, \ (0 < \{d_i\} < 1) \qquad (2)$$

Similarly,

$$e_{ij} = [e_{ij}] + \{e_{ij}\}, (0 \leq \{e_{ij}\} < 1) \qquad (3)$$

Thus

$$v_i = ([d_i] + \{d_i\}) - \sum_{j=1}^{n}([e_{ij}] + \{e_{ij}\})w_j \qquad (4)$$

By simple arithmetical manipulation, we have

$$v_i - [d_i] + \sum_{j=1}^{n}[e_{ij}]w_j = \{d_j\} - \sum_{j=1}^{n}\{e_{ij}\}w_j \qquad (5)$$

Since we need optimal integer solution, v_i and w_j must be integer. That is, if the left hand side of formula (5) is integer then so is the right hand side. By formulae (2) and (3) and the non-negativity of solution ($w_j \geq 0$), we have the following inequality:

$$\{d_i\} - \sum_{j=1}^{n}\{e_{ij}\}w_j \leq 0 \qquad (6)$$

That is

$$\sum_{j=1}^{n}\{e_{ij}\}w_j \geq \{d_i\} \qquad (7)$$

The formula (7) is called Gomory Cut or Gomory Constraint. By adding a new surplus variable s_i, the formula (7) can be transformed into a new constraint equation:

$$\sum_{j=1}^{n} \{e_{ij}\} w_j - s_i = \{d_i\} \qquad (8)$$

By adding new constraints to the original problem and repeating the dual simplex method, we effectively reduce the solution space and finally reach an integer solution while maintaining optimality.

Branch and Bound

The branch and bound method was originally developed by Land and Doig [12];, since then many variations of the method appeared in the literature. The principle of branch and bound is an systematic search for the feasible optimal solution of a given problem. The success of the method is based on the fact that only a small portion of the solution space need actually be enumerated explicitly, the remaining part being implicitly enumerated or eliminated since it contains no optimal solutions. The feasibility criteria is maintained throughout the search process.

The method usually consists of three steps: branching, bounding and fathoming. In a branching step, the feasible solution space is divided into smaller subsets, each corresponding to a subproblem of the original problem. In a bounding step, a upper bound value is determined for the objective function (the minimization problem). The upper bound value should be the best integer solution encountered so far. In a fathoming step, a particular subset of the solution space is excluded from further consideration for one of the following reasons: no feasible integer solution in it, the upper bound value for the subset exceeding the known upper bound value of a feasible integer solution, or its best feasible integer solution has already been reached.

The search process is a repetition of the above three steps. A replacement for the best upper bound value is used once a new feasible integer solution with lower upper bound value is found. The systematic search terminates when the whole solution space is enumerated either explicitly or implicitly and the upper bound value which remains uncontested is the optimal integer solution of the original problem.

Implicit Enumeration

Implicit enumeration method was initiated by Balas [1] in solving 0-1 integer programming problem. Later, many improvements were made by Glover [7], Geoffrion [6] and many others [18]. 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 problems.

The heart of 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 are developed to speed up fixing variables, such as ceiling test, nonnegative cost test, infeasibility test, cancellation zero test, cancellation one test, linear programming, post optimization, and surrogate constraints [18].

A Comparison Based on Computational Results

The implicit enumeration method is the best in solving 0-1 integer programming problems due to its simple principles and machine round-off errors free. So far, it is most widely used. [11], [16] and [18]. The problem with branch and bound method is that it is too general so that large problems can not be solved.

The Gomory cutting plane method is not very reliable in solving 0-1 integer programming problems due to machine round-off errors, a random change of the constraints may increase the number of iterations significantly. Only first few cuts progress towards the optimal solution greatly, then the

solution process remains constant for many iterations. The implementation, detailed in the next section overcomes these limitations.

Requirements Coverage

The requirements coverage analysis algorithm is based on the work of Lemke, Salkin and Spielberg [13] which uses implicit enumeration with linear programming method. The heart of the algorithm is a 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:

   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 whole searching process. One is the optimal (minimal) cost for the corresponding linear programming problem of the original coverage problem. It 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. Another 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 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 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 the 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 violating constraints.

The results of this 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 requirements specified in the system documentation are covered. Where there is a choice between subsets of tests which would satisfy the requirements coverage, the subset which provides the same coverage with the minimum (cumulative) cost is selected. In general the minimal test set is that set which while providing complete coverage of the requirements also minimizes the cost of testing.

Post Coverage Analysis Considerations

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 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) and one (1). 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.

Regression Testing

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

1. Given the set of requirements to be tested that is, the 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 requirements. This process reduces the columns in the matrix to those applicable.

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 requirements covered.

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

Test Identifier

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

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

Test Identifier

	t2	t4	t5
1	1		
2	1	1	
3			1
4		1	
5			1
6		1	
7	1		
8			1

This reduced test matrix, containing only three tests obviously has the same coverage as the original matrix. The cumulative priority of the tests (the sum of the priorities of the requirements covered by each test, and the inverse of the cost) is equal, each test covering three requirements. However, if test t2 were omitted from the suite, then two requirements would not be covered by the remaining tests (requirements 1 and 7). Similarly the omission of test t4 would leave requirements 4 and 6 uncovered. The omission of test t5 however would not provide test coverage for three 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 priority, cost, and number of omissions could be omitted in any order.

If regression testing were to be conducted from this test suite with the need to test (say) requirements 2 and 6 then tests t2 and t4 would be used:

Test Identifier

	t2	t4
1	1	
2	1	1
4		1
6		1
7	1	

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

Recommendations for Less than 100% Coverage

In order to provide guidance on the set of tests which should be applied when less than 100% coverage of the requirements is permissible, a listing of the tests in ascending priority (or descending cost) order can be created. The tests in the latter part of this list are then those with the highest

priority and which should be included in a less than complete coverage test suite. The portion of the list to be used should be the subvector of tests in priority order which includes the test with the highest priority. If it is necessary to ascertain which 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 requirements coverage list.

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. 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 requirements incidently.

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 likely that these partitions will not have 100% coverage of the requirements in the system, and thus to analyze the partition for coverage will require the elimination of the non-applicable requirements.

Once the partitions have been reduced, each partition may then be treated as a single test, the test vector being the sum of the individual 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 in which each column represents the 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 coverage. Thus independent of the possible reduced form of partition 2, all of its coverage can be provided in the other two partitions.

Conclusions

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 system requirements independently of the top level development design specifications. In either case there is a need of ensure 100% coverage of some chosen set of facilities. In the former case this may be based on 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 test certain requirements) in order to initialize the system in preparation for testing a specific feature or 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 requirements that close to 100% coverage (that is, not involving a high degree of redundancy) is achieved. However, if the incidental uses of features or requirements is taken into account considerable overlap and redundancy is present in the test set.

Returning to the triangle classification example, one might expect that a valid program should identify an equilateral triangle as also being isosceles and thus that a single test in which a% = %b% = %c could be substituted for tests t3 and t4, thus eventually permitting a further reduction in the test suite. This is typical of one of the dangers in this reduction process. By only using one test, it would not be possible to reveal the error of assuming that while all equilateral triangles are also isosceles, the reverse is not true. This places a special burden on the specifications to ensure that special cases are not used as tests. Thus the output specifications for this problem might be modified to read:

Output: An indication as to whether:

1. ...
2. They are the sides of an equilateral triangle,
3. They are the sides of a (non-equilateral) isosceles triangle,
4. ...

This cover 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 coverage. Initially the system can be used to confirm 100% 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% coverage.

It must be expected that pre-formal testing of any system provides 100% coverage, but that formal testing may be limited (both by time and resources) to some smaller percentage. By providing realistic priorities to requirements (for example giving higher priorities to requirements which are close to the highest level of the system) the most important requirements can be tested in a formal 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 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% 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 requirements tracking tables.

References

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

[2] Balinski, M.L. and K. Spielberg: "Methods for Integer Programming: Algebraic, Combinatorial and Enumerative", in "Publications in Operations Research Number 16", pp.195-292, John Wiley & Sons, 1969.

[3] Bazaraa, M.S. and J.J. Jarvis: "Linear Programming and Network Flows", John Wiley & Sons, 1977.

[4] Cooper, L. and D. Steinberg: "Methods and Applications of Linear Programming", W.B. Saunders Company, 1974.

[5] Garfinkel, R.S. and G.L. Nemhauser: "Integer Programming", John Wiley & Sons, 1975.

[6] Geoffrion, A.M.: "Integer Programming by Implicit Enumeration and Balas' Method", Journal of the Society of Industrial and Applied Mathematics Rev., Vol.7, pp.178-190, 1967.

[7] Glover, F.: "A Multiphase-Dual Algorithm for the Zero-One Integer Programming Problem", Operations Research, Vol.13, pp.879-913, 1965.

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

[9] J.B. Goodenough 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.

[10] Graves, R.L. and P. Wolfe: "Recent Advances in Mathematical Programming", McGraw-Hill, 1963.

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

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

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

[14] McMillan,Jr., C.: "Mathematical Programming ", 2nd edition, John Wiley & Sons, 1975.

[15] Orchard-Hays, W.: "Advanced Linear - Programming Computing Techniques", McGraw-Hill, 1968.

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

[17] 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.

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

[19] Zionts, S.: "Linear and Integer Programming", Prentice-Hall, 1974.