2009 33rd Annual IEEE International Computer Software and Applications Conference

Software Input Space Modeling with Constraints among Parameters

Sergiy A. Vilkomir
Department of Computer Science
College of Technology and Computer Science
East Carolina University,
Greenville, NC 27858, USA
vilkomirs@ecu.edu

W. Thomas Swain and Jesse H. Poore
Software Quality Research Laboratory
Department of Electrical Eng and Computer Science
University of Tennessee
Knoxville, TN 37996, USA
{swain, poore}@eecs.utk.edu

***Abstract*—This paper considers the task of software test case generation from a large space of values of input parameters. The purpose of the paper is to create a model of software input space with constraints among parameters to serve as a basis of testing. We suggest a procedure to create a directed graph model, where paths through the graph represent all valid (and only valid) input combinations. The procedure accommodates an arbitrary set of dependencies among parameters. It starts from a simple linear graph and sequentially modifies this graph for each dependency between parameters. Modifications include subgraph splitting and elimination of dead nodes and edges. A complete example of a system with six input parameters and five dependencies among them is presented to illustrate the application of the procedure. Applicability of the approach for different types of parameters and dependencies is addressed.**

Keywords – software testing, input space, constraints, dependencies

I. Introduction

The task of test case generation from a large input space of software parameter values has been considered by many researchers. For coverage testing, various combinatorial approaches [3] were used, in particular pairwise testing [9], t-wise testing [15], base choice [1], and others. For statistical testing, it is possible to apply approaches based on software usage models [5, 8, 13].

The problem becomes more complicated when there are some constraints among input parameters (*i.e.*, not all combinations of input values are valid). For combinatorial testing, some approaches to handling constraints were suggested [2, 4, 10], but this problem is far from being solved.

We would like to generate a suitable number of test cases for statistical testing while selecting them from the total population of all valid input combinations (with all constraints among input parameters taken into account). This task can be solved in two steps:

- Creating a model of software input space with constraints among parameters
- Generating the suitable test cases from this model.

The first step is critical for the whole task. For this step, we present a new procedure to create a graphical model, where paths through the graph represent all valid (and only valid) input combinations. The model has Markov property. Thus, adding probabilities for input values, our model becomes a Markov chain usage model. This made the second step (test case generation) routine. Indeed, many well-defined approaches [14] and tools exist to generate tests from Markov chains. In our investigation [11, 12], we used JUMBL [6], a tool developed in the Software Quality Research Laboratory at the University of Tennessee. In this paper, lack of space allows us to consider only the first step (creation of input space model) that is a main part of the approach.

This paper is based on an extension of our earlier investigations. In [11, 12], we presented a procedure for creating a model based on node splitting. In this paper we generalize this approach using subgraph splitting instead of node splitting. We consider a new procedure for creating a model that reflects an arbitrary set of constraints among parameters.

The remainder of the paper is structured as follows. Terminology definitions and potential approaches to the problem are given in Section 2. Section 3 presents a procedure for input space modeling which can accommodate an arbitrary set of dependencies among parameters. An example application with six input parameters and five dependencies among them is considered in Section 4. In Section 5, we investigate applicability of our approach to various situations. Section 6 offers conclusions.

II. Definitions and Approaches

The most general approach to define a constraint between input parameters is to consider it as a relation. In other words, a constraint between variables x and y with domains X and Y is a subset R of the Cartesian product $X \times Y$. We use the more detailed definition of a constraint as a tuple (Q(X), C(Y), f), where

- Q(X) is a partition on X, thus $Q(X)=\{X_1, \ldots, X_t\}$ is a set of non-empty subsets of X such that for all j, m with $1 \leq j,m \leq t$, $j \neq m$, $\cup X_j = X$ and $X_j \cap X_m = \varnothing$.
- C(Y) is a cover of Y, thus $C(Y)=\{Y_1, \ldots, Y_t\}$ is a set of subsets of Y such that for all j with $1 \leq j \leq t$, $\cup Y_j = Y$.
- $|Q(X)|=|C(Y)|$
- f is a one-one function f: $Q(X) \rightarrow C(Y)$; and $x \in X_j \Rightarrow y \in f(X_j)$.

The definition can be expressed in terms of the familiar "if-then" implication: $\forall$ j, $1 \leq j \leq t$, if $x \in X_j$ then $y \in f(X_j)$.

As the general definition, our definition also establishes a relation R between parameters x and y:

$$R = X_1 \times f(X_1) \cup X_2 \times f(X_2)) \ldots \cup X_t \times f(X_t) \qquad (1)$$

For discrete input parameters, any arbitrary relation can be represented in form (1) so both definitions are equivalent in this case. The situation for continuous input parameters is considered in Section 5. For parameters x and y, we say that y depends on x and will use term "dependency" instead of "constraint" with the same meaning. This property is symmetric; x in turn depends on y (of course, with new partition and cover).

In [11, 12], we used a directed graph to model dependency for the purpose of test case selection. The graph has two special nodes: the source ("Enter") and the sink ("Exit"). All other nodes correspond to input parameters and the edges are annotated with the sets of possible parameter values. The nodes are always connected in a specific order, *i.e.* a node for parameter x_i can be connected only with a node for parameter x_{i+1}. An example in Fig. 1 represents three input parameters: x_1 with set of values {a,b,c}, x_2 with set of values {d,e,f,g}, and x_3 with set of values {h,k,l,m}.

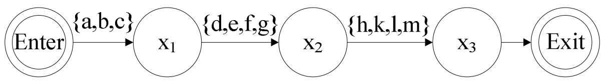

Figure 1. Model with independent parameters

A path through the graph represents a set of test cases. A single test case can be obtained by selecting a single value from the annotation of each edge in the path. The graph in Fig. 1 can be used for test case selection only in the case of independent parameters. For parameters with dependencies, not all combinations of parameter values are valid. To accommodate dependencies, we proposed a graph modification [11], using multiple nodes for one parameter. For example, the node x_1 (Fig. 1) is split into two nodes x_1 and $x_{1,1}$ (Fig. 2) to model the following dependency between x_1 and x_2: if $x_1 \in$ {a,b} then $x_2 \in$ {d,e} and if $x_1 \in$ {c} then $x_2 \in$ {e,f,g}. The paths through the graph in Fig. 2 generate only those test cases which satisfy the dependency between parameters.

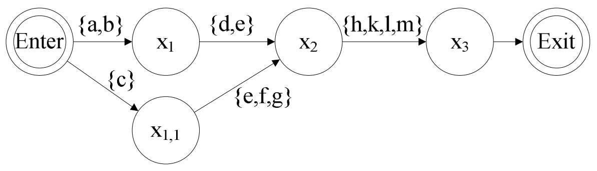

Figure 2. Model for dependency between x_1 and x_2

Splitting nodes allows us to keep the Markov independence property: The path from some node to the sink depends only on the current node and does not depend on the path to reach the current node. If probabilities for each value of each parameter are added, the model becomes a Markov Chain and special tools can be used to analyze the testing process and select test cases in a systematic and/or random way. In particular, the use of JUMBL tool [6] was considered in [12] for NEWTRNX scientific software.

We considered node splitting in [11] with application only to specific combinations of dependencies. In this paper, we provide a new procedure, which can model an arbitrary set of dependencies among parameters. The basis of this procedure is subgraph splitting, so the previous approach of node splitting is a special case of the new procedure.

III. PROCEDURE FOR INPUT SPACE MODELING

Consider k input parameters $x_1, x_2, \ldots, x_k$ and n dependencies among them. We start an input space model from simple linear graph G_0 (similar to the graph in Fig.1), corresponding to the situation with no dependencies. Then we iteratively modify this graph to reflect each dependency between pairs of parameters. First, we modify G_0 and create graph G_1 to reflect the first dependency. Next we modify G_1 and create graph G_2 to reflect the second dependency. Then we modify G_2 and so on until the graph models all dependencies among the parameters.

Below we describe the general procedure of reflecting a dependency between parameters x_i and x_j (modifying graph G_s, $0 \leq s \leq n-1$, to G_{s+1}). A complete example will be presented in Section 4.

The modification procedure is the same for all dependencies and consists of three steps.

A. Step 1. Splitting a subgraph.

Consider a subgraph which contains all nodes for parameters x_i, x_{i+1}, …, x_{j-1} and all edges between these nodes. At the first step, we split this subgraph into t identical subgraphs, where t is a number of partitions of the dependency between x_i and x_j. In our example, the subgraph has nodes x_1, x_2, and $x_{1,1}$. The number of blocks t equals 2, so the subgraph is duplicated (Fig. 3).

All new subgraphs are connected with predecessors (nodes for x_{i-1}) and successors (nodes for x_j) in the same way as for the original subgraph. Thus, if any node is connected with some node in the original subgraph, it will be connected with corresponding nodes in all new subgraphs.

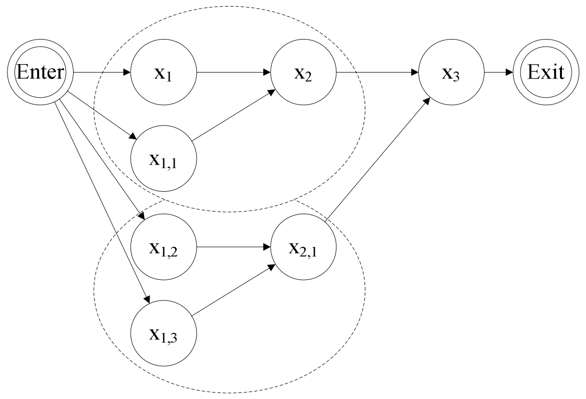

Figure 3. Splitting a subgraph

B. Step 2. Annotating input and output edges.

All input edges (connections with the predecessors) for the split subgraphs in graph G_{s+1} are annotated using corresponding annotation in graph G_s and partitions X_1, …, X_t of the dependency between x_i and x_j. The rule is the following: if some edge between a node for x_{i-1} and a node for x_i in the original graph is annotated with set A then a corresponding edge in subgraph p ($1 \leq p \leq t$) is annotated with set ($A \cap X_p$).

Annotation of output edges of the subgraphs is similar to annotation of input edges. Instead of partitions X_1, …, X_t, covers Y_1, …, Y_t are used. If the edge between a node for x_{j-1} in the original graph and a node for x_j is annotated with set B then a corresponding edge in subgraph p ($1 \leq p \leq t$) is annotated with set ($B \cap Y_p$).

C. Step 3. Eliminating dead nodes and edges.

The following edges and nodes are considered as dead:

- Edges annotated with an empty set.
- Nodes without any input edges.
- Nodes without any output edges.
- Edges connected with dead nodes (output edges for dead nodes without input edges and input edges for dead nodes without output edges).

All dead edges and nodes should be eliminated from the graph.

Note that dead edges can initially appear only after step 2, when some edges become annotated with an empty set. After eliminating this type of dead edges, dead nodes can appear. After eliminated dead nodes and the edges connected with them, new dead nodes can appear. Therefore, eliminating dead edges and nodes is an iterative process: After eliminating some dead elements, it is necessary to check whether new dead elements appeared and to continue elimination if necessary.

IV. A COMPLETE EXAMPLE

Combinations of dependencies can be quite complex. In particular, some parameter can be a first parameter (depender) for one dependency and at the same time be a second parameter (dependee) for another dependency. To illustrate the use of the procedure in such situations, consider a system with six input parameters and five dependencies among them. Denote input parameters as x_1, x_2, …, x_6 and let every parameter take integer values from 1 to 10.

Dependencies among the parameters are described in Table 1. To reflect the first dependency, consider a subgraph with nodes x_1 and x_2 and duplicate it once (because the number of partitions for the first dependency equals two). Edges are annotated as follows:

- Input edge (Enter, x_1) is annotated with $\{1..10\} \cap \{1..5\} = \{1..5\}$.
- Input edge (Enter, $x_{1,1}$) is annotated with $\{1..10\} \cap \{6..10\} = \{6..10\}$.
- Output edge (x_2, x_3) is annotated with $\{1..10\} \cap \{1..5\} = \{1..5\}$.
- Output edge ($x_{2,1}$, x_3) is annotated with $\{1..10\} \cap \{6..10\} = \{6..10\}$.

A subgraph for dependency 2 contains only one node x_5. We split this node into three new ones because the dependency 2 involves three partitions. There are no dead edges after the annotation. The resulting graph for dependencies 1 and 2 is shown in Fig. 4.

TABLE I. DEPENDENCIES AMONG PARAMETERS

Dependency	1-st param	1-st param. partition	2-nd param. cover	2-nd param
1	x_1	{1..5}	{1..5}	x_3
		{6..10}	{6..10}	
2	x_5	{1..8}	{1..3}	x_6
		{9}	{4..10}	
		{10}	{7,8}	
3	x_2	{1..5}	{1..6}	x_4
		{6..10}	{5..10}	
4	x_1	{3..8}	{1..5}	x_5
		{1,2,9,10}	{6..10}	
5	x_4	{1..3}	{1..5}	x_6
		{4..10}	{6..10}	

A subgraph for dependency 3 contains three nodes: x_2, $x_{2,1}$, and x_3. The results of splitting this subgraph are shown in Fig. 5, where new nodes are $x_{2,2}$, $x_{2,3}$, and $x_{3,1}$. There are no dead edges after the annotation.

Dependency 4 is between x_1 and x_5 The subgraph for this dependency includes all nine nodes associated with parameters x_1, x_2, x_3, and x_4 (x_1, $x_{1,1}$, x_2, $x_{2,1}$, $x_{2,2}$, $x_{2,3}$, x_3, $x_{3,1}$, and x_4). The results of splitting this subgraph are shown in Fig. 6.

Edges (x_4, $x_{5,1}$) and (x_4, $x_{5,2}$) are annotated with an empty set so they are dead and should be eliminated. No more nodes or edges are produced by this operation.

Finally, dependency 5 is between x_4 and x_6 and the corresponding subgraph contains five nodes, x_4, $x_{4,1}$, x_5, $x_{5,1}$, and $x_{5,2}$. The final graph, which reflects all five dependencies, is presented in Fig. 7.

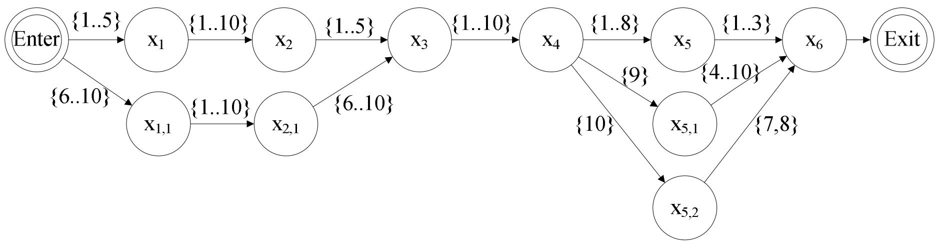

Figure 4. Model for dependencies 1 and 2

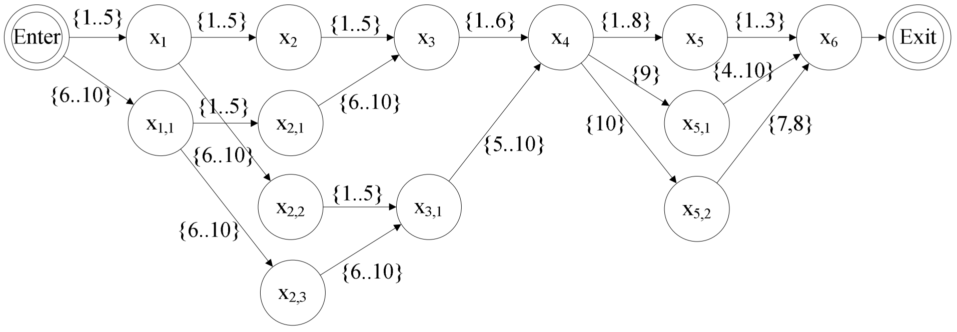

Figure 5. Model for dependencies 1, 2, and 3

The final graph contains 12 different paths which are used to generate all 14,100 valid combinations of test values. The number of valid test cases for each path equals the product of cardinalities of the sets associated with the edges on the path.

V. APPLICABILITY OF THE APPROACH

Section 2 defines constraints in terms of sets of discrete values. In this case, any constraint can be formulated in form (1) and our definition of a constraint between parameters is equivalent to the general definition. For continuous inputs, it is not always possible to present a constraint in form (1). In this case, discretization (partitioning) should be performed before application of the procedure. For some specific situations, constraints for continuous input parameters can also be presented in form (1). For such situations, it is not necessary to perform discretization immediately. We would recommend firstly apply our procedure that provides new continuous partitions only for valid values of the parameters. Specific discrete values of parameters for testing purposes can be then selected from these new partitions.

In this paper, we consider "one-on-one" dependencies (*i.e.* dependency between two parameters). Our definition of dependency can be readily extended to dependencies with many parameters (situations "one-on-many", "many-on-one", or "many-on-many").

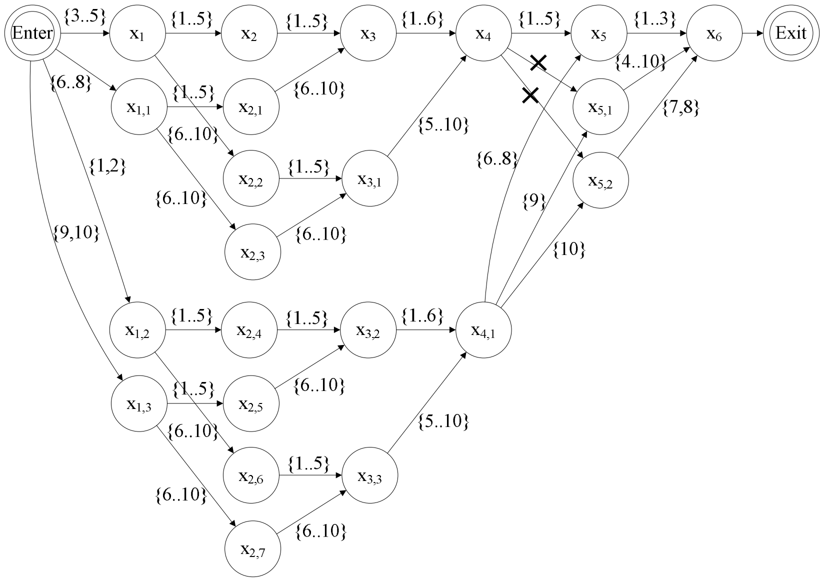

Figure 6. Model for dependencies 1, 2, 3, and 4

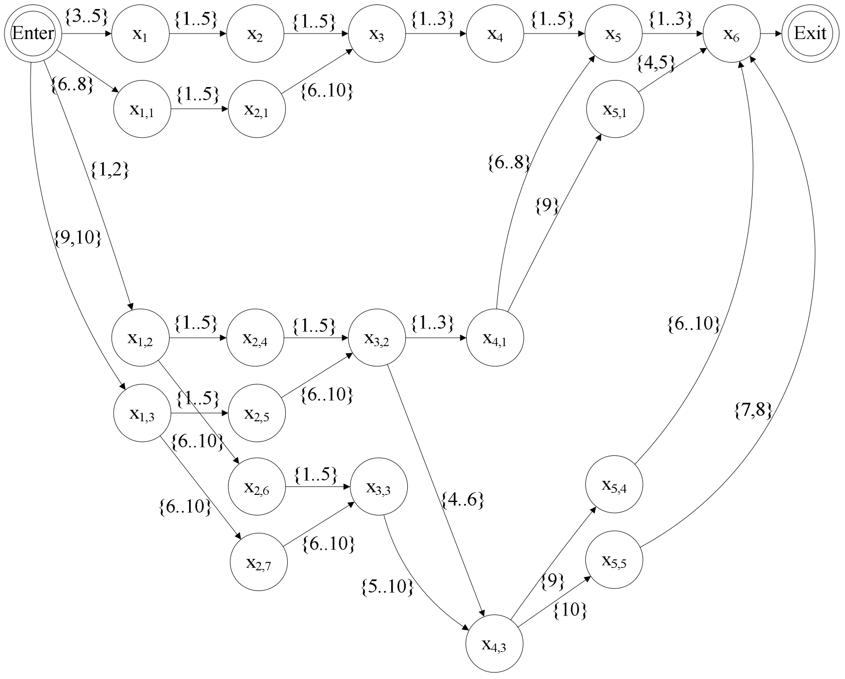

Figure 7. Final model for dependencies 1, 2, 3, 4, and 5

For this case, it is necessary to merge the "many" independent parameters into one derived vector parameter. The domain of this derived parameter equals the Cartesian product of domains of involved parameters. Consideration of the derived parameter instead of "many" parameters brings the situation to "one-on-one" dependency, and then the procedure can be directly applied.

To apply our procedure, the input parameters must be considered in a specific order. This order can be chosen arbitrarily and our procedure can be applied in any order. However, the size and structure of the final graph are different for various orders of the input parameters. By placing input parameters in a "clever" order we can significantly simplify the final graph. A simple recommendation is to put dependent parameters into adjacent places or as close to each other as possible. This reduces the size of split subgraphs and therefore the size of the final graph.

The main factors that determine the size and complexity of the final graph structure are the numbers of different constraints for each parameter. In the simplest structure, each parameter is involved in no more than one constraint. In this case, the number of nodes representing a parameter is equal or less than t, the number of partitions in the constraint for this parameter. When parameters are involved in many constraints, it increases the graph size.

VI. CONCLUSIONS

Modeling the software input space is the first step for automated statistical testing. The main result of this paper is a new procedure that creates the graphical model of the valid input space based on constraints among input parameters. The procedure is based on subgraph splitting and accommodates any set of dependencies among parameters by considering dependencies sequentially one by one. The procedure manages the size of the model while allows generating a large number of test cases.

The main benefit of our model is that the next steps (statistical test case generation) become routine because tools for test case generation from Markov chain usage models are available and can be used. A practical example of test case generation from our model using the JUMBL tool can be found in [12]. For this purpose, the model can be described in a text format, such as TML [7].

REFERENCES

[1] P.E. Ammann and J. Offutt, "Using formal methods to derive test frames in category-partition testing", Proc. of the Ninth Annual Conference on Computer Assurance (COMPASS'94), Gaithersburg, MD, June 1994. IEEE Computer Society Press, pp. 69–80.

[2] A. Calvagna and A. Gargantini, "A logic-based approach to combinatorial testing with constraints", Chapter in Tests and Proofs, Springer Berlin/ Heidelberg, Lecture Notes in Computer Science, Volume 4966, 2008, pp. 66-83.

[3] M. Grindal, J. Offutt, and S.F. Andler, "Combination testing strategies: a survey", Software Testing, Verification, and Reliability, 15(3), September 2005, pp. 167-199.

[4] M. Grindal, J. Offutt, and J. Mellin, "Managing conflicts when using combination srategies to test software", Proc. of the 18th Australian Software Engineering Conference (ASWEC 2007), 10-13 April 2007, Melbourne, Australia, pp. 255-264.

[5] J. Poore and C. Trammell, "Engineering practices for statistical testing", Crosstalk. The Journal of Defense Software Engineering, April 1998, pp. 24-28.

[6] S. Prowell, "JUMBL: A tool for model-based statistical testing", Proc. of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03), January 6-9, 2003, Big Island, HI, USA.

[7] S. Prowell, "TML: A description language for Markov chain usage models", Information and Software Technology, Vol. 42, No. 12, September 2000, 835-844.

[8] W.T. Swain and S.L. Scott, "Model-based statistical testing of a cluster utility", Proc. of the 5th International Conference on Computational Science (ICCS 2005), Atlanta, GA, USA, May 22-25, 2005, Part I, Lecture Notes in Computer Science 3514, pp. 443-450.

[9] K.C. Tai and Y. Lei, "A test generation strategy for pairwise testing", IEEE Transactions on Software Engineering 2002, 28(1):109–111.

[10] K. Vij and W. Feng, "Boundary value analysis using divide-and-rule approach", Proc. of the 5th International Conference on Information Technology: New Generations (ITNG 2008), April 7-9, 2008, Las Vegas, Nevada, USA, IEEE Computer Society, pp. 70-75.

[11] S. Vilkomir, W.T. Swain, and J. Poore, "Combinatorial test case selection with Markovian usage models", Proc. of the 5th International Conference on Information Technology: New Generations (ITNG 2008), April 7-9, 2008, Las Vegas, Nevada, USA, IEEE Computer Society, pp. 3-8.

[12] S. Vilkomir, W.T. Swain, J. Poore, and K. Clarno, "Modeling input space for testing scientific computational software: a case study", Proc. of the International Conference on Computational Science (ICCS 2008), Krakow, Poland, June 23-25, 2008, Part III, LNCS 5103, pp. 291-300.

[13] J. Whittaker and J. Poore, "Markov analysis of software specifications", ACM Transactions on Software Engineering and Methodology, Vol. 2, No. 1, January 1993, pp. 93-106.

[14] J. Whittaker and M.G. Thomason, "A Markov chain model for statistical software testing", IEEE Transactions on Software Engineering, 1994, 20(10): 812-824

[15] A.W. Williams and R.L. Probert, "A measure for component interaction test coverage", Proc. of the ACSI/IEEE International Conference on Computer Systems and Applications (AICCSA 2001), Beirut, Lebanon, June 2001, IEEE Computer Society Press, pp. 304–311.