A Method for Generating a Minimal Functional Set of Test-Cases for Software-Intensive Systems

J. Gericke, M. Wiemann
Siemens AG, Corporate Technology, Production Processes, Simulation and Risk Management
Otto-Hahn-Ring 6, D-81730 Munich, Germany
{joerg.gericke, matthias.wiemann}@siemens.com

Abstract - In testing software components, choosing appropriate test-cases is a difficult task. Not only input data must be identified that results in different behavior of the component under test, but also the resultant high number of test-cases must be handled. This work describes a method for selecting test input data by dividing the input parameter value domains into equivalence classes and reducing the number of resultant test-cases by applying a pair-wise coverage of parameters values. That way, a minimal number of test-cases can be generated that already provides coverage of standard test methods to a high degree and finds a significant number of errors. The described method was implemented and used for automating the testing of automotive systems.

Keywords: Software-intensive systems, robust testing, classification trees method, test generation, test suite reduction

1. Motivation

The most reliable method to ensure the quality of a piece of software is to proof the correctness of source code, e.g. through formal verification [10, 9]. Unfortunately, this approach is mostly too time consuming for a whole system. Only crucial parts of a system can be verified this way. In practice, only the use of test-cases are a way to improve the quality of a software-intensive system [8]. The optimal scenario is to test all possible input values and all their combinations. The outcome is compared to the expected output. That way, it is guaranteed to identify all errors. Unfortunately, this approach is not realistic due to the high number of test-cases and the very limited time and test budgets. So, the challenge is to choose a smaller set of most important test-cases, that means finding the right input data. It can be found that even when testing simple components, important test-cases are overlooked [11]. Classifying the input parameter's value domains helps to identify all relevant input data [12, 11]. The classification tree method is a systematic approach to find equivalence classes utilizing aspects of parameter value domains [13, 12, 14]. In the following, we will introduce a method, how to get the minimal set of test cases out of a classification tree.

2. Classification Tree Method

The process of constructing a classification tree starts with defining the component under test, which means identifying the input parameters as well as the expected output. Possible internal states of components are also regarded as explicit input parameters. Next, all relevant aspects of the input parameters are collected. An aspect is considered as relevant if it is assumed that it influences the behavior of the component. The aspects should be ordered appropriately before application.

Classifications divide the input parameters' value domains in *equivalence classes* [13, 12, 14]. A class name describes the common property of class-members. A classification is named after the distinguishing property. The process of defining classes and subclasses proceeds until all aspects are applied and no further refinements are possible. A classification should divide a class in at least two subclasses to be considered useful.

A *classification tree* is a tree-structured, formal description of input parameters and computa-

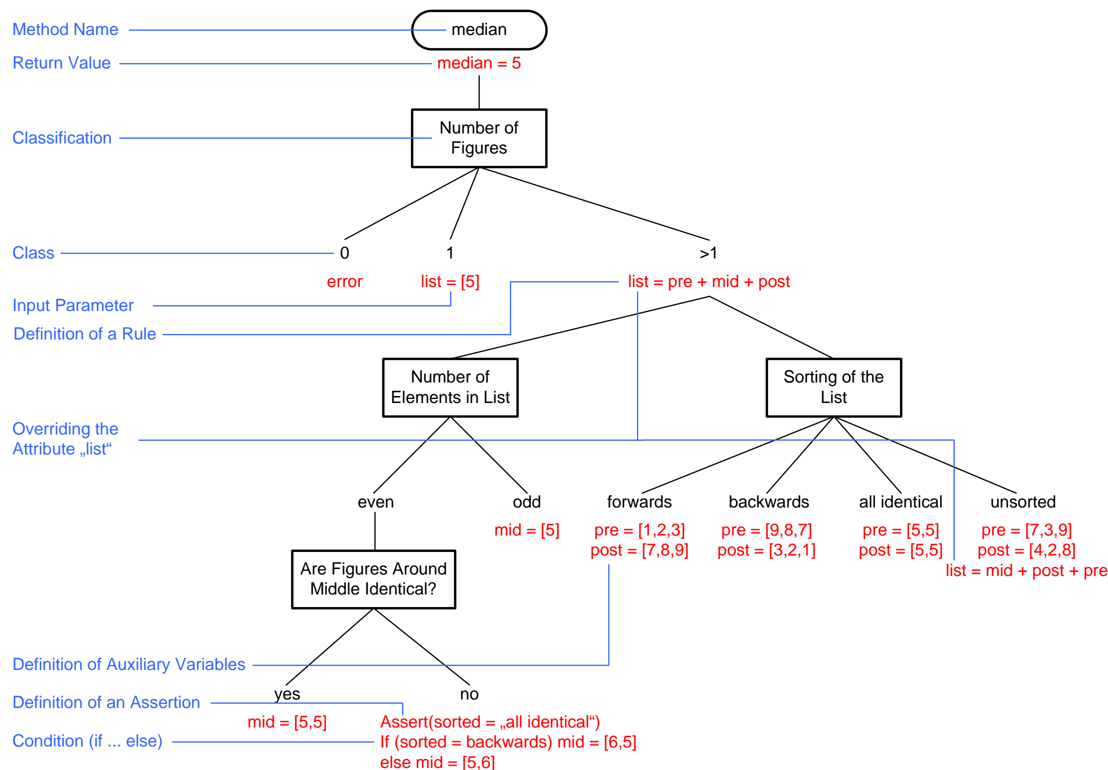

Figure 1. Classification tree enriched with attributes

tion output [11]. It consists alternating of classes and classifications. Additionally, the classification tree method supports assigning attributes to classes and classifications. That way, specific values, constraints and conditions can be defined: *Assertions* can be defined to ignore branches in the latter evaluation of the tree [11]. *Conditions* are used to set parameter values with respect to different (auxiliary) variables. *Attributes* can be overridden similarly to the concept of object-oriented programming; the required hierarchy is defined by the tree-structure [11].

So far, the construction of a classification tree is a manual task, because semantic interpretation is needed. Consequently, resultant trees vary in many ways, depending on which classifications are used and in which order. An example of a classification tree, enriched with attributes, is depicted in Figure 1.

3. Generating Test-Cases

The method for reading off test-cases starts at the root node of a classification tree and proceeds down to leaf nodes. When handling a classification, it must be decided which case should be considered. For instance, at the classification “Number of Figures”, either “zero”, “one” or “more than one” has to be chosen to continue constructing a path to a leaf node. In contrast, when handling a class, several classifications lead to different leaf nodes. The different classifications must be processed simultaneously. For each classification of a class, one leaf node is chosen. Theoretically, there can be an arbitrary number of chosen leaf nodes since the classification tree method is recursive in the respect that the tree consists alternating of classifications and classes. Figure 2 shows how choosing the leaf nodes leads to defining test-cases.

After choosing the leaf nodes for a single test-case, their attributes, either self-defined or inherited, must be processed to set input parameters and *auxiliary variables*. Auxiliary variables are used to evaluate *expressions* in conditions and *formulas*, which put together the parameter values. The return values and object’s internal final state are defined in the same manner. The set of test-cases shown in Table 1 is defined by the tree in Figure 2.

As mentioned above, a classification tree is a formal description of parameters, and attributes add semantic descriptions about parameters values. Thus, that enables computers to generate test-cases automatically. Nevertheless, it is worthy to investigate the method more closely in terms of applicability. One major drawback lies in the fact that it produces an vast number of test-cases. Reduction algorithms, such as *Robust Testing*, should be applied to decrease the number of test-cases. The output is a con-

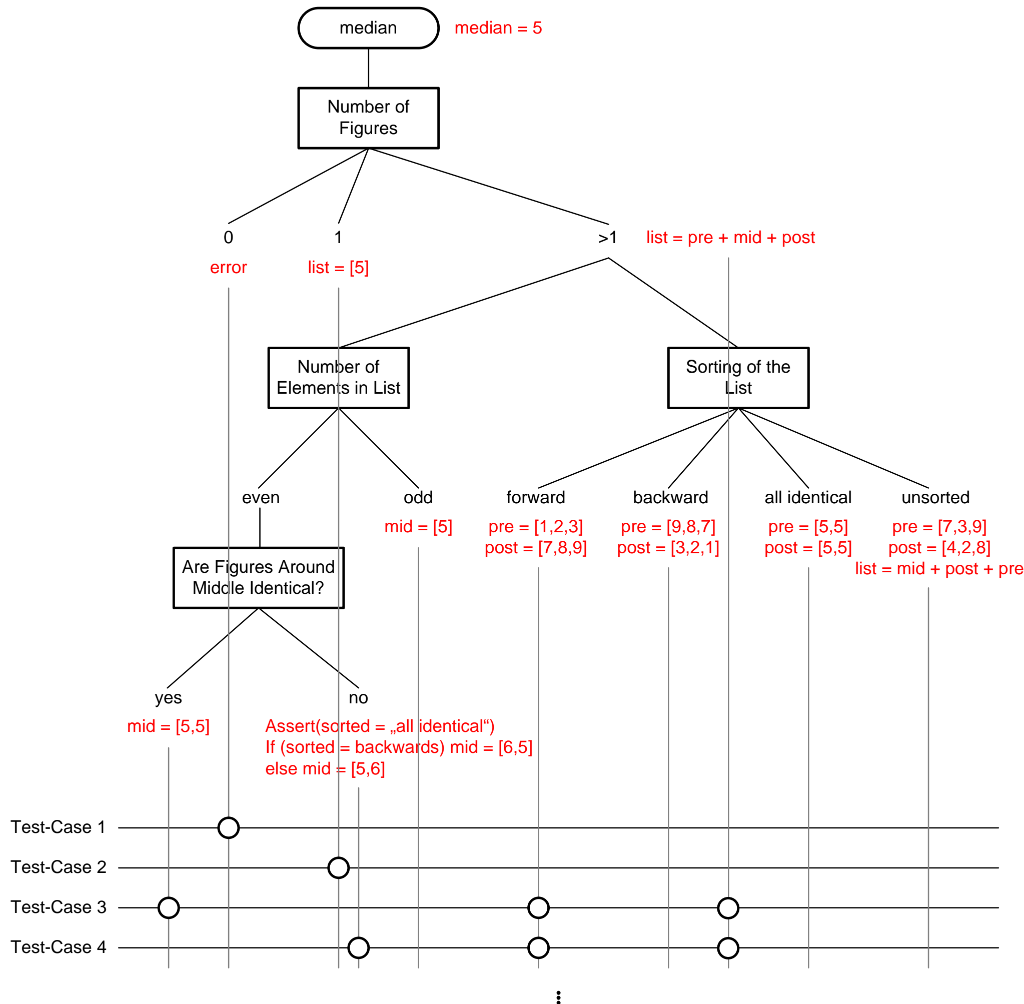

Figure 2. Reading off test-cases

densed set of test-cases with have a high probability to find errors. Both algorithms, constructing equivalence classes for value domains and applying robust testing afterwards, can be seen as filters for finding the important test-cases out of an humongous set.

The need for reduction becomes obvious when looking at a more complex but still very simple example. In the following example, a telephone switch with two terminal telephones should be tested. Both telephones can be provided by three vendors each producing three models. The switch, which manages between the two telephones, can also be provided by three vendors, each having three models. Then, the connection between the telephones and the switch can be of different types. In this example, eight different types of connection are listed. A possible classification tree is depicted in Figure 3.

Table 1. Set of test-cases

No	Median	Pre	Mid	Post	List
1	Error	-	-	-	[]
2	5	-	-	-	[5]
3	5	[1,2,3]	[5,5]	[7,8,9]	[1,2,3,5,5,7,8,9]
4	5	[1,2,3]	[5,6]	[7,8,9]	[1,2,3,5,6,7,8,9]
5	5	[1,2,3]	[5]	[7,8,9]	[1,2,3,5,7,8,9]
6	5	[9,8,7]	[5,5]	[3,2,1]	[9,8,7,5,5,3,2,1]
7	5	[9,8,7]	[6,5]	[3,2,1]	[9,8,7,6,5,3,2,1]
8	5	[9,8,7]	[5]	[3,2,1]	[9,8,7,5,3,2,1]
9	5	[5,5]	[5,5]	[5,5]	[5,5,5,5,5,5]
10	5	[5,5]	[5]	[5,5]	[5,5,5,5,5]
11	5	[7,3,9]	[5,5]	[4,2,8]	[7,3,9,5,5,4,2,8]
12	5	[7,3,9]	[6,5]	[4,2,8]	[7,3,9,5,6,4,2,8]
13	5	[7,3,9]	[5]	[4,2,8]	[7,3,9,5,4,2,8]

The method for generating the test-cases has to choose five leaf nodes simultaneously, since five classifications are defined in parallel. The test-case generation comes up with $9 \times 9 \times 9 \times 8 \times 8 = 46{,}656$ test-cases. Clearly, the complexity and the number of test-cases increase rapidly when the number of possible configurations per component becomes greater. Roughly, the number of test-cases grows exponentially [15]. In general, if all parameters have the same number of parameter values, the number of test-cases is defined by $(\textit{number of values per parameter})^{\textit{number of parameters}}$. At some point, it is impossible to apply all test-cases. In the following we present a method to reduce the number of test-cases systematically to 81.

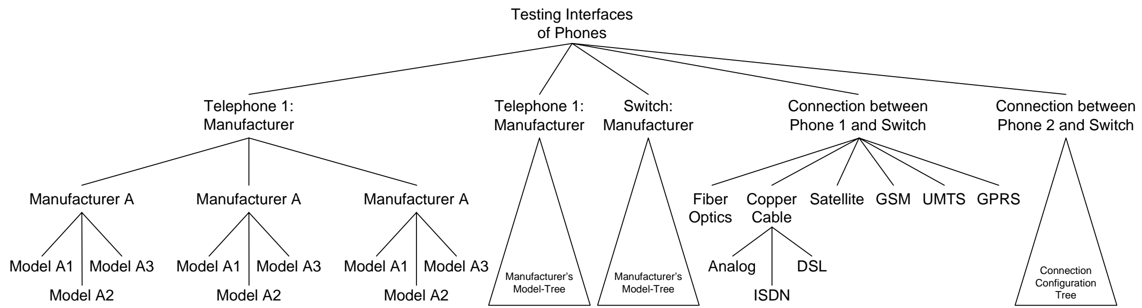

Figure 3. More complex classification tree (simplified visualization)

4. Method to get the minimal set of test cases

When reducing the number of test-cases, it is important to use a standardized and recognized approaches rather than omitting arbitrary test-cases. Furthermore, in general, the neglected test-cases should have a minor chance of revealing errors. To ensure both, we chose a pair-wise coverage of parameters.

The algorithm is known as robust testing [7, 1]. It uses the theory of mutually orthogonal Latin squares, which are constructed using Galois fields. Based of the set of orthogonal Latin squares – a set of squared matrices – a combined matrix can be computed. The entries of this matrix are tuples from which test-cases can be directly deducted. The procedure guarantees finding errors, which are caused by the combination of values of two parameter [7]. Errors that rely on the occurrence of more than two certain parameter values are less likely and not necessarily found by this algorithm [7]. Consequently, robust testing generates test-cases that usually have a higher probability to reveal to errors.

The introductory complex example in Figure 3 showed clearly that the classification tree produces an unmanageable amount of test-cases. When applying a pair-wise coverage of parameters, the robust testing algorithm chooses a feasible amount of test-cases. In order to apply the robust testing algorithm in combination with the classification tree, parameters and parameter values have to be determined first. Then, the list of parameters is sorted according to the number of parameter values. The two parameters with the highest number of parameter values determine the dimension of the Latin

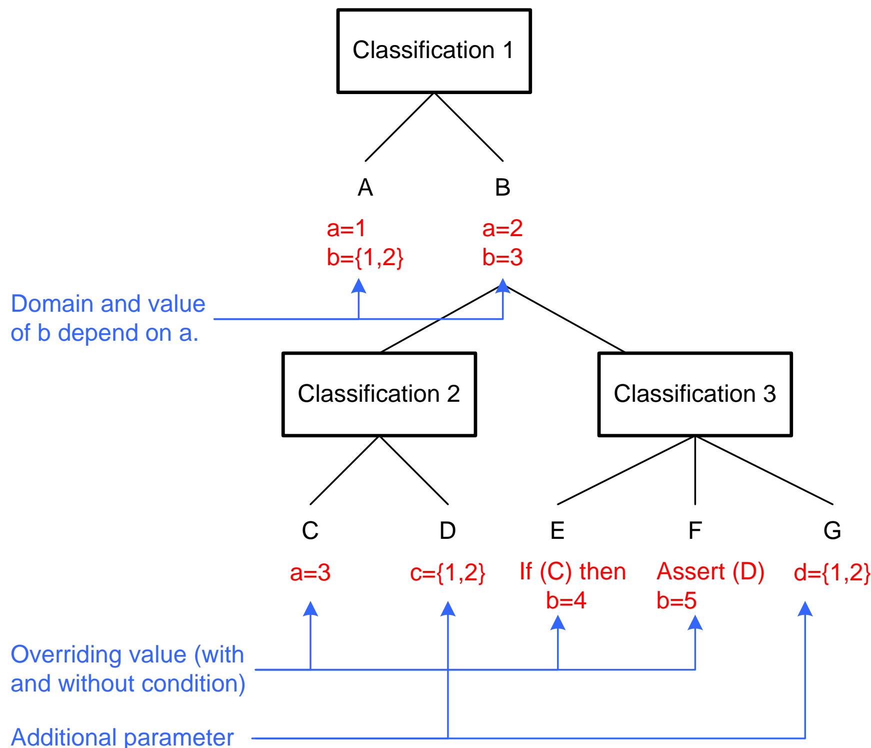

Figure 4. Classification tree with dependencies among parameters

squares. The number of parameters determine the number of required Latin squares, precisely, if k parameters were found, then k-2 Latin squares are to be generated. Thus, it only makes sense to apply robust testing to three or more parameters. In case of two parameters, robust testing returns an exhaustive test. The number of Latin squares and their dimension are the algorithm's only required input parameters.

Going back to the complex example in Figure 3, five parameters can be determined, namely: telephone 1; telephone 2; switch; connection between telephone 1 and switch; and connection between telephone 2 and switch. The values of these parameters are determined at the leaves. For instance, telephone 1 can be realized with model A1, A2 etc. to C2 and C3. The connection between telephone 1 and the switch can be realized with fiber optics, copper cable (analog), copper cable (ISDN) and so forth. The first three parameters in the list (the two telephones and the switch) can have nine different configurations each. The two connections can be configured in eight different ways. Consequently, there are $5\text{-}2=3$ Latin squares needed, each having a dimension of 9×9. Thus, robust testing generates $9\times 9=81$ test-cases. In contrast, an exhaustive test would require $9\times 9\times 9\times 8\times 8=46{,}656$ test-cases. That means, less than 0.2% of the test-cases are needed to guarantee a pair-wise coverage. Experience [5, 4, 2] showed that in many cases the productivity in terms of finding errors could be increased by the factor of two or better when applying robust testing rather than choosing test-cases manually. Robust testing shows a quadratic growth in the number of test-cases, which clearly scales much better than the exponential growth of an exhaustive test.

Robust testing is applied after reading off the parameters and their possible values from the classification tree. Dependencies among parameters must be resolved since this is a prerequisite for the robust testing algorithm. Dependent parameters are resolved by generating a so-called hybrid parameter. A hybrid parameter has tuples for values. Each tuple consists of the parameter values that the numerous dependent parameters represent in this special test-case.

Resolving dependencies among parameters is a non-trivial task. It takes effort to determine the hybrid parameter. This is due to the fact, that the classification tree strongly supports dependencies between parameters by defining classifications in parallel. The following example visualizes resolving the dependencies. Figure 4 shows a classification tree in which parameter values are overridden (parameter a and b), value domains change (parameter a and b), additional parameters are added (parameter c and d) and dependencies between classifica-

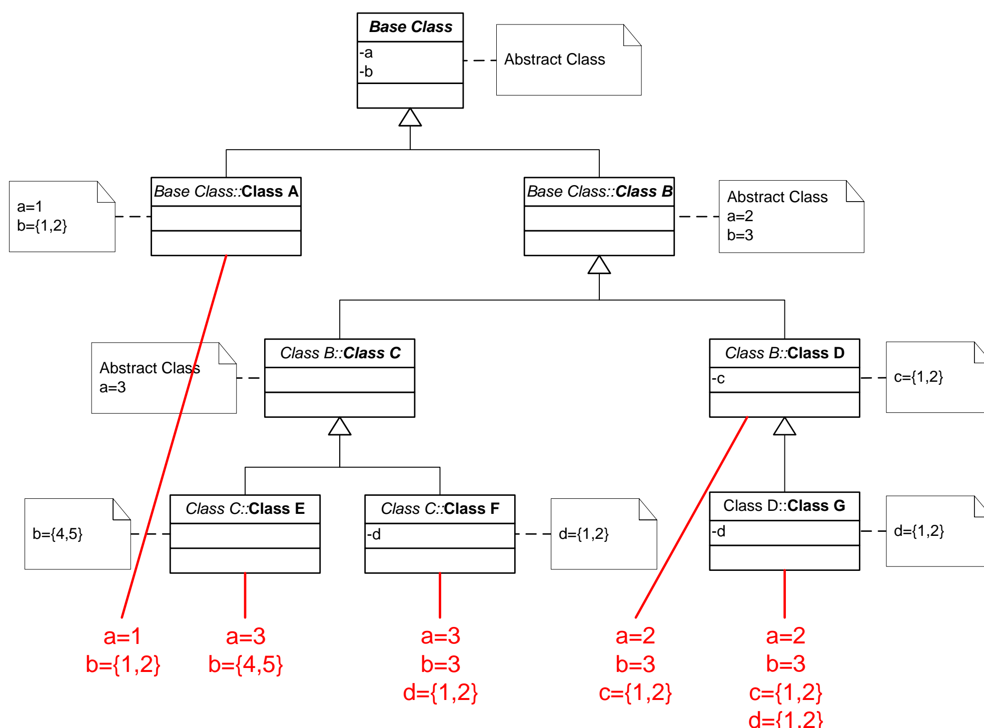

Figure 5. Class diagram, derived from the previous classification tree

Table 2. Values of the hybrid parameter

Hybrid parameter value #	Generating class	Parameter a	Parameter b	Parameter c	Parameter d
1	Class A	1	1		
2		1	2		
	Class B	abstract class			
	Class C	abstract class			
3	Class E	3	4		
4		3	5		
5	Class F	3	3		1
6		3	3		2
7	Class D	2	3	1	
8		2	3	2	
9	Class G	2	3	1	1
10		2	3	2	1
11		2	3	1	2
12		2	3	2	2

tions are defined (“If” and “Assert”).

For visualization purposes it might be useful to convert the classification tree into an UML diagram. Figure 5 shows an UML class diagram which was derived from the classification tree in Figure 4. Since value domains are inherited from upper classes, changes in the value domain can be realized by adding a new subclass and overriding a parameter’s value domain (e.g. class C overrides parameter a of class B). Similarly, additional parameters can be added (e.g. through classes D, F and G). Abstract classes can define value domains and declare additional parameters, but additional information for defining all parameter values depending on other parameter values requires refinements by subclasses.

Based on the given class diagram, the necessary hybrid parameter can be created [5]. Hereby, for non-abstract classes all combinations of parameter values are considered in form of tuples, which describe the parameters’ values. For instance, Figure 5 generates the in Table 2 listed hybrid parameter values. It is obvious that generating the hybrid parameter results in a high number of parameter values. This simple example already generates twelve parameter values.

Remark: For demonstrational purposes, the hybrid parameter of the discussed example covers the total classification tree. In this case, robust testing is not applicable since there is only one parameters - the hybrid parameter. If the tree that generates the hybrid parameter is a sub-tree of bigger classification tree with other parameters, then the hybrid parameter would replace parameters a, b, c and d, and would be covered pair-wise with the residual parameters in the tree. Robust testing would be applicable.

5. Discussion

The above-seen enormous reduction of test-cases to 0.2% clearly raises the question whether applying a pair-wise coverage is sufficient testing. Obviously, robust testing does not test all possibilities, but usually there is not enough time to perform a full test. The main issue is to systematically reduce the number of test-cases by maintaining a high probability to find errors. An often posed question is whether important errors are overseen. Indeed, it is not guaranteed to find all or even all severe errors, but if testers would know in advance which test-cases perform better, then these could be applied and a systematic reduction would not be needed. Robust testing bases on the assumption that it is more important to find errors that are caused by the combination of two parameters rather than the combination of three or more parameters. Combinations of two parameters lead to errors more often. If combinations of more than two parameter values should also be tested, then robust testing can be extended to provide coverage of three or more parameters, but again with the cost of more test-cases.

Robust testing is an acknowledged and successfully applied method for test-case reduction. Application has shown that a pair-wise coverage showed very good results in terms of finding many errors [3]. In average, robust testing provides a good coverage of standard test-procedures like block coverage (93%),

decision coverage (83%), c-use coverage (76%) and p-use coverage (73%) [2].

6. References

1. A. Williams. “Software Component Interaction Testing: Coverage Measurement and Generation of Configurations”. Ph.D. thesis, University of Ottawa. 2002. http://www.site.uottawa.ca/~awilliam/papers/Thesis.pdf
2. D. Cohen, S. Dalal, M. Fredman, G. Patton. “The AETG System: An Approach to Testing Based on Combinatorial Design. Appeared in IEEE Transactions On Software Engineering”, volume 23, number 7, July 1997, pages 437-444.
3. D. Cohen, S. Dalal, J. Parelius, G. Patton, 1996. “The Combinatorial Design Approach to Automatic Test Generation”. IEEE Softw. 13, 5 (Sep. 1996). pages 83-88.
4. D. Kuhn, M. Reilly. “An Investigation of the Applicability of Design of Experiments to Software Testing”. National Institute of Standards and Technology. Reprinted from Proceedings, 27th NASA/IEEE Software Engineering Workshop, NASA Goddard Space Flight Center, 4-6 December, 2002.
5. R. McDaniel, J. D. McGregor. “Testing the Polymorphic Interactions Between Classes”. TR94-103. Clemson University. 1994.
6. Phadke Associates Inc. “Planning Efficient Software Tests”. 1997
7. A. Williams, R. Probert. “A Practical Strategy for Testing Pair-Wise Coverage of Network Interfaces”. In Proceedings of the Seventh international Symposium on Software Reliability Engineering (ISSRE '96) (October 30 - November 02, 1996). ISSRE. IEEE Computer Society, Washington, DC, 246.
8. R. Kurshan. 1997. “Formal Verification in a Commercial Setting”. In Proceedings of the 34th Annual Conference on Design Automation - Volume 00 (Anaheim, California, United States, June 09 - 13, 1997). DAC '97. ACM Press, New York, NY, 258-262.
9. C. Kern, M. Greenstreet. 1999. “Formal Verification in Hardware Design: a Survey”. ACM Trans. Des. Autom. Electron. Syst. 4, 2 (Apr. 1999), 123-193.
10. J. Rushby, F. von Henke. 1991. “Formal Verification of Algorithms for Critical Systems”. In Proceedings of the Conference on Software For Critical Systems (New Orleans, Louisiana, United States, December 04 - 06, 1991). SIGSOFT '91. ACM Press, New York, NY, 1-15.
11. S. Lützendorf, K. Bothe. “Attributierte Klassifikationsbäume zur Testdatenbestimmung“. Vortrag zum Treffen der Fachgruppe TAV der Gesellschaft für Informatik. Köln. Germany. Februar 2003
12. M. Grochtmann, K. Grimm. „Classification Trees for Partition Testing”. Software Testing, Verification and Reliability, S. 63-82. 1993
13. J. Goodenough, S. Gerhart. “Towards a Theory of Test Data Selection”. IEEE Transactions On Software Engineering, 1(2):156-173. June 1975
14. S. Butron. “Automated Testing from Z Specifications”. Report. Department of Computer Science. University of York. 2000
15. D. Brand. “Exhaustive Simulation Need Not Require an Exponential Number of Tests”. In Proceedings of the 1992 IEEE/ACM international Conference on Computer-Aided Design (Santa Clara, California, United States). International Conference on Computer Aided Design. IEEE Computer Society Press, Los Alamitos, CA, 98-101