Pseudo-Exhaustive Testing for Software

D. Richard Kuhn and Vadim Okun
National Institute of Standards and Technology
Gaithersburg, MD 20899
kuhn@nist.gov vadim.okun@nist.gov

Abstract

Pseudo-exhaustive testing uses the empirical observation that, for broad classes of software, a fault is likely triggered by only a few variables interacting. The method takes advantage of two relatively recent advances in software engineering: algorithms for efficiently generating covering arrays to represent software interaction test suites, and automated generation of test oracles using model checking. An experiment with a module of the Traffic Collision Avoidance System (TCAS) illustrates the approach testing pairwise through 6-way interactions. We also outline current and future work applying the test methodology to a large real-world application, the Personal Identity Verification (PIV) smart card.

Keywords

automated testing, combinatorial testing, software testing

1. Introduction

Pseudo-exhaustive testing is an established concept in circuit design. Several different approaches are available for pseudo-exhaustive testing in digital circuits, but all take advantage of the fact that, in general, outputs do not depend on all inputs, but on a subset of them. Circuits are segmented, either logically or physically, and each segment is tested exhaustively. Applying the same scheme to software is problematic. Unit testing concentrates on separate segments of the software, but exhaustive testing of individual units is still intractable. In addition, interactions among functions may cause faults and need to be tested.

We propose an approach to pseudo-exhaustive testing for software. Empirical observation suggests that the number of variables involved in software failures is relatively small, (i.e., on the order of 3 to 6), at least for some classes of software [14]. Therefore, if we know from experience that *t* or fewer variables are involved in failures for a particular application type and we can test all *t*-way (or *t*+1 way) combinations of discrete variables, we have high confidence that the application will function correctly. Looked at another way, if we know in advance that all failures are triggered by *t* or fewer conditions, testing all *t*-way conditions is in some sense equivalent to exhaustive testing. For realistic applications it is not possible to test all *t*-way combinations of values for variables that may have 2^{64} values each, so equivalence classes or other abstraction methods must be used. This approach may thus be considered *pseudo-exhaustive* rather than truly exhaustive.

The methods we propose would not have been possible in the past, because even after partitioning variable values into equivalence classes, testing all *t*-way combinations for realistic software can require an exponential number of tests (tens to hundreds of thousands for much real world software). Faster algorithms for generating covering arrays [5],[12],[17], combined with methods of automatically generating complete test cases, including test oracles [1],[2], make it practical and cost-effective to do pseudo-exhaustive testing of software. We describe this method and illustrate its application through an experiment with a module of the Traffic Collision Avoidance System (TCAS). This system has been used in previous studies of software test methods and provides a benchmark for evaluation of our pseudo-exhaustive testing methodology.

2. Combinatorial Testing

Methods from the field of design of experiments (DOE) have been applied to quality control problems in many engineering fields, including limited use for software [4],[8],[18], DOE seeks to maximize the amount of information gained in an experiment with an economical number of tests. Originally used in agricultural experiments in the 1920s, DOE methods were set up to produce balanced coverage of independent variables. For instance, an experiment might involve six varieties of seed with five fertilizers and five soil types, using an orthogonal array in which every combination occurs exactly once[10]. Early applications of these methods to software were limited because of two significant differences between software and the problems that DOE was originally designed for - needs for balanced coverage and the number of variables involved – and a lack of adequate tools for generating necessary combinations.

Constructing tests that provide combinatorial coverage is a hard problem that has been studied for more than two centuries. Software testing is different from previous applications. For instance, software does not have the same need for balance between parameters: if the combination $\rho_1 = v_3$ and $\rho_4 = v_1$ occurs in multiple test cases, there may be some wasted effort but testing soundness is not affected. Software testing typically has a vast number of parameters and corresponding values. Unlike the application to agriculture, software may have parameters with more than five or six values. Abstraction methods, such as equivalence classes, may reduce the number of values, but the number of variable and value combinations will still be extremely large.

Covering arrays are combinatorial objects that can be generated to represent interaction test suites. A *covering array*, $CA_{\lambda}(N;t,k,v)$, is an N x k array. In every N x t subarray, each t-tuple occurs at least λ times. In our application, t is the *strength* of the coverage of interactions, k is the number of components (degree), and v is the number of symbols for each component (order). In all of our discussions, we treat only the case when $\lambda = 1$, (i.e. that every t-tuple must be covered at least once). The efficient construction of objects to represent these test suites (called *covering* arrays) is an NP-hard problem, but advances have been made within the past decade [5],[17].

Software testing using DOE methods, often referred to as combinatorial testing (or interaction testing), has been advocated as an efficient means of providing a high level of coverage of the input domain with a small number of tests, typically limited to pairwise combinations [8],[9],[12]. For example, consider a device that has 20 inputs, each with 10 settings (or 10 equivalence classes if the variables are continuous), for a total of 10^{20} combinations of settings. The few hundred test cases that can be built under most development budgets would cover an infinitesimally small proportion ($<10^{-15}$) of the possible cases. But the number of *pairs* of settings is small, and since every test case must have a value for each of the ten variables, more than one pair can be included in a single test case. Only 180 – 200 test cases [8] are required for pairwise coverage of the example above with 10^{20} combinations of settings.

Several authors have demonstrated the effectiveness of pairwise testing for software. A test set that covers all possible pairs of variable values can typically detect 50% to 75% of the faults in a program [4],[9],[26]. In other work we found 100% of faults detectable by a relatively low degree of interaction, typically 4-way to 6-way combinations [14],[15],[27]. Advances in covering array algorithms make the generation of all 4-way to 6-way combinations tractable for many realistic testing problems

Even using covering arrays, a large number of combinations will be required, but far fewer than fully exhaustive testing. For the small example used in this experiment, exhaustive coverage would have required 230,400 combinations, but all 4-way combinations were covered with 1,450, all 5-way with 4,347, and all 6-way with 10,902. Clearly, tests for such a large number of combinations could not be constructed manually. However, the application of combinatorial methods with model checking can produce test oracles and more than 17,000 test cases can be generated. These tests can be run with a few minutes of processing time and a few hours of labor. In the next section we describe the model checking approach to test case generation, followed by a review of the proof-of-concept experiment. We also discuss results on how combinatorial designs can be efficiently combined with model checking for software testing.

3. Test Generation Using Model Checking

One practical problem in software testing is determining the correct output for a given set of input variable values, generally referred to as the test oracle problem. If human intervention is required to check outputs, the number of test cases will be limited to a few hundred at most, so some form of automation is essential for thorough testing. One approach to automated generation of test oracles is model checking [1],[6], which uses a formal specification to compute expected output for input values and events.

Model checking is a formal technique based on state exploration. Input to a model checker has two parts. First is a state machine defined in terms of variables, initial values for the variables, environmental assumptions, and a description of the conditions under which variables may change value. Second is temporal logic expressions over states and execution paths. Conceptually, a model checker visits all reachable states and verifies that the temporal logic expressions are satisfied over all paths. If an expression is not satisfied, the model checker attempts to generate a counterexample in the form of a sequence of states.

A common logic for model checking is the branching-time Computation Tree Logic (CTL), which extends propositional logic with temporal operators. For example, a CTL formula `AG safe` means that all reachable states are safe, and `AG (request -> AX response)` means that `request` is always followed by `response` in the next step.

In SMV [20], a CTL symbolic model checker, a specification consists of one or more modules. One module, named main, is the top-level module. Figure 1 is an SMV example. Variables *d*, *b*, and *f* are inputs, *e* and *a* are intermediate variables. The statement *init*(*e*) := 0*;* sets *e* to 0 initially. The next value of *e* is 1 if the guard *f* = *On* is true, otherwise it is 0. The output is the variable *out*, which may be *Low* or *High*. Its value is *High* if *a* is greater than 10, otherwise it is *Low*. The SPEC clause states that if *f* is *On*, it is possible to get to some state where out is *High*. We often drop the keyword *SPEC* when the meaning is clear from the context.

Model checking can be applied to test generation and test coverage evaluation [1],[6]. In both uses, one first chooses a test criterion, that is, decides on a philosophy about what properties of a specification must be exercised to constitute a thorough test.

```
MODULE main
VAR
       d: 0..5; b: 0..11;
       f: {On, Off};
       out: {Low, High};
       a: 0..16; e: 0..1;
ASSIGN
       init(e) := 0;
       next(e) := case
              f = On : 1;
              1 : 0;
       esac;
       a := e * d + b;
       out := case
              a > 10 : High;
              1 : Low;
       esac;
SPEC AG (f = On -> EF out = High)
```

Figure 1. An SMV example.

One applies the chosen test criterion to the specification to derive test requirements, (i.e., a set of individual properties to be tested). To use a model checker, these requirements must be represented as temporal logic formulas [2]. To generate tests, the test criterion is applied to yield *negative requirements*, that is, requirements that are considered satisfied if the corresponding temporal logic formulas are inconsistent with the state machine. For instance, if the criterion is state coverage, the negative requirements are that the machine is never in state 1, never in state 2, etc.

When the model checker finds that a requirement is inconsistent, it produces a counterexample. Again, in the case of state coverage, the counterexamples would have stimulus that puts the machine in state 1 (if it is reachable), another to put the machine in state 2, and so on. Several test criteria have been proposed, including branch coverage [11] and mutation adequacy [1]. We use *t*-way coverage as a test criterion. Different methods can be used to derive the test requirements for *t*-way coverage; we present some possibilities in the Discussion section.

4. TCAS Experiment

Our experiment used a module of Traffic Collision Avoidance System (TCAS). The module is part of a set of C programs that has been used in other evaluations of software testing methods [22],[24].

The program came with 41 faulty versions derived by manually seeding realistic faults. Two thirds of the faulty versions have single changes such as replacing a constant with another constant, replacing >= with >, or dropping a condition. The rest involve either multiple changes or more complex changes. These faulty versions served as the basis of our evaluation. The program has 12 input variables specifying parameters of own aircraft and another aircraft and one output variable, *alt_sep*, a resolution advisory to maintain safe altitude separation between the two aircraft. The program computes intermediate values and prints *alt_sep* to the standard output. A model of the program was written in SMV. The model, together with the C program, was used in evaluations of specification-based mutation testing methods [22]. In this project, we used the SMV model to produce counterexamples.

To make model checking feasible, the domains of input variables were partitioned into equivalence classes and only one representative value from every equivalence class was chosen. The TCAS model has three Boolean and nine scalar input parameters, shown in Figure 2. The output variable can take one of three values.

```
Cur_Vertical_Sep : {299, 300, 601};
High_Confidence : boolean;
Two_of_Three_Reports_Valid : boolean;
Own_Tracked_Alt   : {1, 2};
Other_Tracked_Alt : {1, 2};
Own_Tracked_Alt_Rate : {600,601 };
Alt_Layer_Value : 0..3;
Up_Separation :
{0,399,400,499,500,639,640,739,740,840 };
Down_Separation :
{0,399,400,499,500,639,640,739,740,840};
Other_RAC : { NO_INTENT, DO_NOT_CLIMB,
DO_NOT_DESCEND };
Other_Capability : {TCAS_TA, OTHER};
Climb_Inhibit : boolean;
```

Figure 2. TCAS variables.

There are 230,400 possible combinations of these variables, but covering array methods make it possible to cover all 6-way combinations with only 10,902 input combinations, and all 5-way combinations with only 4,220 combinations (see Table 1).

Table 1. Combinations produced for TCAS example

2-way	3-way	4-way	5-way	6-way
100	405	1375	4220	10902

5. Discussion: Integrating Combinatorial Methods with Model Checking

While model checking has been used for test generation in a number of previous studies, it had not been integrated with combinatorial testing prior to this work. One of the significant questions we investigated was how input variable combinations should be used to generate tests. Given assertions of the form `AG(P -> AX(R))`, and tests consisting of *n* variables of strength *t* (i.e., *t*-way variable combinations), v_1 & v_2 & ... & v_n, counterexamples can be produced in several ways. At least three schemes are possible. Method 1 simply combines the input variable combination (the expression v_1 & v_2 & ... & v_n) in a conjunction with the original predicate P:

Method 1. Use

$AG(v_1$ & v_2 & ... & v_n & P -> AX !(R))

A disadvantage of this method is that as the interaction level is increased, the variable combinations v_1 ... v_n will include more variables in the conjunction, because there are fewer "don't care" conditions (those with no specified value for a particular variable in a combination). As a result, some of the v_1 ... v_n may result in v_1 & v_2 & ... & v_n & P evaluating to 0, preventing the model checker from finding a counterexample (since $0 \rightarrow Q \equiv 1$ for any Q, the expression becomes trivially true and no counterexample is possible).

The problems of method 1 can be prevented by replacing the consequent of the assertion with 0 (or !1), and removing P. This causes the model checker to find counterexamples for all of the variable combinations. But because many of the combinations include "don't care" conditions, and the model checker makes non-deterministic choices if a variable value is not asserted, the counterexamples produced may not cover all values of R.

Method 2. Use

$AG(v_1$ & v_2 & ... & v_n -> AX !(1))

Method 2 can be strengthened by including assertions for each possible value of expression R. This forces the model checker to attempt to produce counterexamples for each, not just one, value of R.

Method 3. Use

$AG(v_1$ & v_2 & ... & v_n -> AX !(R))

The last assertion means that for the chosen input variable combination, R is always false on the next step. So SMV will choose any counterexample where the combination of input variables will result in satisfaction of R. This is sufficient for the SMV example used in this paper, since it simply computes the output based on the inputs. However, reactive systems have a state, and the output depends not only on the inputs but also on the current state. For the case of reactive systems, we can strengthen Method 3 by including a particular state in the conjunction of input variables. Given a set of inputs produced by combinatorial testing, the model checker will produce a counterexample that leads to the chosen state, applies the inputs, and produces the expected outputs.

While Method 3 ensures the production of all result values, it does not guarantee that tests for *t*-tuples at an interaction coverage strength of *t* will be a subset of tests for interaction level *t*+1. Thus in some cases, faults detected by a particular interaction level may not be detected by a higher level, because of "don't care" conditions. For "don't care" conditions, we do not assert any value for the variable, so the model checker will non-deterministically select a value. In other words, tests for *t*-way interaction are not necessarily a subset of tests for (*t*+1)-way interactions

Table 2 shows the number of input combinations and test cases produced for TCAS using pairwise through 6-way interaction levels. There is not a one-to-one mapping between input combinations and test cases in Table 2 because counterexamples are produced for each of the possible outputs, and there are so many combinations with "don't care" conditions, SMV can produce more counterexamples than there are combinations. For example, a boolean input 010XXX could be mapped to two different results, since the model checker will keep trying until it finds values for the "X" - "don't care" - values that will produce a counterexample. So an input of 010XXX could produce a test case with output value UP and another test with output value DOWN, since the X's will get filled in with values that produce the two different counterexamples. This is not a significant problem in practice since they are both valid test cases. The model checker may also produce test cases that are redundant in the sense that one is a prefix of another. This output could be filtered easily, but the small number of extra test cases we generated only cost a few seconds of extra computation time. As can be seen from Table 2, the percentage of redundant tests declines rapidly as a factor as the interaction coverage increases. Of 17,470 tests generated, 17,039, or 97.5%, are unique.

Table 2. Test cases produced for input variable combinations

	2-way	3-way	4-way	5-way	6-way
Comb.	100	405	1375	4220	10902
Cases	156	461	1450	4309	11094

Counterexamples produced by the model checker were post-processed into test harness code and executed on the 41 versions of the TCAS module. We later determined that two pairs of the TCAS versions were equivalent, and a third had a seeded source code error that did not result in a fault in the executable. (The correct version has an array of length 4 that is length 3 in the faulty version, but the array is followed by empty space in memory so an execution fault did not occur; compiled with gcc ver. 3.4.4 under CygWin, on Windows XP.) Results are shown in Figure 3 and Table 3.

Table 3. Fault detection rate by interaction level

	2-way	3-way	4-way	5-way	6-way
Number detected	20	28	34	36	37
Cumulative detected	20	28	34	38	38
Detection Rate	53%	74%	89%	100%	100%

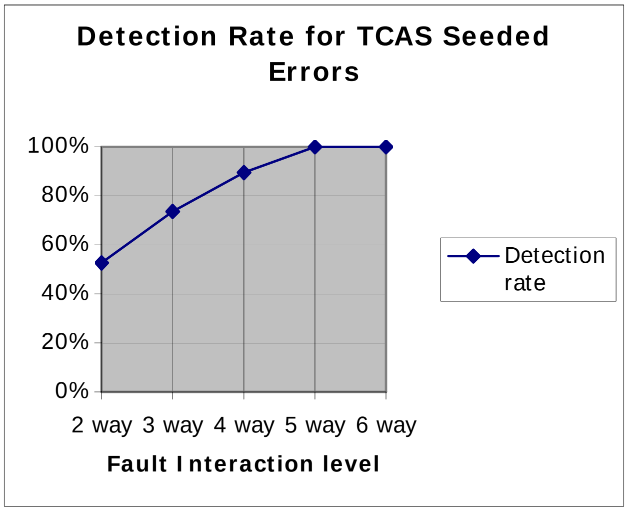

Figure 3. Fault detection rate by interaction level

The percentages of errors per failure triggering fault interaction (FTFI) level shown in Figure 3 are comparable to those we found by analyzing failure reports in large systems [14],[15],[27]. However, the faults-per-FTFI curve grows more slowly for this example than for the real-world software previously tested, suggesting that the seeded errors were relatively difficult. As shown in Figure 4, the number of tests per detected error approximately doubles with each interaction level (up to 5-way, which detected 100%), but since tests are produced and analyzed automatically, the cost in terms of time is relatively unaffected. With realistic software, it is inevitable that more human intervention will be required to review tests and results, so testing to high FTFI levels is likely to have significantly higher costs.

6. Scaling Up: A Ralistic Application

Methods described in this paper will be applied to three modules of the Personal Identity Verification (PIV) smart card, with 43, 26, and 29 variables respectively. Our current version of the extended IPO algorithm can generate combinations up to 4-way interactions in a few hours for each of these modules. For larger problems, we have developed an algorithm that can generate covering arrays for 50 – 500 parameters, depending on the level of interaction [16]. The algorithm is suboptimal in that it produces more than the minimal number of tests, but the increment beyond optimal is small, and additional tests have relatively small cost in execution time, since they are generated automatically.

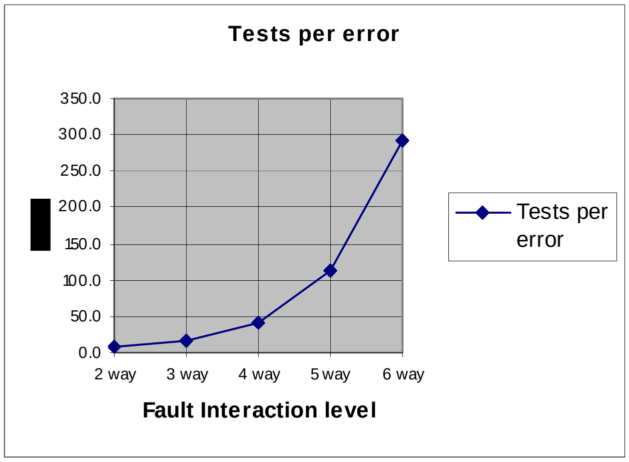

Figure 4. Number of tests per detected error

The second component of the method, model checking, is also subject to scaling problems. Once input combinations have been produced, test generation and execution can be distributed across any number of machines, since there are no dependencies between tests. Using 100 machines it is practical to generate and execute 10^6 tests in a few weeks, a level of effort consistent with most development budgets.

In addition we are investigating the use of combinatorial methods with the TVEC test generation tool [3], [7], which is already being used to produce tests for the PIV card. TVEC is efficient and appears to be suitable for integration with combinatorial methods.

7. Conclusions

This work serves as a proof of concept for integrating combinatorial testing with model checking to provide automated specification based testing. One valuable result from the project was determining the most efficient way to integrate combinatorial testing with model checking. Results suggest that this approach is efficient and can be effective.

8. Acknowledgments

We thank Paul Black for insightful comments about the use of model checking for test generation. We are grateful to Renee Turban for improving our understanding of covering array algorithms and for many helpful suggestions on this paper.

9. References

[1] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proc. Second IEEE Internat. Conf. on Formal Engineering Methods (ICFEM'98)*, pp. 46–54. IEEE Computer Society, Dec. 1998.

[2] P. E. Ammann and P. E. Black. A specification-based coverage metric to evaluate test sets. In *Proc. Fourth IEEE Internat. High-Assurance Systems Engineering Symp. (HASE 99)*, pp. 239–248. IEEE Computer Society, November 1999.

[3] Blackburn, M.R., R.D. Busser, A.M. Nauman, R. Knickerbocker, R. Kasuda, Mars Polar Lander Fault Identification Using Model-based Testing, Proceeding in IEEE/NASA 26th Software Engineering Workshop, p. 163, November 2001.

[4] R. Brownlie, J. Prowse, and M.S. Phadke. Robust Testing of AT&T PMX/StarMail using OATS. *AT&T Technical Journal*, 71(3): 41-47 (May/June 1992).

[5] R. Bryce, C.J. Colbourn, M.B. Cohen. *A Framework of Greedy Methods for Constructing Interaction Tests.* The 27th International Conference on Software Engineering (ICSE), St. Louis, Missouri, pages 146-155. (May 2005).

[6] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using model-checking. In *Proc. 1996 SPIN Workshop*, Aug 1996.

[7] Chandramouli, R. "Security Functional Testing Using Model-Based Test Automation Approach," *Quality Week*, pp. 3 – 6, (Sept. 2002).

[8] D.M. Cohen, S.R. Dalal, J. Parelius, and G.C. Patton. The Combinatorial Approach to Automatic Test Generation. *IEEE Software*, vol. 13, no. 5: 83-88, (September 1996).

[9] S.R. Dalal, A. Jain, N. Karunanithi, J.M. Leaton, C.M. Lott, G.C. Patton, B.M. Horowitz, Model-Based Testing in Practice, International Conference on Software Engineering, 1999.

[10] R.A. Fischer. The Arrangement of Field Experiments. *Journal of Ministry of Agriculture of Great Britain.* 1926. 33:503-513. Available on-line: http://www.library.adelaide.edu.au/digitised/fisher/48.pdf

[11] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In Proceedings of the Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 146-162, Toulouse, France, September 1999.

[12] Grindal, Mats, Offutt, Jeff, and Andler, Sten F. "Combination Testing Strategies: A Survey," *Journal of Software Testing, Verification and Reliability* vol. 15, no. 3, pp. 167-199.

[13] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. Sixteenth Internat. Conf. On Software Engineering*, pp. 191–200, May 1994.

[14] D.R. Kuhn, D.R. Wallace, and A.M Gallo, "Software Fault Interactions and Implications for Software Testing," *IEEE Transactions on Software Engineering*, 30(40):1-4 (2004)

[15] D.R. Kuhn, M.J. Reilly, "An Investigation of the Applicability of Design of Experiments to Software Testing", *27th NASA/IEEE Software Engineering Workshop*, IEEE Computer Society, pp. 91-95, 4-6 December, 2002.

[16] D.R. Kuhn, "An Algorithm for Generating Very Large Covering Arrays", NISTIR 7308, 20 March 2006.

[17] Y. Lei, K.C. Tai. In-parameter order: a Test Generation Strategy for Pairwise Testing. Proceedings of the Third IEEE High Assurance Systems Engineering Symposium, pp. 254-261, IEEE, Nov. 1998.

[18] R. Mandl. Orthogonal Latin squares: An application of experiment design to compiler testing. Communications of the ACM, 28(10): 1054-1058 (October 1985).

[19] J.D.McGregor, D.A. Sykes, *Practical Guide to Testing Object-Oriented Software,*Addison-Wesley, 2001.

[20] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[21] V.N. Nair, D.A. James, W.K. Erlich, J. Zevallos, "A Statistical Assessment of Some Software Testing Strategies and Application of Experimental Design Techniques", *Statistica Sinica*, vol. 8, no. 1, pp 165-184, 1998.

[22] V. Okun, P. E. Black, and Y. Yesha, *Testing with Model Checker: Insuring Fault Visibility*, WSEAS Transactions on Systems, Vol. 2, Issue 1, pages 77-82, January 2003.

[23] R.S. Pressman. *Software Engineering: A Practitioner's Approach 5th edition*, McGraw Hill, 2001.

[24] G. Rothermel and M. J. Harrold. *Empirical studies of a safe regression test selection technique*. IEEE Transactions on Software Engineering, 24(6): 401-419, 1998.

[25] B. D. Smith, M. S. Feather, N. Muscettola. "Challenges and Methods in Validating the Remote Agent Planner", *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems (AIPS-2000)*, Breckenridge, CO.

[26] K.C. Tai, Y. Lei. A Test Generation Strategy for Pairwise Testing. IEEE Trans. Software Eng. vol. 28, no. 1, 109-111 (January 2002).

[27] D.R. Wallace, D.R. Kuhn. "Failure Modes in Medical Device Software: an Analysis of 15 Years of Recall Data", *International Journal of Reliability, Quality, and Safety Engineering*, Vol. 8, No. 4, 2001.