ISSN 1744-1986

The Open University

Technical Report N^o 2007/08

AETG vs. Man: an Assessment of the Effectiveness of Combinatorial Test Data Generation

Michael Ellims
Darrel Ince
Marian Petre

20th June 2008

Department of Computing
Faculty of Mathematics and Computing
The Open University
Walton Hall,
Milton Keynes
MK7 6AA
United Kingdom

http://computing.open.ac.uk

AETG vs. Man: an Assessment of the Effectiveness of Combinatorial Test Data Generation

Michael Ellims
Pi SHURLOK
Milton Hall, Milton, Cambridge,
CB24 6WZ, UK
+44 (0)1223 203 913
mike.ellims@pi-shurlok.com

Darrel Ince
Dept of Computing, Open University
Walton Hall
Milton Keynes, UK
00441908 274066
d.c.ince@open.ac.uk

Marian Petre
Dept of Computing, Open University
Walton Hall
Milton Keynes, UK
00441908 274066
m.petre@open.ac.uk

ABSTRACT

This paper reports on an industrial study of the effectiveness of test data generation. In the literature on the automatic generation of test data a number of techniques stand out as having received a significant amount of interest. One area that has achieved considerable attention is the use of combinatorial techniques to construct data adequate test sets that ensure all pairs, triples etc. of input variables are included in at least one test vector. There has been *some* systematic evaluation of the technique as applied to unit testing and, while there are indications that the technique can be effective, very little work has been performed using industrial code. Moreover, there has been no comparison of effectiveness of the technique for unit testing compared with tests that are generated by hand. In this paper we apply random and combinatorial (AETG) techniques to a number of functions drawn from industrial code with known faults and existing unit test suites. Results indicate that for simple cases combinatorial techniques can be as effective as the human-generated test, but there are instances associated with complex code where the technique performs poorly—but no worse than randomly generated data.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**] - Testing tools - *data generators, coverage testing*.

General Terms

Algorithms, Reliability, Experimentation, Verification.

Keywords

Software testing, random testing, test case selection, automated test generation, partition testing, unit testing, factor covering design, factorial experiments, covering arrays, covering designs, combinatorial deign, pair-wise testing, coverage, *n*-way testing, greedy algorithms.

1. INTRODUCTION

An important technique for the generation of test data involves combinatorial selection, where the tester selects test points of "interest" based on selecting data input ranges, domain partitioning and possibly using heuristic rules. These points are then combined automatically to generate test vectors with specific properties, for example that all pairs of values are present in the test set.

The original work in this area was presented by Mandl [1] using orthogonal arrays. Sherwood [2], [3], [4] developed the Constraint Array Test System (CATS) to generate test sets algorithmically and this work was later extended by Cohen *et al.* [5], [6], [7], [8] as the automatic efficient test generator (AETG) system; this has been the focus of much latter work.

Evidence of the effectiveness of these techniques fall into two main categories: reports of the tools in field use Brownlie *et al.* [9], Cohen *et al.* [5], [6], [10], Burr and Young [11], Dalal *et al.* [12], Smith et al. [13], [14] and a small amount of experimental work Dunietz *et al.* [15], Nair *et al.* [16], Kobayashi, Tsuchiya and Kikuno [17], Schroeder *et al.* [18], Grindal *et al.* [19].

There are a number of useful combinatorial techniques for generating test vectors. The vast majority of reported work reports on the construction of test sets in which all pairs, triples, etc. of data points can be generated and how effective those test sets can be. The motivation for this is derived from the way statistical experiments in various fields are conducted to maximize the amount of information obtained for the minimum amount of effort. The general topic is called design of experiments or factorial experiments; it is covered widely in areas such as industrial quality control and engineering [20].

There are a number of studies that have examined real systems which report on the detection of additional errors using combinational techniques. Table 1 summarizes some of the major results from a series of field studies. By 'field study' we mean an uncontrolled experiment.

It is difficult to draw too many conclusions from the limited data presented in Table 1,however it is obvious that the combinatorial techniques employed were more effective in practice than the alternative techniques that were being employed. What cannot be said is that the increased effectiveness directly results from the combinatorial features techniques employed. One hypothesis is that the alternative techniques were employed poorly or that they were not sufficiently systematic and that some other systematic technique would have produced similar results—since we do not know in any detail what testing techniques were applied as a comparator.

Table 1 : summary of data drawn from field studies.

Study	System or Module	Faults Discovered	Detected by Test Team
Brownlie et al. [9]	mail system	12%	---
Cohen et al. [6]	Bellcore 1994 Release	49	---
	Bellcore 1995 Release	44	---
Dala et al. [10]	Arithmetic tool[1]	43	---
	message parsing[2]	24	3
Dala et al. [12]	rule based personal management	4	1
	user interface	6	3

However, while there is an indication that the use of combinatorial techniques is better than the test process that was then in use, this research provides only weak evidence that the technique is an advance on state of the art. The confounding factor for this in all the publications cited in Table 1 is the uncontrolled nature of the test process which it is being compared to.

Given the period of time in which combinatorial testing using covering arrays has been in use there are surprisingly few controlled experimental studies. There are five major studies: [15] which addressed coverage, [16], [17], [18], [19] which addressed effectiveness at detecting mutants and [21] which compared the strength of *t*-strength covering arrays relative to full factorial designs.

The experiment reported in [15] examined the effectiveness in terms of code coverage of covering arrays *vs*. random designs (i.e. random selection with replacement) and produced results that broadly support the results from field studies. Coverage data reported in [19] for branch coverage is also consistent with field studies, but the authors after examination of the data concluded that code coverage methods may also need to be employed.

Kobayashi *et al*. [17] examined the fault detecting ability of several techniques as applied to the specific problem of testing logic predicates, [18] examined the fault detecting ability of covering array based techniques vs. random testing and [19] examined the fault detecting power of a number of different combinatorial strategies including 1-way (each choice), base choice (a single factor experiment), pair-wise AETG [8] and orthogonal arrays.

Unfortunately there is less overlap between the studies than first appears, e.g. [17] used random testing without replacement where as [18] applied random selection of partitioned values without replacement and [19] ignored random testing completely.

Nair et al. [16] investigated random testing (without replacement, no partitioning) vs. partition based testing and showed that, in general, partition testing was more effective. The particular case of partition testing is an application of experimental design and it shows that the probability of detecting the failure for simple random testing is significantly lower.

In [19] several of the weaker techniques were examined and, as expected, the 1-way test performed poorly compared to other techniques. However it was found that the base choice technique performed as well as orthogonal arrays and AETG in 3 out of 5 problems. No technique detected fewer than 90% of the detectable faults.

Our conclusion is that the literature indicates that there is no overwhelming consensus as to the utility of combinatorial techniques, though there is some evidence that such techniques are better than simple random testing.

Our research addresses this fundamental question about utility and reports on an experimental study on code drawn from a large industrial data set.

1.1 Code Mutation

One of the main limitations to the fidelity of much experimental work that evaluates the effectiveness of testing techniques is the inability of common metrics such as code coverage to distinguish between test sets that reach code but do not stress the code sufficiently to reveal errors and test sets that do.

Code mutation as proposed by DeMillo *et al*. [22] has been used previously in studies to compare test effectiveness [23], [24], [25]. It has the advantage that it subsumes conditional coverage techniques [26] and has recently been applied to evaluating random testing with C programs [27], [28]. As far as the authors are aware this is the first instance where it has been applied to combinatorial data generation techniques.

2. THE EXPERIMENTAL STUDY

2.1 The data set used

The functions that were used in this study were drawn from a system that controls a large industrial engine currently employed in safety-critical applications We will refer to this as the Wallace system. It contains around 70,000 lines of C code. The system is developed in a manner consistent with IEC standard 61508 and code has been subjected to review, unit, integration and system testing on the bench. A fuller description can be found in [29] which describes the effectiveness of the unit test process.

[1] Data for this study are the number of failure classes reported. It is not know if this corresponds to a single fault.

[2] Note Dalal 99 reports 27 failure classes rather than 24.

2.2 The Procedure Employed

The procedure we employed in this experiment consisted of the following steps:

- We developed a tool for generating pair-wise adequate tests based on the work by Cohen *et al.* [6] [8] and Cohen et al. [30].
- We developed a mutation tool that: produced operator mutations, variable type specifier mutations[3] (e.g. `char` for `int`), variable name mutations, constant mutations and statement removal mutations; this was inspired by the tool used in [27].
- We identified a body of industrial code from the Wallace project [29] which had known errors which had a range of code complexities. This contained eight C functions ranging from 12 to 62 executable lines of code (excluding comments, blank lines and braces).
- From the project archive we extracted the hand-generated test vectors and the information used to generate them from the detailed designs and data dictionary.
- For each of the C functions selected we transformed them into the code format that was required for our mutation tool that we used (see [31] for details).
- We carried out a major experiment and two subsidiary experiments as follows:
- The first experiment involved running the mutations for each function with the hand-generated test vectors, the vectors produced by the AETG algorithm [8] for generating pair-wise adequate test sets and randomly generated test data with a similar size.
- The second experiment took a subset of the C functions and examined whether a more sophisticated approach to using the AETG algorithm could produce a significant improvement in performance.
- The third experiment examined whether simply generating more vectors could improve the effectiveness of random testing. This was done in order to establish a comparator for the pair-wise technique [32].

2.3 The Code Selected

Table 2 summarizes the eight functions that we examined. Four were selected on the criteria that they contained known errors discovered running the unit tests (as opposed to designing the tests). The remainder were selected based on their complexity, for example _gov_gen_ffd_rpm was selected because it contains a large number of conditional statements (eleven).

Table 2 gives the following information, the first column is the function name and the second column is the number of executable lines. Columns three and four give the total number of mutants generated and the number of valid mutants that would actually compile (ignoring warning for divide by zero etc). The fifth and sixth columns are the nesting factor and the condition factor as used in [33]. Nesting factor is the maximum depth of nesting and the condition factor is the maximum number of comparisons performed in a single `if` statement. The seventh column is a simple count of the number of `if` statements in the code, each `case` of a `switch` statement being counted as a single `if` statement. The final column is the number of inputs to the function. The function `dip_debounce` stands out here but this is because the underlying data structure is a set of arrays and the original test set contained data values for the first, middle and last elements of the arrays.

2.4 The Experiments

2.4.1 Experiment 1

This directly compared the hand-generated test vectors with test vectors generated by the straightforward application of the AETG algorithm using three values (minimum, middle and maximum) for scalar types and all values for enumerations. As a comparison we included randomly generated sets of test vectors of comparable size.

Table 3 shows the results of this experiment, showing the function name, the number of valid mutants i.e. those that compile with no errors, the number of vectors in the test set for the hand generated data and the number of vectors generated by our implementation of the AETG algorithm. The next two columns show the number of mutants left alive after running each set of test vectors on all valid mutants for hand generated and AETG generated tests. The final two columns detail the results for randomly generated test sets for the same size as the hand generated vectors (Random vs. Hand) and the same size as the AETG generated vectors. Note that where the size of the hand generated test sets and those generated by the AETG algorithm are similar only one size of random test (the larger) was used.

The main results can be summarized as follows;

- Random testing outperforms the AETG algorithm in two of seven cases, the other being a draw.
- In one out of six cases with to draw random testing actually outperforms the hand generated tests.
- For the most complex functions, those in the last two rows, the hand generated tests outperforms both techniques by significant margins.

The functions that have known errors committed by programmers during development of the code are listed on Table 4. As before, column 1 is the function name, column 2 shows whether the error was "found" with the hand generated test set and column three shows whether the AETG test set found the error. Column 3 shows whether the randomly generated test vectors found the error. The last column shows whether the error was a mutant or not.

When we applied the sets of test vectors used against mutants we found that all of the test data generation techniques appeared to be effective at revealing those errors. Therefore, this part of the experiment does not allow us to draw any strong conclusions; but suggests that many real-world errors could, in practice, be quite "shallow" and possibly amenable to being found by any testing (automated or non-automated); this was observed by Duran and Ntafos [34]. However it should be noted that the complex functions were not included in this set.

[3] These were included as they are a known source of errors in embedded systems where memory constraints are tight.

Table 2 : A summary of some properties of the code under study, with the C functions ranked by the number of valid mutations that are generated.

Function Name	Lines	Total Mutants	Valid Mutants	Nesting Factor	Condition Factor	if statements	Inputs
dip_debounce	12	127	81	2	2	2	17
dip_check_cal	19	103	97	1	1	3	2
aip_spike_filter	22	354	178	3	1	4	7
sdc_fuel_control	17	267	213	2	2	5	9
aip_median_filter	25	217	217	1	1	4	3
aip_apply_filters	30	605	311	2	2	4	8
sdc_pre_start	51	1472	1237	3	1	8	3
gov_gen_ffd_rpm	62	1698	1227	4	2	11	16

Table 3: Results of the first experiment showing the number of mutants left alive after all test vectors have been applied. The test set that left the fewest mutants alive is highlighted in bold.

Function Name	Valid Mutants	Hand Vectors	AETG Vectors	Alive Hand	Alive AETG	Random vs. Hand	Random vs. AETG
_dip_debounce	81	18	17	12	***9***	12	---
_dip_check_cal	97	8	8	0	0	0	---
aip_spike_filter	178	40	15	***18***	42	82	90
_sdc_fuel_control	213	15	15	***21***	107	101	---
_aip_median_filter	217	27	9	***41***	47	53	57
_aip_apply_filters	311	68	21	64	58	***57***	57
_sdc_pre_start	1237	14	16	***675***	746	---	891
_gov_gen_ffd_rpm	1227	14	18	***152***	729	---	744

Table 4: Effectiveness of the test vs. known actual errors in the code.

Function Name	Hand : error found	AETG : error found	Random : error found	Error is mutant
_dip_check_cal	Yes	Yes	Yes	Yes
_dip_debounce	Yes	Yes	Yes	No
_sdc_pre_start	Yes	Yes	Yes	No
_aip_apply_filters	Yes	Yes	Yes	Yes
_aip_apply_filters	Yes	Yes	Yes	No

2.5 Experiment 2

This experiment looked at some simple ways of making data generated by the AETG algorithm more effective, for example the following were considered:

- invert: changing the assignment of values i.e. converting elements that had the value `TRUE` to `FALSE` and vice verse;
- interleave: rather than simply selecting the minimum, middle and largest value, we interleaved values that appeared together in conditional statements;
- biasing: altering the distribution of the data generated by adding duplicate elements. For example specifying three values for a Boolean value as `FALSE`, `FALSE`, `TRUE` rather than just `FALSE` and `TRUE`.
- simple combination of the previous techniques e.g. interleaving and biasing.

Interleaving is very similar to domain partitioning, however here the analysis has been less rigorous so the term has not been applied. This is intentional in that the authors are deliberately attempting to avoid performing the type of analysis used in the construction of the original test sets, while attempting to obtain some of the advantages of such analysis. An example of the procedure as applied to the `_sdc_fuel_control` functions is shown below.

The function compare four variables (A to D) against a global value for engine revolutions per minute (R). Values were then allocated as follows;

```
R  0      500      1000        2000       4000
A    400     600       1100
B      450     650         1150
C    400     600       1100        2500
D      450     650         1150       2550
```

One interesting aspect seen here is that the function `_sdc_pre_start` was not amenable to interleaving, no complex comparisons between variables being present. Exactly why this function seems to be so difficult to test is at this point slightly bemusing.

There are some hints in the nature of the code, for example no computed values are assigned, all assignments are from constant values and many of these are to logical `TRUE` which in C is any no-zero value. Explicitly assigning `TRUE` to a fixed value and testing for that may form part of the solution however the driver program does test for equality of returned values so this may not be the cause.

The results from this are shown in Table 5. The first column is the function name, the second column (Base) is the number of mutants left alive as in experiment 1. The column labeled "invert" is an inversion of data fields in the vectors used in experiment 1. The column labeled "Interleave" is a new set of test vectors generated using the technique described above. The column labeled "Interleave + Inverted" has the same values inverted as in column 3 for the new vectors from column 4. The column labeled "biased" has one or more variables biased with the introduction of duplicate values. The seventh column combines the input values for the interleaved and biased test data generation. The last column replicates data for hand generated tests for reference.

Table 5: results for experiment two, to improve the mutant kill rate by modifying the input data points (e.g. interleaving) or the interpretation of those points (inverting). Two set of data are shown for each function, mutants left alive and number of vectors.

Function	Base	Invert	Interleave	Interleave + Inverted	Biased	Interleave + Biased	Hand Generated
_sdc_fuel_control	107	97	96	25	31	26	26
	15	15	23	23	15	23	15
_sdc_pre_start	746	746	N/A	N/A	956	N/A	675
	16	16	---	---	20	---	14
_gov_gen_ffd_rpm	729	614	584	577	637	562	152
	18	18	26	26	19	27	14

The results in Table 5 suggests that:

- a more careful selection of data points for the AETG algorithm can pay dividends, especially with less complex code.
- for more complex code that simple pair-wise testing is not of itself effective and that perhaps tests that ensure triples or even higher orders is required.

Both of these observation are consistent with observation from Cohen *et al*. [6] and Dalal *et al*. [12] that the data models are of prime importance and with the observation that in [6] that modeling the functionality not the interface appears to yield the best results.

2.5.1 Experiment 3

This experiment examined whether test effectiveness can be increased by simply using larger random test vectors. We selected the same functions as in experiment two, and increased the number of vectors generated in each case. Table 6 shows the results, as before the first column is the function name and the other columns show the number of mutant killed for each function for the number of test vectors given in the first row, i.e. 20, 40, 60 etc.

The results from this experiment suggest that:

- For simple functions random testing does indeed appear to be effective. This supports the contention made in [34].
- For complex functions random testing does not appear to be effective. This confirms work reported in [33] which suggests for complex code random testing should not be the method of choice.

Note that the size of the random test sets is constrained by the amount of time it takes to execute all mutants. The larger test set sizes have execution times of several hours, the largest being run over night.

Table 6 : Effect on the number of mutations left alive by increasing the number of randomly generated vectors

Function	20	40	60	100	200
_sdc_fuel_control	103	96	36	32	32
_sdc_pre_start	840	817	817	817	817
_gov_gen_ffd_rpm	726	717	688	688	688

3. THREATS TO VALIDITY

While we feel that this work is a very valuable contribution to the literature on empirical testing it is worth describing some possible weaknesses and our response:

- The code base used for this study may not be representative of the majority of problems. However, one of the author's experience of safety-critical development in the areas of automotive and aerospace engineering indicates that the code used for the study was less complex and cleaner than most he has come into contact with in his career. The message here is that the negative results in this study would hold *more strongly* for other systems.
- The way that we have applied pair-wise testing may be more naïve than would be the case than envisaged by its proponents. We feel that our application of this technique replicates how it may be used in the majority of real-life situations and follows broadly advice presented in text books by Copeland [35] and Kaner *et al.* [36] which will possibly provide the primary source of information on combinatorial techniques for practitioners.
- Another objection is that the tool that we employed to introduce mutations does not reflect real errors. Our reply to this is that it is much better than simple metrics such as structural coverage which are often used as a surrogate for test effectiveness in research studies. It offers much greater degree of discrimination in terms of test set adequacy.

4. CONCLUSIONS

The results of these experiments have been surprising. At the start of this study we all thought that combinatorial techniques offered a valid way of testing critical software. Our results show that:

- pair-wise combinatorial techniques are not adequate for testing complicated critical software which involves a large density of control structures and large interfaces between execution units (in this case C functions); this somewhat challenges the current orthodoxy, though it should be noted that Kuhn and Reilly [37] noted that the technique may not be applicable to real time state based systems.
- random testing is even worse than pair-wise testing for complex code;
- there is a need for further research which replicates the experimental study for triples. We feel that triples may offer an improvement over the results reported here since they increase the probability of complex paths being executed. This will be the subject of further work.
- that for the most complex functions the hand-generated tests outperforms both techniques by significant margins. This suggests a pessimistic message for anyone who wishes to use automated tools based on combinatorial techniques. We hope that the work reported here would give rise to further studies that confirmed or refute this hypothesis.

5. FUTURE WORK

The results of this study suggest some obvious avenues for future work, obviously a larger number of functions need to be investigated to see if the results are consistent. Other functions could be selected from the same project (Wallace) or possibly be taken from other projects.

Other functions could be investigated, for example various sort algorithms have been popular in test research and a preliminary study by the authors using C code for bubble, insertion, shell, heap and quick sort algorithms, conducted while developing the tools used in the study reported here indicated that compared with a small number of hand generated test the AETG algorithm performed very poorly. It may be beneficial to revisit this work.

In addition more advanced applications of the AETG algorithm to generate 3-wise or stronger test sets is also necessary as is work to extend our current tool set to include some of the facilities for building more complex or complete data models such as those provided in the AETG tool itself.

Other possibilities also exist. For example optimization techniques as applied with adaptive testing [38], [33], [39] could be employed, either to improve the test sets after they have been generated by the AETG algorithm or alternately to optimize the input data to the AETG algorithm.

6. ACKNOWLEDGMENTS

Our thanks to J. H. Andrews for making his mutation tool available for evaluation.

7. REFERENCES

[1] R. Mandl, "Orthogonal Latin squares: an application of experiment design to compiler testing," *Commun. ACM*, vol. 28, pp. 1054-1058, 1985.

[2] G. B. Sherwood, "Improving test case selection with constrained arrays," AT&T Report Number e.g. SCE-04-15, 1990.

[3] G. B. Sherwood, "Improving test case selection with constrained arrays II," AT&T Report Number e.g. SCE-04-15, 1990.

[4] G. Sherwood, "Effective Testing of Factor Combinations," *Third Int'l Conf. Software Testing, Analysis and Review*, 1994.

[5] D. M. Cohen, S. R. Dalal, A. Kajla, and G. C. Patton, "The automatic efficient test generator (AETG) system," in *Proceedings., 5th International Symposium on Software Reliability Engineering*. Monterey, CA, USA, 1994.

[6] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, "The Combinatorial Design Approach to Automatic Test Generation," *IEEE Softw.*, vol. 13, pp. 83-88, 1996.

[7] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. Patton, "Method and system for automatically generating efficient test cases for systems having interacting elements," U. S. P. Office, Ed., 1996.

[8] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG System: An Approach to Testing Based on Combinatorial Design," *IEEE Trans. Softw. Eng.*, vol. 23, pp. 437-444, 1997.

[9] R. Brownlie, J. Prowse, and M. S. Phadke, "Robust Testing of AT&T PMX/StarMAIL Using Oats," *AT&T Technical Journal*, vol. 71, 1992.

[10] S. Dalal, A. Jain, N. Karunanithi, J. Leaton, and C. Lott, "Model-based Testing of a Highly Programmable System," in *Proceedings of the The Ninth International Symposium on Software Reliability Engineering*: IEEE Computer Society, 1998.

[11] K. Burr and W. Young, "Combinatorial test techniques: table-based automation, test generation and code coverage," in *Intl. Conf. on Software Testing, Analysis, and Review (STAR)*. San Diego, CA, 1998.

[12] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model-based testing in practice," in *Proceedings of the 21st international conference on Software engineering*. Los Angeles, California, United States: IEEE Computer Society Press, 1999.

[13] B. Smith, W. Millar, Y. W. Tung, P. Nayak, E. Gamble, and M. Clark, "Validation and Verification of the Remote Agent for Spacecraft Autonomy," in *Proceedings of the 1999 IEEE Aerospace Conference*, 1999.

[14] B. Smith, M. S. Feather, and N. Muscettola, "Challenges and Methods in Testing the Remote Agent Planner," presented at Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems, Breckenridge, CO., 2000.

[15] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino, "Applying design of experiments to software testing: experience report," in *Proceedings of the 19th international conference on Software engineering*. Boston, Massachusetts, United States: ACM Press, 1997.

[16] V. N. Nair, D. A. James, W. K. Ehrlich, and J. Zevallos, "A statistical assessment of some software testing strategies and application of experimental design techniques," *Statistica Sinica.*, vol. 8, pp. 165-184, 1998.

[17] N. Kobayashi, T. Tsuchiya, and T. Kikuno, "Non-specification-based approaches to logic testing for software," *Information and Software Technology*, vol. 44, pp. 113-121, 2002.

[18] P. J. Schroeder, P. Bolaki, and V. Gopu, "Comparing the Fault Detection Effectiveness of N-way and Random Test Suites," in *Proceedings of the 2004 International Symposium on Empirical Software Engineering (ISESE'04) - Volume 00*: IEEE Computer Society, 2004.

[19] M. Grindal, B. Lindström, A. J. Offutt, and S. F. Andler., "An Evaluation of Combination Strategies for Test Case Selection," Department of Computer Science, University of Skövde HS-IDA-TR-03-001, 2003 2003.

[20] W. J. Diamond, *Practical Experiment Design For Engineers and Scientists*. New York: John Wiley & Sons, 2001.

[21] D. Hoskins, R. C. Turban, and C. J. Colbourn, "Experimental designs in software engineering: d-optimal designs and covering arrays," in *Proceedings of the 2004 ACM workshop on Interdisciplinary software engineering research*. Newport Beach, CA, USA: ACM Press, 2004.

[22] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: help for the practising programmer," *Computer*, pp. 34-41, 1978.

[23] M. Daran and P. Thevenod-Fosse, "Software error analysis: a real case study involving real faults and mutations," *SIGSOFT Softw. Eng. Notes*, vol. 21, pp. 158-171, 1996.

[24] Frankl, Sweiss, and C. Hu, "All-uses vs mutation testing: an experimental comparison of effectiveness," *J. Syst. Softw.*, vol. 38, pp. 235-253, 1997.

[25] Y. Zhan and J. A. Clark, "Search-based mutation testing for Simulink models," in *Proceedings of the 2005 conference on Genetic and evolutionary computation*. Washington DC, USA: ACM Press, 2005.

[26] A. J. Offutt and J. M. Voas, "Subsumption of condition coverage techniques by mutation testing," Dept. of Information and Software Systems Engineering , George Mason Univ., Fairfax, Va. ISSE-TR-96-1OO, 1996.

[27] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for test experiments?," in *Proceedings of the 27th international conference on Software engineering*. St. Louis, MO, USA: ACM Press, 2005.

[28] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria," *IEEE Transactions on Software Engineering*, vol. 32, pp. 608-624, 2006.

[29] M. Ellims, J. Bridges, and D. C. Ince, "The Economics of Unit Testing," *Empirical Softw. Eng.*, vol. 11, pp. 5-31, 2006.

[30] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn, "Constructing test suites for interaction testing," in *Proceedings of the 25th International Conference on Software Engineering*. Portland, Oregon: IEEE Computer Society, 2003.

[31] E. Ellims, "The Csaw Mutation Tool Users Guide," Open Univerity 2007.

[32] D. C. Ince and S. Hekmatpour, "An empirical investigation of random testing," *The Computer Journal*, vol. 29, pp. 380, 1986.

[33] C. C. Michael, G. McGraw, and M. A. Schatz, "Generating Software Test Data by Evolution," *IEEE Trans. Softw. Eng.*, vol. 27, pp. 1085-1110, 2001.

[34] J. Duran and S. Ntafos, "An evaluation of random testing," *IEEE Transactions on Software Engineering*, vol. 10, 1984.

[35] L. Copeland, *A Practitioner's Guide to Software Test Design*. Boston: Artech House Publishers, 2004.

[36] C. Kaner, J. Bach, and B. Pettichord, *Lessons learned in software testing: a Context driven approach*. New York: John Wiley & Sons, 2002.

[37] D. R. Kuhn and M. J. Reilly, "An Investigation of the Applicability of Design of Experiments to Software Testing," in *Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW-27'02)*: IEEE Computer Society, 2002.

[38] M. J. Gallagher and V. L. Narasimhan, "ADTEST: A Test Data Generation Suite for Ada Software Systems," *IEEE Trans. Softw. Eng.*, vol. 23, pp. 473-484, 1997.

[39] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information and Software Technology*, vol. 43, pp. 841-854, 2001.