The Effectiveness of T-Way Test Data Generation

Michael Ellims[1], Darrel Ince[2], and Marian Petre[2]

[1] Pi-Shurlok, Milton Hall, Cambridge, UK
[2] Dept. of Computing, Open University
Walton Hall, Milton Keynes, UK
mike.ellims@pi-shurlok.com, {d.c.ince,m.petre}@open.ac.uk

Abstract. This paper reports the results of a study comparing the effectiveness of automatically generated tests constructed using random and *t*-way combinatorial techniques on safety related industrial code using mutation adequacy criteria. A reference point is provided by hand generated test vectors constructed during development to establish minimum acceptance criteria. The study shows that 2-way testing is not adequate measured by mutants kill rate compared with hand generated test set of similar size, but that higher factor *t*-way test sets can perform at least as well. To reduce the computation overhead of testing large numbers of vectors over large numbers of mutants a staged optimising approach to applying *t*-way tests is proposed and evaluated which shows improvements in execution time and final test set size.

Keywords: Software testing, random testing, automated test generation, unit test, combinatorial design, pairwise testing, t-way testing, mutation.

1 Introduction

How to generate test sets automatically has been the subject of much research and a wide range of techniques have been proposed ad investigated. These including random generation [1], search techniques such as generic algorithms [2] and combinatorial techniques [3] based on statistical design of experiments [4] used to identify and isolate the effects of interactions between factors of interest.

For unit testing the factors of interest are the input variables of the function under test and the interactions between different values of those variables and how they effect the outcome of running the code. If we generate vectors that cover all 2-way (pairwise) interactions between n input variables v_1 to v_n then there will be a vector in the test set such that for every value that the variable v_i is allowed to take it will be paired with each value the variable v_j is allowed to take for all i and j where $i \neq j$.

An important consideration is which values each variable will be allowed to take on. In general the tester will select data points for each input variable that are of "interest" based on criteria such as data input ranges, domain partitioning and other heuristic rules. Selection all values is impossible except where only a small number of values are allowed such as for enumerations.

To make this more concrete consider a function with three input variables, v_1, v_2 and v_3 that take on the values a_1, a_2, a_3 and b_1, b_2 and c_1, c_2 respectively. Then a 2-way adequate test set that ensures that a vector exits that contains all values of v_1 paired with all values of v_2 and all values of v_3 and all pairs of v_2 and v_3 is shown in Figure 1.

M.D. Harrison and M.-A. Sujan (Eds.): SAFECOMP 2008, LNCS 5219, pp.16–29, 2008.

$$\begin{array}{ccccccc} a_1 & a_2 & a_3 & a_2 & a_1 & a_3 & a_1 \\ b_2 & b_1 & b_1 & b_2 & b_2 & b_2 & b_1 \\ c_1 & c_2 & c_1 & c_1 & c_2 & c_2 & c_1 \end{array}$$

Fig. 1. An example seven vector, 2-way adequate test set for 3 variables

Larger values of t can be used, for example $t = 3$ would involve matching sets of three variables and $t = 4$, four variables in the same way. The advantage of taking this approach is that far fewer vectors are required to construct a t-way adequate test set than would be required for a test set that contained all combination of values. The work presented in this paper is an investigation of the utility of t-way test generation for unit testing and in particular to determine its suitability for safety-related software.

1.1 Contributions of This Work

The work presented here makes the following contributions;

- it provides a direct comparison with t-way adequate test sets against human generated test sets.
- it provides a practical method of incorporating high factor t-way testing and mutation analysis into a development process which can avoid much of the computational overhead that may otherwise be encountered.

2 Related Work

2.1 Combinatorial Techniques

The original work on using combinatorial techniques for testing was presented by Mandl [5] who used orthogonal arrays to select sets of constructs for testing an Ada compiler. Sherwood [6] developed the Constraint Array Test System (CATS) to generate test sets algorithmically. This work was extended by Cohen *et al.* [3] as the automatic efficient test generator (AETG) system and this algorithm has been the focus of much later work.

The literature on combinatorial testing can be divided into two major classes, first algorithms for generating t-way adequate test sets and second, work that evaluates the technique. The latter in turn falls into two main categories: reports of the tools in field use, and a small body of experimental work.

The studies from field use have examined real systems and on the whole report on the detection of additional errors using combinational techniques. Brownlie *et al.* [7] applied the technique to testing of an email system. The effectiveness claimed for the technique is related to the saving in the number of test cases required and not on a direct comparison of faults found by applying this and any other technique. Cohen *et al.* [8] present information on the AETG tool on two releases of software where it found more faults than standard test techniques, however what the standard techniques are is not stated. Dalal *et al.* [9], [10] report briefly on the use of the AETG tool on a number of systems and the fact that more faults were discovered with its use than without. Smith et al. [11] discusses the use of the technique (2-way) on

spaceflight software and compare the number of faults found using the 2-way test sets vs. test sets constructed by other means. Here 2-way adequate test sets did not fair as well as expected.

One further set of field studies is of special interest, these looked at variable interactions leading to the activation of faults in several systems. Wallace and Kuhn [12] looked at software failure modes in data collect by the Federal Drug Administration (FDA) involving the recall of medical equipment. Kuhn and Reilly [13] examined the Mozilla and Apache open source projects using their bug tracking databases to determine the number of conditions required to trigger the fault. Finally Kuhn et al. [14] looked at a large distributed system being developed at NASA. This work suggests that in practice small t factor of between four and six was required to reveal all faults reported.

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]and [19] which addressed effectiveness at detecting seeded faults.

Dunietz et al. [15] compared the code coverage of random designs without replacement vs. the coverage obtained from systematic designs (i.e. t-way adequate test sets) with the same number of vectors. They concluded that for block coverage low factor t-way designs could be effective.

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

Kobayashi *et al*. [17] examined the fault detecting ability of specification based, random, anti-random [20] and t-way techniques applied to the testing logic predicates against mutations of those predicates. The authors concluded that 4-way tests were nearly as effective as specification techniques and better than both random and anti-random.

Grindal et al. [19] examined the fault detecting power of a number of different combinatorial strategies including 1-way (each choice), base choice (a single factor experiment), pairwise AETG and orthogonal arrays. Work was performed on code with hand seeded faults and data reported for branch coverage is consistent with other experimental results. However after examining the data the authors concluded that code coverage methods may also need to be employed. As in [11] it was found that the base choice technique performed as well as orthogonal arrays and 2-way in 3 out of 5 problems. However no technique detected fewer than 90% of the detectable faults.

Schroeder et al. [18] examined effectiveness in terms of code coverage for t-way *vs*. random selection with replacement on code with hand seeded faults. While this produced results that broadly support the results from other experimental work, it was found that higher values of t were required to reveal some faults. They also concluded that t-way test sets were no more effective that test sets constructed random selection for sets of the same size.

Our conclusion is that the literature indicates that there is no overwhelming consensus as to the utility of combinatorial techniques.

2.2 Code Mutation

Much of the empirical work that evaluates the effectiveness of the *t*-way testing technique is constrained by two main limitations. First the reliance on hand seeded faults. Second by 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 Hamlet [21] and DeMillo *et al*. [22] has been used previously in studies to compare test effectiveness [23], [24], [25]. It also has the advantage that it subsumes conditional coverage techniques [26].

Mutation has recently been applied to evaluating random testing with C programs [27] with the aim of determining whether faults inserted using the mutation are representative of real faults. The conclusion is that they are but that they are also possibly more difficult to detect.

3 The Experimental Study

3.1 The Data Set

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 (Wallace). The system was developed in a manner consistent with IEC 61508 [28] and code has been subjected to review, unit, integration and system testing.

Hand generated unit test sets were developed using standard techniques such as boundary value analysis and equivalency partitioning. They also took into account the structure of conditional statements and attempt to ensure that all clauses are tested for both TRUE and FALSE. All sets of test vectors are statement, branch coverage adequate and most are also LCSAJ adequate ([29]). Therefore we have some confidence that the hand generated set of test vectors are of high quality. A full description of how the unit test process is given in [30].

3.2 Procedure Employed

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

- A simple mutation tool was developed that produced operator, variable name, constant and statement removal mutations [31].
- A set of functions from Wallace was selected with a range of complexities from 12 to 62 executable statements (i.e. excluding comments, blank lines and braces).
- The hand-generated vectors for each function was extracted along with input domain information from the detailed designs and data dictionary.
- Each mutant was run on each vector for the automatically and hand generated test sets. For each complete set of mutants vs. test set executed the number mutants left alive was recorded.

In previous work [31] we employed our AETG based tool, however the tool is inherently inefficient as it performs a liner search to match *t*-way tuples generated in candidate vectors with tuples remaining to be covered. Lei et al. [32] reference the

`jenney` tool [33] and compared the performance of their tool `FireEye` against other available tool sets. In terms of execution time `jenny` is far more efficient that our own tool and replaces it in this work.

3.3 Code Selected

Table 1 summarizes the functions examined. Three were selected on the criteria that they contained known errors discovered running the unit tests (vs. designing tests). The remainder were selected based on their complexity, e.g. _gov_gen_ffd_rpm was selected as it contains a large number of conditional statements (eleven).

Table 1. Summary of properties of code used in this study

Function Name	Lines	Valid Mutants	Nesting Factor	Cond'n Factor	if's	Inputs
_dip_debounce	12	81	2	2	2	17
_aip_median_filter	25	217	1	1	4	3
_sdc_fuel_control	17	213	2	2	5	9
aip_spike_filter	22	178	3	1	4	7
_thc_decide_state	16	386	7	2	7	9
_thc_autocal	33	669	5	2	8	6
_aip_apply_filters	30	311	2	2	4	8
_gov_rpm_err	22	783	2	1	5	9
_sdc_pre_start	51	1297	3	1	8	3
_gov_gen_ffd_rpm	62	1227	4	2	11	16

Properties for each of the functions are shown in Table 2 as follows, the first column is the function name and the second is the number of executable statements in the function. Column three gives the number of valid mutants that would actually compile (ignoring warnings for divide by zero etc). The fourth and fifth columns are the nesting factor (maximum depth of nesting) and the condition factor (maximum number of comparisons in a predicate) as used in [34]. The sixth column is a count of the number of `if` statements in the code with each `case` of a switch statement being counted as one. 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 those arrays.

3.4 Experiment 1

3.4.1 Aims

The aims of this experiment are two fold. First to evaluate the effectiveness of *t*-way adequate test sets relative to a set of high quality human generated tests. Second to compare them with other automatic generation techniques that require a comparable level of analysis to allow data to be generated.

3.4.2 Procedure

The Procedure Employed in this Experiment Consisted of the Following Steps for Each Function:

- Generate a *t*-way adequate test set sets for $t = 2$ to $t = 5$. For numeric variables the minimum, median and maximum values in the range were used. For enumeration variables we used all valid values and one out of range value to exercise the default statement in the code. For Boolean variables TRUE and FALSE were used.
- Generate a test set of the same size as the *t*-way test sets using random selection from the same set of values with replacement using the same values as for *t*-way.
- Generate a test set of the same size purely random tests . Numeric values were drawn from the whole range with equal probability and replacement. Enumerations and Boolean values were selected as above. The generator described in [35] was used to ensure long sequences.
- For each function one or more sets of "base choice" [36] test vectors were generated. Base choice is where a base vector is selected, perhaps based on expected or normal use. Additional vectors are generated from this base by changing a single value of one variable in each new vector until all values have been used for all variables.
- For each function, execute each of the valid mutants on each test vector and for each test set recorded the number of mutants that were killed.

3.4.3 Results

Are shown in Table 2. The first column, gives the function name and the second states the information given in the next four rows as follows. For each function the first row (vectors) is the number of test vectors in the set determined by the size of *t*-way test vectors. The second row (t-way) is the number of mutants killed by *t*-way vectors for $t = 2$ to 5.

The final two columns (base, hand) gives the number of vectors in the base choice and hand generated test sets with the number of mutants left alive below it. For each function the smallest test set that had the best performance is highlighted in bold.

Table 3 gives indicative information on the amount of time in seconds that it takes to run each set of *t*-way adequate test sets data for each function.

The primary concern is which of the techniques is best at killing mutants in the selected functions. One approach is to look at which technique kills the most mutants for each function. The results are summaries as follows;

- *t*-way test vectors win or draw in six of the ten cases.
- Test vectors generated via random selection win or draw in half the cases.
- Random data generation wins or draws in four of the ten cases but notably only has a single win in the second half of the table.

The selection of "a winner" here is arbitrary in that it is the test set that killed the most mutants regardless of the number of vectors required and for some code only small numbers of vectors are required. Another way to approach is to examine the number of cases where a method failed to achieve a result comparable with the hand generated tests. Here there is one failure for *t*-way and random selection plus a near miss (_sdc_fuel_control by one) and four failures for random testing.

Table 2. Number of mutants killed for each of the sets of test vectors applied

Function Name	Process	2-way	3-way	4-way	5-way	Base	Hand
_dip_debounce	vectors	19	60	205	634	25	18
	t-way	**9**	9	9	9	28	12
	rand sel	14	9	9	9		
	random	11	10	10	9		
_aip_median_filter	vectors	12	28	54		7	27
	t-way	49	40	40		56	41
	rand sel	46	43	40			
	random	**40**	40	40			
_sdc_fuel_control	vectors	17	57	174	504	17	15
	t-way	101	49	25	22	36	21
	rand sel	126	31	24	22		
	random	84	58	25	**18**		
aip_spike_filter	vectors	16	49	146	400	14	40
	t-way	42	23	23	23	80	18
	rand sel	66	37	32	23		
	random	82	82	66	**16**		
_thc_decide_state	vectors	73	271	972	2883	28	17
	t-way	228	206	100	**57**	313	60
	rand sel	182	146	63	**57**		
	random	348	346	307	232		
_thc_autocal	vectors	20	70	181	377	14	6
	t-way	333	188	**187**	187	270	197
	rand sel	407	299	264	189		
	random	410	335	299	221		
_aip_apply_filters	vectors	34	142	562	1949	23	68
	t-way	47	46	46	46	64	64
	rand sel	**46**	46	46	46		
	random	**46**	46	46	46		
_gov_rpm_err	vectors	17	62	208	662	17	17
	t-way	**443**	443	443	443	444	446
	rand sel	**443**	443	443	443		
	random	465	462	462	460		
_sdc_pre_start	vectors	22	79	228	573	13	14
	t-way	736	**673**	673	673	965	675
	rand sel	700	**673**	673	673		
	random	742	742	742	742		
_gov_gen_ffd_rpm	vectors	21	81	299	1040	29	14
	t-way	701	190	158	**140**	785	152
	rand sel	663	270	148	**140**		
	random	502	265	152	152		

We can also calculate the mean number of vectors required to kill each mutant. Here the number of vectors required achieve the best result is used and we find that *t*-way requires 2.62 vectors per mutant, random selection 2.71 and random 3.70.

In no case was base choice the best performing technique and in only two cases was its performance comparable with the hand generated tests. These results were surprising given that previous work as [11], [19] found the technique to perform rather better.

Table 3. Execution times for the *t*-way adiquate test sets

Function Name	Valid Mutants	2-way	3-way	4-way	5-way	Max (hours)
_dip_debounce	81	76	210	743	1649	0.46
_aip_median_filter	217	64	127	248		0.07
_sdc_fuel_control	213	132	362	808	3667	1.02
aip_spike_filter	178	109	433	858	1665	0.46
_thc_decide_state	311	707	2723	8156	43451	12.07
_thc_autocal	386	139	582	2313	4253	1.18
_aip_apply_filters	669	198	420	675	2788	0.77
_gov_rpm_err	783	212	851	3239	8563	2.34
_sdc_pre_start	1237	906	1506	5083	16,231	4.51
_gov_gen_ffd_rpm	1227	972	2612	17,758	33,653	9.35

3.5 Experiment 2

3.5.1 Aims

There are two obvious issues with the data presented above. First that the execution times are long for some functions compared with the time it takes to generate the tests by hand. Timesheet data gives an average of 5.6 hours for AIP functions, 5.7 hours for DIP and 1.9 hours for SDC function. Second, the number of vectors that would have to be examined to determine if a test passed or failed is infeasiblely large. In practice a large part of the problem with generating tests by hand is determining whether the output is correct. Given the volume of tests generated automatically, determining whether the code passes or fails places an unacceptable burden on the tester and significantly reduces the utility of any automatic generation technique.

Therefore this experiment has two aims. First to investigate the potential of reducing the amount of time required to exercise all the mutants. Second to determine if a minimal test set can be extracted from the process to reduce the oracle problem to a manageable level.

3.5.2 Procedure

For this experiment we modified the test driver to record which vectors killed which mutants for each set of test vectors. After all vectors had been run over all mutants the optimisation routine determines which vector killed the most mutants and it is selected to be retained, mutants it killed are removed from further consideration. This is repeated until there are no new vectors that kill more than one mutant left**.**

The run with the next set of vectors excludes from consideration those mutants that were previously killed by all preceding test sets but otherwise the optimisation process is identical. This continues until the final set of vectors is run when the restriction on not selecting vectors that only kill a single vector is removed.

Other procedures have been made to reduce the number of vectors that need to be considered. A suggestion by Offutt [37] was to simply ignore vectors that do not kill any mutants. However these experiments suggest that savings may not be great as large number of vectors kill at least one mutant which is why we delay selecting these until the final pass. Offutt et al. [38] suggest mechanism for selecting minimal sets of vectors that again removes mutants as they are killed but runs the set of vectors in different orders.

3.5.3 Results

From experiment 2 are shown in Table4 which for each function reports the time to run the largest t-way test set (max), the time using the optimisation procedure outlined above (min) and the percentage time saving for the optimisation (gain). Information on vectors given is the number of hand generated vectors (hand), the size largest single t-way adequate test set (max) and the size of the optimised test set (min). For reference the t value of the test set that first resulted in the maximum number of mutants killed is shown in the second column headed t.

Table 4 shows that in terms of time saved the optimisation procedure can deliver significant saving for possibly the majority of functions, with an average saving of close to 53%. However it is also clear that for functions that show no increase in mutants killed at higher values of *t* the process can be counter productive e.g. _dip_debounce but that it is not always the case e.g. _aip_apply_filters. The benefits where high *t* values do show improvement are more supportive of the idea that the optimisation scheme trialled here is worth while.

Table 4. Summary data for *t*-way optimisation runs

Function Name	*t*	Time (seconds)			Vectors		
		max	min	gain	hand	max	min
_dip_debounce	2	1649	2029	123 %	18	634	6
_aip_median_filter	3	248	67	27 %	27	54	9
_sdc_fuel_control	5	3667	1144	31 %	15	504	12
aip_spike_filter	2	1665	628	37 %	40	400	9
_thc_decide_state	5	43451	6942	16 %	17	2883	13
_thc_autocal	4	4253	1276	30 %	6	377	13
_aip_apply_filters	2	2788	2029	73 %	68	1949	7
_gov_rpm_err	2	8563	6118	71 %	17	662	4
_sdc_pre_start	2	16231	18212	112 %	14	573	12
_gov_gen_ffd_rpm	5	33653	5767	17 %	14	1040	22

Results for the size of the test sets from the optimisation routine are less ambiguous, in eight of the ten cases the test sets are smaller than the hand generated test sets. In the remaining two cases they are not significantly larger in terms of total tests required.

There is however one down side, as reported in [11] vectors that were selected by the optimisation procedure were not very user friendly. That is, it takes a significant effort to understand what is being tested. Here none of the test cases contained tests that would be obvious to an engineer producing the test cases by hand (the first author

was the engineer in charge of Wallace) and many, especially those for the function _aip_apply_filters contained data that in practice would not be used and would be disallowed by the tool that vets the engine control unit calibration data.

3.6 Investigations

There are a small number of interesting features present in Table 2 as follows;

- why is it so difficult to obtain a good kill rate for the _sdc_pre_start function?
- is the fault detecting ability of random testing really static for _sdc_pre_start?
- can we improve on the results for _gov_gen_ffd_rpm if we use more random tests?

Examination of live mutants _sdc_pre_start code revels the fact that the majority of live mutants are connected with manipulating variables that have Boolean values. As has been noted in other work [31] and in a large amount of research on searched based test data generation [34], [39]. Boolean data appears to be intrinsically difficult to deal with.

The _sdc_pre_start code was executed with a number of different randomly generated test sets using different seed for 288, 573, 1200 and 2400 values. While some of the vector sets showed some improvement the best result returned was only 717 killed mutants and all data sets showed the same flat pattern as shown in Table 2.

Code for _gov_gen_ffd_rpm was run with a test set of 2000 and 5000 vectors taking 12 and 32.4 hours to execute. The test set of size 2000 showed no improvement while the test set of 5000 vectors killed only an additional 2 mutants.

4 Threats to Validity

Threats to external validity are that code being tested may not be representative of other code though a variant of aip_median_filter has been used by other researchers [40, 41] and the function itself in [42]. This however is a general problem in testing research and code from different application domains is likely to have different. The code used here is thought to be representative of fixed point integer code real-time embedded applications domain.

A novel threat is that as the code development process was strongly controlled that code actually may be easier to detect faults in than more typical code. The implication is that the results presented here are possibly optimistic. The only approach is to use other data sets, however often these do not have the necessary hand generated test vectors available. Another threat is that code mutation may not be representative of real faults. Results in [27] strongly suggest that test sets that are adequate for mutation will also be effective for real faults.

The major threat to internal validity comes from the way that the data points were used in the *t*-way and random selection data sets, being limited to minimum, median and maximum values. This is simplistic however it should tend to bias the results against success, resulting in a false negative. However the data selection process does follows examples in books such as [43] which will possibly provide the primary source of information on combinatorial techniques for practitioners.

The tool used to insert faults into the code may also presents a risk, while it avoids the bias associated with hand seeded faults it is a relatively simple tool and is not

capable of introducing mutants over multiple lines. Analysis by one of the authors [44] however suggests that the majority of effective operators have been implemented.

5 Conclusions

The results of these experiments have been surprising. At the start of this study we all thought that 2-way techniques offered a valid way of testing critical software. However our results show that:

- 2-way (pairwise) combinatorial techniques using simple selection criteria for selection data points are not adequate with respect to hand generated tests for the more complex functions as measured by mutants generated, nesting level and number of condition statements.
- Test sets that involve higher values of *t*-way adequate tests appear to be as effective as hand generated tests at killing mutants. However this statement holds only relative to being able to distinguish mutated from original code. We have not assessed the relationship with "real" faults. However as noted above results from [27] suggest that a test set for one will be effective on the other.
- Random testing can be surprisingly effective but is not reliable in the sense that it may often provide good results, but this cannot be counted on.

6 Future Work

There are some obvious avenues of work that the authors are either currently pursuing or intend to pursue in the near term. Firstly some initial work has been done using a small number of hand generated vectors as the first step in the optimisation process. This has been done by drawing small random samples from the existing hand generated tests. Initial results suggest that while the number of mutants killed is only minimally affected there may be further savings to be made in execution time.

The Wallace code base contains functions with higher level of complexity than those involved in this study. However these have as inputs large arrays of one or more dimensions and it is not clear how to effectively deal with these data structures. Does one treat them as a collection of individual variables or as a complete unit?

As noted above the data selection model used is possibly too simplistic. Previous work [45] shows that there can be an advantage in using more complete data models. This work should be repeated with higher *t*-way test sets.

The unit tests for the Boar project reported in [30] have been extracted from the project archive and these may provide an interesting comparison. The unit testing for this project was outsourced and it is known that there is a significant difference between Wallace and Boar in what activity in the unit test process (test design vs. test run) errors were revealed.

One area of interest is the effect that the mutant comparison function has on the ability to detect faults. The current comparison functions are derived directly from the hand generated tests and compares not only the output values but in most cases the majority of other input data to check for invalid modification. It would be interesting to determine what effect changing these functions has on the ability of vectors to kill mutants.

Acknowledgments

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

References

1. Duran, J., Ntafos, S.: An Evaluation of Random Testing. IEEE Trans. Softw. Eng. 10(4), 438–444 (1984)
2. Gallagher, M.J., Narasimhan, V.L.: ADTEST: A Test Data Generation Suite for Ada Software Systems. IEEE Trans. Softw. Eng. 23(8), 473–484 (1997)
3. Cohen, D.M., et al.: The AETG System: An Approach to Testing Based on Combinatorial Design. IEEE Trans. Softw. Eng. 23(7), 437–444 (1997)
4. Diamond, W.J.: Practical Experiment Design For Engineers and Scientists. John Wiley & Sons, New York (2001)
5. Mandl, R.: Orthogonal Latin Squares: an Application of Experiment Design to Compiler Testing. Commun. ACM 28(10), 1054–1058 (1985)
6. Sherwood, G.: Effective Testing of Factor Combinations. In: Third Int'l Conf. Software Testing, Analysis and Review, Software Quality Eng. pp. 151–166 (1994)
7. Brownlie, R., Prowse, J., Phadke, M.S.: Robust Testing of AT&T PMX/StarMAIL Using Oats. AT&T Technical Journal 71(3), 41–47 (1992)
8. Cohen, D.M., et al.: The Automatic Efficient Test Generator (AETG) System. In: Proceedings 5th International Symposium on Software Reliability Engineering, pp. 303–309. IEEE Computer Society, Los Alamitos (1994)
9. Dalal, S., et al.: Model-based Testing of a Highly Programmable System. In: Proc. of the Ninth International Symposium on Software Reliability Engineering. IEEE Computer Society, Los Alamitos (1998)
10. Dalal, S.R., et al.: Model-based Testing in Practice. In: Proc. of the 21st Int'l Conf. on Software Engineering, pp. 285–294. IEEE Computer Society, Los Alamitos (1999)
11. Smith, B., Feather, M.S., Muscettola, N.: Challenges and Methods in Testing the Remote Agent Planner. In: Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems, pp. 254–263. AAAI Press, Menlo Park (2000)
12. Wallace, D.R., Kuhn, D.R.: Failure Modes in medical device software: an analysis of 15 years of recall data. International Journal of Reliability, Quality and Safety Engineering 8(4), 351–371 (2001)
13. Kuhn, D.R., Reilly, M.J.: 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 2002). IEEE Computer Society, Los Alamitos (2002)
14. Kuhn, D.R., Wallace, D.R., Gallo, A.M.: Software Fault Interactions and Implications for Software Testing. IEEE Trans. Softw. Eng. 30(6), 418–421 (2004)
15. Dunietz, I.S., et al.: Applying Design of Experiments to Software Testing: Experience Report. In: Proc.of the 19th Int'l Conf. on Software Eng., pp. 205–215. ACM Press, New York (1997)
16. Nair, V.N., et al.: A Statistical Assessment of some Software Testing Strategies and Application of Experimental Design Techniques. Statistica Sinica 8, 165–184 (1998)
17. Kobayashi, N., Tsuchiya, T., Kikuno, T.: Non-Specification-Based Approaches to Logic Testing for Software. Information and Software Technology 44(2), 113–121 (2002)

18. Schroeder, P.J., Bolaki, P., Gopu, V.: Comparing the Fault Detection Effectiveness of N-way and Random Test Suites. In: ISESE 2004: Proceedings of the 2004 International Symposium on Empirical Software Engineering, pp. 49–59. IEEE Computer Society, Los Alamitos (2004)
19. Grindal, M., et al.: An Evaluation of Combination Strategies for Test Case Selection, in Technical Report, Department of Computer Science, University of Skövde (2003)
20. Malaiya, Y.K.: Antirandom testing: getting the most out of black-box testing. In: Proceedings, Sixth International Symposium on Software Reliability Engineering, pp. 86–95 (1995)
21. Hamlet, R.G.: Testing Programs with the Aid of a Compiler. IEEE Trans. Softw. Eng. 3(4), 279–290 (1977)
22. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on Test Data Selection: Help for the Practising Programmer. Computer, 34–41 (1978)
23. Daran, M., Thevenod-Fosse, P.: Software Error Analysis: a Real Case Study Involving Real Faults and Mutations. SIGSOFT Softw. Eng. Notes 21(3), 158–171 (1996)
24. Frankl, P.G., Weiss, S.N., and Hu, C.: All-uses vs. mutation testing: an experimental comparison of effectiveness. J. Syst. Softw. 38(3), 235–253 (1997)
25. Zhan, Y., Clark, J.A.: Search-Based Mutation Testing for Simulink Models. In: Proc. of the 2005 Conference on Genetic and Evolutionary Computation, pp. 1061–1068. ACM Press, New York (2005)
26. Offutt, A.J., Voas, J.M.: Subsumption of Condition Coverage Techniques by Mutation Testing, in Tech. Report, Dept. of Information and Software Systems Engineering, George Mason Univ., Fairfax, Va (1996)
27. Andrews, J.H., Briand, L.C., Labiche, Y.: Is Mutation an Appropriate Tool for Test Experiments? In: Proc. of the 27th Int'l Conf. on Software Engineering, pp. 402–411. ACM Press, New York (2005)
28. Anon.: Functional Safety of Electrical/Electronic/Programmable electronic safety-related systems, Part 1: General Requirements, BS EN 61508-1:2002, British Standards (2002)
29. Woodward, M.R., Hedley, D., Hennel, M.A.: Experience with Path Analysis and Testing of Programs. IEEE Trans. Softw. Eng. 6(6), 228–278 (1980)
30. Ellims, M., Bridges, J., Ince, D.C.: The Economics of Unit Testing. Empirical Softw. Eng. 11(1), 5–31 (2006)
31. Ellims, M., Ince, D., Petre, M.: The Csaw C Mutation Tool: Initial Results. In: Mutation 2007. IEEE Computer Society, Los Alamitos (2007)
32. Lei, Y., et al.: IPOG: A General Strategy for T-Way Software Testing. In: 14th Annual IEEE Int'l Conf. and Workshops on the Engineering of Computer-Based Systems (ECBS 2007), pp. 549–556. IEEE Computer Society, Los Alamitos (2007)
33. Jenny (accessed June 2007), http://www.burtleburtle.net/bob/math
34. Michael, C.C., McGraw, G., Schatz, M.A.: Generating Software Test Data by Evolution. IEEE Trans. Softw. Eng. 27(12), 1085–1110 (2001)
35. Wichmann, B.A., Hill, I.D.: Generating Good Pseudo-Random Numbers. Computational Statistics & Data Analysis 51(3), 1614–1622 (2006)
36. Ammann, P.E., Offutt, J.: Using Formal Methods to Derive Test Frames in Category-Partition Testing. In: Proc. of 9th Annual Conf. on Computer Assurance (COMPASS 1994), pp. 824–830. IEEE Computer Society, Los Alamitos (1994)
37. Offutt, A.J.: A Practical System for Mutation Testing: Help for the Common Programmer. In: Proc. of the IEEE Int'l Test Conference on TEST: The Next 25 Years, pp. 824–830. IEEE Computer Society, Los Alamitos (1994)

38. Offutt, J.A., Pan, J., Voas, J.M.: Procedures for Reducing the Size of Coverage Based Test Sets. In: Twelfth Int. Conf. on Testing Computer Software, pp. 111–123 (1995)
39. Bottaci, L.: Instrumenting Programs with Flag Variables for Test Data Search by Genetic Algorithms. In: Proc. of the Genetic and Evolutionary Computation Conference. Morgan Kaufmann Publishers, San Francisco (2002)
40. Gotlieb, A.: Exploiting Symmetries to Test Programs. In: Proceedings of the 14th International Symposium on Software Reliability Engineering, p. 365. IEEE Computer Society, Los Alamitos (2003)
41. Offutt, A.J., et al.: An Experimental Determination of Sufficient Mutant Operators. ACM Trans. Softw. Eng. Methodol. 5(2), 99–118 (1996)
42. Dillon, E., Meudec, C.: Automatic Test Data Generation from Embedded C Code. In: Heisel, M., Liggesmeyer, P., Wittmann, S. (eds.) SAFECOMP 2004. LNCS, vol. 3219, pp. 180–194. Springer, Heidelberg (2004)
43. Copeland, L.: A Practitioner's Guide to Software Test Design. Artech House Publishers, Boston (2004)
44. Ellims, M.: The Csaw Mutation Tool Users Guide, in Technical Report, Department of Computer Science, Open University (2007)
45. Ellims, M., Ince, D., Petre, M.: AETG vs. Man: an Assessment of the Effectiveness of Combinatorial Test Data Generation, in Technical Report, Department of Computer Science, Open University (2007)