Reprinted from Proceedings, 27th NASA/IEEE Software Engineering Workshop, NASA Goddard Space Flight Center, 4-6 December, 2002.

An Investigation of the Applicability of Design of Experiments to Software Testing

D. Richard Kuhn Michael J. Reilly
National Institute of Standards and Technology
Gaithersburg, MD 20899
kuhn@nist.gov michael.reilly@nist.gov

Abstract

Approaches to software testing based on methods from the field of design of experiments have been advocated as a means of providing high coverage at relatively low cost. Tools to generate all pairs, or higher n-degree combinations, of input values have been developed and demonstrated in a few applications, but little empirical evidence is available to aid developers in evaluating the effectiveness of these tools for particular problems. In this paper we investigate error reports from two large open-source software projects, a browser and web server, to provide preliminary answers to three questions: Is there a point of diminishing returns at which generating all n-degree combinations is nearly as effective as all n+1-degree combinations? What is the appropriate value of n for particular classes of software? Does this value differ for different types of software, and by how much? Our findings suggest that more than 95% of errors in the software studied would be detected by test cases that cover all 4-way combinations of values, and that the browser and server software were similar in the percentage of errors detectable by combinations of degree 2 through 6.

1. Introduction

Methods from the field of design of experiments (DOE) have been applied to quality control problems in many engineering fields for several decades. DOE seeks to maximize the amount of information gained in an experiment by optimizing the combinations of independent variables. Software testing using DOE methods, often referred to as combinatorial testing methods, has been advocated as an efficient means of providing a high level of coverage of the input domain with a small number of tests [1] [2] [3] [4] [5] [6]. 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 many development budgets would cover less than a fraction of a percent ($< 10^{-15}$) of the possible cases. But the number of *pairs* of settings is in fact small by comparison, and since every test case must have a value for each of the ten variables, many pairs can be included in a single test case. Algorithms based on orthogonal arrays are available that can generate test data for all 2-way (or higher order *n*-way) combinations at a reasonable cost. One such method makes it possible to test all pairs of values for this example using only 180 cases [7]. This level of test effort would be practical for many small software-controlled devices, or critical components of larger systems. In general, for k parameters with v values each, the number of test cases is proportional to $(v/2)\log_n k$ (for small n).

Combinatorial testing methods have an industrial appeal in their potential to reduce test costs, but there is also a significant productivity advantage to applying these methods in testing high integrity software. If we were able to know with certainty that all faults in a system can be triggered by a combination of n or fewer parameters, then testing all n-way or fewer interactions is effectively equivalent to exhaustive testing for variables with a small set of discrete values (possibly using equivalence classes for continuous value variables). In reality, of course, we can never know in advance what degree of interaction is required to trigger all faults in a system. A practical

alternative, however, may be to collect empirical data on faults that occur in real systems in various application domains. For example, if long term failure data show that a particular type of application has never required the interaction of more than 5 parameters to reveal a failure, then an appropriate testing goal for that class of applications is to test all 5-way or fewer interactions.

Proponents of these methods have reported that little empirical work exists to support their use [8]. Many applications of combinatorial testing methods have focused on *configuration testing* [9]. For example, a client-server information system may include five types of client operating systems, three browsing programs, and five types of server operating systems. Rather than test all 75 configurations of the variables - client operating system, browser, server operating system - a smaller number of tests can be used to consider all pairs of variables. A less common application of combinatorial methods is in selection of input data. The earliest such example is probably that of Mandl [10], who used orthogonal arrays to select data types in testing Ada compilers. Since then, combinatorial methods have been used in a number of other applications, and tools have been developed to simplify their use in test data selection. Dalal et al. [11] demonstrated the effectiveness of pair-wise testing in four case studies, but did not investigate higher-degree interactions. The Remote Agent Experiment (RAX) software on NASA's Deep Space 1 mission [12] is another example of applied combinatorial methods. The RAX is an expert system that generates plans to carry out spacecraft operations without human intervention. This work found that testing all 2-degree pairs of input values and all individual values detected 88% of bugs classified as correctness and convergence flaws (i.e. successfully finding a feasible path), but detected only 50% of engine interface bugs. The NASA study did not investigate higher-degree interactions required to trigger a failure. Another study [13] reviewed 15 years of medical device recall data from the US Food and Drug Administration to characterize the types of faults that occur in this application domain. Only 109 of the 342 recalls of software controlled devices contained enough information to determine how many conditions were required to replicate a failure. Of these 109 cases, 98% of the reported flaws could be detected by testing all pairs of parameter settings, and only three of the recalls indicated that more than two conditions were required to cause a failure. The most complex of these failures required four conditions. A serious limitation of this study was the limited data set.

It is noted in Smith, Feather, and Muscetolla [12] that pairwise testing detected only 20% more errors than all-values testing. Although they did not test beyond pairwise combinations, the authors proposed the reasonable hypothesis that a point of diminishing returns is reached for some small value of n, so that an effective test strategy is to test n-way combinations of parameter values, with additional tests for selected higher order combinations. Some important questions in this regard are:

1. Is there in fact such a point of diminishing returns?
2. What is the appropriate value of n for particular classes of software?
3. Does this value differ for different types of software, and by how much?
4. Does the value increase as software moves from development to stable use?
5. Does it continue to increase with version upgrades?

In this paper we report on work that begins to answer the first three of these questions.

2. Procedures

We characterized faults in two large open source software projects by the number of conditions required to trigger the fault. That is, what percentages of known faults were triggered by a single condition, an interaction between two conditions, three conditions, and so on? The Mozilla web browser and Apache web server projects provide publicly accessible databases of bugs for use by developers. Each bug is classified according to characteristics such as severity, priority for repair, and current state (e.g. fixed or pending). A description of each bug is given with instructions on how to replicate the bug when available. We reviewed a total of 194 bug reports in the browser database (http://bugzilla.mozilla.org) - all entries classified as "Verified, Fixed, Critical." Using the descriptions in the database, bugs were categorized by the number of conditions required to trigger the associated fault. For example, bug 106763 has a description which states "Subscribe window is blank until you enter a search term," has one condition: the subscribe window should be opened. A corresponding procedure was used to collect data on 171 bugs from the server bug database, although the database used was "Old Apache Bug Database," which has a slightly different classification scheme than Bugzilla. (The new version of the Apache database is the same as that for the browser, but does not contain a sufficient number of bug reports for review.) The server bug database

organizes bugs according to the module in which they occur, e.g., access control, CGI processing, cookie handling.

3. Findings and Discussion

For the two software projects analyzed in this paper, some conclusions can be suggested from the results shown in Table 1, although more software projects must be analyzed to provide a reasonable level of confidence

Conditions (values of *n*)	**Browser (194 bugs)**		**Server Modules (171 bugs)**	
	(percent)	**(cumulative percent)**	**(percent)**	**(cumulative percent)**
1	28.6	28.6	41.7	41.7
2	47.5	76.1	28.6	70.3
3	18.9	95.0	19.0	89.3
4	2.2	97.2	7.1	96.4
5	2.2	99.4	0.0	96.4
6	0.6	100.0	3.6	100.0

Table 1. Number and Percent of Faults Triggered by *n*-way Conditions

- For these projects, there was in fact a point of diminishing returns reached at a small number of conditions. Testing all 3-way or lower degree combinations would detect approximately 90% of the reported bugs, and all 6-way and lower degree combinations would detect all faults reported in the bug databases.
- The review conducted for this paper was not sufficient to determine whether higher degree combinations are required to detect faults as the software was upgraded in later releases.
- Both databases contained bug data that were unclassifiable in terms of 'Number of Conditions.' Some bugs were either not adequately described to give a determinate number of conditions, while others were not traditional bugs in the sense that they were caused by incorrect use of the product by a user (e.g. improper configuration).

Before returning to the questions posed in the Introduction, there are a number of caveats and sources of error to be considered before conclusions are drawn.

- The bug reports indicate conditions required to trigger faults, but do not describe the level of testing conducted. It is possible that the easier bugs – requiring fewer conditions to detect – were being found.

- Conditions needed to replicate failures for the server modules were frequently reported as a list of configuration settings. It may be that many of the settings were "don't care" conditions, and that only one or two of the conditions were essential to triggering the fault. In this case, the number of conditions required to detect a fault would be artificially increased.

- A third possible source of error is attributed to the analysts who reported the bugs. The methods for counting the number of conditions required to trigger a bug in each database are not necessarily the same. Since the bug database for the web browser describes specific lines of code and functions that are invoked to trigger the bug, the number of ways that these functions are activated is not addressed. For example there could be two separate ways of calling a specific function that triggers a bug. Also, it is not clear how many conditions are set or activated by the code in question. If a function that seems to trigger a bug, which would be classified as one condition, is responsible for setting a number of conditions that are, at a finer level of

scrutiny, ultimately the true cause of the bug, the recorded number of conditions for that bug would be less than the actual number.

- Viewing the bugs in the server database, we also find a number of bugs that are described only in terms of a specific web page that, when viewed, causes the fault. These page-specific bugs cannot be classified with as much certainty as bugs that are described directly in terms of the conditions required to cause them, because their descriptions do not allow insight into what conditions are required on the coding level to recreate the fault (e.g. how to make a separate web page that would cause the same fault to occur).

Returning to the questions from the Introduction, some preliminary conclusions and implications for testing can be suggested:

1. For the systems reviewed, there was in fact a point of diminishing returns reached at a fairly low level of *n*-way combinations. More than 70% of bugs were detected with two or fewer conditions (75% for browser and 70% for server) and approximately 90% of the bugs reported were detected with three or fewer conditions (95% for browser and 89% for server). This result is consistent with the hypothesis proposed in [12].

2. Depending on reliability requirements, cost considerations, and other assurance methods available, the appropriate value of *n* could be $n \leq 3$ to $n \leq 6$. It is interesting that a small number of conditions ($n \leq 6$) are sufficient to detect all reported errors for the browser and server software. For the medical device software reported in [13] this value is even smaller: $n \leq 4$. Testing all combinations up to these small values of *n* would provide a form of "pseudo-exhaustive" testing, although clearly not truly exhaustive because of uncertainty as to whether remaining errors would be triggered by a higher-order combination of *n*+1 or more conditions, and the uncertainty introduced by using equivalence classes rather than all values for some variables.

3. There is some degree of variation among the different types of software discussed in this paper for the level of *n* required to detect a high percentage of bugs. It is somewhat surprising that a higher degree of combinations was required to detect close to 100% of browser and server errors ($n \leq 4$ conditions for 97% of errors) than was required for the medical device software ($n \leq 2$ to detect 98% of errors) reviewed in [13]. The medical device bug reports were from software installed in mature, fielded products, while the browser and server bug reports came from development efforts. However, the sample size for the medical device software was smaller and bug reports were much less complete than those provided by the browser and server developers, so the $n \leq 2$ value may not be truly representative of this software. Another possibility is that the browser and server software are simply larger and more complex than the majority of the medical device applications. The percentage of bugs detected by 2-way combinations (approximately 70%) for the browser and server software falls in between the values reported by [12] for the RAX planner (88%) and engine-interface (50%) software, again possibly as a result of differences in complexity and application domain.

For combinatorial testing to be effective in providing a high level of assurance, two conditions must hold:

- For all *n*-way combinations of values that trigger faults, *n* must be relatively small, to make test case development tractable.
- The number of values to be tested can be large, but it must be possible to enumerate these conditions from specifications, and to create test cases that cover these conditions up to the required degree.

For both projects reviewed, the first condition clearly holds. Three or fewer conditions triggered about 90% of the browser and server bugs. For high quality software, simply testing all 3-way conditions is not sufficient, but other testing approaches might be as good as or better than combinatorial testing from a cost/benefit standpoint. Empirical studies comparing the cost of detecting complex faults through combinatorial testing versus other methods would be helpful to test planners. The second condition is more problematic. For the browser, almost all of the conditions reported in bug reports were unique, while for the server it was not unusual to see a particular condition, say *P*, involved in more than one bug

report. For example, one server bug might be triggered by $P \wedge Q \vee R$, another by $P \wedge S$, and another by $P \wedge T$, but it was rare to see a particular condition *P* appear in more than one browser bug report.

4. Conclusions

For the software investigated in this paper, a web browser and server, a relatively low degree of *n*-way combinations of values would detect nearly all errors in the database. Appropriate levels of *n* could be $n \leq 3$ to $n \leq 6$, according to dependability requirements, suggesting that combinatorial testing would be effective for this type of software. If experience shows that all errors in a particular class of software are triggered by combinations of *n* values or less, then testing all combinations of *n* or fewer values would provide a form of "pseudo-exhaustive" testing. Since most variables are likely to have a very large range of values, equivalence classes would need to be used in practice. Because the effectiveness of combinatorial testing depends on the fact that a single test case can include a large number of pairs (or higher degree combination) of values, this approach would not be effective for most real-time or other software that depends on testing event sequences, but it may be applicable to subsystems within real-time software. Empirical studies of other classes software would be helpful in evaluating the applicability of combinatorial testing.

References

[1] 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).

[2] K. Burroughs, A. Jain, and R.L. Erickson. Improved Quality of Protocol Testing Through Techniques of Experimental Design. In Proceedings of Supercomm/ICC '94, 1994, pp. 745-752 1994.

[3] D.M. Cohen, S.R. Dalal, J. Parelius, and G.C. Patton. The Combinatorial Approach to Automatic Test Generation. IEEE Software, 13(5): 83-88, (September 1996).

[4] I.S. Dunietz, W.K. Ehrlich, B.D. Szablak, C.L. Mallows, A. Iannino. Applying Design of Experiments to Software Testing. In Proceedings of ICSE '97, pages 205-215, Boston MA USA, (1997).

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

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

[7] 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 Transactions on Software Engineering, 23(7): 437-444, (July 1997).

[8] J.M. Harrell, "Orthogonal Array Testing Strategy Technique", http://www.cvc.uab.es/shared/teach/a21291/apunts/provaOO/OATS.pdf

[9] W.B. Perkinson. A Methodology for Designing and Executing ISDN Feature Tests Using Automated Test Systems. In Proceedings of IEEE GLOBECOMM '92, 1992.

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

[11] 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.

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

[13] 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.