Software Fault Interactions and Implications for Software Testing

D. Richard Kuhn, *Senior Member*, *IEEE*,
Dolores R. Wallace, *Member*,
IEEE Computer Society, and
Albert M. Gallo Jr.

Abstract—Exhaustive testing of computer software is intractable, but empirical studies of software failures suggest that testing can in some cases be effectively exhaustive. Data reported in this study and others show that software failures in a variety of domains were caused by combinations of relatively few conditions. These results have important implications for testing. If all faults in a system can be triggered by a combination of n or fewer parameters, then testing all n-tuples of parameters is effectively equivalent to exhaustive testing, if software behavior is not dependent on complex event sequences and variables have a small set of discrete values.

Index Terms—Statistical methods, testing strategies, test design.

1 Introduction

A software tester's task is extremely difficult. Seeking to locate the maximum number of latent errors under generally immovable deadlines is daunting, to say the least. Consider, for example, a device that has 20 inputs, each having 10 possible values (or 10 equivalence classes if the variables are continuous). This scenario yields a total of 10^{20} combinations of settings. Only a few hundred test cases can be built and executed under most budgets, yet this would cover less than a fraction of one percent ($< 10^{-15}$) of the possible cases.

Empirical research into quality and reliability, for at least some types of software, suggests that relatively few parameters are actually involved in triggering failures—a phenomenon that has significant implications for testing. This leads one to suspect: If we were able to know with certainty that all faults in a system are triggered by a combination of n or fewer parameters, then testing all n-tuples of parameters is effectively equivalent to exhaustive testing at least for variables with a small set of discrete values (or possibly using equivalence classes for continuous value variables). For variables with a continuous range of values, partition testing of all n-way combinations of equivalence classes might be considered *pseudoexhaustive*.

In reality, of course, we can never know in advance what degree of interaction is required to trigger all faults in a system. A somewhat more practical alternative, however, may be to collect empirical data on faults that occur among similar systems in various application domains. For example, if a long history of failure data shows that a particular type of application has never required the interaction of more than four parameters to reveal a failure, then an appropriate testing goal for that class of applications might be to test all 5-way or fewer interactions. We will refer to the number of conditions required to trigger a failure as the *failure-triggering fault interaction* (FTFI) number. For example, if a microwave oven control module fails when power is set on "High" and time is set to 20 minutes, the FTFI number is 2. Combinatorial testing [1], [2] that exercised all 2-tuples of test data would have detected this failure. In this paper, we analyze the fault interactions of a large distributed system, compare the results with data reported for systems in other domains, and explore the implications of these results for software testing.

- *D.R. Kuhn is with the National Institute of Standards and Technology, Gaithersburg, MD. E-mail: kuhn@nist.gov.*
- *D.R. Wallace and A.M. Gallo, Jr. are with NASA Goddard Space Flight Center, Greenbelt, MD.*
  E-mail: dwallac@pop300.gsfc.nasa.gov, al.gallo@nasa.gov.

Manuscript received 24 Sept. 2003; revised 18 Feb. 2004; accepted 15 Mar. 2004.
Recommended for acceptance by J. Offutt.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0172-1003.

2 Related Work

To our knowledge, only three studies prior to this one attempted to characterize fault interactions using empirical data. Nair et al. [3] described a case study of combinatorial testing for a small subsystem of a screen-based administrative database. The system was designed to present users with input screens, accept data, then process it and store it in a database. Size was not given, but similar systems normally range from a few hundred to a few thousand lines of code. This study was extremely limited in that only one screen of a subsystem with two known faults was involved, but pairwise testing was sufficient to detect both faults.

Wallace and Kuhn [4] reviewed 15 years of medical device recall data gathered by the US Food and Drug Administration (FDA) to characterize the types of faults that occur in this application domain. These applications include any devices under FDA authority, but are primarily small to medium sized embedded systems, and would range from roughly 10^4 to 10^5 lines of code. All of the applications in the database were fielded systems that had been recalled because of reported defects. A limitation of this study, however, was that only 109 of the 342 recalls of software-controlled devices contained enough information to determine the number of conditions required to replicate a given failure. Of these 109 cases, 97 percent of the reported flaws could be detected by testing all pairs of parameter settings, and only three of the recalls had an FTFI number greater than 2. (The number of failures triggered by a single condition was not given in [4], but we reviewed the data and report this figure in Table 1.) The most complex of these failures required four conditions. Kuhn and Reilly [5] analyzed reports in bug tracking databases for open source browser and server software, the Mozilla web browser and Apache server. Both were early releases that were undergoing incremental development. This study found that more than 70 percent of documented failures were triggered by only one or two conditions, and that no failure had an FTFI number greater than 6. Difficulty in interpreting some of the failure reports (e.g., in some cases, it was not clear whether some conditions were "don't care" or were required to reproduce the failure) led to conservative assumptions regarding failure causes. Thus, some of the failures with high FTFI numbers may actually have been less than 6.

Three other studies provided some limited information regarding fault interactions. Dalal et al. [6] demonstrated the effectiveness of pairwise testing in four case studies but did not investigate higher-degree interactions. Smith et al. [7] investigated pairwise testing of the Remote Agent Experiment (RAX) software on NASA's Deep Space 1 mission. The RAX is an expert system that generates plans to carry out spacecraft operations without human intervention. This study found that testing all pairs of input values detected over 80 percent of the bugs classified as either "correctness" or "convergence" flaws in onboard planning software (i.e., successfully finding a feasible path), but only about half of engine and interface bugs [7]. (Figures for these four components are shown separately in Table 1.) The authors did not investigate higher-degree combinations required to trigger a failure. Pan [8] found that testing all values triggered more than 80 percent of detected errors in a selection of POSIX operating system function

TABLE 1
Cumulative Percent of Faults Triggered by n-way Conditions

FTFI No.	RAX convergence	RAX correctness	RAX interf	RAX engine	POSIX modules	Medical Devices	Browser	Server	NASA GSFC
1	61	72	48	39	82	66	29	42	68
2	97	82	54	47	*	97	76	70	93
3	*	*	*	*	*	99	95	89	98
4	*	*	*	*	*	100	97	96	100
5	*	*	*	*	*		99	96	
6	*	*	*	*	*		100	100	

** = not reported.*

calls from 15 fielded, commercial systems. Higher-degree combinations were not reported.

3 Empirical Data

We analyzed 329 error reports from development and integration testing of a large distributed system being developed at NASA Goddard Space Flight Center. This application is a data management system that gathers and receives large quantities of raw scientific data. The system is comprised of numerous subsystems for scientific analysis of the data as well as the storage of all results. Multiple standalone copies of this system are deployed at several locations. Faults are initially corrected at the site where they were first discovered, and subsequently all sites receive the correction as there are new releases of the system. Regardless of the point of origin, faults are characterized in a database by date submitted, severity, priority for fix, the location where found, status, the activity being performed when found, and several other features. Several text fields provide additional context, including one to describe how the fault was found as well as one to discuss its resolution. Results of this analysis are shown in the last column of Table 1. System type, release stage, and approximate system size (or size of similar applications, where this information was not provided) are summarized in Table 2 for comparison purposes. Also, note that the distribution of failure-triggering conditions appears to follow a power law (Fig. 1, last four columns of Table 1), but many more data sets would be required to make this generalization.

The analyses discussed above raise some interesting questions. Perhaps most intriguing is the absence of any clear differences in fault interaction complexities between development projects and fielded products. Intuition suggests that bugs should be more difficult to trigger, hence occur less frequently, once a system has been developed. Some spectacular software failures seem to bear out this thought. For example, the Mars Pathfinder failed as a result of a complex series of events leading to a priority inversion, which deadlocked critical system processes [9]. This intuition has been referred to as the "Heisenbug" hypothesis, which posits that bugs in fielded systems are likely to be transient, hard to reproduce, and not consistently observable.

Yet surprisingly, this expectation does not clearly hold for the two sets of fielded products reviewed above. For all levels of fault interactions reported, the development project failures were harder to trigger than those in both classes of fielded products. In fact, bugs with an FTFI number of 2 accounted for a higher proportion of the medical device failures than for any of the development projects (ignoring the administrative database, which had too few data points to be statistically significant). Much more analysis across a variety of application domains will be needed to provide a comprehensive picture of the fault interactions of fielded systems, but these data suggest that it is not safe to assume that such failures are always due to rare combinations of conditions. We note

TABLE 2
Characteristics of Systems Reviewed

System	System type	Release Stage	Size (LOC)
Admin database	Database user interface	Development - integration test	approx. 10^3 (size of similar applications)
RAX Planner	Artificial intelligence	Development	approx. 3,000
POSIX modules	Operating system function calls	Fielded products	10^3 (varies)
Medical Devices	Embedded	Fielded products	10^3 -10^4 (varies)
Browser	Web browser	Development/ beta release	approx.2x10^6
Server	HTTP server	Development/ beta release	approx.10^5
NASA	Distributed scientific database	Development - integration test	approx.10^5

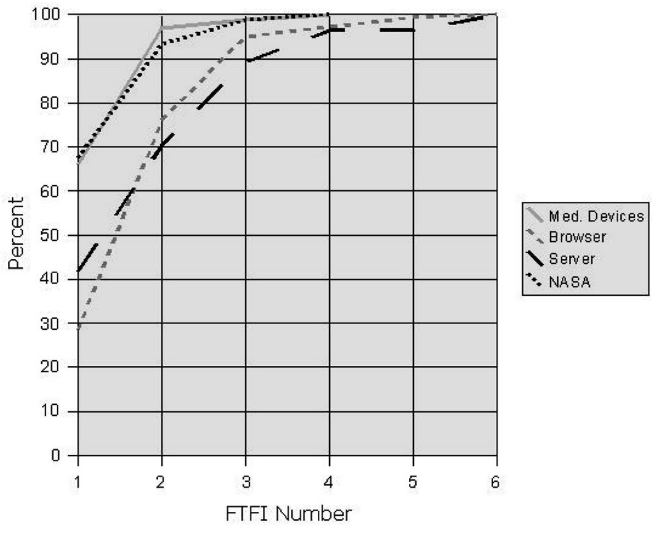

Fig. 1. Failure triggering fault interactions, cumulative distribution.

also that there are a number of famous software failures with an FTFI number of only 1 or 2. One such example, the Ariane 5 disaster [10], occurred because the horizontal velocity of the rocket exceeded that of Ariane 4. (The software-related cause of the error was a failed numerical conversion, but the operational condition required to trigger this situation was simply a horizontal velocity greater than earlier systems.) The USS Yorktown failure is another example of a spectacular failure resulting from a single fault condition. Assigning a value of zero in a particular database field caused a divide-by-zero error, which caused the local network to crash, disabling the entire ship [11].

4 Implications for Testing

Consider the previously discussed system with 20 inputs, each of which can assume 10 possible values. Exhaustive testing would, of course, require 10^{20} test cases, but the empirical results described above show that most failures were actually triggered by a single erroneous parameter; however, nearly all could be triggered by fewer than 4 or 5 and at most 6 for the software that was studied. This section gives some "back of the envelope" calculations of the number of tests required to cover all n-way combinations.

Now, consider the effort required to exercise all n-tuples of k parameters, each of which has v possible values (known in the combinatorics literature as the problem of *covering array construction* [12], [13], [14]). The number of n-tuples drawn from k parameters is $C(k,n) = \frac{k!}{n!(k-n)!}$ and, since each parameter has v values, the total number of test cases required to test exhaustively would be $C(k,n) \cdot v^n$. This calculation uses the simplifying assumption that each parameter has the same number, v, of values, but, in practice, v can be the maximum with "don't care" values for parameters with less than v values. Attempting to test all 4-tuples for the example described above would require 48,450,000 test cases. Fortunately, this prohibitively large number can be reduced to a reasonable level.

Since each test case will contain 20 parameters, there are $C(20,4) = 4,845$ 4-tuples of parameters and $C(20,6) = 38,760$ 6-tuples in each test case. If test case generation is perfectly efficient, then each test case would contain unique sets of n-tuples, i.e., ensure there are no duplicate tests. A rough best case estimate for the total number of test cases would therefore be $\frac{C(k,n)v^n}{C(k,n)} = v^n$, although avoiding all duplicates is not possible in practice [12], [13], [14], so v^n is in fact a naïve best-case estimate. The actual number of tests cases will be within a multiplicative constant factor of optimal [15], which increases proportional to the log of the number of parameters [12], [13], [14]. In practice, this naïve estimate may be multiplied by as much as two to three, so our earlier examples of 20 inputs with 10 values each, v^n may reasonably require 20,000 tests to cover all 4-tuples. Manually generating an extremely large number of test cases is hardly practical, but new automated test case generation tools [16], [17] render such a task possible. Finding efficient methods for generating n-way covering test combinations is an active research area [2], [18], [19], [20]. The results reported in this paper suggest that this work could be of significant benefit to software testers.

Real systems are, of course, rarely as simple as the example. Rather than parameters with only 10 discrete values each, most or all parameters are either continuous or have significantly larger sets of discrete input values. Therefore, this form of testing should, for most cases, be considered pseudoexhaustive, rather than effectively exhaustive. The traditional approach to dealing with the problem of continuous variables is to partition the parameter values into equivalence classes, where values in each set are assumed to be equivalent from a testing standpoint, i.e., correct (incorrect) system operation for one value is assumed to imply correct (incorrect) operation for another value from the same equivalence class. In many cases, this assumption is not unreasonable provided the input is partitioned into an appropriate set of classes.

TABLE 3
Maximum Value of v for Combinations of n-Tuples and Test Cases

n	10^2 tests	10^3 tests	10^4 tests	10^5 tests	10^6 tests
All 2-tuples	10	31	100	316	1000
All 3 tuples	4	10	21	46	100
All 4 tuples	3	5	10	17	31
All 5 tuples	2	3	6	10	15
All 6 tuples	2	3	4	6	10

When planning for needed testing resources, the first question to define is the scope of the effort. For a given number, N, of test cases, and a specified level of n-tuple, how many values, or equivalence classes, can or must be covered (keeping in mind that the actual number of test cases will be a small multiple of N)? Using v^n as the best-case approximation of the number of n-tuples covered by the set of test cases, we have $v^n \leq N$, so $n \log v \leq \log N$. So, for $N = 10^x$ tests, $v \leq 10^{\frac{x}{n}}$. Maximum values for v, in various combinations of n and number of test cases, are shown in Table 3. Thus, testing all 2-tuples of parameters using 100 tests would require that each parameter have no more than 10 values. Looked at another way, producing pairwise tests for parameters with 10 values each would require a minimum of 100 tests. One combinatorial testing tool makes it possible to test all pairs of values for this example using 180 cases [21].

Because testing occurs at the end of the development lifecycle, it must be both thorough and efficient in order to maximize effectiveness. Consider the case where deadlines are fixed and management has opted to conduct pseudoexhaustive testing. If it is believed that any fault present can be triggered by interactions of no more than five variables, the following line of reasoning is used. First, variable values are partitioned into some number of equivalence classes. If we assumed six for each variable, then a small multiple of 10,000 tests would be needed to cover all 5-tuples. Using automated test generation tools, this number of tests is feasible to generate. Practical trials of automated test tools generating this number of tests are needed to evaluate this approach.

5 Conclusions

All failures of software reviewed in this paper were triggered by low FTFI number faults-at most four to six parameters were involved. If experience shows that all errors in a particular class of software are triggered by finite combinations of n values or less, then testing all combinations of n or fewer values would provide a form of "pseudoexhaustive" testing. Since most variables actually have very large ranges of values, equivalence classes would need to be used in practice. Appropriate levels of n appear to be $4 \leq n \leq 6$ when considering "pseudoexhaustive" testing, according to dependability requirements. 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 may not be as effective for real-time or other software that depends on testing event sequences, but may be applicable to subsystems within real-time software. Many more empirical studies of other classes of software are needed to evaluate the applicability of combinatorial testing for other classes of systems.

Acknowledgments

The authors would like to thank Tim Grance for his support of this project. They want to thank Jim Lyle for his careful review and the *TSE* reviewers for many helpful suggestions.

References

[1] R. Brownlie, J. Prowse, and M.S. Phadke, "Robust Testing of AT&T PMX/StarMail Using OATS," *AT&T Technical J.,* vol. 71, no. 3, pp. 41-47, May/June 1992.

[2] D.M. Cohen, S.R. Dalal, J. Parelius, and G.C. Patton, "The Combinatorial Approach to Automatic Test Generation," *IEEE Software,,* vol. 13, no. 5, pp. 83-88, Sept. 1996.

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

[4] D.R. Wallace and D.R. Kuhn, "Failure Modes in Medical Device Software: An Analysis of 15 Years of Recall Data," *Int'l J. Reliability, Quality and Safety Eng.,* vol. 8, no. 4, 2001.

[5] D.R. Kuhn and M.J. Reilly, "An Investigation of the Applicability of Design of Experiments to Software Testing," *Proc. 27th NASA/IEEE Software Eng. Workshop,* Dec. 2002.

[6] 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," *Proc. Int'l Conf. Software Eng.,* 1999.

[7] B. Smith, M.S. Feather, and N. Muscettola, "Challenges and Methods in Testing the Remote Agent Planner," *Proc. Fifth Int'l Conf. Artificial Intelligence Planning Systems,* 2000.

[8] J. Pan, "The Dimensionality of Failures—A Fault Model for Characterizing Software Robustness," *Proc. Int'l Symp. Fault-Tolerant Computing,* June 1999.

[9] M. Jones, "What Really Happened on Mars Pathfinder Rover," *RISKS Digest,* vol. 19, no. 49, Dec. 1997.

[10] J.L. Lions, "Ariane 5, Flight 501, Report of the Inquiry Board," *European Space Agency,* July 1996.

[11] G. Slabodkin, "Software Glitches Leave Navy Smart Ship Dead in the Water," *Government Computer News,* July 1998.

[12] B. Stevens, L. Moura, and E. Mendelsohn, "Lower Bounds for Transversal Covers," *Design, Codes, and Cryptography,* vol. 15, pp. 279-299, 1998.

[13] *The CRC Handbook of Combinatorial Designs.* C.J. Colbourn and J.H. Dinitz, eds., CRC Press, 1996.

[14] A.W. Williams and R.L. Probert, "A Measure for Component Interaction Test Coverage," *Proc. ACS/IEEE Int'l Conf. Computer Systems and Applications (AICCSA 2001),* pp. 304-311, June 2001.

[15] C. Cheng, A. Dumitrescu, and P. Schroeder, "Generating Small Test Suites for Non-Uniform Instances ," *Third Int'l Conf. Quality Software,* Nov. 2003.

[16] P.E. Ammann, P.E. Black, and W. Majurski, "Using Model Checking to Generate Tests from Specifications ," *Proc. Second IEEE Int'l Conf. Formal Eng. Methods (ICFEM '98),* pp. 46-54, Dec. 1998.

[17] M.R. Blackburn, R.D. Busser, A.M. Nauman, and R. Chandramouli, "Model Based Approach to Security Test Automation," *Proc. Quality Week,* June 2001.

[18] K.C. Tai and Y. Lie, "A Test Generation Strategy for Pairwise Testing ," *IEEE Trans. Software Eng.,* vol. 28, no. 1, pp. 109-111, 2002.

[19] A.W. Williams and R.L. Probert, "Formulation of the Interaction Test Coverage Problem as an Integer Program ," *Proc. TestCom Conf.,* pp. 283-298, 2002.

[20] M.B. Cohen, C.J. Colbourn, P.B. Gibbons, and W.B. Mugridge, "Constructing Test Suites for Interaction Testing," *Proc. 25th Int'l Conf. Software Eng. (ICSE 2003),* pp. 38-48, May 2003.

[21] 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. Software Eng.,* vol. 23, no. 7, pp. 437-444, July 1997.