This article is part of the Reliability Society 2010 Annual Technical Report

Advanced Combinatorial Test Methods for System Reliability

D. Richard Kuhn*, Raghu N. Kacker*, Yu Lei**

**National Institute of Standards & Technology*
Gaithersburg, MD 20899

***University of Texas at Arlington*
Arlington, TX

Every computer user is familiar with software bugs. Many seem to appear almost randomly, suggesting that the conditions triggering them must be complex, and some famous software bugs have been traced to highly unusual combinations of conditions. For example, the 1997 Mars Pathfinder mission began experiencing system resets at seemingly unpredictable times soon after it landed and began collecting data. Fortunately, engineers were able to deduce and correct the problem, which occurred only when (1) a particular type of data was being collected and (2) intermediate priority tasks exceeded a certain load, allowing a blocking condition that eventually triggered a reset.

At 155,000 lines of code (not including the operating system), the Pathfinder program is small compared with commercial software: a Boeing 777 airliner flies on 6.5 million lines of code, the Microsoft Windows XP operating system is estimated at 40 million, and within the next two years the average new car may have more than 100 million lines of code in various subsystems. Ensuring correct operation of complex software is so difficult that more than half of a software development budget – frequently tens of millions of dollars – is normally devoted to testing, and even then errors often escape detection. A 2002 NIST-funded study by the Research Triangle Institute estimated an annual cost of inadequate software testing infrastructure at $22.2 to $59.5 billion for the US economy [1].

As with any engineered system, cost is a critical issue for quality software. Any improvement in software testing efficiency can have a huge impact when testing consumes over half of the development budget. It is clearly possible to build ultra-dependable software (we bet our lives on this proposition each time we board a commercial aircraft), but the process is extremely expensive. Much of the cost results from the human effort involved in attempting to ensure that the software functions correctly in every situation.

Even before the Pathfinder incident, NASA researchers had shown that the fault density (number of faults per line of code) can be over 100 times greater in rarely executed code than in frequently executed portions of a program [2]. In a 1999 study that considered faults arising from rare conditions, NIST reviewed 15 years of medical device recall data, in an effort to determine what types of testing could detect the reported faults [3]. For example, one recall report indicated that the "upper limit CO_2 alarm can be manually set above upper limit without alarm sounding." In this case, a single parameter – CO_2 alarm value – caused the problem, and a test with the upper limit value exceeded could have detected it. Another report gave an example of a problem triggered only when two conditions were met simultaneously: "the ventilator could fail when the altitude adjustment feature was set on 0 meters and the total flow volume was set at a delivery rate of less than 2.2 liters per minute". In this case, a test in which the pair of conditions was true – altitude is 0 and rate is less than 2.2 lpm – could have detected the flaw.

Recognizing that system failures can result from the interaction of conditions that might be innocuous individually, software developers have long used "pairwise testing", in which all possible pairs of parameter values are covered by at least one test. For example, suppose we wanted to show that a new software application works correctly on PCs that use Windows or

Linux operating systems, SQL or Access databases, and IPv4 or IPv6 protocols. This is a total of $2^3 = 8$ possibilities, but only four tests are required to test every component interacting with every other component at least once. A reduction in test set size from 8 to 4 is not that impressive, but consider a larger example: a manufacturing automation system that has 20 controls, each with 10 possible settings, a total of 10^{20} combinations. Surprisingly, we can check all *pairs* of these values with less than 200 tests, if the tests are carefully constructed. Several empirical investigations suggest individual values or pairs of values of two parameters are responsible for roughly 2/3 to more than 95% of faults in real-world applications.

Rationale and Method of Combinatorial Testing

But what about the remaining faults? How many failures will be triggered only by an unusual combinatorial interaction of more than two parameters? The medical device study discussed above found one case in which a failure involved a four-way interaction between parameter values. Subsequent investigations [4, 5] found a similar distribution of failure-triggering conditions: usually, many were caused by a single parameter value, a smaller proportion resulted from an interaction between two parameter values, and progressively fewer were triggered by 3, 4, 5, and 6-way interactions. Figure 1 summarizes these results. With the web server application, for example, roughly 40% of the failures were caused by a single value, such as a file name exceeding a maximum length. Another 30% of the problems were triggered by the interaction of two parameters, and a cumulative total of almost 90% triggered by three or fewer parameters. Curves for the other applications have a similar shape, reaching 100% fault detection with 4 to 6-way interactions. While not conclusive, these results suggest that combinatorial testing which exercises high degree (4-way or above) of combinatorial interactions is necessary to reach a higher level of software assurance. If we know from experience that *t* or fewer variables are involved in failures for a particular application type and we can test all *t*-way combinations of discrete variable settings, then we can have reasonably high confidence that the application will function correctly.

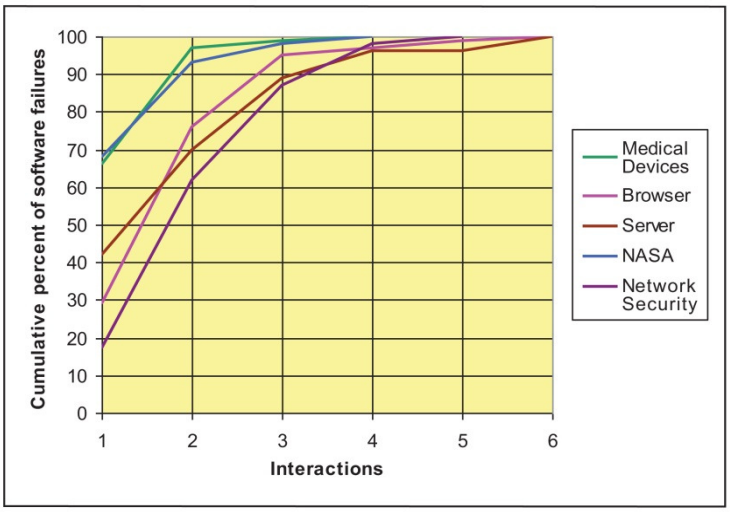

Figure 1. Percentage of failures triggered by *t*-way interactions

The key ingredient for this form of testing is known as a *covering array*, a mathematical object in which all *t*-way combinations of parameter values are covered at least once [6]. Generating covering arrays for complex interactions (beyond pairwise) is a difficult problem, but

new algorithms have been developed that make it possible to generate covering arrays several orders of magnitude faster than previous algorithms, making up to 6-way covering arrays tractable for many applications.

Test sets based on high strength covering arrays for *t*-way testing, for $t = 2$ to 6 or more, are extensions of traditional Design of Experiments (DoE) for testing software/hardware systems, but there are significant differences between traditional DoE and software/hardware combinatorial testing. An objective in DoE (based on fractional factorials, Latin squares, orthogonal arrays, and other such mathematical objects) is to predict an 'optimal combination' using a statistical model estimated from results of experiment. In testing based on covering arrays, statistical models are not used. Tests sets based on DoE are *balanced* (each 2-way combination appears same number of times), but tests sets based on covering arrays are *unbalanced* (each *t*-way combination appears at least once, where $t = 2, 3, 4, 5, 6$ or more) because no statistical prediction model needs to be estimated. In DoE, generally, 3-way and higher order interactions are regarded as error. In contrast, the aim in high strength software/hardware combinatorial testing is to detect possible *t*-way combinatorial interactions, for $t = 2, 3, 4, 5, 6$ or more, which may cause system failure. In conventional DoE, the number of parameters is generally less than 10 and the number of test settings for each parameter is 4 or less. But for software testing, there is no arbitrary limit on the number of parameters or the number of test settings for each parameter. For example, it is not uncommon for a software system to have 50 to 100 parameters (input or configuration parameters) with many test settings. Parameters also typically vary a great deal in the number of settings, e.g., some binary, some with many possible discrete values, and many that represent continuous variables; for these parameters, a subset of roughly 10 or fewer representative values is used. Because balance is not required, the number of test runs in a test suite for combinatorial testing based on a covering array is often less than the number of test runs in the corresponding DoE.

Designs of experiments have been used in agricultural research since the 1920s. Subsequently, their use was extended to animal science and the chemical industry. Designs of experiments based on orthogonal arrays have been used in manufacturing industry since the 1960s. Since the 1980s orthogonal arrays have been used for pairwise (2-way) black-box testing of software. Use of covering arrays for pairwise software testing has begun to attract interest in the past decade, and within the past few years, higher strength testing ($t > 2$) has become possible with new algorithms to efficiently generate test suites based on high strength covering arrays. One such tool is ACTS, freely available from NIST.

Figure 2 gives an example of a covering array for all 3-way interactions of 10 binary parameters (columns) in only 13 tests (rows). It can be seen that any three columns chosen out of ten columns in Fig. 2, contain all eight possible values of the three parameters: 000,001,010,011,100,101,110,111. Referring back to Fig. 1, 3-way interaction testing detected roughly 90% of bugs or more. Exhaustive testing (all possible combinations of the values of ten variables) would require $2^{10} = 1{,}024$ tests.

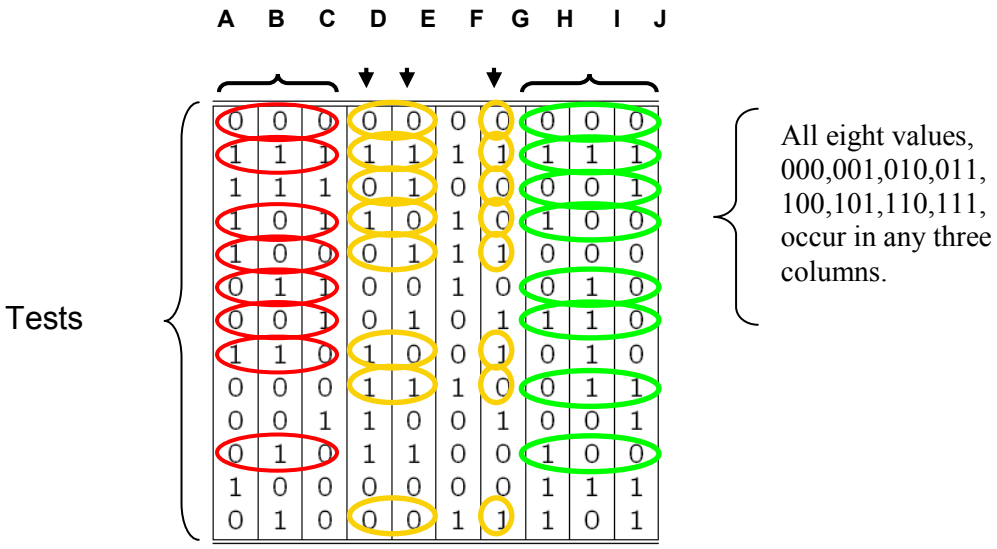

Figure 2. 3-way covering array for 10 parameters with 2 values each.

But even with efficient algorithms to produce covering arrays, some other problems must be solved for practical application of combinatorial testing. Fortunately, since these remaining problems are common to all types of software testing, reasonable solutions exist that are compatible with combinatorial testing. An obvious challenge is simply the enormous number of possible values for input parameters. A 64-bit integer can store more than 18 million trillion possible values, and software may be manipulating a vast number of integers. Clearly, only a small selection of discrete values can be tested for each parameter, so testers establish *equivalence classes*, representing ranges of values that should be treated differently by the software. For example, tested values for a bank account may be: a negative value, 0, a positive value, the largest allowable value, and more than the largest allowable value. Many variations of this partition may be used, such as the previously listed boundary values +/- 1, depending on the objective of testing, but the goal is to reduce the testing effort by selecting a reasonably small number of representative values for each parameter. Some testers also select random test values from each equivalence class.

The Oracle Problem

A more challenging problem is determining the correct result that should be expected from the system under test (SUT) for each set of test inputs. For example, an e-commerce bookstore application that receives inputs of book titles, prices, shipping type, and location, should produce correct outputs for total order amount, tax, and other accounting information. Generating 1,000 test data inputs is of little help if we cannot determine what the SUT should produce as correct (expected) output for each of the 1,000 tests. A test component that determines the expected result for each set of inputs is known as the *test oracle*, and producing a test oracle can be almost as complex as producing the original system under test. Approaches to solving the oracle problem for combinatorial testing include model-based testing, crash testing,

and embedded assertions.

Model-based test oracle generation uses a mathematical model of the SUT and one or more tools that can interpret the model and generate tests together with the expected result for each input. Since tests can be produced independently of each other, the test generation effort can be distributed across a cluster of processors, so even if tens of thousands of test oracles are needed, they can be produced. Model based test generation can be expensive, but its effectiveness has been demonstrated with a variety of test methods, and it has been successfully integrated with combinatorial testing.

Crash testing: A much simpler approach is to simply run tests against the SUT to check whether any unusual combination of input values causes a crash or other easily detectable failure. This is much easier and less expensive to implement than model-based test generation, which requires a full mathematical model of the system. But crash testing clearly produces much less information – a bookstore application that crashes is clearly faulty, but one that runs and produces incorrect results may cost the e-commerce firm its business. Nonetheless, crash testing using combinatorial methods can be an inexpensive way of checking a system's reaction to rare input combinations that might take months or years to occur in normal operation.

Embedded assertions: A method that falls between simple crash testing and model-based test generation is embedding *assertions* within code to ensure proper relationships between data. Assertions are a commonly used programming practice in which logical statements that should always be true are placed at critical points in a program. For example, if a program contains the statement "y = sqrt(x);", an assertion "assert(y*y == x);" may be placed immediately following it to ensure that the square root function worked correctly. Sufficiently thorough placement of assertions at the beginning and end of program functions can be highly effective in ensuring correct operation. Notations such as the Java Modeling language (JML) [7] can be used to introduce very powerful assertions, effectively embedding a formal specification within the code. The embedded assertions serve as an executable form of the specification, thus providing an oracle for the testing phase. With embedded assertions, exercising the application with all *t*-way combinations can provide reasonable assurance that the code works correctly across a very wide range of inputs. This approach has been used successfully in testing smart cards, with one experiment showing that embedded JML assertions acting as a test oracle found up to 90% of seeded faults [8].

Configuration Testing with Combinatorial Methods

In addition to testing input values, combinatorial methods can be used for testing configurations of a software system. Many, if not most, software systems have a large number of configuration parameters, such as alternative operating systems, CPUs, and databases. Many of the earliest applications of combinatorial testing were in testing all pairs of system configurations for telephone switching equipment. For example, telecommunications software may be configured to work with different types of call (local, long distance, international), billing (caller, phone card, 800), access (ISDN, VOIP, PBX), and server for billing (Windows Server, Linux/MySQL, Oracle). The software must work correctly with all combinations of these, so a single test suite could be applied to all pairwise combinations of these four major configuration items. Any system with a variety of configuration options is a suitable candidate for this type of testing. Configuration coverage is perhaps the most developed form of combinatorial testing. It has been used for years with pairwise coverage, particularly for applications that must be shown to work across a variety of combinations of operating systems, databases, and protocols.

Conclusions

While the most basic form of combinatorial testing – pairwise – is well developed, it is only in the past few years that efficient algorithms for complex covering arrays – for up to 6-way coverage – have become available. New algorithms, coupled with fast, inexpensive processors, are making sophisticated combinatorial testing a practical approach that may hold the promise of better software testing at a lower cost.

References

1. NIST. *Economic Impacts of Inadequate Infrastructure for Software Testing* (May 2002).
2. H. Hecht, P. Crane, "Rare Conditions and their Effect on Software Failures", *Proceedings of the 1994 Reliability and Maintainability Symposium,* pp. 334 - 337, January 1994.
3. 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.
4. D.R. Kuhn, D.R. Wallace, and A. Gallo, "Software Fault Interactions and Implications for Software Testing," *IEEE Trans. Software Engineering*, 30(6): 418-421, 2004
5. K.Z. Bell, Optimizing Effectiveness and Efficiency of Software Testing: a Hybrid Approach, PhD Dissertation, North Carolina State University, 2006.
6. Y. Lei, R. Kacker, D.R. Kuhn, V. Okun, J. Lawrence, "IPOG/IPOG-D: Efficient Test Generation for Multi-Way Combinatorial Testing", *Software Testing, Verification, and Reliability.*
7. G.T. Leavens, A.L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*. Kluwer, 1999.
8. L. du Bousquet, Y. Ledru, O. Maury, C. Oriat, J.-L. Lanet, A case study in JML-based software validation. *Proceedings of 19th Int. IEEE Conf. on Automated Sofware Engineering*, pp. 294-297, Linz, Sep. 2004.