Combinatorial Software Testing

Rick Kuhn, Yu Lei, Raghu Kacker, Justin Hunter

Developers of large data-intensive software often notice an interesting – though not surprising – phenomenon: when usage of an application jumps dramatically, components that have operated for months without trouble suddenly develop previously undetected errors. For example, newly added customers may have account records with an oddball combination of values that have not been seen before. Some of these rare combinations trigger faults that have escaped previous testing and extensive use. Or, the application may have been installed on a different OS-hardware-DBMS-networking platform. Combinatorial testing, which exercises all *t*-way combinations up to a pre-specified level of *t*, can help find problems like this early in the testing life-cycle.

For example, suppose we wanted to show that a new software application works correctly on PCs that use Windows or Linux operating systems, Intel or AMD processors, and IPv4 or IPv6 protocols. This is a total of 2 x 2 x 2 = 8 possibilities, but only four tests are required to test every component interacting with every other component at least once (see Table 1). This is only the most basic combinatorial method, "pairwise testing", in which all possible pairs of parameter values are covered by at least one test.

Test case	OS	CPU	Protocol
1	Windows	Intel	IPv4
2	Windows	AMD	IPv6
3	Linux	Intel	IPv6
4	Linux	AMD	IPv4

Table 1. Pairwise test configurations

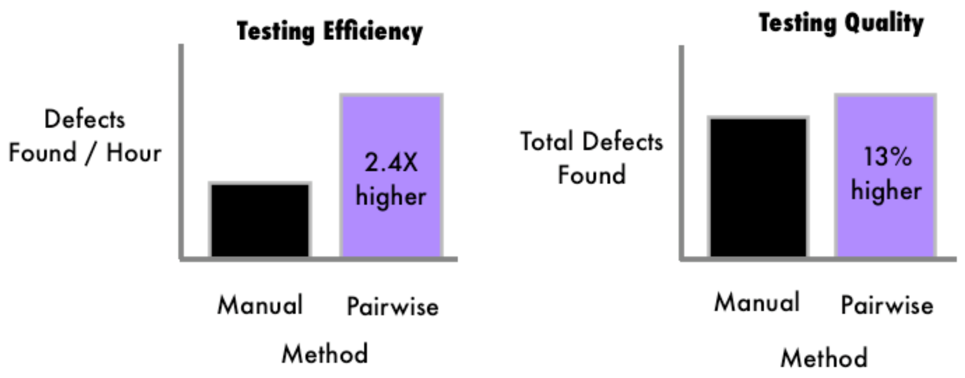
It should be noted that while all pairs of possible values (e.g. (a) OS = Linux & (b) protocol = Ipv4), are tested for by at least one test case, several combinations of three specific values are not tested (e.g., (a) OS = Windows & (b) CPU = Intel & (c) Protocol = IPv6). We will address this potential "lack of thoroughness" issue in a moment.

How does combinatorial testing work in practice?

Even with this acknowledged deficiency, pairwise testing is used because it often works very well. The reduction in test set size from 8 to 4 shown in Table 1 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, which is far more combinations than a software tester would be able to test in a lifetime. Surprisingly, we can check all *pairs* of these values with only 180 tests if the tests are carefully constructed. Pairwise testing has become popular because of this; it can check for simple, potentially problem-causing interactions with relatively few tests. Several empirical investigations suggest individual values or pairs of values of two parameters are responsible for roughly 50% to more than 97% of faults.

One of the authors, Justin Hunter, conducted a 10-project empirical study that compared the effectiveness of pairwise combinatorial testing with manual test case selection methods. The findings, shown in Figure 1, speak for themselves.

Figure 1: Average Efficiency and Quality Improvements found in 10 Projects from Pairwise Testing (vs. Manual Test Case Selection)



The ten testing projects were conducted at six companies and tested commercial applications in development; in each project, two small teams of testers were asked to test the same application at the same time using different methods. One group of testers selected tests manually; they relied on “business as usual” methods such as developing tests based on (a) functional and technical requirements and (b) potential use cases mapped out on white boards. The other group used a software tool to identify 2-way combinatorial (pairwise) tests. Test execution productivity was significantly higher in all ten projects for the testers using combinatorial methods, with test execution productivity more than doubling on average and more than tripling in three projects. The groups using the pairwise combinatorial testing approach also achieved the same or higher quality in all ten projects; all of the defects identified by the teams using manual test case selection methods were identified by the teams using combinatorial methods, and in five projects, the combinatorial teams found additional defects that had not been identified by the teams using manual test case identification methods. These were “proof of concept” projects which successfully demonstrated to the teams involved that their manual “business as usual” methods of test case selection were not nearly as effective as the newly-introduced pairwise combinatorial methods (the simplest form of combinatorial testing) to find the largest number of defects in the least amount of time.

How much combinatorial testing is needed?

We noted that other empirical studies have concluded that from about 50% to 97% of faults could be identified by well-selected pairs of parameter settings. But what about the remaining faults? How many failures will be triggered only by an unusual interaction involving more than two parameters? In a 1999 study that considered faults arising from rare conditions, the National Institute of Standards and Technology (NIST) reviewed 15 years of medical device recall data in an effort to determine what types of testing could detect the reported faults. This study found one case in which a fault involved a four-way interaction between parameter values. In that example, an error could be triggered when: (1) demand dose = administered, and (2) days elapsed = 31, and (3) pump time = unchanged, and (4) battery status = charged. Pairwise testing is unlikely to detect faults like this, because pairwise testing only guarantees that all *pairs* of parameter values will be tested. A particular 4-way combination of values is statistically unlikely to occur in a test set that only ensures 2-way combination coverage, so in order to ensure thorough testing of complex applications, it is necessary to generate test suites for 4-way, or higher degree, interactions.

Investigations of other applications found similar distributions of fault-triggering conditions: usually, many faults 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 2 summarizes these results.

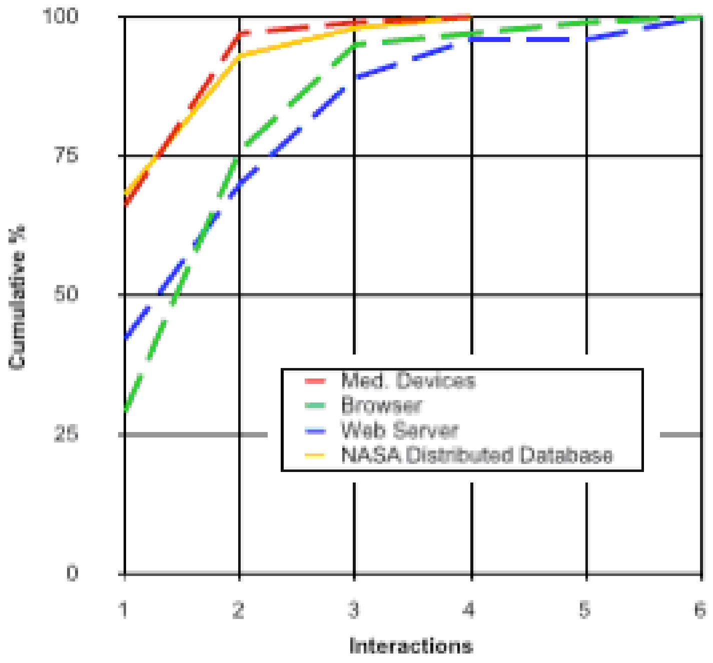


Figure 2. Cumulative error detection rate at interactions 1 to 6.

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 certain 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. While not conclusive, these results suggest that combinatorial testing which exercises high degree interaction combinations (4-way or above) can be very effective for achieving a higher level of thoroughness in software testing.

The key ingredient for this kind 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. For the pairwise testing example in Table 1, $t = 2$, and it is relatively easy to generate tests that cover all pairs of parameter values. Generating covering arrays for more complex interactions (beyond pairwise) is a much more difficult problem, but new algorithms have been developed that make it possible to generate covering arrays orders of magnitude faster than previous algorithms, making up to 6-way covering arrays tractable for many applications.

Figure 3 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 of Fig. 3, selected in any order, contain all eight possible values of three parameters: 000,001,010,011,100,101,110,111.

					Parameters					
Test	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	0	0	0	0	0	0	0	0	0	0
2	1	1	1	1	1	1	1	1	1	1
3	1	1	1	0	1	0	0	0	0	1
4	1	0	1	1	0	1	0	1	0	0
5	1	0	0	0	1	1	1	0	0	0
6	0	1	1	0	0	1	0	0	1	0
7	0	0	1	0	1	0	1	1	1	0
8	1	1	0	1	0	0	1	0	1	0
9	0	0	0	1	1	1	0	0	1	1
10	0	0	1	1	0	0	1	0	0	1
11	0	1	0	1	1	0	0	1	0	0
12	1	0	0	0	0	0	0	1	1	1
13	0	1	0	0	0	1	1	1	0	1

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

Referring back to Fig. 2, we saw that three-way interaction testing detected roughly 90% of bugs or more in all four empirical studies. In Figure 3, exhaustive testing (all possible combinations) would require 2^{10} = 1,024 tests. What are the pragmatic implications of being able to achieve 100% 3-way coverage in 13 test cases on real-world software testing projects? If we assume for the sake of argument that there are 10 defects in this hypothetical application and that 9 of them are identified through the 13 tests listed here, testing these 13 cases (and thereby uncovering 9 of 10) would result in finding *71 times more defects per test case* ((9/13) / (10/1024)) than testing exhaustively(and uncovering all 10).

What issues are important when considering combinatorial testing?

There are a number of practical testing takeaways for software testing practitioners considering combinatorial methods. Below we compare pairwise and more thorough methods:

- **Resources:** Teams in a hurry seeking to efficiently maximize testing thoroughness given tight time and/or resource constraints may want pairwise (2-way) testing. When more time is available or more thorough testing is required, *t*-way testing for $t > 2$ is better.
- **Fault detection:** Although much more empirical research is needed, available data suggest that pairwise testing may find 50% to > 90% of faults, but higher strength combinations ($t > 2$) can detect 90% to 100% of faults, and variability among detection rates appears to decrease as *t* increases.
- **Awareness and adoption:** Pairwise testing is well known among researchers and practitioners, although still not widely used, so training may be needed for many testers. Research interest in higher strength *t*-way testing has increased recently as better algorithms have become available.

While the most basic form of combinatorial testing – pairwise – is established, and adoption by practitioners continues to increase, usage in industry remains patchy at best. Practitioners who face significant time and resource pressure (and who currently use manual test case selection methods) will find pairwise methods deliver large efficiency improvements. Practitioners who require very high quality software will find *t*-way combinatorial testing efficiently detects many hard-to find faults. It is only in the past few years that efficient algorithms for complex covering arrays – for 4-way coverage or more – have become available, so empirical experience is sparse, but these methods appear to enable extremely thorough testing of applications with manageable numbers of test cases. New algorithms (packaged in an increasing number of user-friendly tools) with fast, inexpensive processors, are making sophisticated combinatorial testing a practical approach that holds considerable promise for better software testing at a lower cost.