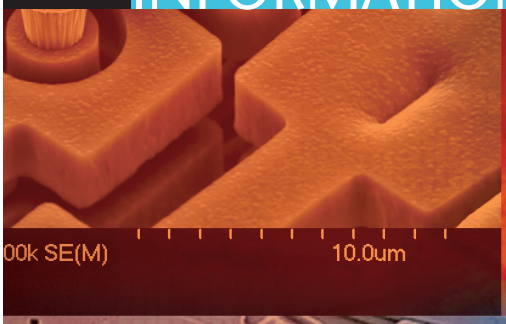

Practical Combinatorial Testing: Beyond Pairwise

Rick Kuhn, *US National Institute of Standards and Technology*
Yu Lei, *University of Texas, Arlington*
Raghu Kacker, *US National Institute of Standards and Technology*

With new algorithms and tools, developers can apply high-strength combinatorial testing to detect elusive failures that occur only when multiple components interact.

Suppose we want to test a process-control system that has 20 sensors, each with 10 possible values. As usual, exhaustive testing is out of the question: 10^{20} tests would require billions of years to execute. We could test all 10 values for each sensor with only 10 tests, but that's unlikely to be enough. Experience suggests that some faults are triggered only by unusual combinations—hence, the popularity of *pairwise testing*, which is based on the observation that software faults often involve interactions between two parameters. In pairwise testing, all possible pairs of parameter values are covered by at least one test, and good tools are available to generate arrays with the value pairs.[1] By selecting parameter values carefully, we could test all value pairs for the 20 sensors in our example with only 180 tests.

But what if some failure is triggered only by a very unusual combination of three, four, or more values? It's unlikely that our 180 tests would detect this unusual case. We would need to test at least three- and four-way value combinations. Combinatorial testing beyond pairwise is rare, however, because good algorithms for higher-strength combinations haven't been available or were too slow for practical use. In the past few years, advances in covering-array algorithms, integrated with model checking or other testing approaches, have made it practical to extend combinatorial testing beyond pairwise tests. The US National Institute of Standards and Technology (NIST) and the University of Texas, Arlington, are now distributing freely available methods and tools for constructing large *t*-way

1520-9202/08/$25.00 © 2008 IEEE Published by the IEEE Computer Society

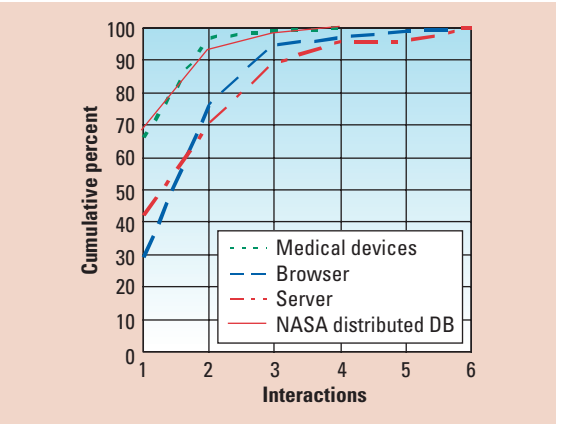

Figure 1. Error-detection rates for four- to six-way interactions in four application domains: medical devices, a Web browser, an HTTP server, and a NASA distributed database.

combination test sets (known as covering arrays), converting covering arrays into executable tests, and automatically generating test oracles using model checking (http://csrc.nist.gov/acts). In this review, we focus on real-world problems and empirical results from applying these methods and tools.

Motivation and Challenges

If some failure is triggered only by an unusual combination of more than two sensor values, how many testing combinations are enough to detect all errors? What degree of interaction occurs in real system failures? Surprisingly, researchers hadn't studied these questions when NIST began investigating causes of software failures in 1996. Study results showed that, across various domains, all failures could be triggered by a maximum of four- to six-way interactions.[2] As Figure 1 shows, the detection rate increased rapidly with interaction strength. Within the NASA database application, for example, 67 percent of the failures were triggered by only a single parameter value, 93 percent by two-way combinations, and 98 percent by three-way combinations.[2] The detection-rate curves for the other applications studied are similar, reaching 100 percent detection with four- to six-way interactions.

These results are not conclusive, but they suggest that the degree of interaction involved in faults is relatively low, even though pairwise testing is insufficient. Testing all four- to six-way combinations might therefore provide reasonably high assurance. As with most things, however, the situation isn't that simple. Quite a few practical problems remain. Efficiently generating test suites to cover all t-way combinations is a difficult mathematical problem that researchers have studied for nearly a century. In addition, not many applications have only a few discrete values for each variable; most include continuous variables with huge ranges of values, depending on the hardware on which the program runs. Most glaring of all is the *oracle problem*: determining the correct result that should be expected from the system under test (SUT) for each set of test inputs. Generating 1,000 test data inputs is of little help if we can't determine what the SUT should produce as output for each of the 1,000 tests.

With the exception of generating covering arrays, these stumbling blocks are common to all types of software testing, and researchers have developed good techniques for dealing with them. The challenge in making combinatorial testing practical, then, is to find efficient algorithms to generate covering arrays and effective methods of integrating the tests produced into the testing process.

Generating Covering Arrays

The first step in combinatorial testing is to find a set of tests (a covering array) that will cover all t-way combinations of parameter values for the desired strength t. In pairwise testing, $t = 2$, and good algorithms are widely available. For stronger assurance, the results discussed earlier suggest that we need a set of tests that covers all four-way or higher-strength combinations of parameter values. The covering array specifies test data, where we can regard each row of the array as a set of parameter values for an individual test. Collectively, the array's rows include every t-way combination of parameter values at least once.

Figure 2 gives an example that shows a three-way covering array for 10 variables with two values each. In this array, any three columns contain all eight possible values for three binary variables. Therefore, this set of tests will exercise all three-way combinations of input values in only 13 tests, compared with 1,024 for exhaustive coverage. We can generate similar

arrays to cover up to all six-way combinations. New algorithms, such as In-Parameter Order, General (IPOG) and the Bryce-Colbourn Density algorithm,[3] make this possible with greater efficiency than previous tools.For example, an existing tool required 5,400 seconds to produce a less optimal test set than the IPOG algorithm generated in 4.2 seconds. These new tools make it possible to apply combinatorial testing to applications that previously were prohibitive in time and cost.

Tackling the Oracle Problem

Although Figure 2 shows that we can accomplish interaction testing beyond pairwise with far fewer tests than are required for exhaustive testing, real systems typically have more than 10 binary parameters. How many tests are needed for real-world systems?

In general, the number of tests required for t-way combinatorial testing of n parameters with v values apiece is proportional to $v^t \log n$. So, the number of tests needed for four-way testing is several times that required for three-way testing. On the other hand, testing 30 parameters requires a modest increase over the number of tests needed for 20. For example, a system with 20 variables, five values each, requires 444 tests for three-way coverage but 3,019 tests for four-way coverage with IPOG. A much smaller penalty is incurred for covering more variables: increasing the number of variables to 30 requires 3,749 tests for four-way coverage, a 24 percent increase.

However, even with efficient algorithms to produce covering arrays, the oracle problem remains. Taking advantage of combinatorial testing might require numerous tests in some cases, although not always. Approaches to addressing the oracle problem for combinatorial testing include *crash testing*, *embedded assertions* , and *model checker-based test generation*.

Crash testing is the easiest and least expensive approach: simply run tests against the SUT to check whether any unusual combination of input values causes a crash or other easily detectable failure. This form of combinatorial testing could be regarded as a disciplined form of fuzz testing, which sends random values against the SUT.[5] Although pure random testing will generally cover a high percentage of t-way combinations,

	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 2. Three-way covering array for 10 parameters with two values each. Any three columns contain all eight possible values for three binary variables.

100 percent coverage of combinations requires a random test set much larger than a covering array. For example, all three-way combinations of 10 parameters with four values each can be covered with 151 tests using IPOG. Purely random generation requires approximately 914 tests to provide full three-way coverage.

Embedded assertions are increasingly popular. This lightweight formal method embeds assertions within the code to ensure proper relationships between data, such as preconditions, postconditions, or input value checks. Tools such as the Java Modeling Language (JML) can help us introduce complex assertions, effectively embedding a formal specification within the code.[6] 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 range of inputs. JML has been used successfully as a high-level (that is, not fully complete) specification of system behavior to test smart cards,[7] with embedded JML assertions acting as an oracle for combinatorial tests. The results showed that 80 to 90 percent of errors could be found in this way.

In a more comprehensive approach, model-checker based test generation uses a mathematical model of the SUT and a model checker to generate expected results for each input. Conceptually, the

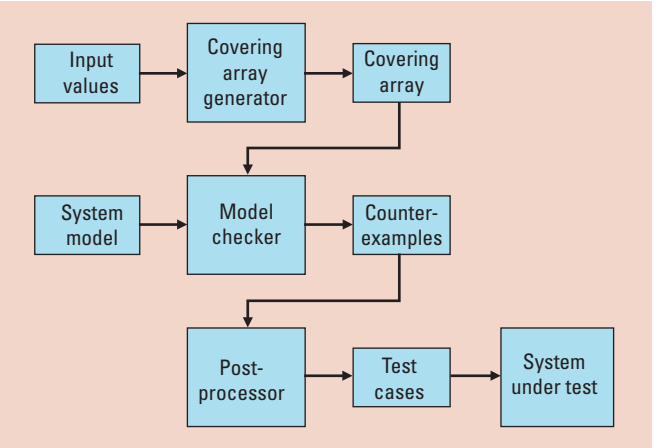

Figure 3. Generating test cases with a model checker.

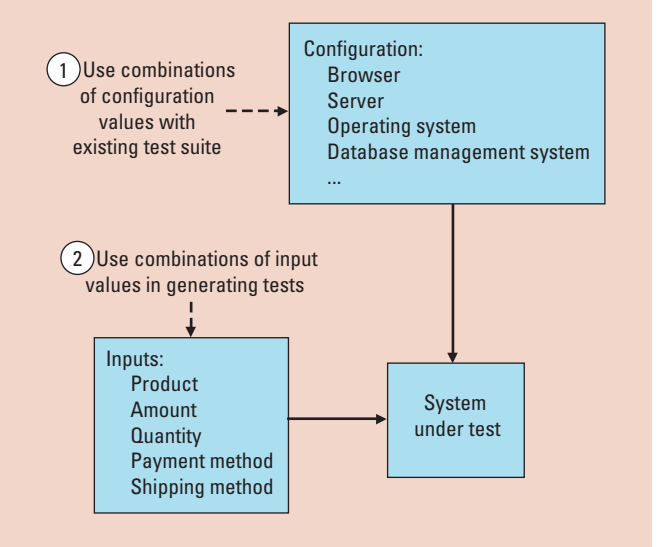

Figure 4. Two approaches to combinatorial testing: using configuration parameter values or input data values.

model checker explores all system model states to determine if a property claimed in a specification statement is true. A model checker is particularly valuable because it not only reports that a claim is false but also provides a counterexample that includes a trace of parameter input values and states that will prove it is false. In effect, this is a complete test case—that is, a set of parameter values and the expected result. It's then simple to map these values into complete test cases in the syntax needed for the SUT.[8] Figure 3 illustrates this process.

As an example, we applied the model checking approach to combinatorial testing on a module from the US Federal Aviation Administration (FAA) Traffic Collision Avoidance System (TCAS), which had been used in previous test-method studies. The module used 12 parameters: seven Boolean, two three-value, one four-value, and two with 10 values. Researchers from Siemens developed 41 versions seeded with realistic faults. Two thirds of the faulty versions had single changes such as replacing a constant with another constant, replacing ">=" with ">," or dropping a condition. The rest involved either multiple changes or more complex single changes. Covering all two- to six-way combinations for this module required 17,000 tests. Creating the formal system specification required time and expertise in formal methods, but once the specification was in place, we generated and executed all 17,000 tests in a few minutes.[9] Pairwise testing detected 53 percent of the faults, but testing through five-way combinations provided 100 percent detection.

Two Ways to Use Covering Arrays

The two basic approaches to combinatorial testing use combinations of either configuration parameter values or input parameter values. In the first approach, we use the covering array to select values of configurable parameters, possibly with the same tests run against all configuration combinations. For example, we might test a server by setting up all four-way combinations of configuration parameters—such as the number of simultaneous connections allowed, the amount of memory, the operating system, or the database size—and then run the same test suite against each configuration. The tests might have been constructed using any methodology, not necessarily combinatorial coverage. In the second approach, we use the covering array to select input data values, which then become part of complete test cases, creating a test suite for the application. This approach requires combinatorial coverage of input data values for the tests constructed.

Figure 4 contrasts these two approaches for an example e-commerce system. With the first approach, we might run the same test set against all three-way combinations of configuration options. For the second approach, we would

construct a test suite that covers all three-way combinations of input transaction fields. Of course, we could combine these approaches, with the combinatorial tests (approach 2) run against all the configuration combinations (approach 1).

Many, if not most, software systems have a large number of configuration parameters. Many of the earliest applications of combinatorial testing were in testing all pairs of system configurations. For example, telecommunications software can be configured to work with different types of calls (local, long distance, international), billing (caller, phone card, 800), access (ISDN, VOIP, PBX), and billing servers (Windows Server, Linux/MySQL, Oracle). The software must work correctly with all these combinations, so we could apply a single test suite 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. Testers have used it for years with pairwise coverage, particularly for applications that must be shown to work across various combinations of operating systems, databases, and network characteristics. Recently, a more sophisticated version has proved effective for applications with elaborate configuration options, such as Web browsers and office tools.[10]

This example demonstrated the feasibility of higher-strength combinatorial testing for small- to medium-sized modules. We're currently working with developers of real-world software to measure the costs and benefits of this approach for full-scale systems. Interested testers can find more on the methods and tools we describe here at http://csrc.nist.gov/acts.

References

1. K. Burr and W. Young, "Combinatorial Test Techniques: Table-Based Automation, Test Generation, and Test Coverage," *Proc. Int'l Conf. Software Testing, Analysis, and Review* (STAR), 1998; http://aetgweb.argreenhouse.com/papers/1998-star.pdf.
2. D.R. Kuhn, D.R. Wallace, and A. Gallo, "Software Fault Interactions and Implications for Software Testing," *IEEE Trans. Software Eng.*, vol. 30, no. 6, 2004, pp. 418–421.
3. R. Bryce and C.J. Colbourn, "A Density-Based Greedy Algorithm for Higher Strength Covering Arrays," to be published in *J. Software Testing, Verification, and Reliability*; www.egr.unlv.edu/~rbryce/research.htm.
4. R. Bryce and C.J. Colbourn, "The Density Algorithm for Pairwise Interaction Testing," *J. Software Testing, Verification, and Reliability*, vol. 17, no. 3, Aug. 2007, pp. 159–182.
5. M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*, Addison-Wesley, 2007.
6. G.T. Leavens, A.L. Baker, and C. Ruby. "JML: A Notation for Detailed Design," H. Kilov, B. Rumpe, and I. Simmonds, eds., *Behavioral Specifications of Businesses and Systems*, Kluwer, 1999, pp. 175–188.
7. L. du Bousquet et al., "A Case Study in JML-Based Software Validation," *Proc. 19th Int'l IEEE Conf. Automated Software Eng.* (ASE 04), IEEE CS Press, 2004, pp. 294–297.
8. P. Ammann and P.E. Black, "Abstracting Formal Specifications to Generate Software Tests via Model Checking," *Proc. 18th Digital Avionics Systems Conf.*, vol. 2., IEEE Press, 1999, pp. 10.A.6.1–10.
9. D.R. Kuhn and V. Okun, "Pseudo-Exhaustive Testing for Software," *Proc. 30th NASA/IEEE Software Eng. Workshop*, 2006, pp. 153–158; http://csrc.nist.gov/acts/PID258305.pdf.
10. M.B. Cohen, J. Snyder, and G. Rothermel, "Testing Across Configurations: Implications for Combinatorial Testing," *Proc. Workshop on Advances in Model-Based Software Testing*, IEEE Press, 2006, pp. 1–9.

Rick Kuhn *is a computer scientist in the Computer Security Division of the US National Institute of Standards and Technology. Contact him at kuhn@nist.gov.*

Yu Lei *is an assistant professor of computer science at the University of Texas, Arlington. Contact him at ylei@uta.edu.*

Raghu Kacker *is a mathematical statistician in the Mathematical and Computational Sciences Division of the US National Institute of Standards and Technology. Contact him at raghu.kacker@nist.gov.*