Chapter 14
Combinatorial Testing

Renée C. Bryce
Utah State University, USA

Yu Lei
University of Texas, Arlington, USA

D. Richard Kuhn
National Institute of Standards and Technology, USA

Raghu Kacker
National Institute of Standards and Technology, USA

ABSTRACT

Software systems today are complex and have many possible configurations. Products released with inadequate testing can cause bodily harm, result in large economic losses or security breaches, and affect the quality of day-to-day life. Software testers have limited time and budgets, frequently making it impossible to exhaustively test software. Testers often intuitively test for defects that they anticipate while less foreseen defects are overlooked. Combinatorial testing can complement their tests by systematically covering t-way interactions. Research in combinatorial testing includes two major areas (1) algorithms that generate combinatorial test suites and (2) applications of combinatorial testing. The authors review these two topics in this chapter.

I. INTRODUCTION

Software systems are complex and can incur exponential numbers of possible tests. Testing is expensive and trade-offs often exist to optimize the use of resources. Several systematic approaches to software testing have been proposed in the literature. Category partitioning is the base of all systematic approaches as finite values of parameters are identified for testing. Each of these finite parameter-values may be tested at least once, in specified combinations together, or in exhaustive combination. The simplest approach tests all values at least once. The most thorough approach exhaustively tests all parameter-value combinations. While testing only individual values may not be enough, exhaustive testing of all possible combinations is not always feasible. Combination strategies are a reasonable alternative that falls in between these two extremes.

DOI: 10.4018/978-1-60566-731-7.ch014

Copyright © 2010, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Table 1. Four parameters that have three possible settings each for an on-line store

Log-in Type	Member Status	Discount	Shipping
New member - not logged in	Guest	None	Standard (5-7 day)
New-member - logged in	Member	10% employee discount	Expedited (3-5 day)
Member - logged in	Employee	$5 off holiday discount	Overnight

Consider an on-line store that has four parameters of interest as shown in Table 1. There are three log-in types; three types of member status; three discount options; and three shipping options. Different end users may have different preferences and will likely use different combinations of these parameters. To exhaustively test all combinations of the four parameters that have 3 options each from Table 1 would require $3^4 = 81$ tests.

In this example, exhaustive testing requires 81 test cases, but pair-wise combinatorial testing uses only 9 test cases. Instead of testing every combination, all individual pairs of interactions are tested. The resulting test suite is shown in Table 2, and is contains only 9 tests. All pairs of combinations have been combined together at least once during the testing process. For instance, the first test from Table 2 covers the following pairs: (New member - not logged in, Guest), (New member - not logged in, $5 off holiday discount), (New member - not logged in, Standard (5-7 day)), (Guest, None), (Guest, Standard (5-7 day)), and (None, Standard (5-7 day)). The entire test suite covers every possible pairwise combination between components. This reduction in tests amplifies on larger systems - a system with 20 factors and 5 levels each would require 5^{20} = 95,367,431,640,625 exhaustive tests! Pairwise combinatorial testing for 5^{20} can be achieved in as few as 45 tests.

II. BACKGROUND

Combinatorial testing is simple to apply. As a specification-based technique, combinatorial testing requires no knowledge about the implementation under test. Note that the specification required by some forms of combinatorial testing is lightweight, as it only needs to identify a set of parameters and their possible values. This is in contrast with other testing techniques that require a complex operational model of the system under test. Finally, assuming that the parameters and

Table 2. A pair-wise combinatorial test suite

Test No.	Log-in Type	Member Status	Discount	Shipping
1	New member - not logged in	Guest	None	Standard (5-7 day)
2	New member - not logged in	Member	10% employee discount	Expedited (3-5 day)
3	New member - not logged in	Employee	$5 off holiday discount	Overnight
4	New-member - logged in	Guest	$5 off holiday discount	Expedited (3-5 day)
5	New-member - logged in	Member	None	Overnight
6	New-member - logged in	Employee	10% employee discount	Standard (5-7 day)
7	Member - logged in	Guest	10% employee discount	Overnight
8	Member - logged in	Member	$5 off holiday discount	Standard (5-7 day)
9	Member - logged in	Employee	None	Expedited (3-5 day)

values are properly identified, the actual combination generation process can be fully automated, which is a key to industrial acceptance.

In this section, we review applications of combinatorial testing. Two major research themes exist on the empirical effectiveness of combinatorial testing:

1. *What measures of effectiveness exist for combinatorial testing?*
2. How much combinatorial testing is enough? (i.e.: what is largest number of variables which may be involved in failures)

Combinatorial testing is based on the premise that many errors in software can only arise from the interaction of two or more parameters. A number of studies have investigated the application of combinatorial methods to software testing (Burr 1998; Cohen 1997; Dunietz 1997; Kuhn 2002; Kuhn 2004; Wallace 2001; Williams 2001; Yilmaz 2006). Early research focused on pairwise testing, i.e., testing all 2-way combinations of parameter values, thus exercising all interactions between parameters or components at least once. Some of these were designed to determine the degree of test coverage obtained using combinatorial methods, e.g. (Dunietz 1997), (Cohen 1996). These studies use code coverage, rather than fault detection, to measure the effectiveness of combinatorial testing. They show that combinatorial methods can produce coverage results that are comparable or better than other test schemes. Code coverage is an important metric, but only an indirect one. Testers seek to detect faults in an application, so a direct measure of effectiveness for a test method is the fault detection rate.

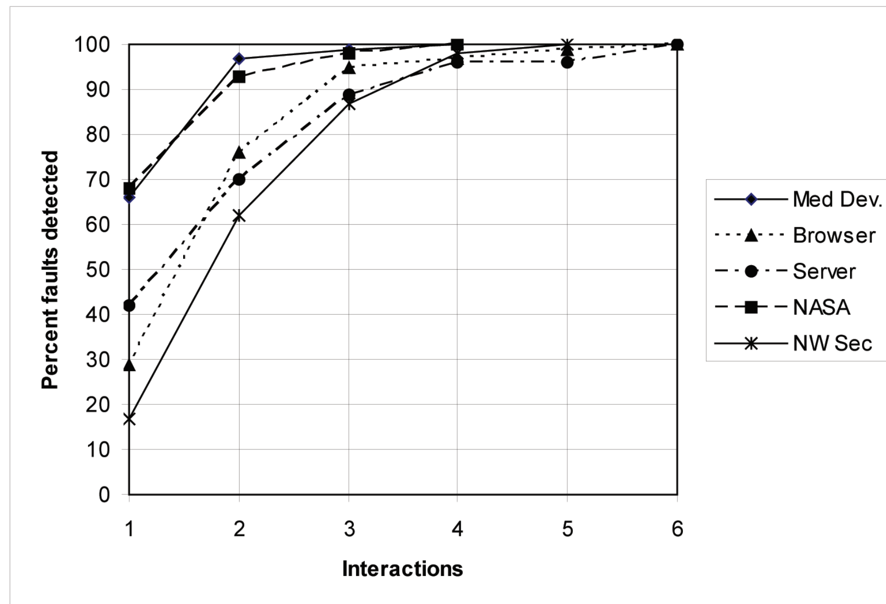
Many studies demonstrated the effectiveness of pairwise testing in a variety of applications. But what if some failure is triggered only by a very unusual combination of more than two values? What degree of interaction occurs in real failures in real systems? Studies that investigated the distribution of *t*-way faults are summarized in Figure 1 and Table 3 (Kuhn 2002; Kuhn 2004). As can be seen from the data, across a variety of domains, all failures could be triggered by a maximum of 4-way to 6-way interactions. Figure 1 shows that the detection rate increases rapidly with interaction strength. With the server, for example, 42% of the failures were triggered by only a single parameter value, 70% by 2-way combinations, and 89% by 3-way combinations. The detection rate curves for the other applications are similar, reaching 100% detection with 4 to 6-way interactions. That is, six or fewer variables were involved in all failures for the applications studied, so 6-way testing could in practice detect nearly all of the failures. So far we have not seen a failure from combinatorial interaction involving more than six variables. While not conclusive, these results suggest that combinatorial testing which exercises high strength interaction combinations can be an effective approach to software assurance. Much more empirical work will be needed to understand the effectiveness of combinatorial testing in different domains. Note that the detection rate at different interaction strengths varies widely for the studies shown in Figure 1. Additional research will help determine the extent to which these limited results can be generalized to other types of software.

III. TOOLS THAT GENERATE COMBINATORIAL TEST SUITES

Three main types of algorithms construct combinatorial test suites: algebraic, greedy, or heuristic search algorithms. A high-level overview of the major advantages and disadvantages of the algorithms that construct combinatorial test suites include:

1. *Algebraic methods* offer efficient constructions in regards to time; however, it is difficult to produce accurate results on a broad and general variety of inputs with algebraic

Figure 1. Fault detection at interaction strengths 1 to 6 (Source: Kuhn 2004; Bell 2006)



methods. (See Colbourn 2004 and references given therein.)

2. *Greedy algorithms* are a well-studied type of algorithm for the construction of covering arrays because they have been found to be relatively efficient in regards to time and accuracy (Bryce 2007; Bryce to appear; Cohen 1996; Cohen 1997; Lei 2008; Tai 2002; Tung 2000).
3. *Heuristic search*, particularly through the application of Simulated Annealing (SA) has provided the most accurate results in several instances to date. This local search method has provided many of the smallest test suites for different system configurations; however, at a cost in execution time to generate test suites (Cohen 2008).

We refer the reader to the papers above for more details of these algorithms and here we include an overview of a freely available research tool, called FireEye, to generate combinatorial test suites. The IPO algorithm was first proposed for pairwise testing (Tai 2002), and was later extended

Table 3. Percent fault detection at interaction strengths 1 to 6 (Source: Kuhn 2004; Bell 2006)

Interaction strength	Med Devices	Browser	Server	NASA Database	Network Security
1	66%	29%	42%	68%	17%
2	97	76	70	93	62
3	99	95	89	98	87
4	100	97	96	100	98
5	100	99	96	100	100
6	100	100	100	100	100

Figure 2. Illustration of IPO

P1	P2
0	0
0	1
1	0
1	1

(a)

P1	P2	P3
0	0	0
0	1	1
1	0	2
1	1	0

(b)

P1	P2	P3
0	0	0
0	1	1
1	0	2
1	1	0
0	1	2
1	0	1

(c)

to general *t*-way combinatorial testing (Lei 2008). The FireEye tool implements the general version of the IPO algorithm, called IPOG (Lei 2008). We provide both an overview of the algorithm and screenshots of the tool that uses the algorithm. Thus the readers have an example that helps them to build their own test suites.

Overview of IPOG

The general framework of the IPOG algorithm can be described as follows: For a system with at least *t* parameters, the IPOG strategy first builds a *t*-way test set for the first *t* parameters, it then extends the test set to a *t*-way test set for the first *t* + 1 parameters, and then continues to extend the test set until it builds a *t*-way test set for all the parameters. The extension of an existing *t*-way test set for an additional parameter begins with horizontal growth, which extends each existing test by adding one value for the new parameter. The *t*-way tuples covered by the addition of those new values are tracked and removed from the set of uncovered tuples. Note that horizontal growth does not add any new tests, but only extends the existing ones. After horizontal growth, if all the tuples have not yet been covered the test set is extended vertically, i.e., new tests are added to cover the remaining tuples. The IPOG algorithm utilizes local optimums to provide a bound of accuracy for worst case scenarios.

In the following we use an example system to illustrate the working of the IPO algorithm. This example system consists of three parameters P_1, P_2, and P_3, where P_1, and P_2 have two values 0 and 1, and P_3 has three values 0, 1, and 2. Figure 2 shows the construction of a 2-way test set for the example system using the IPO algorithm.

The IPO algorithm first builds a 2-way test set for the first two parameters P1 and P2, which is shown in Figure 2 (a). This test set simply contains 4 tests, each of which is one possible combination of values of P1 and P2. Next, this test set is extended to cover parameter P3. In order to cover P3, we only need to cover all the combinations involving P3 and P1 and those involving P3 and P2. This is because all the combinations involving P1 and P2 have already been covered. There are in total 12 combinations to cover: 6 combinations involve P3 and P1, and 6 combinations involve P3 and P2. Those combinations are covered in the following two steps:

- *Horizontal growth*: This step extends each of the four existing tests by adding a value for the P3, as shown in Figure 2 (b). These values are chosen in a greedy manner. That is, each value is chosen and added into a test such that it covers the most combinations that have not been covered yet. In Figure 2 (b), the four existing tests are extended with values 0, 1, 2, 0, respectively, each of

which covers two new combinations. Note that since there are only three parameters, adding a value of P3 can cover at most two combinations. For example, the 1st test is extended with 0, which covers two new combinations {(P_1.0, P_3.0), (P_2.0, P_3.0)}. Note that adding value 1 or 2 into the 1st test also covers two new combinations. In this case, the tie is broken arbitrarily for value 0. As another example, the last test is extended with value 0, which covers two new combinations combinations {(P_1.1, P_3.0), (P_2.1, P_3.0)}. If the fourth test was extended with value 1 or 2, it would only cover one new combination. This is because (P2.1, P3.1) (or (P1.1, P3.2)) has already been covered in the second test (or in the third test) when it is extended with value 1 (or value 2).

- *Vertical growth*: This step adds two new tests to cover the remaining uncovered combinations, as shown in Figure 2 (c). After horizontal growth, there are four combinations that have not been covered yet: {(P_1.0, P_3.2), (P_1.1, P_3.1), (P_2.0, P_3.1), (P_2.1, P_3.2)}. To cover (P_1.0, P_3.2), we add a new test (P_1.0, P_2.*, P_3.2), where P2.* indicates that the value of P2 is not determined yet. To cover (P_1.1, P_3.1), we add a new test (P_1.1, P_2.*, P_3.1). To cover (P_2.0, P_3.1), we change the value of P2 from * to 0 in the last test. To cover (P_2.1, P_3.2), we change the value of P2 from * to 1 in the 5th test. At this point, we have covered all the 2-way combinations, and thus have built a pairwise test set for the example system.

Due to space limitation, the readers are referred to (Lei 2008) for a detailed presentation of the IPO algorithm.

Generating a Combinatorial Test Suite with FireEye

The FireEye tool implements the IPOG algorithm to generate combinatorial test suites for users. Consider the input from Table 1 which has four parameters: *Log-in type, Member status, Discount, and Shipping*. Each of these parameters can take on one of three possible options. The FireEye tool can generate a combinatorial test suite for this input. Users enter the parameters and their possible values and FireEye then automatically generates a *t*-way test suite, where *t* is the strength of coverage that the user specifies. Figure 3 provides an example of the input from Table 1. The user entered the parameters and values on the left side of the window. The right side shows a summary of the parameters and values that have been entered. The user may additionally specify relations or constraints on other tabs from this window if there are combinations of parameters and values that can only occur together, or can not be combined together. The user may then choose to save the data that they enter on this screen and choose to generate a combinatorial test suite. In this example, we choose for FireEye to build a pairwise combinatorial test suite. The test suite is shown in Figure 4. (Note that the asterisks in the test cases are "don't care" values, meaning that a tester can use any option for a parameter and still cover all pairwise combinations.) The test suite can be saved in multiple formats for testing purposes. Figure 5 shows an subset of the test case from our example in XML format. The left side of the figure shows that the parameters and values are saved in a simple format and the right side shows a few tests in XML format.

IV. RESEARCH DIRECTIONS

In this section, we will discuss research directions of both algorithms and applications. We can

Figure 3. Example inputs to FireEye

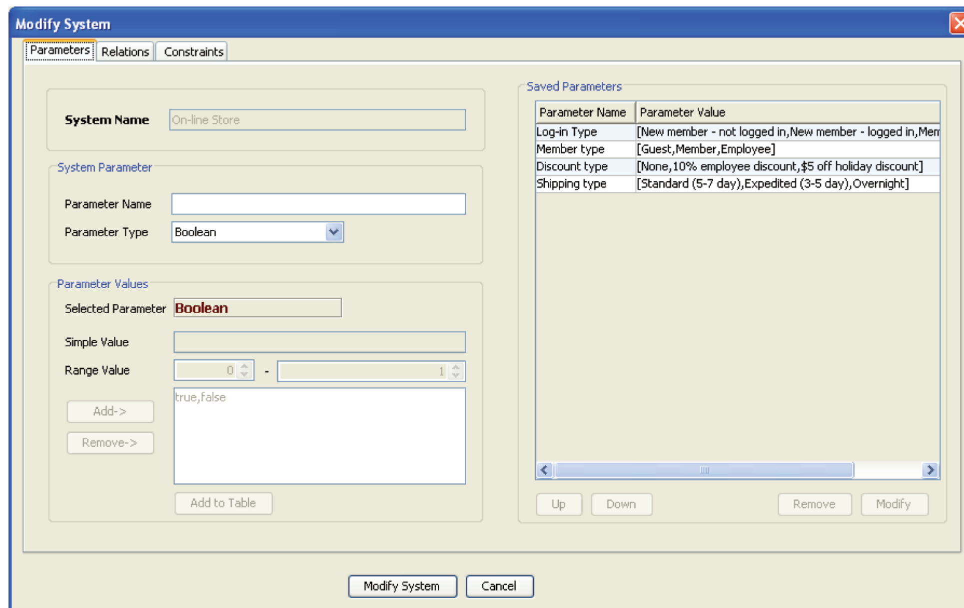
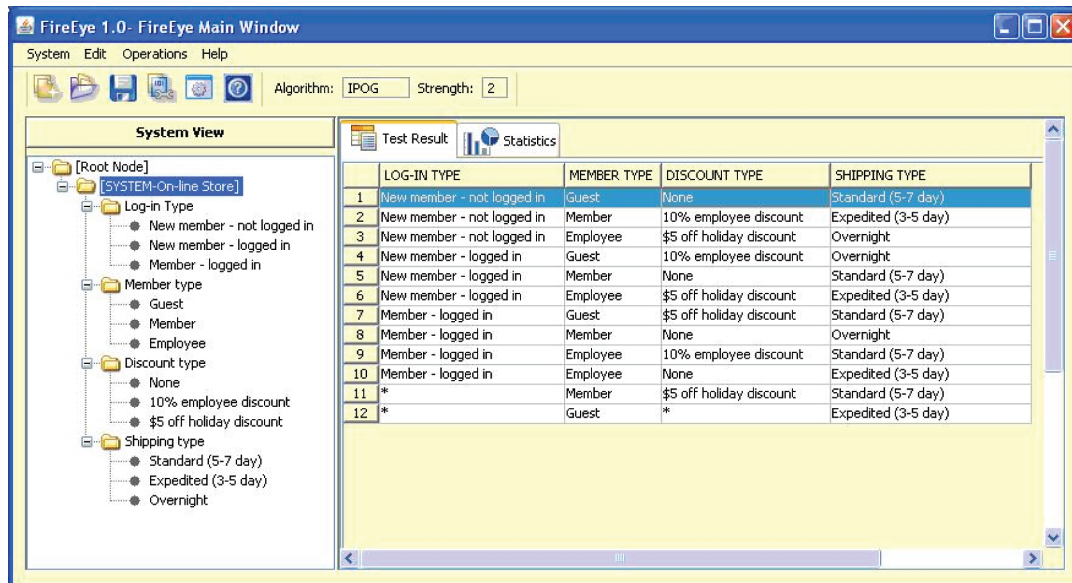


Figure 4. Example Combinatorial Test Suite created with FireEye



	LOG-IN TYPE	MEMBER TYPE	DISCOUNT TYPE	SHIPPING TYPE
1	New member - not logged in	Guest	None	Standard (5-7 day)
2	New member - not logged in	Member	10% employee discount	Expedited (3-5 day)
3	New member - not logged in	Employee	$5 off holiday discount	Overnight
4	New member - logged in	Guest	10% employee discount	Overnight
5	New member - logged in	Member	None	Standard (5-7 day)
6	New member - logged in	Employee	$5 off holiday discount	Expedited (3-5 day)
7	Member - logged in	Guest	$5 off holiday discount	Standard (5-7 day)
8	Member - logged in	Member	None	Overnight
9	Member - logged in	Employee	10% employee discount	Standard (5-7 day)
10	Member - logged in	Employee	None	Expedited (3-5 day)
11	*	Member	$5 off holiday discount	Standard (5-7 day)
12	*	Guest	*	Expedited (3-5 day)

categorize these issues as algorithms for *t*-way combinatorial testing and approaches to the application of combinatorial testing.

Algorithms for *t*-way Combinatorial Testing

Combinatorial test suites can also provide higher strength *t*-way coverage. Generating covering

Figure 5. An abbreviated example of a test case in XML format

Sample of Abbreviated Output of the first two parameters and values from the test suite in Figure 4, using XML formatted output	Sample of Abbrevieted Output of the first three test cases from the test suite in Figure 4, using XML formatted output
<?xml version="1.0" encoding="UTF-8" ?> - <System name="On-line Store"> - <Parameters> - <Parameter id="1" name="Log-in Type" type="1"> - <values> <value>New member - not logged in</value> <value>New member - logged in</value> <value>Member - logged in</value> </values> </Parameter> - <Parameter id="2" name="Member type" type="1"> - <values> <value>Guest</value> <value>Member</value> <value>Employee</value> </values> </Parameter> …	- <Testcase TCNo="0"> <Value>1</Value> <Value>New member - not logged in</Value> <Value>Guest</Value> <Value>None</Value> <Value>Standard (5-7 day)</Value> </Testcase> - <Testcase TCNo="1"> <Value>2</Value> <Value>New member - not logged in</Value> <Value>Member</Value> <Value>10% employee discount</Value> <Value>Expedited (3-5 day)</Value> </Testcase> - <Testcase TCNo="2"> <Value>3</Value> <Value>New member - not logged in</Value> <Value>Employee</Value> <Value>$5 off holiday discount</Value> <Value>Overnight</Value> </Testcase>

arrays of higher *t*-way coverage can consume significant computational resources and produce large results. For instance, Table 5 shows a sample of inputs and the combinatorial growth of tuples that occur as *t* increases. The input 3^{13} (read as 13 parameters have 3 possible values each) includes 702 pairs, 7,722 triples, and reaches over a million 6-tuples. As the size of the tuples and their number increase, the size of corresponding test suites increase. Managing this combinatorial growth with regard to both accuracy and execution time is still an open research issue.

The majority of algorithms for combinatorial testing focus on the special case of 2-way combinatorial testing. Two greedy algorithms recently appeared for *t*-way combinatorial testing (Bryce to appear; Lei 2008). However, the efficient generation of *t*-way combinatorial test suites remains an ongoing research topic.

Approaches for Combinatorial Testing

There are basically two approaches to combinatorial testing – use combinations of *configuration* parameter values, or combinations of *input* parameter values. In the first case, the covering array is used to select values of configurable parameters, possibly with the same tests run against all configuration combinations. For example, a

Table 5. A sample of the exponential growth of t-tuples as t increases

	10^4	3^{13}	11^{16}
t=2	600	702	14,520
t=3	4,000	7,722	745,360
t=4	10,000	57,915	26,646,620
t=5	-	312,741	703,470,768
t=6	10,000	1,250,954	1,301,758,600
t=k	10,000	1,594,323	45,949,729,863,572,200

server might be tested by setting up all 4-way combinations of configuration parameters such as number of simultaneous connections allowed, memory, OS, database size, etc., with the same test suite run against each configuration.

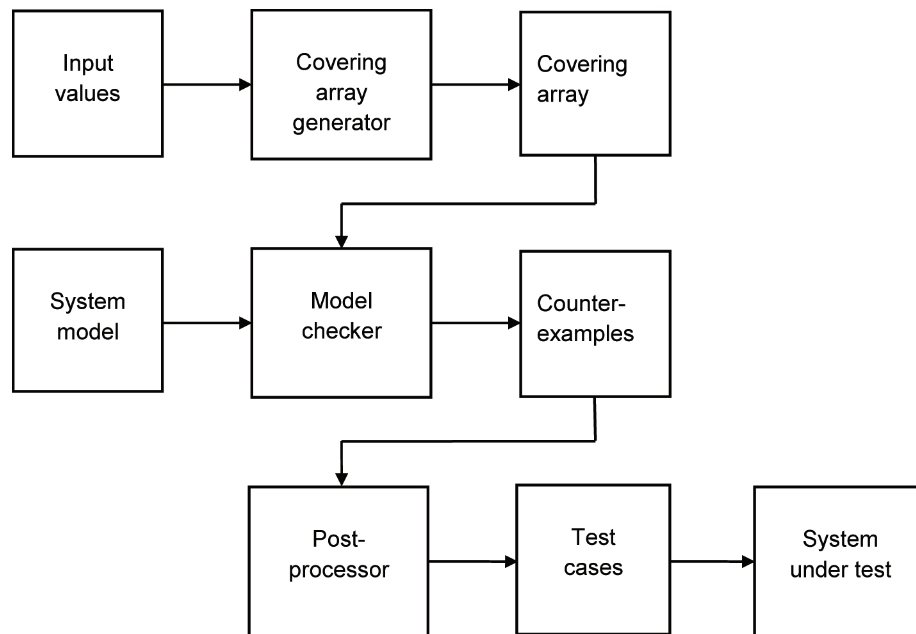
In the second approach, the covering array is used to select input data values, which then become part of complete test cases, creating a test suite for the application. Applying this form of combinatorial testing to real-world software presents a significant challenge: for higher degree interactions, a very large number of tests can be required. Thousands of tests may be needed to cover all 4-way to 6-way combinations for many typical applications, and for each test, the expected result from the application under test must be determined. Approaches to solving this "oracle problem" for combinatorial testing include:

Crash testing: the easiest and least expensive approach is to simply run tests against the system under test (SUT) to check whether any unusual combination of input values causes a crash or other easily detectable failure. This approach clearly produces limited 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. Crash testing using combinatorial methods can be an inexpensive yet thorough basic method of checking a system's reaction to rare input combinations that might take months or years to occur in normal operation.

Embedded assertions: An increasingly popular "light-weight formal methods" technique is to embed assertions within code to ensure proper relationships between data, for example as preconditions, post-conditions, or input value checks. Tools such as the Java Modeling language (JML) (Leavens 1999) can be used to introduce very complex 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 for testing smart cards detecting 80% - 90% of application faults (du Bousquet 2004).

Model-checker based test generation uses a mathematical model of the SUT and a model checker to generated expected results for each input. Conceptually, the model checker can be viewed as exploring all states of a system model to determine if a property claimed in a specification statement is true. What makes a model checker particularly valuable is that if the claim is false, the model checker not only reports this, but also provides a "counterexample" showing how the claim can be shown false. If the claim is false, the model checker indicates this and provides a trace of parameter input values and states that will prove it is false. In effect this is a complete test case, i.e., a set of parameter values and expected result. It is then simple to map these values into complete test cases in the syntax needed for the

Figure 6. A process for model-checker based test generation with covering arrays



system under test (Ammann 1999). This process is illustrated in Figure 6.

V. FUTURE AND INTERNATIONAL IMPACT

Combinatorial testing has attracted a lot of attention from both academia and industry. Several studies have indicated that combinatorial testing could dramatically reduce the number of tests while remaining effective for detecting software faults. Moreover, combinatorial testing is relatively easy to apply. As a black-box technique, combinatorial testing does not require analysis of source code, which is often difficult for practical applications. To apply combinatorial testing, a set of parameters, as well as their possible values, need to be identified. This information is often much easier to obtain than an operational model as required by many other black-box techniques. After the parameters and their values are identified, the actual test generation process can be fully automated, which is the key to industrial acceptance.

Combinatorial testing research has made significant progress in recent years, and continues to make progress every day, especially in the directions outlined in the previous section. With these progresses, combinatorial testing is expected to be fully integrated with the existing testing processes and become an important tool in the toolbox of software practitioners. The wide use of combinatorial testing will help to significantly reduce the cost of software testing while increasing software quality. It will also improve the productivity of software developers by reducing the time and effort they spend on testing.

VI. CONCLUSION

Software is growing in complexity. This poses a challenge to testers since there are often more combinations of system settings than there is time for testing. Combinatorial testing is an approach

that can systematically examine system settings in a manageable number of tests. This chapter provides a summary of the different types of algorithms that exist to efficiently generate tests. This provides readers not only with one example of how test suites can be constructed, but they are also pointed to a publicly available tool that generates tests. Further, we provide a discussion of empirical studies that reveal the effectiveness of combinatorial testing for different types of applications. These studies show how and when to apply the techniques, but also open questions for future research.

Certain commercial equipment or materials are identified in this paper. Such identification is not intended to imply recommendation or endorsement by NIST, nor is it intended to imply that the equipment or materials identified are necessarily the best available for the purpose.

REFERENCES

Ammann, P., & Black, P. E. (1999). Abstracting Formal Specifications to Generate Software Tests via Model Checking. *IEEE 18th Digital Avionics Systems Conference, 2*(10), 1-10.

Bell, K. Z. (2006). *Optimizing Effectiveness and Efficiency of Software Testing: a Hybrid Approach*. PhD Dissertation, North Carolina State University.

Bryce, R., & Colbourn, C. J. (2007). The Density Algorithm for Pairwise Interaction Testing. *Journal of Software Testing . Verification and Reliability*, *17*(3), 159–182. doi:10.1002/stvr.365

Bryce, R., & Colbourn, C. J. (in press). A Density-Based Greedy Algorithm for Higher Strength Covering Arrays. *Journal of Software Testing, Verification, and Reliability*.

Burr, K., & Young, W. (1998). Combinatorial test techniques: Table-based automation, test generation, and code coverage. *International Conference on Software Testing Analysis and Review* (pp. 503-513).

Cohen, D.M., Dalal, S. R., Fredman, M. L., & Patton, G. C. (1996). *Method and system for automatically generating efficient test cases for systems having interacting elements* [United States Patent, Number 5, 542, 043].

Cohen, D. M., Dalal, S. R., Fredman, M. L., & Patton, G. C. (1997). The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, *23*(7), 437–444. doi:10.1109/32.605761

Cohen, D. M., Dalal, S. R., Parelius, J., & Patton, G. C. (1996). The combinatorial design approach to automatic test generation. *IEEE Software*, *13*(5), 83–88. doi:10.1109/52.536462

Cohen, M. B., Colbourn, C. J., Gibbons, P. B., & Mugridge, W. B. (2003). Constructing test suites for interaction testing. *International Conference on Software Engineering* (pp. 38-48).

Cohen, M. B., Colbourn, C. J., & Ling, A. C. H. (2008). Constructing Strength 3 Covering Arrays with Augmented Annealing. *Discrete Mathematics*, *308*, 2709–2722. doi:10.1016/j.disc.2006.06.036

Colbourn, C. J. (2004). Combinatorial aspects of covering arrays. *Le Matematiche (Catania)*, *58*, 121–167.

Dalal, S. R., Jain, A., Karunanithi, N., Leaton, J. M., Lott, C. M., Patton, G. C., & Horowitz, B. M. (1999). Model-Based Testing in Practice. *International Conference on Software Engineering* (pp. 285-294).

du Bousquet, L., Ledru, Y., Maury, O., Oriat, C., & Lanet, J.-L. (2004). A case study in JML-based software validation. *IEEE International Conference on Automated Software Engineering* (pp. 294-297).

Dunietz, S. Ehrlich, W. K. Szablak, B. D. Mallows, C. L. & Iannino, A. (1997). Applying design of experiments to software testing. *International. Conference on Software Engineering* (pp. 205-215).

Fischer, R. A. (1926). The Arrangement of Field Experiments. *Journal of Ministry of Agriculture of Great Britain*, *33*, 503–513.

Hartman, A., Klinger, T., & Raskin, L. (2008). *IBM Intelligent Test Case Handler*. Retrieved on August 30, 2008, from http://www.alphaworks.ibm.com/tech/whitch

Hartman, A., & Raskin, L. (2004). Problems and algorithms for covering arrays. *Discrete Mathematics*, *284*, 149–156. doi:10.1016/j.disc.2003.11.029

Kuhn, R., & Reilly, M. (2002). An investigation of the applicability of design of experiments to software testing. *NASA Goddard/IEEE Software Engineering Workshop* (pp. 91-95).

Kuhn, R., Wallace, D., & Gallo, A. (2004). Software Fault Interactions and Implications for Software Testing. *IEEE Transactions on Software Engineering*, *30*(6), 418–421. doi:10.1109/TSE.2004.24

Leavens, G. T., Baker, A. L., & Ruby, C. (1999). JML: A notation for detailed design. In H. Kilov, B. Rumpe, & I. Simmonds, (Ed.) *Behavioral Specifications of Businesses and Systems.*

Lei, Y. Kacker, R. Kuhn, D. Okun, & V. Lawrence J. (2008). IPOG/IPOD: Efficient Test Generation for Multi-Way Combinatorial Testing. *Journal of Software Testing, Verification, and Reliability*, *18*(3), 125–148. doi:10.1002/stvr.381

Lei, Y., & Tai, K. C. (1998). In-parameter-order: a test generation strategy for pairwise testing. *International High-Assurance Systems Engineering Symposium* (pp. 254-261).

NIST. (2003). *The Economic Impacts of Inadequate Infrastructure for software testing.* Retrieved from http://www.nist.gov/director/prog-ofc/report02-3.pdf

Tai, K. C., & Lei, Y. (2002). A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering*, *28*(1), 109–111. doi:10.1109/32.979992

Tung, Y. W., & Aldiwan, W. S. (2000). Automating test case generation for the new generation mission software system. *IEEE Aerospace Conference* (pp. 431-437).

Wallace, D. R., & Kuhn, D. R. (2001). Failure Modes in Medical Device Software: an Analysis of 15 Years of Recall Data. *International Journal of Reliability Quality and Safety Engineering*, *8*(4). doi:10.1142/S021853930100058X

Williams, A., & Probert, R. L. (2001). A measure for component interaction test coverage. *ACS/IEEE International Conference on Computer Systems and Applications* (pp. 301-311).

Yilmaz, C., Cohen, M. B., & Porter, A. (2006). Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, *32*(1), 20–34. doi:10.1109/TSE.2006.8

KEY TERMS AND DEFINITIONS

Covering Array: $CA_{\lambda}(N;t,k,v)$, is an N x k array. In every N x t subarray, each t-tuple occurs at least λ times. In combinatorial testing, t is the *strength* of the coverage of interactions, k is the number of components (degree), and v is the number of symbols for each component.

Covering Array Number (CAN): The size of a covering array, or a mixed-level covering array and is considered optimal when it is as small as possible.

Pairwise Combinatorial Testing: An interaction test in which the strength, t, is equal to two. Pair-wise permutations of factors are executed during testing.

Pseudo-Exhaustive Testing: The term used when interaction testing is considered effectively exhaustive. Interaction testing at levels of 4-way to 6-way coverage have been suggested to be pseudoexhaustive (Kuhn, Reilly 2002)

***n*-way Combinatorial Testing:** An interaction test in which the strength, t, is equal to a specified value n. Permutations of n-tuples of factors are executed during testing