Preface

Software testing has always faced a seemingly intractable problem: for real-world programs, the number of possible input combinations can exceed the number of atoms in the ocean, so as a practical matter it is impossible to show through testing that the program works correctly for all inputs. Combinatorial testing offers a (partial) solution. Empirical data show that the number of variables involved in failures is small. Most failures are triggered by only one or two inputs, and the number of variables interacting tails off rapidly, a relationship called the *interaction rule*. Therefore, if we test input combinations for even small numbers of variables, we can provide very strong testing at low cost. As always, there is no "silver bullet" answer to the problem of software assurance, but combinatorial testing has grown rapidly because it works in the real world.

This book introduces the reader to the practical application of combinatorial methods in software testing. Our goal is to provide sufficient depth that readers will be able to apply these methods in their own testing projects, with pointers to freely available tools. Included are detailed explanations of how and why to use various techniques, with examples that help clarify concepts in all chapters. Sets of exercises or questions and answers are also included with most chapters. The text is designed to be accessible to an undergraduate student of computer science or engineering, and includes an extensive set of references to papers that provide more depth on each topic. Many chapters introduce some of the theory and mathematics of combinatorial methods. While this material is needed for thorough knowledge of the subject, testers can apply the methods using tools (many freely available and linked in the chapters) that encapsulate the theory, even without in-depth knowledge of the underlying mathematics.

We have endeavored to be as prescriptive as possible, but experienced testers know that standardized procedures only go so far. Engineering judgment is as essential in testing as in development. Because analysis of the input space is usually the most critical step in testing, we have devoted roughly a third of the book to it, in Chapters 3 through 6. It is in this phase that experience and judgment have the most bearing on the success of a testing project. Analyzing and modeling the input space is also a task that is easy to do poorly, because it is so much more complex than it first appears. Chapters 5 and 6 introduce systematic methods for dealing with this problem, with examples to illustrate the subtleties that make the task so challenging to do right.

Chapters 7 through 9 are central to another important theme of this book—combinatorial methods can be applied in many ways during the testing process, and can improve conventional test procedures not designed with these methods in mind. That is, we do not have to completely re-design our testing practices to benefit from combinatorial methods. Any test suite, regardless of how it is derived, provides some level of combinatorial coverage, so one way to use the methods introduced in this book is to create test suites using an organization's conventional procedures, measure their combinatorial coverage, and then supplement them with additional tests to detect complex interaction faults. The oracle problem—determining the correct output for a given test—is covered in Chapters 10 and 11. In addition to showing how formal models can be used as test oracles, Chapter 11 introduces an approach to integrating testing with formal specifications and proofs of properties by model checkers. Chapters 12 through 15 introduce advanced topics that can be useful in a wide array of problems. Except for the first four chapters, which introduce core terms and techniques, the chapters are designed to be reasonably independent of each other, and pointers to other sections for additional information are provided throughout.

The project that led to this book developed from joint research with Dolores Wallace, and we are grateful for that work and happy to recognize her contributions to the field of software engineering. Special thanks are due to Tim Grance for early and constant support of the combinatorial testing project. Thanks also go to Jim Higdon, Jon Hagar, Eduardo Miranda, and Tom Wissink for early support and evangelism of this

work, and especially Jim Lawrence who has been an integral part of the team since the beginning. Donna Dodson, Ron Boisvert, David Ferraiolo, and Lee Badger at NIST (US National Institute of Standards and Technology) have been strong advocates for this work. Jon Hagar provided many recommendations for improving the text. Mehra Borazjany, Michael Forbes, Itzel Dominguez Mendoza, Tony Opara, and Linbin Yu made major contributions to the software tools developed in this project. We have benefitted tremendously from interactions with researchers and practitioners, including Bob Binder, Paul Black, Renee Bryce, Myra Cohen, Charles Colbourn, Howard Deiner, Elfriede Dustin, Mike Ellims, Al Gallo, Vincent Hu, Justin Hunter, Aditya Mathur, Josh Maximoff, Carmelo Montanez-Rivera, Michael Reilly, Jenise Reyes Rodriguez, Rick Rivello, Sreedevi Sampath, Itai Segall, Mike Trela, Sergiy Vilkomir, and Tao Xie. We also gratefully acknowledge NIST SURF (Summer Undergraduate Research Fellowships) students William Goh, Evan Hartig, Menal Modha, Kimberley O'Brien-Applegate, Michael Reilly, Malcolm Taylor, and Bryan Wilkinson who contributed to the software and methods described in this document. We are especially grateful to Randi Cohen, editor at Taylor & Francis, for making this book possible and for timely guidance throughout the process. Certain software products are identified in this document, but such identification does not imply recommendation by the US National Institute for Standards and Technology, nor does it imply that the products identified are necessarily the best available for the purpose.

Appendix A – MATHEMATICS REVIEW

This appendix reviews a few basic facts of combinatorics and regular expressions that are necessary to understand the concepts in this publication.

Combinatorics

Permutations and Combinations

For n variables, there are $n!$ permutations and $\binom{n}{t} = \frac{n!}{t!(n-t)!}$ ("n choose t") combinations of t variables, also written for convenience as $C(n, t)$. To exercise all of the t-way combinations of inputs to a program, we need to cover all t-way combinations of variable values, and each combination of t values can have v^t configurations, where v is the number of values per variable. Thus the total number of combinations instantiated with values that must be covered is

$$v^t \binom{n}{t} \qquad (1)$$

Fortunately, each test covers $C(n, t)$ combination configurations. This fact is the source of combinatorial testing's power. For example, with 34 binary variables, we would need $2^{34} = 1.7 * 10^{10}$ tests to cover all possible configurations, but with only 33 tests we can cover all 3-way combinations of these 34 variables. This happens because each test covers C(34, 3) combinations.

Example. If we have five binary variables, *a, b, c, d,* and *e,* then expression (1) says we will need to cover $2^3 * C(5, 3) = 8*10 = 80$ configurations. For 3-way combinatorial testing, we will need to take all 3-variable combinations, of which there are 10:

abc, abd, abe, acd, ace, ade, bcd, bce, bde, cde

Each of these will need to be instantiated with all 8 possible configurations of three binary variables:

000, 001, 010, 011, 100, 101, 110, 111

The test [0 1 0 0 1] covers the following C(5, 3) = 10 configurations:

abc	*abd*	*abe*	*acd*	*ace*	*ade*	*bcd*	*bce*	*bde*	*cde*
010	000	011	001	001	001	100	101	101	001

Orthogonal Arrays

Many software testing problems can be solved with an orthogonal array, a structure that has been used for combinatorial testing in fields other than software for decades. An orthogonal array, $OA_\lambda(N; t, k, v)$ is an N x k array. In every N x t subarray, each t-tuple occurs exactly λ times. We refer to t as the *strength* of the coverage of interactions, k as the number of parameters or components (degree), and v as the number of possible values for each parameter or component (order).

Example. Suppose we have a system with three on-off switches, controlled by an embedded processor. The following table tests all pairs of switch settings exactly once each. Thus $t = 2$, $\lambda = 1$, $v = 2$. Note that there are $v^t = 2^2$ possible combinations of values for each pair: 00, 01, 10, 11. There are C(3,2) = 3 ways

to select switch pairs: (1,2), (1,3), and (2,3), and each test covers three pairs, so the four tests cover a total of 12 combinations which implies that each combination is covered exactly once. As one might suspect, it can be very challenging to fit all combinations to be covered into a set of tests exactly the same number of times.

Test	Sw 1	Sw 2	Sw 3
1	0	0	0
2	0	1	1
3	1	0	1
4	1	1	0

Covering Arrays

An alternative to an orthogonal array is a set called a *covering array*, which includes all *t*-way combinations of parameter values, for the desired strength *t*. A covering array, $CA_\lambda(N;t,k,v)$, is an $N \times k$ array. In every $N \times t$ subarray, each *t*-tuple occurs *at least* λ times. Note this distinction between covering arrays and orthogonal arrays discussed in the previous section. The covering array relaxes the restriction that each combination is covered exactly the same number of times. Thus covering arrays may result in some test duplication, but they offer the advantage that they can be computed for much larger problems than is possible for orthogonal arrays. Software described elsewhere in this book can efficiently generate covering arrays up to strength $t = 6$, for a large number of variables.

The problems discussed in this publication deal only with the case when $\lambda = 1$, (i.e. that every *t*-tuple must be covered at least once). In software testing, each row of the covering array represents a test, with one column for each parameter that is varied in testing. Collectively, the rows of the array include every *t*-way combination of parameter values at least once. For example, Figure 1 shows a covering array that includes all 3-way combinations of binary values for 10 parameters. Each column gives the values for a particular parameter. It can be seen that any three columns in any order contain all eight possible combinations of the parameter values. Collectively, this set of tests will exercise all 3-way combinations of input values in only 13 tests, as compared with 1,024 for exhaustive coverage.

0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	0	0	0	0	1
1	0	1	1	0	1	0	1	0	0
1	0	0	0	1	1	1	0	0	0
0	1	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0
1	1	0	1	0	0	1	0	1	0
0	0	0	1	1	1	0	0	1	1
0	0	1	1	0	0	1	0	0	1
0	1	0	1	1	0	0	1	0	0
1	0	0	0	0	0	0	1	1	1
0	1	0	0	0	1	1	1	0	1

Figure 1. 3-way covering array for 10 parameters with 2 values each.

Number of Tests Required

The challenge in computing covering arrays is to find the smallest possible array that covers all

configurations of t variables. If every new test generated covered all previously uncovered combinations, then the number of tests needed would be

$$\frac{v^t \binom{n}{t}}{\binom{n}{t}} = v^t$$

Since this is not generally possible, the covering array will be significantly larger than v^t, but still a reasonable number for testing. It can be shown that the number of tests in a t-way covering array will be proportional to

$$v^t \log n \qquad (2)$$

for n variables with v values each.

It's worth considering the components of this expression to gain a better understanding of what will be required to do combinatorial testing. First, note that the number of tests grows exponentially with the interaction strength t. The number of tests required for t+1-way testing will be in the neighborhood of v times the number required for t-way testing. The table below shows how v^t, grows for values of v and t. Although the number of tests required for high-strength combinatorial testing can be very large, with advanced software and cluster processors it is not out of reach.

v↓ t→	2	3	4	5	6
2	4	8	16	32	64
4	16	64	256	1024	4096
6	36	216	1296	7776	46656

Table 1. Growth of v^t

Despite the possibly discouraging numbers in the table above, there is some good news. Note that formula (2) grows only logarithmically with the number of variables, n. This is fortunate for software testing. Early applications of combinatorial methods were typically involved with small numbers of variables, such as a few different types of crops or fertilizers, but for software testing, we must deal with tens, or in some cases hundreds of variables.

Regular Expressions

Regular expressions are formal descriptions of strings of symbols, which may represent text, events, characters, or other objects. They are developed within automata theory and formal languages, where it is shown that there are direct mappings between expressions and automata to process them, and are encountered in many areas within computer science. In combinatorial testing they may be encountered in sequence covering or in processing test input or output. Implementations vary, but standard syntax is explained below.

Expression Operators

Basic elements of regular expressions include:

|	"or" alternation. Ex: ab|ac matches "ab" or "ac"
?	0 or 1 of the preceding element. Ex: ab?c matches "ac" or "abc"
*	0 or more of the preceding element. Ex: ab* matches "a", "ab", "abb", "abbb" etc. + 1 or more of the preceding element. Ex: ab+ matches "ab", "abb", "abbb" etc.
()	grouping. Ex: (abc|abcd) matches "abc" or "abcd"
.	matches any single character. Ex: a.c matches "abc", "axc", "a@c" etc.
[ ]	matches any single character within brackets. Ex: [abc] matches "a" or "b" or "c". A range may also be specified. Ex: [a-z] matches any single lower case character. (This option depends on the character set supported.)
[^]	matches any single character that is not contained in the brackets. Ex: [^ab] matches any character except "a" or "b"
^	matches start position, i.e., before the first character
$	matches end position, i.e., after the last character

Combining Operators

The operators above can be combined with symbols to create arbitrarily complex expressions. Examples include:

.*a.*b.*c.*	"a" followed by "b" followed by "c" with zero or more symbols prior to "a", following "c", or interspersed with the three symbols
a|b*	null or "a" or zero or more occurrences of "b"
a+	equivalent to aa*

Many regular expression utilities such as *egrep* support a broader range of operators and features. Readers should consult documentation for *grep*, *egrep*, or other regular expression processors for detailed coverage of the options available on particular tools.

Appendix B - EMPIRICAL DATA ON SOFTWARE FAILURES

One of the most important questions in software testing is "how much is enough"? For combinatorial testing, this question includes determining the appropriate level of interaction that should be tested. That is, if some failure is triggered only by an unusual combination of more than two values, how many testing combinations are enough to detect all errors? What degree of interaction occurs in real system failures? This section summarizes what is known about these questions based on research by NIST and others [4, 7, 35, 36, 37, 69].

Table 1 below summarizes what we know from empirical studies of a variety of application domains, showing the percentage of failures that are triggered by the interaction of one to six variables. For example, 66% of the medical devices were triggered by a single variable value, and 97% were triggered by either one or two variables interacting. Although certainly not conclusive, the available data suggest that the number of interactions involved in system failures is relatively low, with a maximum from 4 to 6 in the six studies cited below. (Note: TCAS study used seeded errors, all others are "naturally occurring", * = not reported.)

Vars	Medical Devices	Browser	Server	NASA GSFC	Network Security	TCAS
1	66	29	42	68	17	*
2	97	76	70	93	62	53
3	99	95	89	98	87	74
4	100	97	96	100	98	89
5		99	96		100	100
6		100	100			

Table 1. Number of variables involved in triggering software failures

System	System type	Release stage	Size (LOC)
Medical Devices	Embedded	Fielded products	10^3 – 10^4 (varies)
Browser	Web browser	Development/ beta release	approx. 2 x 10^5
Server	HTTP server	Development/ beta release	approx. 10^5
NASA database	Distributed scientific database	Development, integration test	approx. 10^5
Network security	Network protocols	Fielded products	10^3 – 10^5 (varies)

Table 2. System characteristics

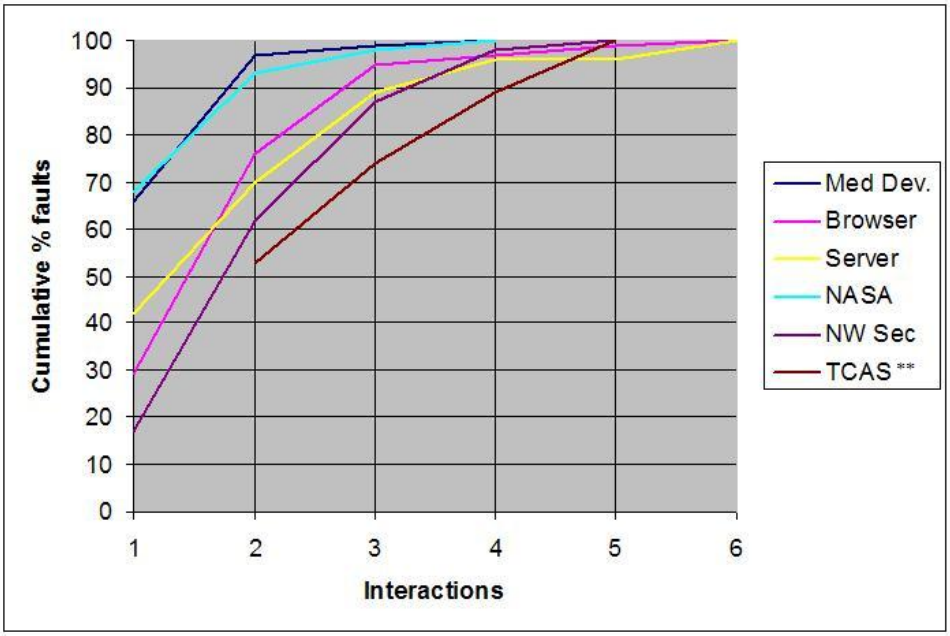

Figure 1. Cumulative percentage of failures triggered by *t*-way interactions.

We have also investigated a particular class of vulnerabilities, denial-of-serivce, using reports from the National Vulnerability Database (NVD), a publicly available repository of data on all publicly reported software security vulnerabilities. NVD can be queried for fine-granularity reports on vulnerabilities. Data from 3,045 denial-of-service vulnerabilities have the distribution shown in Table 3. We present this data separately from that above because it covers only one particular kind of failure, rather than data on any failures occurring in a particular program as shown in Figure 1.

Vars	NVD cumulative %
1	93%
2	99%
3	100%
4	100%
5	100%
6	100%

Table 3. Cumulative percentage of denial-of-service vulnerabilities triggered by *t*-way interactions.

Why do the failure detection curves look this way? That is, why does the error rate tail off so rapidly with more variables interacting? One possibility is that there are simply few complex interactions in branching points in software. If few branches involve 4-way, 5-way, or 6-way interactions among variables, then this degree of interaction could be rare for failures as well. The table below (Table 4 and Fig. 2) gives the number and percentage of branches in avionics code triggered by one to 19 variables. This distribution was developed by analyzing data in a report on the use of MCDC testing in avionics software [17], which contains 20,256 logic expressions in five different airborne systems in two different airplane models. The table below includes all 7,685 expressions from *if* and *while* statements;

expressions from assignment (:=) statements were excluded.

Table 4. Number of variables in avionics software branches

Vars	Count	Pct	Cumulative
1	5691	74.1%	74.1%
2	1509	19.6%	93.7%
3	344	4.5%	98.2%
4	91	1.2%	99.3%
5	23	0.3%	99.6%
6	8	0.1%	99.8%
7	6	0.1%	99.8%
8	8	0.1%	99.9%
9	3	0.0%	100.0%
15	1	0.0%	100.0%
19	1	0.0%	100.0%

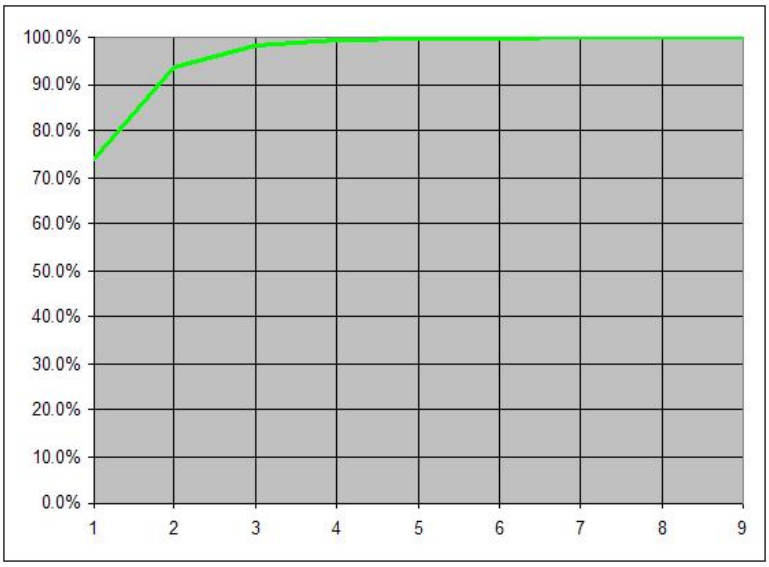

Figure 2. Cumulative percentage of branches containing *n* variables.

As shown in Fig. 2, most branching statement expressions are simple, with over 70% containing only a single variable. Superimposing the curve from Fig. 2 on Fig. 1, we see (Fig. 3) that most failures are triggered by more complex interactions among variables. It is interesting that the NASA distributed database failures, from development-phase software bug reports, have a distribution similar to expressions in branching statements. This distribution may be because this was development-phase rather than fielded software like all other types reported in Fig. 1. As failures are removed, the remaining failures may be harder to find because they require the interaction of more variables. Thus testing and use may push the curve down and to the right.

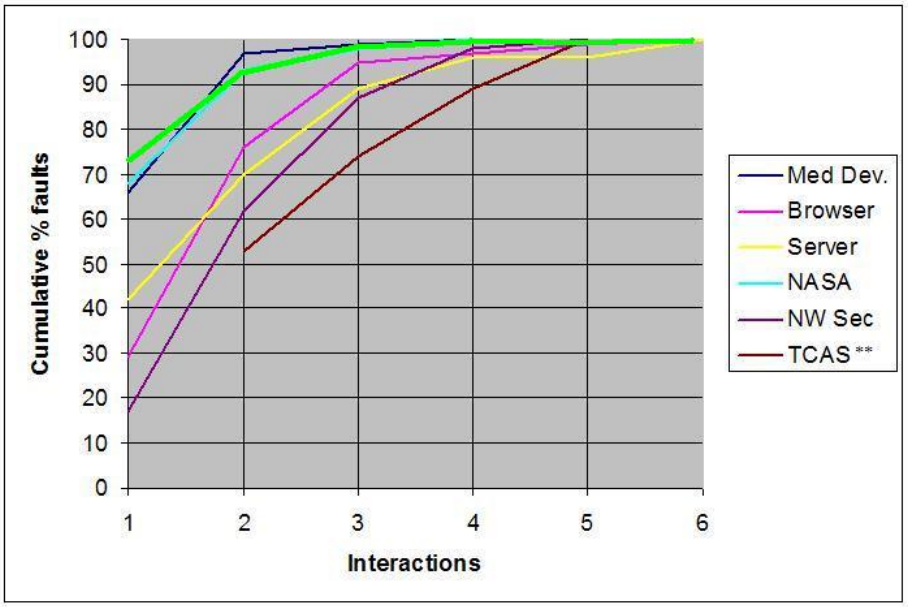

Figure 3. Branch distribution (green) superimposed on Fig. 1.