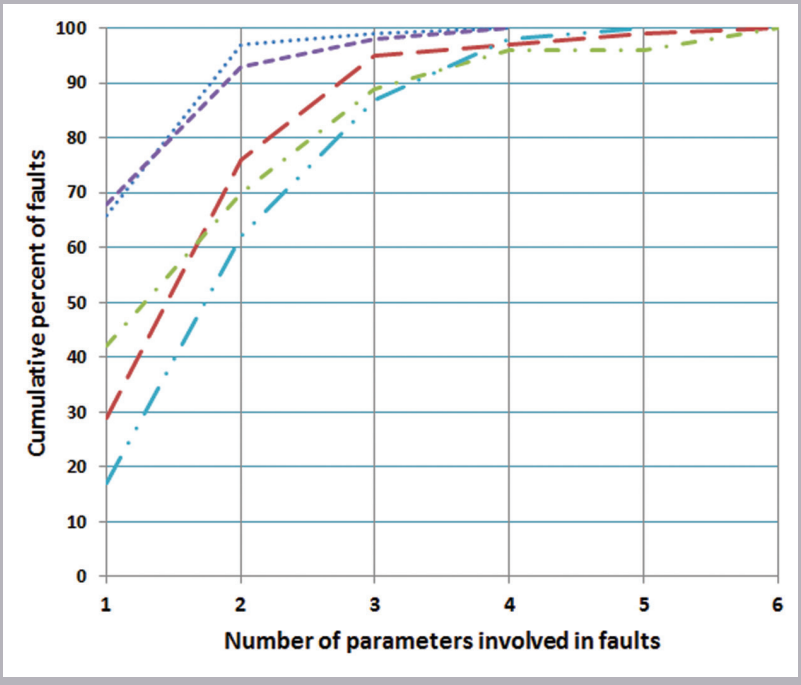
CHAPMAN & HALL/CRC INNOVATIONS IN
SOFTWARE ENGINEERING AND SOFTWARE DEVELOPMENT

Introduction to Combinatorial Testing



D. Richard Kuhn
Raghu N. Kacker
Yu Lei

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

Introduction to Combinatorial Testing

Chapman & Hall/CRC Innovations in Software Engineering and Software Development

Series Editor
Richard LeBlanc
Chair, Department of Computer Science and Software Engineering, Seattle University

AIMS AND SCOPE

This series covers all aspects of software engineering and software development. Books in the series will be innovative reference books, research monographs, and textbooks at the undergraduate and graduate level. Coverage will include traditional subject matter, cutting-edge research, and current industry practice, such as agile software development methods and service-oriented architectures. We also welcome proposals for books that capture the latest results on the domains and conditions in which practices are most effective.

PUBLISHED TITLES

Software Development: An Open Source Approach
Allen Tucker, Ralph Morelli, and Chamindra de Silva

Building Enterprise Systems with ODP: An Introduction to Open Distributed Processing
Peter F. Linington, Zoran Milosevic, Akira Tanaka, and Antonio Vallecillo

Software Engineering: The Current Practice
Václav Rajlich

Fundamentals of Dependable Computing for Software Engineers
John Knight

Introduction to Combinatorial Testing
D. Richard Kuhn, Raghu N. Kacker, and Yu Lei

CHAPMAN & HALL/CRC INNOVATIONS IN
SOFTWARE ENGINEERING AND SOFTWARE DEVELOPMENT

Introduction to Combinatorial Testing

D. Richard Kuhn
Raghu N. Kacker
Yu Lei

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group an **informa** business
A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Version Date: 20130426

International Standard Book Number-13: 978-1-4665-5230-2 (eBook - PDF)

Contents

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. Donna Dodson, Ron Boisvert, Geoffrey McFadden, David Ferraiolo, and Lee Badger at NIST (U.S. 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, Linbin Yu, Wenhua Wang, and

Laleh SH. Ghandehari 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, Greg Hutto, Aditya Mathur, Josh Maximoff, Carmelo Montanez-Rivera, Jeff Offutt, Vadim Okun, 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 Kevin Dela Rosa, 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 U.S. National Institute for Standards and Technology, nor does it imply that the products identified are necessarily the best available for the purpose.

Authors

D. Richard Kuhn is a computer scientist in the Computer Security Division of the National Institute of Standards and Technology (NIST). He has authored or coauthored more than 100 publications on information security, empirical studies of software failure, and software assurance, and is a senior member of the IEEE. He co-developed the role-based access control model (RBAC) used throughout the industry and led the effort establishing RBAC as an ANSI (American National Standards Institute) standard. Before joining NIST, he worked as a systems analyst with NCR Corporation and the Johns Hopkins University Applied Physics Laboratory. He earned an MS in computer science from the University of Maryland College Park, and an MBA from the College of William & Mary.

Raghu N. Kacker is a senior researcher in the Applied and Computational Mathematics Division (ACMD) of the Information Technology Laboratory (ITL) of the U.S. National Institute of Standards and Technology (NIST). His current interests include software testing and evaluation of the uncertainty in outputs of computational models and physical measurements. He has a PhD in statistics and has coauthored over 100 refereed papers. Dr. Kacker has been elected Fellow of the American Statistical Association and a Fellow of the American Society for Quality.

Yu Lei is an associate professor in Department of Computer Science and Engineering at the University of Texas, Arlington. He earned his PhD from North Carolina State University. He was a member of the Technical Staff in Fujitsu Network Communications, Inc. from 1998 to 2002. His current research interests include automated software analysis and testing, with a special focus on combinatorial testing, concurrency testing, and security testing.

Note of Appreciation

Two people who have made major contributions to the methods introduced in this book are James Lawrence of George Mason University, and James Higdon of Jacobs Technology, Eglin AFB. Jim Lawrence has been an integral part of the team since the beginning, providing mathematical advice and support for the project. Jim Higdon was co-developer of the sequence covering array concept described in Chapter 10, and has been a leader in practical applications of combinatorial testing.

Nomenclature

n = number of variables or parameters in tests

t = interaction strength; number of variables in a combination

N = number of tests

v_i = number of values for variable i

CA(N, n, v, t) = t-way covering array of N rows for n variables with v values each

CAN(t, n, v) = number of rows in a t-way covering array of n variables with v values each

OA(N, v^k, t) = t-way orthogonal array of entries from the set {0, 1, …, (v − 1)}

$C(n,t) = \binom{n}{t} = \frac{n!}{t!(n-t)!}$, the number of t-way combinations of n parameters

SCA(N, S, t) = an $N \times S$ sequence covering array where entries are from a finite set S of symbols, such that every t-way permutation of symbols from S occurs in at least one row

(p, t)-completeness = proportion of the $C(n, t)$ combinations in a test array of n rows that have configuration coverage of at least p

Φ_t = proportion of combinations with full t-way variable-value configuration coverage

M_t = minimum t-way variable value configuration coverage for a test set

S_t = proportion of t-way variable value configuration coverage for a test set

F = set of t-way combinations in failing tests

P = set of t-way combinations in passing tests

F^+ = augmented set of combinations in failing tests

P^+ = augmented set of combinations in passing tests

CHAPTER 1

Combinatorial Methods in Testing

Developers of large software systems often notice an interesting phenomenon: usage of an application suddenly increases, and components that worked correctly for years develop previously undetected failures. For example, the application may have been installed with a different operating system (OS) or database management system (DBMS) than used previously, or newly added customers may have account records with combinations of values that have not occurred before. Some of these rare combinations trigger failures that have escaped previous testing and extensive use. Such failures are known as *interaction failures*, because they are only exposed when two or more input values interact to cause the program to reach an incorrect result.

1.1 SOFTWARE FAILURES AND THE INTERACTION RULE

Interaction failures are one of the primary reasons why software testing is so difficult. If failures only depended on one variable value at a time, we could simply test each value once, or for continuous-valued variables, one value from each representative range or equivalence class. If our application had inputs with v values each, this would only require a total of v tests—one value from each input per test. Unfortunately, the real world is much more complicated than this.

Combinatorial testing can help detect problems like those described above early in the testing life cycle. The key insight underlying t-way

combinatorial testing is that not every parameter contributes to every failure and most failures are triggered by a single parameter value or interactions between a relatively small number of parameters. For example, a router may be observed to fail only for a particular protocol when packet volume exceeds a certain rate, a 2-way interaction between protocol type and packet rate. Figure 1.1 illustrates how such a 2-way interaction may happen in code. Note that the failure will only be triggered when both *pressure* < *10* and *volume* > *300* are true. To detect such interaction failures, software developers often use "pairwise testing," in which all possible pairs of parameter values are covered by at least one test. Its effectiveness results from the fact that most software failures involve only one or two parameters.

Pairwise testing can be highly effective and good tools are available to generate arrays with all pairs of parameter value combinations. But until recently only a handful of tools could generate combinations beyond 2-way, and most that did could require impractically long times to generate 3-way, 4-way, or 5-way arrays because the generation process is mathematically complex. Pairwise testing, that is, 2-way combinations, is a common approach to combinatorial testing because it is computationally tractable and reasonably effective.

But what if some failure is triggered only by a very unusual combination of 3, 4, or more values? It is unlikely that pairwise tests would detect this unusual case; we would need to test 3- and 4-way combinations of values. But is testing all 4-way combinations enough to detect all errors? It is important to understand the way in which interaction failures occur in real systems, and the number of variables involved in these failure triggering interactions.

```
if (pressure < 10) {
      // do something
      if (volume > 300)  {
            faulty code!  BOOM!
      }
      else {
            good code, no problem
      }
}
else {
      // do something else
}
```

FIGURE 1.1 2-Way interaction failures are triggered when two conditions are true.

What degree of interaction occurs in real failures in real systems? Surprisingly, this question had not been studied when the National Institute of Standards and Technology (NIST) began investigating interaction failures in 1999. An analysis of 15 years of medical device recall data [212] included an evaluation of fault-triggering combinations and the testing that could have detected the faults. For example, one problem report said that "if device is used with old electrodes, an error message will display, instead of an equipment alert." In this case, testing the device with old electrodes would have detected the problem. Another indicated that "upper limit CO_2 alarm can be manually set above upper limit without alarm sounding." Again, a single test input that exceeded the upper limit would have detected the fault. Other problems were more complex. One noted that "if a bolus delivery is made while pumps are operating in the body weight mode, the middle LCD fails to display a continual update." In this case, detection would have required a test with the particular pair of conditions that caused the failure: bolus delivery while in body weight mode. One description of a failure manifested on a particular pair of conditions was "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." The most complex failure involved four conditions and was presented as "the error can occur when demand dose has been given, 31 days have elapsed, pump time hasn't been changed, and battery is charged."

Reviews of failure reports across a variety of domains indicated that all failures could be triggered by a maximum of 4-way to 6-way interactions [103–105,212] for the applications studied. As shown in Figure 1.2, the detection rate increased rapidly with interaction strength (the interaction level t in t-way combinations is often referred to as *strength*). With the NASA application, for example, 67% of the failures were triggered by only a single parameter value, 93% by 2-way combinations, and 98% by 3-way combinations. The detection rate curves for the other applications studied are similar, reaching 100% detection with 4-way to 6-way interactions. Studies by other researchers [14,15,74,222] have been consistent with these results.

Failures appear to be caused by interactions of only a few variables, so tests that cover all such few-variable interactions can be very effective.

These results are interesting because they suggest that, while pairwise testing is not sufficient, the degree of interaction involved in failures is

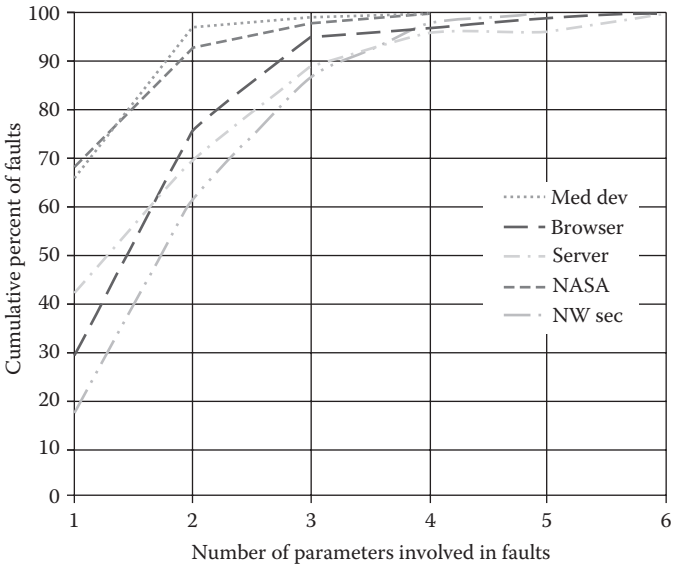


FIGURE 1.2 (**See color insert.**) The Interaction Rule: Most failures are triggered by one or two parameters interacting, with progressively fewer by 3, 4, or more.

relatively low. We summarize this result in what we call the *interaction rule*, an empirically derived [103–105] rule that characterizes the distribution of interaction faults:

> **Interaction Rule**: Most failures are induced by single factor faults or by the joint combinatorial effect (interaction) of two factors, with progressively fewer failures induced by interactions between three or more factors.

The maximum degree of interaction in actual real-world faults so far observed is six. This is not to say that there are no failures involving more than six variables, only that the available evidence suggests they are rare (more on this point below). Why is the interaction rule important? Suppose we somehow know that for a particular application, any failures can be triggered by 1-way, 2-way, or 3-way interactions. That is, there are some failures that occur when certain sets of two or three parameters have particular values, but no failure that is only triggered by a 4-way interaction. In this case, we would want a test suite that covers all 3-way combinations of parameter values (which automatically guarantees 2-way coverage as well). If there are some 4-way interactions that are not covered, it will

not matter from a fault detection standpoint, because none of the failures involve 4-way interactions. Therefore in this example, covering all 3-way combinations is in a certain sense equivalent to exhaustive testing. It will not test all possible inputs, but those inputs that are not tested would not make any difference in finding faults in the software. For this reason, we sometimes refer to this approach as "pseudo-exhaustive" [103], analogous to the digital circuit testing method of the same name [131,200]. The obvious flaw in this scenario is our assumption that we "somehow know" the maximum number of parameters involved in failures. In the real world, there may be 4-way, 5-way, or even more parameters involved in failures, so our test suite covering 3-way combinations might not detect them. But if we can identify a practical limit for the number of parameters in combinations that must be tested, and this limit is not too large, we may actually be able to achieve the "pseudo-exhaustive" property. This is why it is essential to understand interaction faults that occur in typical applications.

Some examples of such interactions were described previously for medical device software. To get a better sense of interaction problems in real-world software, let us consider some examples from an analysis of over 3000 vulnerabilities from the National Vulnerability Database, which is a collection of all publicly reported security issues maintained by NIST and the Department of Homeland Security:

- Single variable (1-way interaction): Heap-based *buffer_overflow* in the SFTP protocol handler for Panic Transmit … allows remote attackers to execute arbitrary code via a long ftps:// URL.
- 2-Way interaction: *Single character search string* in conjunction with a *single character replacement string*, which causes an "off by one overflow."
- 3-Way interaction: Directory traversal vulnerability when *register_globals is enabled* and *magic_quotes is disabled* and.. *(dot dot) in the page parameter.*

The single-variable case is a common problem: someone forgot to check the length of an input string, allowing an overflow in the input buffer. A test set that included any test with a sufficiently long input string would have detected this fault. The second case is more complex, and would not necessarily have been caught by many test suites. For example, a requirements-based test suite may have included tests to ensure that the software was

capable of accepting search strings of 1 to *N* characters, and others to check the requirement that 1 to *N* character replacement strings could be entered. But unless there was a single test that included *both* a one-character search string and a one-character replacement string, the application could have passed the test suite without detecting the problem. The 3-way interaction example is even more complex, and it is easy to see that an *ad hoc* requirements-based test suite might be constructed without including a test for which all three of the italicized conditions were true. One of the key features of combinatorial testing is that it is designed specifically to find this type of complex problem, despite requiring a relatively small number of tests.

As discussed above, an extensive body of empirical research suggests that testing 2-way (pairwise) combinations is not sufficient, and a significant proportion of failures result from 3-way and higher strength interactions. This is an important point, since the only combinatorial method many testers are familiar with is pairwise/2-way testing, mostly because good algorithms to produce 3-way and higher strength tests were not available. Fortunately, better algorithms and tools now make high strength *t*-way tests possible, and one of the key research questions in this field is thus: What *t*-way combination strength interaction is needed to detect all interaction failures? (Keep in mind that not all failures *are* interaction failures—many result from timing considerations, concurrency problems, and other factors that are not addressed by conventional combinatorial testing.) As we have discussed, failures seen thus far in real-world systems seem to involve six or fewer parameters interacting. However, it is *not* safe to assume that there are no software failures involving 7-way or higher interactions. It is likely that there are some that simply have not been recognized. One can easily construct an example that could escape detection by *t*-way testing for any arbitrary value of *t*, by creating a complex conditional with $t + 1$ variables:

```
if (v1 && . . . && vt && vt+1) {/* bad code */}.
```

In addition, analysis of the branching conditions in avionics software shows up to 19 variables in some cases [42]. Experiments on using combinatorial testing to achieve code coverage goals such as line, block, edge, and condition coverage have found that the best coverage was obtained with 7-way combinations [163,188], but code coverage is not the same as fault detection. Our colleague Linbin Yu has found up to 9-way interactions in some conditional statements in the Traffic Collision Avoidance System software [216] that is often used in testing research, although 5-way

combinations were sufficient to detect all faults in this set of programs [103] (*t*-way tests always include some higher strength combinations, or the 9-way faults may also have been triggered by <9 variables). Because the number of branching conditions involving *t* variables decreases rapidly as *t* increases, it is perhaps not surprising that the number of failures decreases as well. The available empirical research on this issue is covered in more detail in a web page that we maintain [143], and summarized in Appendix B. Because failures involving more than six parameters have not been observed in fielded software, most combinatorial testing tools generate up to 6-way arrays.

Because of the interaction rule, ensuring coverage of all 3-way, possibly up to 6-way combinations may provide high assurance. As with most issues in software, however, the situation is not that simple. Efficient generation of test suites to cover all *t*-way combinations is a difficult mathematical problem that has been studied for nearly a century, although recent advances in algorithms have made this practical for most testing. An additional complication is that most parameters are continuous variables which have possible values in a very large range ($\pm 2^{31}$ or more). These values must be discretized to a few distinct values. Most glaring of all is the problem of determining the correct result that should be expected from the system under test (SUT) for each set of test inputs. Generating 1000 test data inputs is of little help if we cannot determine what SUT should produce as output for each of the 1000 tests.

With the exception of covering combinations, these challenges are common to all types of software testing, and a variety of good techniques have been developed for dealing with them. What has made combinatorial testing practical today is the development of efficient algorithms to generate tests covering *t*-way combinations, and effective methods of integrating the tests produced into the testing process. A variety of approaches introduced in this book can be used to make combinatorial testing a practical and effective addition to the software tester's toolbox.

Advances in algorithms have made combinatorial testing beyond pairwise finally practical.

Notes on terminology: we use the definitions below, following the Institute of Electrical and Electronics Engineers (IEEE) Glossary of Terms [97]. The term "bug" may also be used where its meaning is clear.

Error: A mistake made by a developer. This could be a coding error or a misunderstanding of requirements or specification.

Fault: A difference between an incorrect program and one that correctly implements a specification. An error may result in one or more faults.

Failure: A result that differs from the correct result as specified. A fault in code may result in zero or more failures, depending on inputs and execution path.

The acronym SUT (system under test) refers to the target of testing. It can be a function, a method, a complete class, an application, or a full system including hardware and software. Sometimes, a SUT is also referred as a test object (TO) or artifact under test (AUT). That is, SUT is not meant to imply only the system testing phase.

1.2 TWO FORMS OF 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, we select combinations of values of configurable parameters. For example, a 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, DBMS type, and others, with the same test suite run against each configuration. The tests may have been constructed using any methodology, not necessarily combinatorial coverage. The combinatorial aspect of this approach is in achieving combinatorial coverage of all possible *t*-way configuration parameter values. (Note that the terms *variable* and *factor* are often used interchangeably with *parameter* to refer to inputs to a function or a software program.)

Combinatorial testing can be applied to configurations, input data, or both.

In the second approach, we select combinations of *input data* values, which then become part of complete test cases, creating a test suite for the application. In this case, combinatorial coverage of input data values is required for tests constructed. A typical *ad hoc* approach to testing involves subject matter experts setting up use case scenarios, then selecting input values to exercise the application in each scenario, possibly

supplementing these tests with unusual or suspected problem cases. In the combinatorial approach to input data selection, a test data generation tool is used to cover all combinations of input values up to some specified limit. One such tool is automated combinatorial testing for software (ACTS) (described in Appendix C), which is available freely from NIST.

Aspects of both configuration testing and input parameter testing may appear in a great deal of practical testing. Both types may be applied for thorough testing, with combinations of input parameters applied to each configuration combination. In state machine approaches (Chapter 6), other variations appear—parameters are inputs that may determine the presence or absence of other parameters, or both program variables and states may be treated as test parameters. But a wide range of testing problems can be categorized as either configuration or input testing, and these approaches are analyzed in more detail in later chapters.

1.2.1 Configuration Testing

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 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.

For example, suppose we had an application that is intended to run on a variety of platforms comprised of five components: an operating system (Windows XP, Apple OS X, Red Hat Enterprise Linux), a browser (Internet Explorer, Firefox), protocol stack (IPv4, IPv6), a processor (Intel, AMD), and a database (MySQL, Sybase, Oracle), a total of $3 \times 2 \times 2 \times 2 \times 3 = 72$ possible platforms. With only 10 tests, shown in Table 1.1, it is possible to test every component interacting with every other component at least once, that is, all possible pairs of platform components are covered. While this gain in efficiency—10 tests instead of 72—is respectable, the improvement for larger test problems can be spectacular, with 2- and 3-way tests often requiring <1% of the tests needed for exhaustive testing. In general, the larger the problem, the greater the efficiency gain from combinatorial testing.

TABLE 1.1 Pairwise Test Configurations

Test	OS	Browser	Protocol	CPU	DBMS
1	XP	IE	IPv4	Intel	MySQL
2	XP	Firefox	IPv6	AMD	Sybase
3	XP	IE	IPv6	Intel	Oracle
4	OS X	Firefox	IPv4	AMD	MySQL
5	OS X	IE	IPv4	Intel	Sybase
6	OS X	Firefox	IPv4	Intel	Oracle
7	RHEL	IE	IPv6	AMD	MySQL
8	RHEL	Firefox	IPv4	Intel	Sybase
9	RHEL	Firefox	IPv4	AMD	Oracle
10	OS X	Firefox	IPv6	AMD	Oracle

1.2.2 Input Testing

Even if an application has no configuration options, some form of input will be processed. For example, a word-processing application may allow the user to select 10 ways to modify some highlighted text: *subscript, superscript, underline, bold, italic, strikethrough, emboss, shadow, small caps*, or *all caps*. The font-processing function within the application that receives these settings as input must process the input and modify the text on the screen correctly. Most options can be combined, such as bold and small caps, but some are incompatible, such as subscript and superscript.

Thorough testing requires that the font-processing function work correctly for all valid combinations of these input settings. But with 10 binary inputs, there are $2^{10} = 1024$ possible combinations. Fortunately, the empirical analysis reported above shows that failures appear to involve a small number of parameters, and that testing all 3-way combinations can often detect 90% or more of bugs. For a word-processing application, testing that detects better than 90% of bugs may be a cost-effective choice, but we need to ensure that all 3-way combinations of values are tested. To do this, or to construct the configuration tests shown in Table 1.1, we create a matrix that covers all t-way combinations of variable values, where $t = 2$ for the configuration problem described previously and $t = 3$ for the 10 binary inputs in this section. This matrix is known as a *covering array* [27,31,51,53,58,97,116,205].

How many t-way combinations must be covered in the array? Consider the example of 10 binary variables. There are $C(10, 2) = 45$ pairs of variables (*ab*, *ac*, *ad*,…). For each pair, the two binary variables can be assigned $2^2 = 4$