# 1 Combinatorial Methods in Testing

Developers of large software systems often notice an interesting phenomenon: if usage of an application suddenly increases, components that have been working correctly develop previously undetected failures. For example, the application may have been installed with a different OS or DBMS system from what was 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. 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 situation 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 more on the number of parameters interacting in failures, see Appendix B). 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 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 is based on the observation that most software failures involve only one or two parameters.

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

**Figure 1.** 2-way interaction failures are triggered when two conditions are true.

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, i.e. 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 very unlikely that pairwise tests would detect this unusual case; we would need to test 3-way 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.

What degree of interaction occurs in real failures in real systems? Surprisingly, this question had not been studied when NIST began investigating interaction failures in 1999. An analysis of 15 years of medical device recall data [190] 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 CO2 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 suggest that all failures could be triggered by a maximum of 4-way to 6-way interactions [91, 92, 93, 190]. As shown in Figure 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 to 6-way interactions. Studies by other researchers [14, 15, 61, 199] 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.*

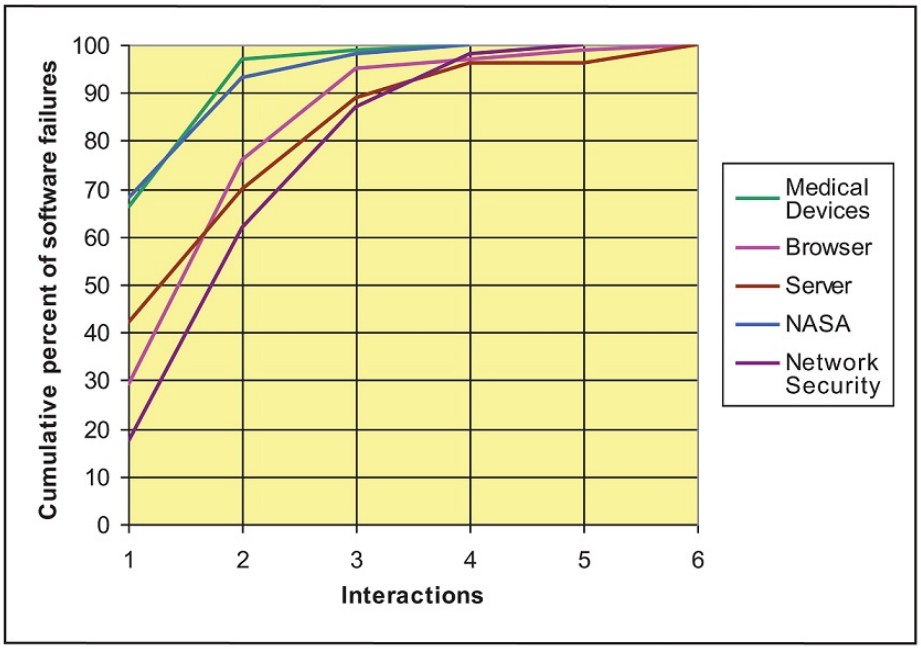



**Figure 2.** The Interaction Rule: Most failures are triggered by one or two parameters interacting, with progressively fewer by 3, 4, or more.

These results are interesting because they suggest that, while pairwise testing is not sufficient, the degree of interaction involved in failures is relatively low. We summarize this result in what we call the *interaction rule*, an empirically-derived [93, 92, 91] 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 all the failures are triggered by 1-way, 2-way, or 3-way interactions. Therefore in this example, covering all 3-way combinations is in a certain sense equivalent to exhaustive testing. It won't 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" [91], analogous to the digital circuit testing method of the same name [116, 179]. 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's consider some examples from an analysis of over 3,000 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, causing an overflow in the input buffer. A test set that included any test with a sufficiently long input string would have detected this failure. 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* 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 detection of the error. 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 underlined conditions were true. One of the key features of combinatorial testing is that it is specifically designed 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 many testers are familiar with 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 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.) Thus far, failures seen 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 [39]. 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 [141, 167], 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 [0] that is often used in testing research, although 5-way covering arrays were sufficient to detect all faults in this set of programs [91] (*t*-way tests always include some higher strength combinations, or the 9-way faults may also have been triggered by less than 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 [128], and summarized in **Error! Reference source not found.**. Because failures involving more than six parameters have not been observed in fielded software, most covering array 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 (+/- $2^{31}$ 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 for each set of test inputs. Generating 1,000 test data inputs is of little help if we cannot determine what the system under test (SUT) should produce as output for each of the 1,000 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 [85]. 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 TO (test object) or AUT (Artifact Under Test). 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 configuration parameter values. (Note, the terms *variable* and *factor* are often used interchangeably with *parameter* to refer to inputs to a function or a software program.)

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 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 ACTS (described in **Error! Reference source not found.**), which is available freely from NIST.

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

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 a covering array 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.

### *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.

Configuration coverage is perhaps the most developed form of combinatorial testing. It has been used for years with pairwise coverage, particularly for applications that must be shown to work across a variety of combinations of operating systems, databases, and network characteristics.

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 $3 \times 2 \times 2 \times 2 \times 2 = 48$ possible platforms. With only 10 tests, shown in Table 1, it is possible to test every component interacting with every other component at least once, i.e., all possible pairs of platform components are covered. While this gain in efficiency – 10 tests instead of 48 – is respectable, the improvement for larger test problems can be spectacular, with 2-way and 3-way tests often requiring less than 1% of the tests needed for exhaustive testing. In general, the larger the problem, the greater the efficiency gain from combinatorial testing.

| 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 |

Table 1. Pairwise test configurations

*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 $2^{10}$ = 1,024 possible combinations. But the empirical analysis reported above shows that failures appear to involve a small number of parameters, and that testing all 3-way combinations 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, 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* [25, 30, 49, 85, 103, 184].

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 possible values: 00, 01, 10, 11. So the number of 2-way combinations that must be covered in the array is $2^2 \times$ C(10, 2) = 4×45 = 180. For 3-way combinations, the variables can be assigned eight possible values: 000, 001, 010, …. Selecting three variables can be done in C(10, 3) = 120 ways, so there are $2^3 \times$C(10, 3) = 960 possible parameter settings to be covered. In general, there are $v^t$ $t$-way combinations of $v$ values, so for $n$ parameters we have

$$\textit{total combinations} = v^t \binom{n}{t}.$$

Generally not all parameters have the same number of test values. In combinatorics parlance, these are referred to as "mixed level" parameters. For $n$ different parameters, with $v_i$ values for the $i$th parameter, we need to cover:

$$\textit{total mixed level combinations} = \sum_i v_{i1} \times \ldots \times v_{it} \quad \forall\, i = 1..\binom{n}{t}\ t\text{-way combinations}$$

As we will see in the next section, a very large number of such combinations can be covered in remarkably few tests. Algorithms to compute covering arrays efficiently have been developed and are now implemented in practical tools.

### *1.3 Covering Arrays*

*The key component is a covering array, which includes all t-way combinations. Each column is a parameter. Each row is a test.*

An example of a covering array is given in Figure 3, which shows a 3-way covering array for 10 variables with two values each. The interesting property of this array is that any three columns contain all eight possible values for three binary variables. For example, taking columns F, G, and H, we can see that all eight possible 3-way combinations (000, 001, 010, 011, 100, 101, 110, 111) occur somewhere in the three columns together. In fact, any combination of three columns chosen in any order will also contain all eight possible values. Collectively, therefore, 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. Similar arrays can be generated to cover all $t$-way combinations, for whatever value of $t$ is appropriate to the problem.

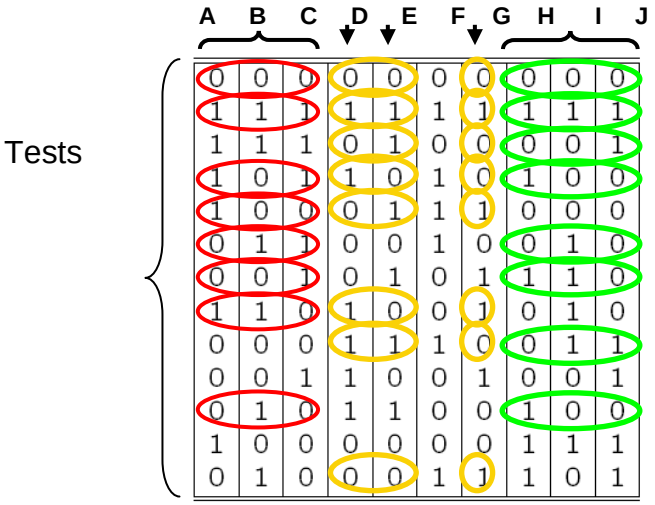



**Figure 3.** A 3-way covering array includes all 3-way combinations of values.

### *Covering Array Definition*

A covering array CA($N$, $n$, $s$, $t$) is an $N$ x $n$ matrix in which entries are from a finite set $S$ of $s$ symbols such that each $N$ x $t$ subarray contains each possible $t$-tuple at least once. For example, in the matrix above, we saw that all eight possible 3-tuples (3-way combinations) of the binary variables occurred at least once. The number $t$ is referred to as the *strength* of the array. A covering array must satisfy the $t$-covering property: when any $t$ of the $k$ columns are chosen, all $v^t$ of the possible $t$-tuples must appear

among the rows. The "size" of an array is usually given as its number $N$ of rows, where the number of columns is fixed.

This definition can be generalized to the case where $k_1$ columns have $v_1$ distinct values, $k_2$ columns have $v_2$ distinct values, and so on. A covering array with $n_1$ columns of $v_1$ distinct values, $n_2$ columns of $v_2$ distinct values, etc., is designated ${v_1}^{n1}\ {v_2}^{n2}\ \ldots\ v^{nk}$. Example: An array that has three columns with two distinct values each, two columns with 5 distinct values each, and four columns with six distinct values each is called a $2^3 5^2 6^4$ array. Note that if the columns represent nine parameters and their input values for a system under test, the number of tests required for exhaustive testing would be $2^3 5^2 6^4 = 259{,}200$ tests. The covering array in Fig. 3 is a $2^{10}$ array, since it has 10 columns of binary variables.

### *Size of Covering Arrays*

It is important to understand how covering array size is affected by the attributes of a testing problem to get a sense of how to apply combinatorial testing in practice. Since we are discussing tests and parameters the notation is a bit different than as used above in the formal definition of a covering array. It has been shown [52, 70] that in general, the number of rows (tests) for a covering array constructed with a greedy algorithm grows as

$$v^t \log n \qquad (1)$$

where
$v$ = number of possible values that each variable can take on.
$t$ = interaction strength, i.e., $t$-way interactions
$n$ =number of variables or parameters for the tests

When a covering array is produced, the number of tests will be proportional to this expression, not equal to it, but taking a look at the components of this expression will help in understanding how the characteristics of a testing problem affect the number of tests needed. This is a "good news/bad news" situation. The good news is that the number of tests increases only logarithmically with the number of parameters, $n$. Thus, testing systems with 50 inputs will not require significantly more tests than for 40 inputs. However, the bad news is that the number of tests increases exponentially with $t$, the interaction strength. So 4-way testing will be much more expensive than 3-way testing. Note another aspect of the first component, $v^t$, of expression (1). The exponent $t$ applies to $v$, the number of values that each variable can take on, so the value of $v$ can have an enormous effect on the number of tests.

Since many or most variables will be continuous-valued (within the limitations of digital hardware), values must be discretized from some range of integer or floating point numbers. The input range must be partitioned into a relatively small number of discrete values (see Sect. 4.1) to keep the number of tests to a minimum. In practice, it is generally a good idea to keep the number of values per variable to 10 or fewer. Figure 4 shows the number of tests required for 10 through 100 parameters for various values of $v$ for $t = 2$.

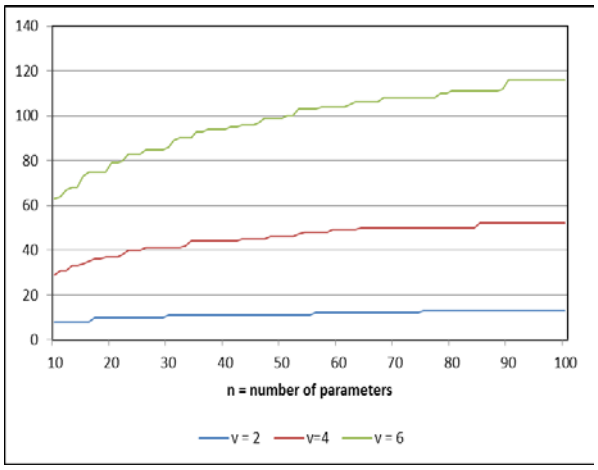



**Figure 4.** Number of tests, t = 2

There is no known formula for computing the smallest possible covering array for a particular problem. A database maintained by Charles Colbourn at Arizona State University collects the best known sizes of covering arrays for a broad range of configurations ranging from $t = 2$ to $t = 6$ (see http://www.public.asu.edu/~ccolbou/src/tabby/catable.html). Many algorithms have been developed for computing covering arrays, but there is no uniformly best algorithm, in the sense of computing the smallest possible array. Certain algorithms produce very compact arrays for some configurations, but perform poorly on others. More on algorithm design can be found in Chapter **Error! Reference source not found.**.

At this point it is important to point out that covering arrays are not the only way to produce combinatorial coverage. Any test set may cover a large number of parameter value combinations, and ways to measure such coverage are introduced in Chapter 7. As introduced previously in this chapter, the motivation for our interest in combinatorial methods is the empirical observation – the interaction rule – that a relatively small number of parameters interact in producing failures in real-world software. We thus want to cover in testing as many combinations as possible, and covering arrays are just one approach (although usually the most efficient). We can measure the combinatorial coverage of just about any test set, regardless of how it is produced. A combinatorial approach to testing is thus compatible with a broad range of test strategies, and this approach can improve testing in a variety of ways that will be introduced in this book.

## *1.4 The Test Oracle Problem*

Even with efficient algorithms to produce covering arrays, the oracle problem remains – testing requires both test data and results that should be expected for each data input. High interaction strength combinatorial testing may require a large number of tests in some cases, although not always. This section summarizes some approaches to solving the oracle problem that are particularly suited to automated or semi-automated combinatorial testing. Note that there are other test oracle methods as well. One of the most widely used approaches is of course to have human experts analyze test cases and determine the expected results. It is also possible that some or all of the functionality of the SUT will exist in another program. For example, the new code may be modifying one part of an existing program, so old tests may be re-used. In some cases, all of the functions may exist in another program whose results can be compared with the SUT, for example in an version that runs on another platform or a separate implementation of a compiler or network protocol standard. Here we summarize some approaches for the general case where the SUT presents all or mostly new functionality.

*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. Execution traces and memory dumps may then be analyzed to determine the cause of the crash. This is similar to the procedure used in some types of "fuzz testing" [159], which sends random values against the SUT. It should be noted that although pure random testing will generally cover a high percentage of *t*-way combinations, 100% coverage of combinations requires a random test set much larger than a covering array. For example, all 3-way combinations of 10 parameters with 4 values each can be covered with 151 tests. A purely random generation requires over 900 tests to provide full 3-way coverage.

*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, postconditions, or consistency checks. Tools such as the Java Modeling language (JML) 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, with embedded JML assertions acting as an oracle for combinatorial tests [57]. Results showed that 80% - 90% of failures could be found in this way.

*Several types of test oracle can be used, depending on resources and the system under test.*

*Model based test generation* uses a mathematical model of the SUT and a simulator or model checker to generate expected results for each input [1,16,18,118,134]. If a simulator can be used, expected results can be generated directly from the simulation, but model checkers are widely available and can also be used to prove properties such as liveness in parallel processes, in addition to generating tests. Conceptually, a 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 system under test. Chapter 12 develops detailed procedures for applying model based test oracle generation.

## *1.5 Quick Start – How to Use the Basics of Combinatorial Methods Right Away*

This book introduces a wide range of topics in combinatorial methods for software testing, sufficient for handling many practical challenges in software assurance. Most testers, however, will not face all of the types of test problems covered in this book, at least not on every project. Many test problems require a core set of methods, possibly with one or two specialized topics. As with many subjects, one of the best ways to approach combinatorial testing is to start small; try the basics to get a feel for how it works, then supplement these methods as needed. This book is designed for such an approach. Readers anxious to learn by applying some of the methods introduced here can use the following steps:

1. Read Chapter 1, to learn why combinatorial methods are effective and what to expect.
2. Read Chapter 3 and 4, for step-by-step approaches to input testing and configuration testing (as introduced in Section 1.2

3. Download and install the Java program ACTS or another covering array tool (see **Error! Reference source not found.**)
4. Develop a covering array of tests using ACTS or other tool, then run the tests.

After reading this chapter to understand why combinatorial testing works, readers can also review the two case studies in Chapter 2. These two testing problems are practical examples that illustrate the basics in situations that include many features of web application testing problems. Following the steps above is really just getting started, of course. But trying these methods on one of your own small testing problems will likely make the rest of the topics introduced in the book easier and more interesting to apply.

## *1.6 Chapter Summary*

1. Empirical data suggest that software failures are caused by the interaction of relatively few parameter values, and that the proportion of failures attributable to *t*-way interactions declines very rapidly with increase in *t*. That is, usually single parameter values or a pair of values are the cause of a failure, but increasingly smaller proportions are caused by 3-way, 4-way, and higher order interactions. This relationship is called the *Interaction Rule*.

2. Because a small number of parameters are involved in failures, we can attain a high degree of assurance by testing all *t*-way interactions, for an appropriate interaction strength *t* (2 to 6 usually). The number of *t*-way tests that will be required is proportional to $v^t \log n$, for $n$ parameters with $v$ values each.

3. A mathematical construct called a *covering array* can be used to produce tests that cover all *t*-way combinations. A covering array with $k_1$ columns of $v_1$ distinct values, $k_2$ columns of $v_2$ distinct values, etc., is designated $v_1^{k1}\ v_2^{k2}\ \ldots\ v_n^{kn}$, which is also equal to the number of tests that would be required for exhaustive testing. There is no "best" covering array construction algorithm, in the sense of always producing an optimal array.

4. As with all other types of testing, the oracle problem must be solved – i.e., for every test input, the expected output must be determined in order to check if the application is producing the correct result for each set of inputs. A variety of methods can be used to solve the oracle problem.

5. Combinatorial methods can be applied to *configurations* of the SUT or to *input values*, or in some cases both. Figure 5 contrasts the two approaches to combinatorial testing. With the first approach, we may run the same test set against all 3-way combinations of configuration options, while for the second approach, we would construct a test suite that covers all 3-way combinations of input transaction fields. Of course these approaches could be combined, with the combinatorial tests run against all the configuration combinations.

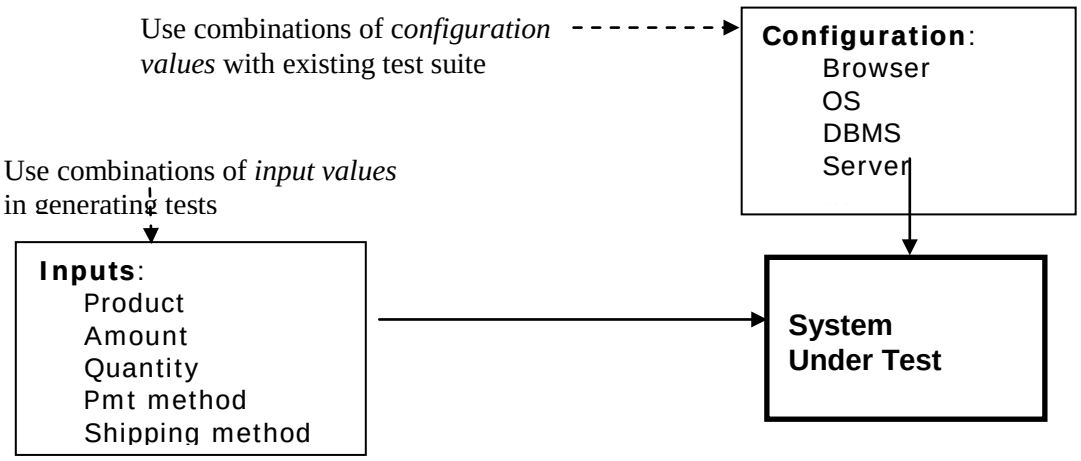



**Figure 5.** Combinatorial testing may be used on input values or configurations.

# 2 Combinatorial Testing Applied

# 3 Configuration Testing

The term "configuration" may be used in slightly different ways with respect to software. In some cases it may refer to options that are settable through an external file or other source. For example, a database management system may have configurable options for storage location and size, maximum size of various tables, key length, and other aspects of databases. These configurable options are read in when the system is initialized and used to set properties of the application. In other cases, configuration refers to characteristics of the platform on which the application is running, such as the presence or absence of a hard keyboard on a smartphone, the network protocol used, or the type of database. In this case, the configurable options are expected to provide essentially the same functions to the software – network interface or searchable storage – but low-level functions in the application must interface differently depending on the protocol or database in use. The software is built to operate correctly on a variety of platforms, and different parts of the code may be exercised depending on the configuration.

## *3.1 Runtime Environment Configurations*

One of the most common problems in software testing is assuring that an application can run on a variety of platforms. Different operating systems, web browsers, network protocols, or databases may be operated by customers, but developers would like to ensure that their software runs correctly on all platforms. An example illustrating the complexity of the problem occurred in July, 2012. A major antivirus program suffered crashes on certain configurations of Windows XP machines. According to a *Register* news article [104], *"Subsequent analysis has revealed that a three-way clash between third-party encryption drivers, Symantec's own security software and the Windows XP Cache manager resulted in the infamous Blue Screen of Death (BSOD) on vulnerable machines, as this advisory explains:*

> *"The root cause of the issue was an incompatibility due to a three-way interaction between some third-party software that implements a file system driver using kernel stack based file objects – typical of encryption drivers, the SONAR signature and the Windows XP Cache manager. The SONAR signature update caused new file operations that create the conflict and led to the system crash."*

Combinatorial testing of runtime configurations can help in catching this type of problem. While it is rarely practical to test all possible runtime platforms, methods described in this chapter can be used for efficient testing of all *t*-way combinations of platform configurations.

Returning to the simple example introduced in Chapter 1, we illustrate development of test configurations, and compare the size of test suites for various interaction strengths versus testing all possible configurations. For the five configuration parameters, we have $3 \times 2 \times 2 \times 2 \times 3 = 72$ configurations. Note that at $t = 5$, the number of tests is the same as exhaustive testing for this example, because there are only five parameters. The savings as a percentage of exhaustive testing are good, but not that impressive for this small example. With larger systems the savings can be enormous, as will be seen in the next section.

| Parameter | Values |
|---|---|
| Operating system | XP, OS X, RHL |
| Browser | IE, Firefox |
| Protocol | IPv4, IPv6 |
| CPU | Intel, AMD |
| DBMS | MySQL, Sybase, Oracle |

Table 2. Simple example configuration options.

After the parameters and possible values for each have been determined, a covering array can be generated using a software tool. In this book, the generation process will be illustrated using the ACTS covering array tool, which is described in more detail in **Error! Reference source not found.**, but other tools may have similar features. In addition to the summary in **Error! Reference source not found.**, a comprehensive user manual is included with the ACTS download.

The first step in creating test configurations is to specify the parameters and possible values, as shown in Figure 6. Another covering array tool or the GUI version of ACTS would of course have a different specification, but the essential features will be similar to Figure 6.

```
[System]

[Parameter]
OS (enum): XP,OS_X,RHL
Browser (enum): IE, Firefox
Protocol(enum): IPv4,IPv6
CPU (enum):  Intel,AMD
DBMS (enum): MySQL,Sybase,Oracle

[Relation]
[Constraint]
[Misc]
```

**Figure 6.** ACTS input includes parameter names, types, and possible values.

The degree of interaction must also be specified: 2-way, 3-way, etc. coverage. Output can be created as a matrix of numbers, comma separated value, or Excel spreadsheet format. If the output will be used by human testers rather than as input for further machine processing, the format in Figure 7 is useful.

The complete test set for 2-way combinations is shown in Table 1 in Section 1.3. Only 10 tests are needed. Moving to 3-way or higher interaction strengths requires more tests, as shown in Table 3.

| t | # Tests | % of Exhaustive |
|---|---|---|
| 2 | 10 | 14 |
| 3 | 18 | 25 |
| 4 | 36 | 50 |
| 5 | 72 | 100 |

Table 3. Number of combinatorial tests for a simple example.

In this example, substantial savings could be realized by testing *t*-way configurations instead of all possible configurations, although for some applications (such as a small but highly critical module) a full exhaustive test may be warranted. As we will see in the next example, in many cases it is impossible to test all configurations, so we need to develop reasonable alternatives.

## *3.2 Highly Configurable Systems and Software Product Lines*

*A software product line with n features may produce $2^n$ products.*

Software product lines are an increasingly attractive approach to application development. A software product line (SPL) uses standardized development procedures on systems that "share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets" [166]. The basic idea of a SPL is that enterprises, or their subunits, tend to produce families of software products for a particular application domain or market [76, 77, 129, 148]. For example, a company may develop software products for point-of-sale (POS) and retail store management. By combining software that implements various features, a wide variety of products can be provided with far less effort than traditional development approaches. In the retail store management example, a basic POS terminal application may allow for input from the cashier's keyboard or a laser scanner embedded in the checkout counter, while a more sophisticated terminal application may add features for a handheld scanner and a scale. Thus in some cases a product line can thus be viewed as a framework that can produce $2^n$ products, where there are $n$ different features [76]. With the high degree of customization and configurable feature sets, combinatorial testing can be especially effective when applied to SPLs [49, 50, 86, 135].

Telecommunications and mobile phone vendors have been among the early adopters of the SPL approach, with significant success [159]. Smart phones have become enormously popular because they combine communication capability with powerful graphical displays and processing capability. Literally tens of thousands of smart phone applications, or 'apps', are developed annually. Among the platforms for smart phone apps is the Android, which includes an open source development environment and specialized operating system. Android units contain a large number of configuration options that control the behavior of the device. Android apps must operate across a variety of hardware and software platforms, since not all products support the same options. For example, some smart phones may have a physical keyboard and others may present a soft keyboard using the touch sensitive screen. Keyboards may also be either only numeric with a few special keys, or a full typewriter keyboard. Depending on the state of the app and user choices, the keyboard may be visible or hidden. Ensuring that a particular app works across the enormous number of options is a significant challenge for developers. The extensive set of options makes it intractable to test all possible configurations, so combinatorial testing is a practical alternative.

```
Degree of interaction coverage: 2
Number of parameters: 5
Maximum number of values per parameter: 3
Number of configurations: 10
------------------------------------
Configuration #1:

1 = OS=XP
2 = Browser=IE
3 = Protocol=IPv4
4 = CPU=Intel
5 = DBMS=MySQL
------------------------------------
Configuration #2:

1 = OS=XP
2 = Browser=Firefox
3 = Protocol=IPv6
4 = CPU=AMD
5 = DBMS=Sybase
------------------------------------
Configuration #3:

1 = OS=XP
2 = Browser=IE
3 = Protocol=IPv6
4 = CPU=Intel
5 = DBMS=Oracle
------------------------------------
Configuration #4:

1 = OS=OS_X
2 = Browser=Firefox
3 = Protocol=IPv4
4 = CPU=AMD
5 = DBMS=MySQL
etc .
```

**Figure 7.** Excerpt of test configuration output covering all 2-way combinations.

Figure 8 shows a resource configuration file for Android apps. A total of 35 options may be set. Our task is to develop a set of test configurations that allow testing across all 4-way combinations of these options. The first step is to determine the set of parameters and possible values for each that will be tested. Although the options are listed individually to allow a specific integer value to be associated with each, they clearly represent sets of option values with mutually exclusive choices. For example, “Keyboard Hidden” may be “yes”, “no”, or “undefined”. These values will be the possible settings for parameter names that we will use in generating a covering array. Table 4 shows the parameter names and number of possible values that we will use for input to the covering array generator. For a complete specification of these parameters, see:
http://developer.android.com/reference/android/content/res/Configuration.html

```
int     HARDKEYBOARDHIDDEN_NO;
int     HARDKEYBOARDHIDDEN_UNDEFINED;
int     HARDKEYBOARDHIDDEN_YES;
int     KEYBOARDHIDDEN_NO;
```

```
int     KEYBOARDHIDDEN_UNDEFINED;
int     KEYBOARDHIDDEN_YES;
int     KEYBOARD_12KEY;
int     KEYBOARD_NOKEYS;
int     KEYBOARD_QWERTY;
int     KEYBOARD_UNDEFINED;
int     NAVIGATIONHIDDEN_NO;
int     NAVIGATIONHIDDEN_UNDEFINED;
int     NAVIGATIONHIDDEN_YES;
int     NAVIGATION_DPAD;
int     NAVIGATION_NONAV;
int     NAVIGATION_TRACKBALL;
int     NAVIGATION_UNDEFINED;
int     NAVIGATION_WHEEL;
int     ORIENTATION_LANDSCAPE;
int     ORIENTATION_PORTRAIT;
int     ORIENTATION_SQUARE;
int     ORIENTATION_UNDEFINED;
int     SCREENLAYOUT_LONG_MASK;
int     SCREENLAYOUT_LONG_NO;
int     SCREENLAYOUT_LONG_UNDEFINED;
int     SCREENLAYOUT_LONG_YES;
int     SCREENLAYOUT_SIZE_LARGE;
int     SCREENLAYOUT_SIZE_MASK;
int     SCREENLAYOUT_SIZE_NORMAL;
int     SCREENLAYOUT_SIZE_SMALL;
int     SCREENLAYOUT_SIZE_UNDEFINED;
int     TOUCHSCREEN_FINGER;
int     TOUCHSCREEN_NOTOUCH;
int     TOUCHSCREEN_STYLUS;
int     TOUCHSCREEN_UNDEFINED;
```

**Figure 8.** Android resource configuration file.

| Parameter Name | Values | # Values |
|---|---|---|
| HARDKEYBOARDHIDDEN | NO, UNDEFINED, YES | 3 |
| KEYBOARDHIDDEN | NO, UNDEFINED, YES | 3 |
| KEYBOARD | 12KEY, NOKEYS, QWERTY, UNDEFINED | 4 |
| NAVIGATIONHIDDEN | NO, UNDEFINED, YES | 3 |
| NAVIGATION | DPAD, NONAV, TRACKBALL, UNDEFINED, WHEEL | 5 |
| ORIENTATION | LANDSCAPE, PORTRAIT, SQUARE, UNDEFINED | 4 |
| SCREENLAYOUT_LONG | MASK, NO, UNDEFINED, YES | 4 |
| SCREENLAYOUT_SIZE | LARGE, MASK, NORMAL, SMALL, UNDEFINED | 5 |
| TOUCHSCREEN | FINGER, NOTOUCH, STYLUS, UNDEFINED | 4 |

Table 4. This set of Android options has 172,800 possible configurations.

Using Table 4, we can calculate the total number of configurations: $3\times3\times4\times3\times5\times4\times4\times5\times4$ = 172,800 configurations (i.e., a $3^3 4^4 5^2$ system). Like many applications, thorough testing will require some human intervention to run tests and verify results, and a test suite will typically include many tests. If each test suite can be run in 15 minutes, it will take roughly 24 staff-years to complete testing for an app. With salary and benefit costs for each tester of $150,000, the cost of testing an app will be more than $3 million, making it virtually impossible to return a profit for most apps. We saw in Section 0 that combinatorial methods can reduce the number of tests needed for strong assurance, but will the reduction in test set size be enough to provide effective testing for apps at a reasonable cost?

Using the covering array generator, we can produce tests that cover *t*-way combinations of values. Table 5 shows the number of tests required at several levels of *t*. For many applications, 2-way or 3-way testing may be appropriate, and either of these will require less than 1% of the time required to cover all possible test configurations. This example illustrates the power of combinatorial testing for real-world testing, and how its advantages increase with the size of the problem.

| t | # Tests | % of Exhaustive |
|---|---|---|
| 2 | 29 | 0.02 |
| 3 | 137 | 0.08 |
| 4 | 625 | 0.4 |
| 5 | 2532 | 1.5 |
| 6 | 9168 | 5.3 |

Table 5. The number of combinatorial tests is a fraction of an exhaustive test set.

## *3.3 Invalid Combinations and Constraints*

So far we have assumed that the set of possible values for parameters never changes. Thus a covering array of *t*-way combinations of possible values would contain combinations that either would occur in the systems under test, or could occur and must therefore be tested. But look more closely at the configurations in Figure 7. In practice, the Internet Explorer browser is never used on Linux systems, so it would be impossible to create a configuration that specified IE on a Linux system. This is an example of a constraint between possible values of parameters. Some combinations never occur in practice, or occur only sometimes. Practical testing requires consideration of constraints.

### *Constraints Among Parameter Values*

The system described earlier illustrates a common situation in all types of testing: some combinations cannot be tested because they don't exist for the systems under test. In this case, if the operating system is either OS X or Linux, Internet Explorer is not available as a browser. Note that we cannot simply delete tests with these untestable combinations, because that would result in losing other combinations that are essential to test but are not covered by other tests. For example, deleting tests 5 and 7 in Section 0 would mean that we would also lose the test for Linux with the IPv6 protocol.

*Some combinations never occur in practice.*

One way around this problem is to delete tests and supplement the test suite with manually constructed test configurations to cover the deleted combinations, but covering array tools offer a better solution. With ACTS we can specify constraints, which tell the tool not to include specified combinations in the generated test configurations. ACTS supports a set of commonly used logic and arithmetic operators to specify constraints. In this case, the following constraint can be used to ensure that invalid combinations are not generated. It says that if the OS is not XP, then the Browser will be Firefox:

```
(OS != "XP") => (Browser = "Firefox")
```

The covering array tool will then generate a set of test configurations that does not include the invalid combinations, but does cover all those that are essential. The revised test configuration array is shown in Figure 9. Parameter values that have changed from the original configurations

are underlined. Note that adding the constraint also resulted in reducing the number of test configurations by one. This will not always be the case, depending on the constraints used, but it illustrates how constraints can sometimes reduce the problem. Even if particular combinations are testable, the test team may consider some combinations unnecessary, and constraints could be used to prevent these combinations, possibly reducing the number of test configurations.

In many practical cases, the situation will not be quite as simple as the example above. For example, instead of dealing only with one Windows OS variety (in this case XP) we may have several: XP, Vista, Win7, and Win8. Similarly, there may be many Linux releases to consider, such as Red Hat, Ubuntu, Fedora, and many others plus different releases of the individual Linux versions. Such a situation could lead to very complicated constraint expressions. One approach proposed for handling this problem is the notion of *properties* [158], which can be used to combine related values. For the example here, there could be an "OSfamily" property defined for the OS parameter, so the constraint could be expressed as

```
(OS.OSfamily != "Windows") => (Browser = "Firefox")
```

Without the properties feature, we would need to write something like:

```
(OS != "XP" && OS != "Vista" && OS != "Win7" && OS != "Win8")
  => (Browser = "Firefox")
```

If we needed other constraints to also include references to the OSfamily property, the constraint set could become complicated very quickly. Such situations are not uncommon in practical testing.

Although the "properties" feature is not available on most covering array generators, we can achieve the goal of simplifying constraint expression in a different (though somewhat less elegant) way by taking advantage of the power of constraint solvers in ACTS or other tools, along with a little textual substitution. For example, define a term "WindowsVersion" as

```
(OS = "XP" || OS = "Vista" || OS = "Win7" || OS = "Win8")
```

Then constraints can be written such as `!WindowsVersion => (Browser = "Firefox")`. Substituting the parenthetical expression above for "WindowsVersion" using a preprocessor, or simply a text editor, will then introduce the necessary expression throughout the constraint set.

### *Constraints Among Parameters*

A second way in which untestable combinations may arise in practice is where some parameters become inactive when others are set to particular values. In the previous section, we considered situations where particular parameter values do not occur in combination with other particular values, but the parameters themselves were always present. For example, every test configuration included both operating system and browser, even though certain OS/browser value combinations did not occur. But for some test problems, a value in one parameter affects not just the possible values for another parameter, but the presence of other parameters themselves, regardless of values. Returning to the testing problem described in Section 3.1, suppose testers wanted to also consider additional software that may be present in configurations. Java and Microsoft .Net are used by many applications, and it is important to test for compatibility with different versions of these platforms. Thus is may be desirable to add two additional parameters: "java_version" and "dot_net_version". However, Java can be present on both Windows and Linux platforms, but we must deal with the problem that .Net will not be present on a Linux system. This restriction cannot be handled with an ordinary constraint, because if the platform is Linux, the "*dot_net_version*"

parameter does not make any sense. Instead we end up with two different parameter sets: for Windows, the parameters are *OS, browser, protocol, cpu, dbms, java_version, and dot_net_version*; for Linux the parameters are *OS, browser, protocol, cpu, dbms, and java_version*. Practical testing problems may be more complex than this somewhat contrived example, and may have multiple constraints among parameters. A variety of approaches can be used to deal with this type of problem:

*Split test suite*: The simplest and perhaps most obvious method is to switch from a single configuration test suite to one for each combination of parameters that control the applicability of others. In this case, there would be one test suite for Linux and one for Windows systems. This setup is easy to accomplish, but results in some duplicate combinations. For example, the same 3-way combinations for *browser*, *protocol*, and *dbms* will occur in both test suites. The situation is helped a bit by the fact that splitting the tests into two separate arrays means two covering arrays for $n$-1 parameters instead of one for $n$ parameters, and we will have fewer tests with one less parameter to cover. But since the number of tests grows with log $n$, the number of tests for $n$-1 parameters is just slightly smaller than for $n$. In general, therefore, splitting the problem into two test suites will result in almost twice the number of tests. For example, for $t = 3$, $v = 3$, a covering array for 10 parameters has 66 tests, and for 9 parameters there are 62 tests.

*Covering arrays with shielding parameters*: It is also possible to use an algorithm that allows the specification of "shielding" parameters [33]. In the example above, *dot_net_version* does not apply where the OS parameter is Linux. A parameter that does not always appear (in this case, *dot_net_version*) is called a *dependent* parameter, one that controls whether the dependent parameter is used is called the *shielding* parameter, and values of the shielding parameter that control use of the dependent parameter are controlling values (here, OS = Linux). This method prevents the generation of a large number of duplicate combinations. However, this approach requires modification of the covering array generation algorithm, and the shielded parameter approach is not yet implemented in covering array tools.

*Combine parameters*: An alternative approach is to combine parameters that do not apply to all configurations with other parameters, then use constraints. This is essentially a way of using the "shielded parameters" concept without requiring a modified covering array algorithm. In this case, "*java_version*" and "*dot_net_version*" could be combined into a single "*platform_version*". Constraints could be used to prevent the occurrence of invalid platform versions. For example, if the Java versions being included in tests are 1.6, and 1.7, and .Net versions are 3 and 4, then the following parameter can be established:

```
platform_version:  {java1.6, java1.7, dot_net3, dot_net4}
constraint: (OS = "Linux" => platform_version = "java1.6"|| platform_version =
"java1.7")
```

This approach prevents the generation of duplicate 3-way combinations for *java_version*, *protocol*, and *dbms* in both test suites. That is, a particular 3-way combination of these parameters will occur in association with at least one, but not necessarily both OSes in the test suite. The advantage of this approach is that it can be used with any covering array tool that implements constraints. It also produces reasonably compact covering arrays that are suitable for practical testing.

| 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 | Firefox | IPv4 | Intel | Sybase |
| 6 | OS X | Firefox | IPv6 | AMD | Oracle |
| 7 | RHL | Firefox | IPv6 | Intel | MySQL |
| 8 | RHL | Firefox | IPv4 | Intel | Oracle |
| 9 | XP | IE | IPv4 | AMD | Sybase |

**Figure 9.** Test configurations for simple example with constraint.

## *3.4 Cost and Practical Considerations*

Applying combinatorial methods to testing configurations can be highly cost-effective. Most software applications are required to run on a variety of systems, and must work correctly on different combinations of OS, browser, hardware platform, user interface, and other variables. Constraints among parameter values are very common in practical testing. Depending on the constraints needed, the size of the test suite may either decrease or increase with constraints, because the covering array algorithm has less opportunity to compress combinations in tests. The increase in test set size is not always significant, but must be kept in mind in initial planning.

One of the key questions in any software assurance effort concerns how many tests are required. Unfortunately, there is no general formula to compute the size of a covering array with constraints and parameters with varying numbers of values (mixed level arrays). If all parameters have the same number of values, or at least little variation among values (e.g., mostly binary with a few having three values), then tables of covering arrays may be used to determine the number of tests needed in advance. See **Error! Reference source not found.** for links to pre-computed covering arrays and best-known sizes of arrays for particular configurations. For mixed level arrays, particularly where there is significant variation among the number of values per parameter, the situation is more complex. If $v_l$ is the least number of values for among $n$ parameters, and $v_m$ is the greatest, the number of tests will lie somewhere between the size of a covering array for $(v_l)^n$ and $(v_m)^n$, but the interpolation is not linear. For example, a 3-way array for a configuration of $2^8 10^2$ has 375 tests, while the $2^{10}$ configuration has 66 tests and the $10^{10}$ configuration has 2367 tests. The situation is even more complex with more variability among parameter values, or in the presence of constraints, so there is generally no practical way to determine the number of tests without running the covering array generator.

## *3.5 Chapter Summary*

Configuration testing is probably the most commonly used application of combinatorial methods in software testing. Whenever an application has roughly five or more configurable attributes, a covering array is likely to make testing more efficient. Configurable attributes usually have a small number of possible values each, which is an ideal situation for combinatorial methods. Because the number of t-way tests is proportional to $v^t \log n$, for $n$ parameters with $v$ values each, as long as configurable attributes have less than around 10 possible values each, the number of tests generated will probably be reasonable. The real-world testing problem introduced in Section 3.2 is a fairly typical size, where 4-way interactions can be tested with a few hundred tests.

Because many systems have certain configurations that may not be of interest (such as the Internet Explorer browser on a Linux system), constraints are an important consideration in any

type of testing. With combinatorial methods, it is important that the covering array generator allows for the inclusion of constraints so that all relevant interactions are tested, and important information is not lost because a test contains an impossible combination. Constraints may exist between parameter values or even affect the presence of certain parameters in testing. An example of the former is the constraint "OS = Linux => browser ≠ IE", where the value of the "OS" parameter affects the value of the "browser" parameter. The second type of constraint involves what have been termed "shielding parameters", such as the case where "OS = Linux" means that the parameter "dot_net_version" should not appear in a test, but if "OS = Windows" the a test may have both a .Net version and a Java version. A practical workaround for this situation is to merge the dependent parameter into an abstract parameter such as "platform" and then use constraints among values to prevent the production of tests with non-existent configurations.

# 4 Input Testing

As noted in the introduction, the key advantage of combinatorial testing derives from the Interaction Rule: all, or nearly all, software failures involve interactions of only a few parameters. Using combinatorial testing to select configurations can make testing more efficient, but it can be even more effective when used to select input parameter values. Testers traditionally develop scenarios of how an application will be used, then select inputs that will exercise each of the application features using representative values, normally supplemented with extreme values to test performance and reliability. The problem with this often ad hoc approach is that unusual combinations will usually be missed, so a system may pass all tests and work well under normal circumstances, but eventually encounter a combination of inputs that it fails to process correctly. By testing all *t*-way combinations, for some specified level of *t*, combinatorial testing can help to avoid this type of situation.

## *4.1 Partitioning the Input Space*

To get a sense of the problem, we will consider a simple example. The system under test is an access control module that implements the following policy:

Access is allowed if and only if:

- the subject is an employee
  AND current time is between 9 am and 5 pm
  AND it is not a weekend
- OR subject is an employee with a special authorization code
- OR subject is an auditor
  AND the time is between 9 am and 5 pm
  (not constrained to weekdays).

The input parameters for this module are shown in Figure 10. In an actual implementation, the values for a particular access attempt would be passed to a module that returns a "grant" or "deny" access decision, using a function call such as "`access_decision(emp, time, day, auth, aud)`".

```
emp:  boolean;
time:  0..1440;  // time in minutes
day:   {m,tu,w,th,f,sa,su};
auth: boolean;
aud:  boolean;
```

**Figure 10.** Access control module input parameters.

Our task is to develop a covering array of tests for these inputs. The first step will be to develop a table of parameters and possible values, similar to that in Section 0 in the previous chapter. The only difference is that in this case we are dealing with input parameters rather than configuration options. For the most part, the task is simple: we just take the values directly from the specifications or code, as shown in Figure 11. Several parameters are boolean, and we will use 0 and 1 for false and true values respectively. For day of the week, there are only seven values, so these can all be used. However, hour of the day presents a problem. Recall that the number of tests generated for *n* parameters is proportional to $v^t$, where *v* is the number of values and *t* is the interaction level (2-way to 6-way). For all boolean values and 4-way testing, $v^t$ is $2^4$. But consider

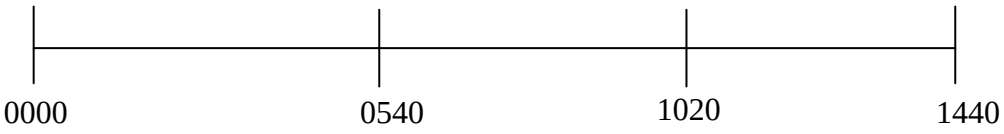
what happens with a large number of possible values, such as 24 hours. The number of tests will be proportional to $24^4 = 331{,}736$. Even worse in this example, time is given in minutes, which would obviously be completely intractable. Therefore, we must select representative values for the hour parameter. This problem occurs in all types of testing, not just with combinatorial methods, and good methods have been developed to deal with it. Most testers are already familiar with one or more of these: *category* [135] *or equivalence* [141] *partitioning* and *boundary value analysis*. These methods are reviewed here to introduce the examples. A much more systematic treatment, in the context of data modeling, is provided in Sect. **Error! Reference source not found.**. Additional background on these methods can be found in software testing texts such as Ammann and Offutt [4], Beizer [9], Copeland [48], Mathur [107], and Myers [118].

| Parameter | Values |
|---|---|
| emp | 0,1 |
| time | ?? |
| day | `m,tu,w,th,f,sa,su` |
| auth | 0, 1 |
| aud | 0, 1 |

**Figure 11.** Parameters and values for access control example.

Both of these intuitively appealing methods will produce a smaller set of values that should be adequate for testing purposes, by dividing the possible values into partitions that are meaningful for the program being tested. One value is selected for each partition. The objective is to partition the input space such that any value selected from the partition will affect the program under test in the same way as any other value in the partition. Thus, ideally if a test case contains a parameter *x* which has value *y*, replacing *y* with any other value from the partition will not affect the test case result. This ideal may not always be achieved in practice.

How should the partitions be determined? One obvious, but not necessarily good, approach is to simply select values from various points on the range of a variable. For example, if capacity can range from 0 to 20,000, it might seem sensible to select 0, 10,000, and 20,000 as possible values. But this approach is likely to miss important cases that depend on the specific requirements of the system under test. Engineering judgment is involved, but partitions are usually best determined from the specification. In this example, 9 am and 5 pm are significant, so 0540 (9 hours past midnight) and 1020 (17 hours past midnight) determine the appropriate partitions:




*Use a maximum of 8 to 10 values per parameter to keep testing tractable.*

Ideally, the program should behave the same for any of the times within the partitions; it should not matter whether the time is 4:00 am or 7:03 am, for example, because the specification treats both of these times the same. Similarly, it should not matter which time between the hours of 9 am and 5 pm is chosen; the program should behave the same for 10:20 am and 2:33 pm. One common strategy, *boundary value analysis*, is to select test values at each boundary and at the smallest possible unit on either side of the boundary, for three values per boundary. The intuition, backed by em pirical research, is that errors are more likely at boundary conditions because errors in programming may be made at these points. For example, if the requirements for automated teller machine software

say that a withdrawal should not be allowed to exceed $300, a programming error such as the following could occur:

```
if (amount > 0 && amount < 300) {
     //process withdrawal
} else {
     // error message
}
```

Here, the second condition should have been "`amount <= 300`", so a test case that includes the value `amount = 300` can detect the error, but a test with `amount = 305` would not. It is generally also desirable to test the extremes of ranges. One possible selection of values for the time parameter would then be: 0000, 0539, 0540, 0541, 1019, 1020, 1021, and 1440. More values would be better, but the tester may believe that this is the most effective set for the available time budget. With this selection, the total number of combinations is $2 \times 8 \times 7 \times 2 \times 2 = 448$. Generating covering arrays for $t = 2$ through 4 results in the following number of tests:

| t | # Tests |
|---|---|
| 2 | 56 |
| 3 | 112 |
| 4 | 224 |

**Figure 12.** Number of tests for access control example.

It is important to keep in mind that parameters may not always appear in a single function call, such as our example `access_decision(emp, time, day, auth, aud)`. Sometimes inputs to a particular operation may be spread through many lines of code in a program. For instance, consider an automated teller machine processing input from a user and the user's ATM card. The code may contain a series of calls such as the following:

```
get_acct_num(); // read acct number from card
get_PIN();      // read PIN from keyboard
get_tran_type();//  read  transaction  type,  withdrawal  or
deposit
get_amt();      // read transaction amount from keyboard
process_tran(); // process transaction
```

In this case, a series of values will be established in memory before finally being processed. So account number, PIN, transaction type, and amount are all parameters used in tests, but they are being entered one at a time instead of all at once. This situation is common in real-world systems.

## *4.2 Input Variables vs. Test Parameters*

*For some applications, we test combinations of input characteristics, not just inputs.*

In the example above, we assumed that the parameters to be included in tests were taken from function calls in the program, $f(p_1, p_2, \ldots, p_n)$, where each parameter had defined values or a range of values. In many cases, it will not be so obvious how to identify what should be included in the covering array and tests. The classic Ostrand and Balcer [135] software testing paper illustrates this common situation with the example of a "find" command, which takes user input of a string and a file name and locates all lines containing the

string. The format of the command is "find <*string*> <*filename*>, where <*string*> is one or more quoted strings of characters such as "john", "john smith", or "john" "smith". Search strings may include the escape character (backslash) for quotes, to select strings with embedded quotes in the file, such as "\"john\"" to report the presence of lines containing *john* in quotes within the file. The command displays any lines containing one or more of the strings. This command has only two input variables, *string* and *filename*, so is combinatorial testing really useful here?

In fact, combinatorial methods can be highly effective for this common testing problem. To check the "find" command, testers will want to ensure that it handles inputs correctly. The input variables in this case are *string* and *filename*, but it is common to refer to such variables as *parameters*. We will distinguish between the two here, but follow conventional practice where the distinction is clear. The *test parameters* identify characteristics of the command input variables. So the *test parameters* are in this case different from the two *input parameters*, *string* and *filename*. For example, the *string* input has characteristics such as length and presence of embedded blanks. Clearly, there are many ways to select test parameters, so engineering judgment must be used to determine what are most important. One selection could be the following, where *file_length* is the length in characters of the file being searched:

String length: {0, 1, 1..*file_length*, >*file_length*}
Quotes: {yes, no, improperly formatted quotes}
Blanks: {0, 1, >1}
Embedded quotes: {0, 1, 1 escaped, 1 not escaped}
Filename: {valid, invalid}
Strings in command line: {0, 1, >1}
String presence in file: {0, 1, >1}

For these seven test parameters, we have $2^1 3^4 4^2 = 2{,}592$ possible combinations of test parameter values. If we choose to test all 2-way interactions we need only 19 tests. For 3 and 4-way combinations, we need only 67 and 218 tests respectively. Because the number of tests grows only as log *n* for *n* parameters, we can do very thorough testing at relatively low cost for problems like this. That is, we can include a large number of characteristics to be used as test parameters without significantly increasing the test burden. In the problem above, if we used only the first four of the test parameters, instead of all seven, the number of tests required for $t = 2$, 3, and 4 respectively are 16, 54, and 144. Using all seven characteristics means much more thorough testing with relatively little increase in test set size.

When testing combinations of input characteristics as above, we must be careful that the test set captures enough important cases. For the find command, testing 3-way or 4-way combinations of the seven characteristics should be an excellent sample of test cases that can detect problems. That is, the tests will include both valid and invalid strings. In some cases, there may be a need to ensure the presence of test cases with a number of specific characteristics. For example, passwords may be required to (1) exceed a certain length, (2) contain numerics, and (3) contain special characters. A 2-way covering array might not include any valid cases, because it contains all pairs but three characteristics must be true to constitute a valid test case. We may need to supplement the covering array with some additional tests in this case. Sect. **Error! Reference source not found.** discusses this situation in more detail, along with ways to deal with it.

## *4.3 Fault Type and Detectability*

Consider the code snipped introduced in Fig. Figure 1 again. As seen below, if two boolean conditions are true, faulty code is executed, resulting in a failure:

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

In this case, the branches `pressure < 10` and `volume > 300` are correct and the fault occurs in the code that is reached when these conditions are true. Thus any covering array with values for pressure and volume that will make the conditions true can detect the problem. But consider another type of fault, in which branching statements may be faulty. The difference between these two types of faults is illustrated below, which we will refer to as (a) *code block faults* and (b) *condition faults*:

Example 1.

(a) Code block fault example:

```
if (correct condition) {faulty code}
else                   {correct code}
```

(b) Condition fault example:

```
if (faulty condition)  {correct code}
else                   {correct code}
```

Now suppose the code is as follows:

Example 2.

```
if ( (a || !b) && c) {faulty code}
else                {correct code}
```

*Condition faults are much more difficult to detect than code block faults.*

In this case, a 2-way covering array that includes values for `a`, `b`, and `c` is guaranteed to trigger the faulty code, since a branch to the faulty code occurs if either `a && c` or `!b && c` is true. A 2-way array will contain both of these conditions, so only pairs of values are needed even though the branch condition contains three variables. Suppose however that the fault is not in the code block that follows from the branch, but in the branch condition itself, as shown in the following code block. In this case, block 1 should be executed when `(a || !b) && c` evaluates to true and block 2 should be executed in all other cases, but a programming error has replaced `||` with `&&`.

```
if ( (a && !b) && c)  { block 1, correct code }
else                  { block 2, different correct code }
```

A 2-way covering array may fail to detect the error. A quick analysis shows that the two expressions `(a && !b) && c` and `(a || !b) && c` evaluate differently for two value settings: `a,b,c = 0,0,1` and `a,b,c = 1,1,1`. A 2-way array is certain to include all pairs of these values, but not necessarily all three in the same test. A 3-way array would be needed to ensure detecting the error, because it would be guaranteed to include `a,b,c = 0,0,1` and `a,b,c = 1,1,1`, either of which will detect the error.

Detecting condition faults can be extremely challenging. Experimental evaluations of the effectiveness of pairwise (2-way) combinatorial testing [9] show the difficulty of detecting condition faults. Using a set of 20 complex boolean expressions that have been used in other testing studies (see [10] or [191] for complete list of expressions), detection was evaluated for five different types of seeded faults. For the full set of randomly seeded faults, pairwise testing had an effectiveness of only 28%, although this was partially because different types of faults occurred with different frequency. For the five fault types, detection effectiveness was only 22% for one type, but the other four ranged from 46% to 73%, averaging 51% across all types. This is considerably below the occurrence rates of 2-way interaction failures reported in Sect. 1.1 and shown in Figure 2, which reflect empirical data on failures that result from a combination of condition faults and code block faults. Even 6-way combinations are not likely to detect all errors in complex conditions. A study [181] of fault detection effectiveness for expressions of five to 15 boolean variables found detection rates for randomly generated faults as shown in Figure 13 (2,000 trials; 200 per set). Note that even for 6-way combinations, fault detection was just above 80%.

How can we reconcile these results with the demonstrated effectiveness of combinatorial testing? First, note that the expressions used in this study were quite complex, involving up to 15 variables. Consider also that software nearly always includes code blocks interspersed with nested conditionals, often several levels deep. Furthermore, the input variables used in covering arrays often are not used directly in conditions internal to the program. Their values may be used in computing other values that are then propagated to the variables in the Boolean conditions inside the program, and using high strength covering arrays of input values in testing may be sufficient for a high rate of error detection. Nevertheless, the results in [181] are important because they illustrate an additional consideration in using combinatorial methods. For high assurance, it may be necessary to inspect conditionals in the code (if source code is available) and determine the correctness of branching conditions through non-testing means, such as formally mapping conditionals to program specifications.

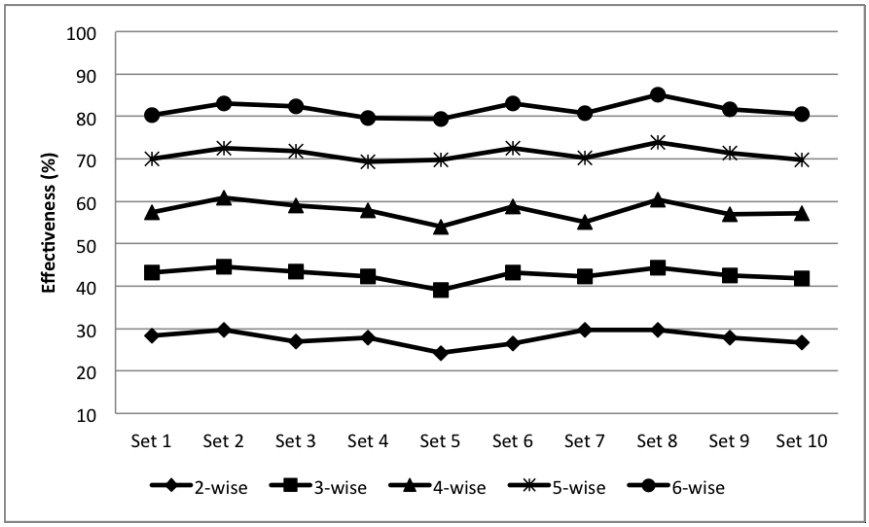


**Figure 13.** Effectiveness of t-way testing for expressions of 5 to 15 boolean variables [181]

What do these observations mean for practical testing, and what interaction strengths are needed to detect condition faults that occur in actual product software? In general, code with complex conditions may require higher strength (higher level of *t*-way combinations) testing, which is not surprising. But it also helps to explain why relatively low-strength covering arrays may be so effective. Although the condition in Example 1 above includes three terms, it expands to a disjunctive normal form of `a && c || b && c`, so only two terms are needed to branch into the faulty code. Even a more complex expression with many different terms, such as:

```
( (a || b) && c || d && e && (!f || g) || !a && (d || h || j) )
```

expands to:

```
a && c || b && c || d && !a  || h && !a || d && e && g || d && e && !f
```

which has three clauses with two terms each, and two clauses with three terms. Note that a test which includes any of the pairs `[a c], [b c], [d !a], [h !a]` will trigger a branch into code that follows this conditional. Thus if that code is faulty, a 2-way covering array will cause it to be executed so that the error can be detected.

These observations lead us to an approach for detecting condition faults: Given any complex condition, *P*, convert *P* to DNF, then let *t* equal the smallest number of literals in any term. A *t*-way covering array will then include at least one test in which the conditional will evaluate to true, thus branching into the code that follows the conditional. For example, convert `(a || !b) && c)` to `(a && c) || (!b && c);` then *t* = 2. Again, however, an important caveat to this approach is that in most software, conditions are nested, interspersed with blocks of code, so the relationship between code block faults and condition faults is complex. A faulty condition may branch into a section of code that is not correct for that condition, which then computes values that may be used in a nested conditional statement, and so on.

## *4.4 Building Tests to Match an Operational Profile*

*Combinatorial test sets can approximate an operational profile with some loss of efficiency.*

Many test projects require the use of an operational profile [118, 121], which attempts to use the same probability distribution of inputs for test data as occurs in live system operation. For example, if a web banking system typically receives 40% balance inquiries, 40% payroll deposit transactions, and 20% bill-pay transactions, then the test data would include these three transaction types in approximately the same proportion. Similarly, an operational profile may be applied to input data in each transaction, and the test data would be matched to this distribution. For example, an input partition for the "amount" field in the bill-pay transaction might include inputs of 96% with amounts under the user's balance, 3% with insufficient funds, and 1% zero amounts (user error), similar to the proportion of values that the bank experiences in day to day use of their system. How can the operational profile approach be used in conjunction with combinatorial testing?

One way we can approximate an operational profile for some problems is to assign sets of values to variables in proportion to their occurrence in the operational profile, if the chances of their occurrence in input are independent of each other. For example, if we have 5 binary variables, *a..e*, where *a* and *b* have value 0 two-thirds of the time and value 1 a third of the time, and the rest have

0, 1 with equal chance. Then use this as input to ACTS, assigning 0 and 1 in proportion to the occurrence of 0 for a and b (2/3), and 2 in proportion to the occurrence of 1 (1/3):

a: 0,1,2
b: 0,1,2
c: 0,1
d: 0,1
e: 0,1

In the covering array, change 1 to 0 for variables a and b, then change 2 to 1:

| *a b c d e* | | *a b c d e* |
|---|---|---|
| 0,0,0,0,0 | | 0,0,0,0,0 |
| 0,1,1,1,1 | | 0,0,1,1,1 |
| 0,2,0,1,0 | | 0,1,0,1,0 |
| 1,0,1,0,1 | | 0,0,1,0,1 |
| 1,1,0,0,0 | becomes → | 0,0,0,0,0 |
| 1,2,1,1,1 | | 0,1,1,1,1 |
| 2,0,0,1,1 | | 1,0,0,1,1 |
| 2,1,1,0,0 | | 1,0,1,0,0 |
| 2,2,*,0,* | | 1,1,*,0,* |

We will have inputs where *a*,*b* = 0,0 4/9 of the time, *a*,*b*=0,1 2/9 of the time, etc. It's just an approximation to the correct distribution though, since the distribution isn't quite right for some combinations, e.g., *b*,*c* = 1,0 only 1/9, instead of 1/6, depending on what we do with the * in the last row. This approach would obviously be a lot messier if we were trying to do distributions with lots of values per variable. There are no doubt lots of ways to make this more efficient, but we should probably stick with things we can do using ACTS, and not implementing new algorithms, since practical problems will require constraint handling.

Limitations: Fine-grained control of the distribution of test values is not practical with this approach, because it relies on using multiple values that are then mapped into a smaller set of desired values to produce the distribution. Thus if the desired distribution is 60/20/20 for three values of parameter P1, we can specify the input to the covering array generator as follows:

P1: a1, a2, a3, b, c.

Then the covering array will have approximately three times as many values of "a" for P1 if we map a1, a2, and a3 to a. We will refer to the values a1, a2, and a3 as "temporary" values, which are mapped to the "actual" value a. A distribution such as 45/25/20/10 for four values a, b, c, and d, would be much more difficult to approximate. It requires that value a appear in the covering array 4.5 X as frequently as value d, value b appear 2.5 X for each occurrence of d, and c must be twice as common as d. Since we obviously are limited to whole numbers of value occurrences, the way to do this would be as follows:

P1: a1, a2, a3, a4, a5, a6, a7, a8, a9, b1, b2, b3, b4, b5, c1, c2, d.

Unfortunately, this results in 17 temporary values for parameter P1. Recall from Chapter 1 that the number of tests is proportional to $v^t$, so even if $t = 2$ or $t = 3$, the resulting covering array of tests will be extremely large. A more practical approach to this problem is to trade some of the precision in the distribution for a smaller test set. If we are willing to accept an approximate distribution of 40/20/20/10 instead of 45/25/20/10, then we reduce the number of values for P1 to 9 instead of 17 (a1, a2, a3, a4, b1, b2, c1, c2, d). One heuristic that helps make it more practical to generate test arrays meeting an operational distribution is to require that the proportions of different values all be divisible by at least 10, to ensure that no more than 10 temporary values are used. For example, a 60/20/10/10 distribution can be produced with six for the first value, two for the second, etc. Of course, limiting temporary values to 10 or less means that actual values must be constrained to significantly less than 10, depending on the distribution being modeled. Once again, engineering judgment is required to find a tradeoff that works for the problem at hand.

We also note that operational profile testing is focused on approximating the type and number of inputs normally encountered, while combinatorial testing's forte is exercising the very rare cases that normal testing might miss. An additional complication is that not all failures have the same consequence in terms of economic or other impact. The more commonly used functions of the system may be much more important to a company's revenue, for example, because of the large number of customers impacted when one of them fails. Such considerations argue for the need to consider the operational distribution in test planning, looking at the cost of failure for different functions [191, 192]. For example, a retail operation may place a higher priority on customer purchase transactions than on item return, on the basis of both volume and impact on revenue. In this case it makes sense to do more testing of purchase transactions, reflecting the operational distribution of transaction types. Combinatorial testing would then be applied to testing of purchase transactions to detect obscure input combinations that might cause a failure. Very heavily used transaction types are eventually likely to encounter almost any combination, so it is important to find these rare cases in testing.

## *4.5 Scaling Considerations*

*The larger the system, the greater the benefit from combinatorial testing.*

With the first of the examples above, the advantage over exhaustive testing is not large, because of the small number of parameters. The second example provided a respectable gain, but what happens with really big problems? For larger problems, the advantages of combinatorial testing can be spectacular. For example, consider the problem of testing the software that processes switch settings for the panel [125] shown in Figure 14. There are 34 switches, which can each be either on or off, for a total of $2^{34} = 1.7 \times 10^{10}$ possible settings. We clearly cannot test 17 billion possible settings, but all 3-way interactions can be tested with only 33 tests, and all 4-way interactions with only 85. This may seem surprising at first, but it results from the fact that every test of 34 parameters contains $\binom{34}{3} = 5{,}984$ 3-way and $\binom{34}{4} = 46{,}376$ 4-way combinations.


**Figure 14.** Panel with 34 switches.

This example illustrates the fact that the testing efficiency gain from combinatorial methods is much greater with larger problems. Recall from Section 0 that the number of tests required for $n$ parameters with $v$ values each increases proportional to $v^t \log n$, for $t$-way testing, but exhaustive testing for the same problem would require $v^n$ tests. Figure 15 shows the sizes of 2-way and 4-way covering arrays for different levels of $v$ with 10 through 50 variables. Notice the logarithmic growth of covering array sizes with increasing values of $n$, and the fact that the covering arrays are extremely tiny compared with what would be required for exhaustive testing.

| | *v*=2 | | | *v*=4 | | | *v*=6 | | |
|---|---|---|---|---|---|---|---|---|---|
| *n* | 2-way CA | 4-way CA | exhaustive | 2-way CA | 4-way CA | exhaustive | 2-way CA | 4-way CA | exhaustive |
| 10 | 8 | 41 | 1024 | 29 | 725 | 1048576 | 63 | 3713 | 6.046e+7 |
| 20 | 10 | 65 | 1048576 | 37 | 1165 | 1.099e+12 | 79 | 6015 | 3.656e+15 |
| 30 | 11 | 80 | 1.073e+9 | 41 | 1448 | 1.1529e+18 | 86 | 7473 | 2.210e+23 |
| 40 | 11 | 90 | 1.099e+12 | 44 | 1661 | 1.2089e+24 | 94 | 8550 | 1.336e+31 |
| 50 | 11 | 98 | 1.125e+15 | 46 | 1839 | 1.267e+30 | 99 | 9466 | 8.082e+38 |

**Figure 15.** 2-way and 4-way covering array sizes compared with exhaustive tests for various values of $n$ and $v$.

## *4.6 Cost and Practical Considerations*

Combinatorial methods can be highly effective and reduce the cost of testing substantially. For example, Justin Hunter has applied these methods to a wide variety of test problems and consistently found both lower cost and more rapid error detection [85]. But as with most aspects of engineering, tradeoffs must be considered. Among the most important is the question of when to stop testing, balancing the cost of testing against the risk of failing to discover additional failures. An extensive body of research has been devoted to this topic, and sophisticated models are available for determining when the cost of further testing will exceed the expected benefits [19, 107]. Existing models for when to stop testing can be applied to the combinatorial test approach also, but there is an additional consideration: What is the appropriate interaction strength to use in this type of testing?

To address these questions consider the number of tests at different interaction strengths for an avionics software example [91] shown in Figure 16. While the number of tests will be different (probably much smaller than in Figure 16) depending on the system under test, the magnitude of difference between levels of $t$ will be similar to Figure 16, because the number of tests grows with $v^t$, for parameters with $v$ values. That is, the number of tests grows with the exponent $t$, so we want

to use the smallest interaction strength that is appropriate for the problem. Intuitively, it seems that if no failures are detected by *t*-way tests, then it may be reasonable to conduct additional testing only for *t*+1 interactions, but no greater if no additional failures are found at *t*+1. In the empirical studies of software failures, the number of failures detected at $t > 2$ decreased monotonically with *t*, so this heuristic seems to make sense: *start testing using 2-way (pairwise) combinations, continue increasing the interaction strength t until no errors are detected by the t-way tests, then (optionally) try t+1 and ensure that no additional errors are detected.* As with other aspects of software development, this guideline is also dependent on resources, time constraints, and cost-benefit considerations.

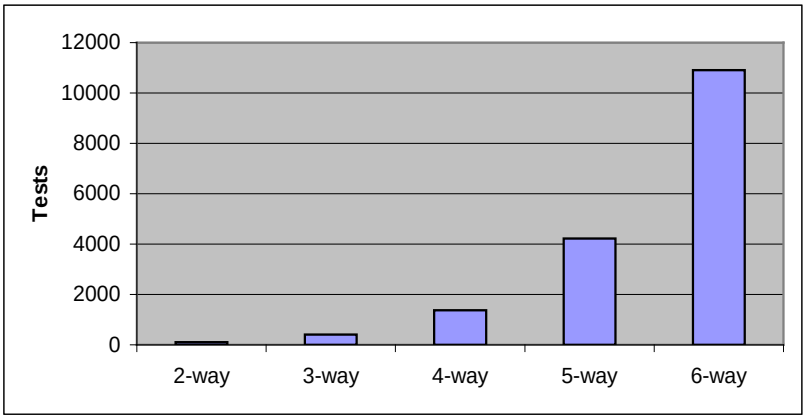



**Figure 16.** Number of tests for avionics example.

When applying combinatorial methods to input parameters, the key cost factors are the number of values per parameter, the interaction strength, and the number of parameters. As shown above, the number of tests increases rapidly as the value of t is increased, but the rate of increase depends on the number of values per parameter. Binary variables, with only two values each, result in far fewer tests than parameters with many values each. As a practical matter, when partitioning the input space, it is best to keep the number of values per parameter below 8 or 10 if possible, since the number of tests increases with $v^t$ (consider the difference between $4^3 = 64$ and $11^3 = 1{,}331$, for example).

Because the number of tests increases only logarithmically with the number of parameters, test set size for a large problem may be only somewhat larger than for a much smaller problem. For example, if a project uses combinatorial testing for a system that has 20 parameters and generates several hundred tests, a much larger system with 40 to 50 parameters may only require a few dozen more tests. Combinatorial methods may generate the best cost benefit ratio for large systems.

## *4.7 Chapter Summary*

1. The key advantage of combinatorial testing derives from the fact that all, or nearly all, software failures appear to involve interactions of only a few parameters. Generating a covering array of input parameter values allows us to test all of these interactions, up to a level of 5-way or 6-way combinations, depending on resources.

2. Practical testing often requires abstracting the possible values of a variable into a small set of equivalence classes. For example, if a variable is a 32-bit integer, it is clearly not possible to test the full range of values in $+/- 2^{31}$. This problem is not unique to combinatorial testing, but occurs in most test methodologies. Simple heuristics and engineering judgment are required to

determine the appropriate portioning of values into equivalence classes, but once this is accomplished it is possible to generate covering arrays of a few hundred to a few thousand tests for many applications. The thoroughness of coverage will depend on resources and criticality of the application.

# 5 Test Parameter Analysis (E. Miranda)

[non-NIST author]

# 6 Managing System State in Combinatorial Test Designs (G. Sherwood)

[non NIST author]

# 7 Measuring Combinatorial Coverage

[noted separately]

# 8 Test Suite Prioritization by Combinatorial Coverage (R. Bryce and S. Sampath)

**[non NIST authors]**

# 9 Combinatorial Testing and Random Test Generation

For combinatorial testing to be most efficient and effective, we need an understanding of when a particular test development method is most appropriate. That is, what characteristics of a problem lead us to use one approach over another, and what are the tradeoffs with respect to cost and effectiveness? Some studies have reviewed the effectiveness of combinatorial and random approaches to testing, comparing the use of covering arrays with randomly generated tests, but have reached conflicting results [5, 6, 9, 139, 151, 94, 95]. Any single test containing values for $n$ parameters, no matter how it is constructed, covers C($n$,2) 2-way combinations (pairs), C($n$,3) 3-way combinations, and so on. Naturally as additional tests are added, more combinations are covered. A covering array packs combinations together closely, but as long as test $i$+1 differs from previously produced tests, additional combinations will be covered. Generating values randomly naturally leads to differences between tests, resulting in good combinatorial coverage for certain classes of problems. This chapter discusses the use of covering arrays and random test generation. As we will see, there is an interesting connection between these two concepts.

## *9.1 Coverage of Random Tests*

By definition, a covering array covers all $t$-way combinations for the specified value of $t$ at least once. If enough random tests are generated, they will eventually also cover all $t$-way combinations. One key question is *how many random tests are needed to cover all t-way combinations*? In general, as the number of parameters increases, the probability that a random test set covers all $t$-way combinations increases as well, so that with thousands of parameters, these two methods begin to converge to the same number of tests. It has been shown [6] that where there is a large number of parameters (i.e., 1000s) and parameter values, and no constraints among parameters or parameter values, the number of tests required for $t$-way coverage (for arbitrary $t$) is approximately the same for covering arrays and randomly generated tests. This is an encouraging result, because of the difficulty of generating large covering arrays. We can produce thousands of random tests in seconds, but existing covering array algorithms cannot produce arrays for such large problems in a practical amount of time. If $t$-way coverage is needed for such problems, then random tests can be generated with a known probability of producing a full covering array. For $N$ randomly generated tests containing parameters with $v_i$ values each, there is a probability $P_t$ of detecting at least one $t$-way fault [6]:

$$P_t \geq 1 - \left(1 - \frac{1}{\prod_{i=1}^{t} v_i}\right)^N \tag{9.1.1}$$

For the more common case where there are multiple faults, we need to also consider the ways in which combinations of faults can be discovered, leading to a probability $P_{t,z}$ to detect z different faults of [6]:

$$P_{t,z} \geq \sum_{j=0,z} (-1)^j \binom{z}{j} \left(1 - \frac{1}{\prod_{i=1}^{t} v_i}\right)^N \tag{9.1.2}$$

These probabilities converge to $\lim_{k\to\infty} P_t = 1$ and $\lim_{k\to\infty} P_{t,z} = 1$, for $k$ parameters. For very large $N$, a randomly generated test set almost assures full $t$-way coverage. However, note that full

*A large number of random tests can provide a high level of combinatorial coverage.*

coverage is not guaranteed because values are generated randomly. Using the coverage methods presented in Chapter 7 (which are easily parallelizable), we could determine if all *t*-way combinations have been covered, and supplement the test set with any missing ones. After all, our goal is not to use covering arrays, but to cover all *t*-way combinations, for the appropriate level of *t*. It doesn't matter how tests providing the necessary coverage are generated. As mentioned, an important caveat to this probability calculation is that it does not hold when constraints are involved, as they often are in practical testing problems. We can still generate tests randomly, but cannot rely on this calculation to estimate how many tests to produce.

For smaller test problems involving 10s of parameters, covering array algorithms are entirely practical and can cover all *t*-way combinations in a fraction of the number of tests required by random generation. Table 6 gives the percentage of *t*-way combinations covered by a randomly generated test set of the same size as a *t*-way covering array, for various combinations of $k$ = number of variables and $v$ = number of values per variable. Note that the coverage could vary with different realizations of randomly generated test sets. That is, a different random number generator, or even multiple runs of the same generator, may produce slightly different coverage (perhaps a few tests out of thousands, depending on the problem). Figure 18 through Figure 22 summarize the coverage for arrays with variables of 2 to 10 values. As seen in the figures, the coverage provided by a random test suite versus a covering array of the same size varies considerably with different configurations.

| Vars | Values/ Variable | ACTS 2-way tests | Random 2-way coverage | ACTS 3-way tests | Random 3-way coverage | ACTS 4-way tests | Random 4-way coverage |
|---|---|---|---|---|---|---|---|
| 10 | 2 | 10 | 89.28% | 20 | 92.18% | 42 | 92.97% |
| 10 | 4 | 30 | 86.38% | 151 | 89.90% | 657 | 92.89% |
| 10 | 6 | 66 | 84.03% | 532 | 91.82% | 3843 | 94.86% |
| 10 | 8 | 117 | 83.37% | 1214 | 90.93% | 12010 | 94.69% |
| 10 | 10 | 172 | 82.21% | 2367 | 90.71% | 29231 | 94.60% |
| 15 | 2 | 10 | 96.15% | 24 | 97.08% | 58 | 98.36% |
| 15 | 4 | 33 | 89.42% | 179 | 93.75% | 940 | 97.49% |
| 15 | 6 | 77 | 89.03% | 663 | 95.49% | 5243 | 98.26% |
| 15 | 8 | 125 | 85.27% | 1551 | 95.21% | 16554 | 98.25% |
| 15 | 10 | 199 | 86.75% | 3000 | 94.96% | 40233 | 98.21% |
| 20 | 2 | 12 | 97.22% | 27 | 97.08% | 66 | 98.41% |
| 20 | 4 | 37 | 90.07% | 209 | 96.40% | 1126 | 98.79% |
| 20 | 6 | 86 | 91.37% | 757 | 97.07% | 6291 | 99.21% |
| 20 | 8 | 142 | 89.16% | 1785 | 96.92% | 19882 | 99.22% |
| 20 | 10 | 215 | 88.77% | 3463 | 96.85% | 48374 | 99.20% |
| 25 | 2 | 12 | 96.54% | 30 | 98.26% | 74 | 99.18% |
| 25 | 4 | 39 | 91.67% | 233 | 97.49% | 1320 | 99.43% |
| 25 | 6 | 89 | 92.68% | 839 | 97.94% | 7126 | 99.59% |
| 25 | 8 | 148 | 90.46% | 1971 | 97.93% | 22529 | 99.59% |
| 25 | 10 | 229 | 89.80% | 3823 | 97.82% | 54856 | 99.58% |

Table 6. Percent of *t*-way combinations covered by equal number of random tests

Now consider the size of a random test set required to provide 100% combination coverage. With the most efficient covering array algorithms, the difficulty of finding tests with high coverage increases as tests are generated. Thus even if a randomly generated test set provides better than 99% of the coverage of an equal sized covering array, it should not be concluded that only a few

more tests are needed for the random set to provide 100% coverage. Table 7 gives the sizes of randomly generated test sets required for 100% combinatorial coverage at various configurations, and the ratio of these sizes to covering arrays computed with ACTS. Although there is considerable variation among configurations, note that the ratio of random to covering array size for 100% coverage exceeds 3 in most cases, with average ratios of 3.9, 3.8, and 3.2 at $t$ = 2, 3, and 4 respectively. Thus, combinatorial testing retains a significant advantage over random testing for problems of this size if the goal is 100% combination coverage for a given value of $t$.

| | | 2-way Tests | | | 3-way Tests | | | 4-way Tests | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Vars | Values | ACTS Tests | Random Tests | Ratio | ACTS Tests | Random Tests | Ratio | ACTS Tests | Random Tests | Ratio |
| 10 | 2 | 10 | 18 | 1.80 | 20 | 61 | 3.05 | 42 | 150 | 3.57 |
| 10 | 4 | 30 | 145 | 4.83 | 151 | 914 | 6.05 | 657 | 2256 | 3.43 |
| 10 | 6 | 66 | 383 | 5.80 | 532 | 1984 | 3.73 | 3843 | 13356 | 3.48 |
| 10 | 8 | 117 | 499 | 4.26 | 1214 | 5419 | 4.46 | 12010 | 52744 | 4.39 |
| 10 | 10 | 172 | 808 | 4.70 | 2367 | 11690 | 4.94 | 29231 | 137590 | 4.71 |
| 15 | 2 | 10 | 20 | 2.00 | 24 | 52 | 2.17 | 58 | 130 | 2.24 |
| 15 | 4 | 33 | 121 | 3.67 | 179 | 672 | 3.75 | 940 | 2568 | 2.73 |
| 15 | 6 | 77 | 294 | 3.82 | 663 | 2515 | 3.79 | 5243 | 17070 | 3.26 |
| 15 | 8 | 125 | 551 | 4.41 | 1551 | 6770 | 4.36 | 16554 | 60568 | 3.66 |
| 15 | 10 | 199 | 940 | 4.72 | 3000 | 15234 | 5.08 | 40233 | 159870 | 3.97 |
| 20 | 2 | 12 | 23 | 1.92 | 27 | 70 | 2.59 | 66 | 140 | 2.12 |
| 20 | 4 | 37 | 140 | 3.78 | 209 | 623 | 2.98 | 1126 | 3768 | 3.35 |
| 20 | 6 | 86 | 288 | 3.35 | 757 | 2563 | 3.39 | 6291 | 18798 | 2.99 |
| 20 | 8 | 142 | 630 | 4.44 | 1785 | 8450 | 4.73 | 19882 | 59592 | 3.00 |
| 20 | 10 | 215 | 1028 | 4.78 | 3463 | 14001 | 4.04 | 48374 | 157390 | 3.25 |
| 25 | 2 | 12 | 34 | 2.83 | 30 | 70 | 2.33 | 74 | 174 | 2.35 |
| 25 | 4 | 39 | 120 | 3.08 | 233 | 790 | 3.39 | 1320 | 3520 | 2.67 |
| 25 | 6 | 89 | 327 | 3.67 | 839 | 2890 | 3.44 | 7126 | 19632 | 2.75 |
| 25 | 8 | 148 | 845 | 5.71 | 1971 | 7402 | 3.76 | 22529 | 61184 | 2.72 |
| 25 | 10 | 229 | 1031 | 4.50 | 3823 | 16512 | 4.32 | 54856 | 191910 | 3.50 |
| Ratio Average: | | | | 3.90 | | | 3.82 | | | 3.21 |

Table 7. Size of random test set required for 100% $t$-way combination coverage.

| Values per variable | Ratio, 2-way | Ratio, 3-way | Ratio, 4-way |
|---|---|---|---|
| 2 | 2.14 | 2.54 | 2.57 |
| 4 | 3.84 | 4.04 | 3.04 |
| 6 | 4.16 | 3.59 | 3.12 |
| 8 | 4.70 | 4.33 | 3.44 |
| 10 | 4.68 | 4.59 | 3.86 |

Table 8. Average ratio of random/ACTS for covering arrays by values per variable, variables = 10, 15, 20, 25

## *9.2 Adaptive Random Testing*

A recently developed testing strategy that can work quite well with combinatorial methods is called adaptive random testing (ART) [33, 35, 36]. The ART strategy seeks to deal with the problem that

faults tend to cluster together [2, 16, 34], by choosing tests one at a time such that each newly chosen test is as "different" as possible from previous tests. The difference, or distance, metric is chosen based on problem characteristics. The basic ART algorithm is shown in Figure 17.

```
T = {} /* T is the set of previously executed test cases */
randomly generate an input t
test the program using t as a test case
add t to T
while (stopping criteria not reached) {
    D = 0
    randomly generate next k candidates c1, c2, . . . , ck
    for each candidate ci {
        calculate the minimum distance di from T
        if di > D { D = di;   t = ci }
    }
    add t to T
    test the program using t as a test case
} // end while
```

**Figure 17.** Adaptive Random Testing algorithm

ART generates a set of random tests, determines the best test, i.e., with the greatest distance from the existing test set *T*, then adds that test to *T*, continuing until some stopping criterion is fulfilled. If the distance metric is based on the number of previously uncovered *t*-way combinations that are covered in the candidate tests, then this algorithm is essentially a greedy algorithm [125] for computing a covering array one test at a time. The distance measures for this approach were originally developed for numeric processing. Many application domains, however, must deal with enumerated values with relatively little complex calculation. In these cases, distance measures tailored to covering arrays can help in choosing test order, that is, in prioritizing tests. Chapter **Error! Reference source not found.** explains the use of prioritization methods.

## *9.3 Tradeoffs: Covering Arrays and Random Generation*

The comparisons between random tests and covering arrays for combinatorial testing suggest a number of conclusions:

- *For binary variables (v=2), random tests compare reasonably well with covering arrays* (96% to 99% coverage) for all three values (2, 3, and 4) of *t* for 15 or more variables. Thus random testing for a SUT with all or mostly binary variables may compare favorably with covering arrays.

- *Combination coverage provided by random generation of the equivalent number of pairwise tests at (t = 2) decreases as the number of values per variable increases*, and the coverage provided by pairwise testing is significantly less than 100%. The effectiveness of random testing relative to pairwise testing should be expected to decline as the average number of values per variable increases.

- *For 4-way interactions, coverage provided by random test generation increases with the number of variables*. Using a covering array for a module with approximately 10 variables should be significantly more effective than random testing, while the difference between the two methods should be less for modules with 20 or more variables.

- *For 100% combination coverage, the efficiency advantage of covering arrays varies directly with the number of values per variable and inversely with the interaction strength t.* Figure

23 illustrates how these factors (interaction strength *t* and values per variable *v*) combine: the ratio of random/covering array coverage is highest for 10 variables with *t* = 2, but declines for other pairings of *t* and *v*. To obtain 100% combination coverage, random testing is significantly less efficient, requiring 2 to nearly 5 times as many tests as a covering array generated by ACTS. Thus if 100% combination coverage is desired, using covering arrays may be less expensive than random test generation.

- *For very large sets of parameters with no constraints, random test generation can produce a set of tests that cover all t-way combinations* that is not significantly larger than the corresponding covering array. Generating the tests randomly will be much faster, and for very large problems covering array generation with existing tools is likely to be intractable.

An important practical consideration in comparing combinatorial with random testing is the efficiency of the covering array generator. Algorithms have a very wide range in the size of covering arrays they produce. In some cases, the better algorithms to produce arrays that are 50% smaller than other algorithms. We have found in comparisons with other tools that there is no uniformly "best" algorithm. Other algorithms may produce smaller or larger combinatorial test suites, so the comparable random test suite will vary in the number of combinations covered. Thus random testing may fare better in comparison with combinatorial tests produced by one of the less efficient algorithms.

*A less optimal (by size) array may provide better failure detection because it includes more interactions at t+1, t+2, etc.*

However, there is a less obvious but important tradeoff regarding covering array size. An algorithm that produces a very compact array, i.e., with few tests, for *t*-way combinations may include fewer (*t*+1)-way combinations because there are fewer tests. Table 9 and Table 10 illustrate this phenomenon for an example. Table 9 shows the percentage of *t*+1 up to *t*+3 combination coverage provided by the ACTS tests and in Table 10 the equivalent number of random tests. Although ACTS pairwise tests provide better 3-way coverage than the random tests, at other interaction strengths and values of *t*, the random tests are roughly the same or slightly better in combination coverage than ACTS. Recall from Section 9.1 that pairwise combinatorial tests detected slightly fewer events than the equivalent number of random tests. One possible explanation may be that the superior 4-way and 5-way coverage of the random tests allowed detection of more events. Almost paradoxically, an algorithm that produces a larger, sub-optimal covering array may provide better failure detection because the larger array is statistically more likely to include *t*+1, *t*+2, and higher degree interaction tests as a byproduct of the test generation. Again, however, the less optimal covering array is likely to more closely resemble the random test suite in failure detection.

Note also that the number of failures in the SUT can affect the degree to which random testing approaches combinatorial testing effectiveness. For example, suppose the random test set covers 99% of combinations for 4-way interactions, and the SUT contains only one 4-way interaction failure. Then there is a 99% probability that the random tests will contain the 4-way interaction that triggers this failure. However, if the SUT contains *m* independent failures, then the probability that combinations for all *m* failures are included in the random test set is $.99^m$. Hence with multiple failures, random testing may be significantly less effective, as its probability of detecting all failures will be $c^m$, for *c* = percent coverage and *m* = number of failures.

| t | 3-way coverage | 4-way coverage | 5-way coverage |
|---|---|---|---|

| | | | |
|---|---|---|---|
| 2 | .758 | .429 | .217 |
| 3 | | .924 | .709 |
| 4 | | | .974 |

Table 9. Higher interaction coverage of t-way tests

| t | 3-way coverage | 4-way coverage | 5-way coverage |
|---|---|---|---|
| 2 | .735 | .499 | .306 |
| 3 | | .917 | .767 |
| 4 | | | .974 |

Table 10. Higher interaction coverage of random tests

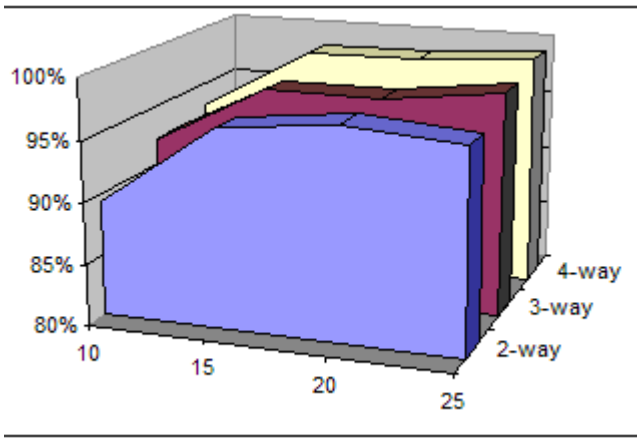



**Figure 18.** Percent coverage of *t*-way combinations for v=2.

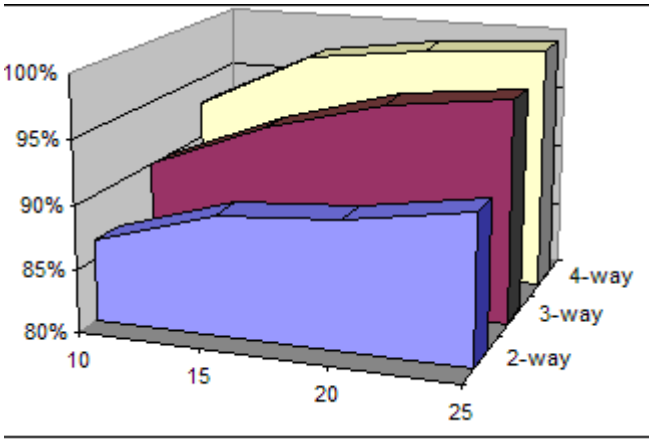



**Figure 19.** Percent coverage of *t*-way combinations for v=4.

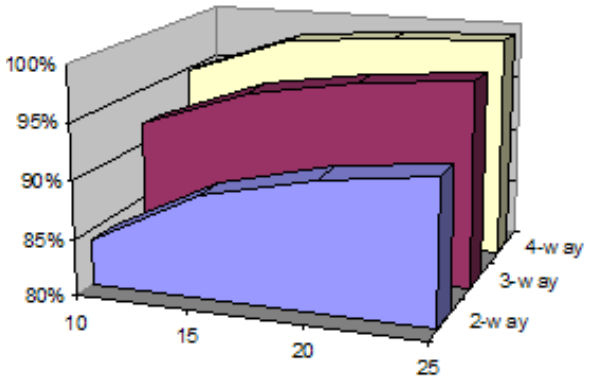



**Figure 20.** Percent coverage of *t*-way combinations for v=6.

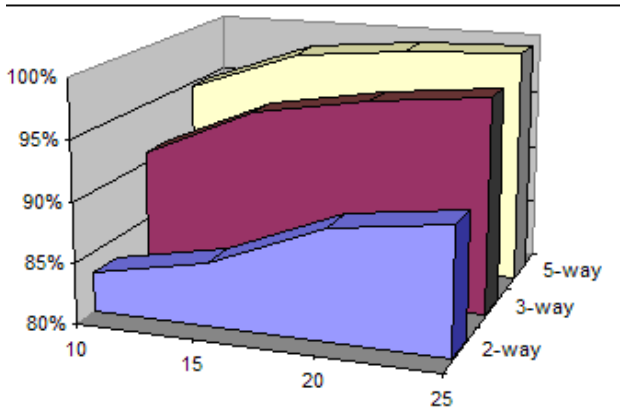



**Figure 21.** Percent coverage of *t*-way combinations for v=8.

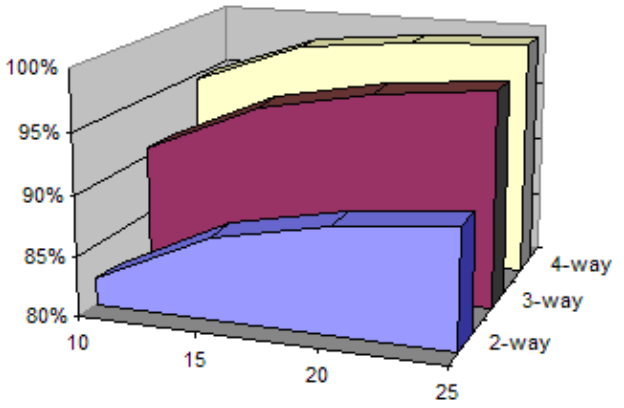



**Figure 22.** Percent coverage of *t*-way combinations for v=10

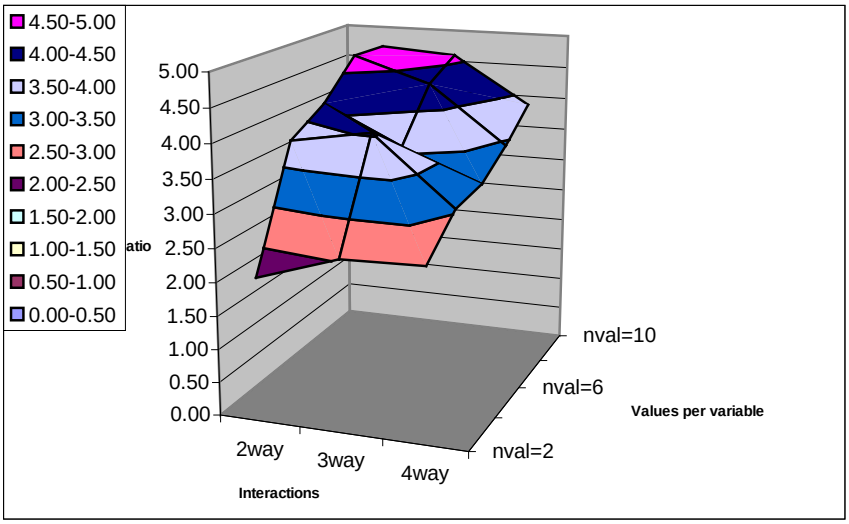



**Figure 23.** Average ratio of random/ACTS for covering arrays by values per variable

## *9.4 Cost and Practical Considerations*

The relationship between covering arrays and randomly generated tests presents some interesting issues. Generating covering arrays for combinatorial tests is complex; it has been shown to be an NP-hard problem. But generating tests randomly is trivial. Thus for large problems, we can compare the cost and time of generating a covering array versus producing tests randomly, measuring their coverage (Chapter 7), then adding tests as needed to provide full combinatorial coverage. Notice the last column of Table 6. For 4-way tests, once the number of parameters exceeds roughly 20, random generation will cover 99% or more of 4-way combinations. If a problem requires tests for 100 parameters, for example, covering array generators may require hours or days, or may simply be unable to handle that many parameters, but random tests could be generated quickly and easily. The test generation time for these two approaches is one factor among many that must be considered in test planning. Analyzing test parameters (Chapters 3 through 6), oracle development (Chapters **Error! Reference source not found.** and 12), and other essential tasks such as test execution and managing test runs will generally be much more expensive than generating tests, regardless of the test generation method used.

While the analyses reported here do not indicate that combinatorial testing is uniformly better than random, it does support a preference for combinatorial methods if the cost of applying the two test approaches is approximately the same. Most of the cost of testing goes into test planning, test oracle development, running and reporting tests, and the generation of test data – either randomly or with covering array tools – can be fully automated and run in parallel with other tasks. This preference may be particularly relevant if the SUT is likely to contain multiple failures (as is usually the case). Single failures that depend on the interaction of two or more variables have a high likelihood of being detected by random tests, because the random test set may cover a high percentage of all *t*-way combinations. But the probability of detecting multiple failures declines rapidly as $c^m$, for $c$ = percent coverage and $m$ = number of independent failures. Unfortunately many testing problems are too large (too many parameters) to be handled entirely using covering arrays, so random test generation may be used to achieve the combinatorial coverage desired.

## *9.5 Chapter Summary*

Covering array algorithms are significantly more efficient than random test generation if the goal is 100% combination coverage. The table below summarizes the test set size comparison for a variety of problem configurations. The difference is especially striking for binary parameters, where the ACTS covering array generator produces (*t*+1)-way coverage with roughly the same number of tests required by random generation for *t*-way coverage. Table 7 provides additional detail.

| | | t = 2 | | t = 3 | | t = 4 | |
|---|---|---|---|---|---|---|---|
| n | v | ACTS | random | ACTS | random | ACTS | random |
| 10 | 2 | 10 | 18 | 20 | 61 | 42 | 150 |
| 10 | 4 | 30 | 145 | 151 | 914 | 657 | 2256 |
| 10 | 8 | 117 | 499 | 1214 | 5419 | 12010 | 52744 |
| 15 | 2 | 10 | 20 | 24 | 52 | 58 | 130 |
| 15 | 4 | 33 | 121 | 179 | 672 | 940 | 2568 |
| 15 | 8 | 125 | 551 | 1551 | 6770 | 16554 | 60568 |
| 20 | 2 | 12 | 23 | 27 | 70 | 66 | 140 |
| 20 | 4 | 37 | 140 | 209 | 623 | 1126 | 3768 |
| 20 | 8 | 142 | 630 | 1785 | 8450 | 19882 | 59592 |
| 25 | 2 | 12 | 34 | 30 | 70 | 74 | 174 |
| 25 | 4 | 39 | 120 | 233 | 790 | 1320 | 3520 |
| 25 | 8 | 148 | 845 | 1971 | 7402 | 22529 | 61184 |

Table 11. Summary, ACTS and random test set sizes for 100% *t*-way combination coverage.

Existing research has shown either no difference (for some problems) or higher failure detection effectiveness (for most problems) for combinatorial testing. Analyzing random test sets suggests a number of reasons for this result. In particular, a highly optimized *t*-way covering array may include fewer *t*+1, *t*+2, and higher degree interaction tests than an equivalent sized random test set. Similarly, a covering array algorithm that produces a larger, sub-optimal array may provide better failure detection because the larger array is statistically more likely to include *t*+1, *t*+2, and higher degree interaction tests as a byproduct of the test generation. In some applications, it may make sense to combine aspects of both approaches. Adaptive random testing is a systematic method that can be used in this manner.

# 10 Sequence-Covering Arrays

In testing event-driven software, the critical condition for triggering failures often is whether or not a particular event has occurred prior to a second one, establishing a particular state that must be reached before a given failure can be triggered. For example, a failure might occur when connecting device *A* only if device *B* is already connected, or only if devices *B* and *C* were both already connected. Events may be repeatable in some systems, but this is not always the case. In the testing problem that motivated this work, the critical issue was the sequence of connecting a large number of peripherals, so it was physically impossible to connect an already connected device (without unplugging, which would be a separate event). As a different example, a memory management function may fail on an attempt to allocate memory if it failed to properly release memory at some prior time. Another common class of problems of this type occurs with graphical user interfaces that use callbacks. User actions may trigger the creation or release of resources, or the enabling or disabling of GUI controls. But the user may invoke these callbacks in any order, and errors may result if a prior callback left the system in an unexpected state.

## *10.1 Sequence Covering Array Definition*

*In many systems, the order of inputs is important.*

For this problem we can define a *sequence-covering array* [96, 97, 98, 99], which is a set of tests that ensure all *t-way sequence*s of events have been tested. The *t* events in the sequence may be interleaved with others, but all permutations will be tested. For example, we may have a component of a factory automation system that uses certain devices interacting with a control program. We want to test the events defined in Table 12.

There are 6! = 720 possible sequences for these six events, and the system should respond correctly and safely no matter the order in which they occur. Operators may be instructed to use a particular order, but mistakes are inevitable, and should not result in injury to users or compromise the enterprise. Because setup, connections and operation of this component are manual, each test can take a considerable amount of time. It is not uncommon for system-level tests such as this to take hours to execute, monitor, and complete. We want to test this system as thoroughly as possible, but time and budget constraints do not allow for testing all possible sequences, so we will test all 3-event sequences.

With six events, *a, b, c, d, e,* and *f,* one subset of three is {*b, d, e*}, which can be arranged in six permutations: [*b d e*], [*b e d*], [*d b e*], [*d e b*], [*e b d*], [*e d b*]. A test that covers the permutation [*d b e*] is: [*a d c f b e*]; another is [*a d c b e f*]. A larger example system may have 10 devices to connect, in which case the number of permutations is 10!, or 3,628,800 tests for exhaustive testing. In that case, a 3-way sequence covering array with 14 tests covering all $10 \cdot 9 \cdot 8 = 720$ 3-way sequences is a dramatic improvement, as is 72 tests for all 4-way sequences (see **Error! Reference source not found.**).

| Event | Description |
|---|---|
| *a* | connect air flow meter |
| *b* | connect pressure gauge |
| *c* | connect satellite link |
| *d* | connect pressure readout |
| *e* | engage drive motor |
| *f* | engage steering control |

Table 12. System events

**Definition.** A sequence covering array, SCA(*N*, *S*, *t*) is an *N* x *S* matrix where entries are from a finite set *S* of *s* symbols, such that every *t*-way permutation of symbols from *S* occurs in at least one row; the *t* symbols in the permutation are not required to be adjacent. That is, for every *t*-way arrangement of symbols $x_1$, $x_2$, ..., $x_t$, the regular expression $.*x_1.*x_2.*x_t.*$ matches at least one row in the array. Sequence covering arrays, as the name implies, are analogous to standard covering arrays (see Sect. 1.3), which include at least one of every *t*-way combination of any *n* variables, where $t<n$. A variety of algorithms are available for constructing covering arrays, but these are not usable for generating *t*-way sequences because they are designed to cover combinations in any order.

**Example 1.** Consider the problem of testing four events, *a*, *b*, *c*, and *d*. For convenience, a *t*-way permutation of symbols is referred to as a *t-way sequence*. There are 4! = 24 possible permutations of these four events, but we can test all 3-way sequences of these events with only six tests (see **Error! Reference source not found.**).

| Test | | | | |
|---|---|---|---|---|
| **1** | *a* | *d* | *b* | *c* |
| **2** | *b* | *a* | *c* | *d* |
| **3** | *b* | *d* | *c* | *a* |
| **4** | *c* | *a* | *b* | *d* |
| **5** | *c* | *d* | *b* | *a* |
| **6** | *d* | *a* | *c* | *b* |

Table 13. Tests for four events.

## *10.2 Size and Construction of Sequence Covering Arrays*

Sequence covering arrays can be constructed with a variety of methods. A 2-way sequence covering array can be constructed simply by listing the events in some order for one test and in reverse order for the second test:

| | | | | |
|---|---|---|---|---|
| **1** | *a* | *b* | *c* | *d* |
| **2** | *d* | *c* | *b* | *a* |

To see that this procedure generates tests that cover all 2-way sequences, note that for 2-way sequence coverage, every pair of variables *x* and *y*, *x..y* and *y..x* must both be in some test (where *a..b* means that *a* is eventually followed by *b*). All variables are included in each test, therefore any sequence *x..y* must be in either test 1 or test 2 and its reverse *y..x* in the other test. Thus only 2 tests are needed to cover all 2-way sequences, regardless of the number of events to be included in the tests. This can be an effective way of doing initial tests on a GUI with multiple buttons, text input boxes, selection lists, and other features. Invoking each of the features on screen in some order and then reversing the order may uncover problems in memory management or initialization (often as a result of developers' assumptions about the order in which the user will interact with the system.)

The number of tests required for *t*-way coverage of *n* events is proportional to *t*! log *n*., and the lower bound for a sequence covering array grows logarithmically in *n* [97]. Therefore, a large number of events can be tested using a reasonable number of tests for most applications, as can be confirmed in **Error! Reference source not found.**. Greedy methods produce good results across a broad range of problem sizes. Construction methods for sequence covering arrays also include answer-set programming [11, 58]. Answer set programming can generate more compact test sets than greedy methods, but this advantage may not hold for larger problem sizes.

*Generalized t-way Sequence Covering*

For *t-way sequence* test generation, where $t > 2$, one method is to use a greedy algorithm that generates a large number of tests, scores each by the number of previously uncovered sequences it covers, then chooses the highest scoring test. This simple approach produces surprisingly good results, in both test set size and execution time.

---

**Algorithm** *t-seq*(int *t*, int *n*)
// *t* = interaction strength; *n* = # parameters, $n > t$;
*N* = # candidate tests to generate
initialize test set *ts* to be an empty set;
initialize set *chk* of $n \times (n-1) \times ... \times (n-t+1)$ bits to 0;
**while** (all *t-way sequence*s not marked in *chk*) {
1. *tc* := set of *N* test candidates generated with random values of each of the *n* parameters
2. $test_1$ := test from set *tc* that covers the greatest number of sequences not marked as covered in *chk*;
3. **for each** new sequence covered in $test_1$, mark corresponding bit in set *chk* to 1;
4. *ts* := *ts* ∪ $test_1$ ;
5. **if** (symmetry && all *t-way sequence*s not marked in *chk*) { $test_2$ := reverse($test_1$);
   *ts* := *ts* ∪ $test_2$ ;
   **for each** new sequence cover in $test_2$,
   mark corresponding bit in set *chk* to 1; }

}
**return** *ts*;

---

**Figure 24.** Algorithm *t-seq*

The complexity of the algorithm is dominated by the selection of a candidate test that covers the greatest number of previously uncovered sequences. An array of bits for each possible *t*-way sequence is used so that marking and testing the array for a particular sequence can be done in constant time for each of the *t*-way sequences This selection process checks each of the $n \times (n-1) \times ... \times (n-t+1)$ possible *t*-way sequences to determine if the sequence has previously been covered or is newly covered by the candidate test. The check is done for each of the *N* candidate tests, with constant *N*, so the time complexity of the algorithm is $O(n^t)$. Storage required for the algorithm is $O(n^t)$ also, because of the set *chk* for keeping track of which sequences have been covered at each step.

It is shown in [97] that the number of tests generated by a greedy algorithm grows logarithmically with *n*. At each step, a greedy algorithm that selects the test which covers the largest number of previously uncovered sequences will progress at a rate of at least $1/t!$ of the remaining sequences at each iteration. Thus uncovered sequences are reduced as $U_{i+1} = U_i(1 - 1/t!)$, and after *k* iterations, remaining uncovered sequences will be $U_0(1 - 1/t!)^k$. Initially, $U_0 = n \times (n-1) \times ... \times (n-t+1)$. For small *n*, it may be possible to implement an optimal greedy algorithm that tests all *n*! possible tests. For larger values of *n*, the algorithm may be reasonably close to finding an optimal next test, with sufficient candidates.

### *10.3 Using Sequence Covering Arrays*

Sequence covering arrays have been incorporated into operational testing for a mission-critical system that uses multiple devices with inputs and outputs to a laptop computer. The test procedure has 8 steps: boot system, open application, run scan, connect peripherals P-1 through P-5. It is expected that for some sequences, the system will not function properly, thus the order of connecting peripherals is a critical aspect of testing. In addition, there are constraints on the sequence of events: can't scan until the app is open; can't open app until system is booted. There are 40,320 permutations of 8 steps, but some are redundant (e.g., changing the order of peripherals connected before boot), and some are invalid (violates a constraint). Around 7,000 are valid, and non-redundant, but this is far too many to test for a system that requires manual, physical connections of devices.

The system was tested using a seven-step sequence covering array, incorporating the assumption that there is no need to examine strength-3 sequences that involve boot-up. The initial test configuration (**Error! Reference source not found.**) was drawn from the library of pre-computed sequence tests. Some changes were made to the pre-computed sequences based on unique requirements of the system test. If 6='Open App' and 5='Run Scan', then cases 1, 4, 6, 8, 10, and 12 are invalid, because the scan cannot be run before the application is started. This was handled by 'swapping 0 and 1' when they are adjacent (1 and 4), out of order. For the other cases, several cases were generated from each that were valid mutations of the invalid case. A test was also embedded to see whether it mattered where each of three USB connections were placed. The last test case ensures at least strength 2 (sequence of length 2) for all peripheral connections and 'Boot', i.e., that each peripheral connection occurs prior to boot. The final test array is shown in Table 15.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Test 1** | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| **Test 2** | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| **Test 3** | 2 | 1 | 0 | 6 | 5 | 4 | 3 |
| **Test 4** | 3 | 4 | 5 | 6 | 0 | 1 | 2 |
| **Test 5** | 4 | 1 | 6 | 0 | 3 | 2 | 5 |
| **Test 6** | 5 | 2 | 3 | 0 | 6 | 1 | 4 |
| **Test 7** | 0 | 6 | 4 | 5 | 2 | 1 | 3 |
| **Test 8** | 3 | 1 | 2 | 5 | 4 | 6 | 0 |
| **Test 9** | 6 | 2 | 5 | 0 | 3 | 4 | 1 |
| **Test 10** | 1 | 4 | 3 | 0 | 5 | 2 | 6 |
| **Test 11** | 2 | 0 | 3 | 4 | 6 | 1 | 5 |
| **Test 12** | 5 | 1 | 6 | 4 | 3 | 0 | 2 |

**Figure 25.** Seven-event tests from pre-computed test library.

### *10.4 Cost and Practical Considerations*

As with other forms of combinatorial testing, some combinations may be either impossible or not exist on the system under test. For example, ‘receive message’ must occur before ‘process message’. One algorithm for sequence covering arrays makes it possible to specify pairs *x,y*, where the sequence *x..y* is to be excluded from the generated covering array. Typically this will lead to extra tests, but does not increase the test array significantly.

Sequence covering can be realtively inexpensive as a test technique. As noted previously, only two tests are needed to produce 2-way covering, and the number of tests grows only as log *n* for *n* events for $t > 2$. **Error! Reference source not found.** shows the number of tests for 3-way and 4-way

sequences. Different algorithms may produce slightly fewer or more tests than shown, but numbers will be similar.

| Events | 3-seq Tests | 4-seq Tests |
|---|---|---|
| **5** | 8 | 29 |
| **6** | 10 | 38 |
| **7** | 12 | 50 |
| **8** | 12 | 56 |
| **9** | 14 | 68 |
| **10** | 14 | 72 |
| **11** | 14 | 78 |
| **12** | 16 | 86 |
| **13** | 16 | 92 |
| **14** | 16 | 100 |
| **15** | 18 | 108 |
| **16** | 18 | 112 |
| **17** | 20 | 118 |
| **18** | 20 | 122 |
| **19** | 22 | 128 |
| **20** | 22 | 134 |
| **21** | 22 | 134 |
| **22** | 22 | 140 |
| **23** | 24 | 146 |
| **24** | 24 | 146 |
| **25** | 24 | 152 |
| **26** | 24 | 158 |
| **27** | 26 | 160 |
| **28** | 26 | 162 |
| **29** | 26 | 166 |
| **30** | 26 | 166 |
| **40** | 32 | 198 |
| **50** | 34 | 214 |
| **60** | 38 | 238 |
| **70** | 40 | 250 |
| **80** | 42 | 264 |

Table 14. Number of tests for combinatorial 3-way and 4-way sequences.

## *10.5 Chapter Summary*

Sequence covering arrays are a new application of combinatorial methods, developed to solve problems with interoperability testing. A sequence-covering array is a set of tests that ensures all *t-way sequence*s of events have been tested. The *t* events in the sequence may be interleaved with others, but all permutations will be tested. All 2-way sequences can be tested simply by listing the events to be tested in any order, then reversing the order to create a second test. Algorithms have been developed to create sequence covering arrays for higher strength interaction levels. For a given interaction strength, the number of tests generated is proportional to the log of the number of events.

As with other types of combinatorial testing, constraints may be important, since it is very common that certain events depend on others occurring first. The tools developed for this problem allow the user to specify constraints in the form of excluded sequences which will not appear in the generated test array.

Table 15. Final sequence covering array used in testing.

| Original Case | Case | Step1 | Step2 | Step3 | Step4 | Step5 | Step6 | Step7 | Step8 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | Boot | P-1 (USB-RIGHT) | P-2 (USB-BACK) | P-3 (USB-LEFT) | P-4 | P-5 | Application | Scan |
| 2 | 2 | Boot | Application | Scan | P-5 | P-4 | P-3 (USB-RIGHT) | P-2 (USB-BACK) | P-1 (USB-LEFT) |
| 3 | 3 | Boot | P-3 (USB-RIGHT) | P-2 (USB-LEFT) | P-1 (USB-BACK) | Application | Scan | P-5 | P-4 |
| 4 | 4 | Boot | P-4 | P-5 | Application | Scan | P-1 (USB-RIGHT) | P-2 (USB-LEFT) | P-3 (USB-BACK) |
| 5 | 5 | Boot | P-5 | P-2 (USB-RIGHT) | Application | P-1 (USB-BACK) | P-4 | P-3 (USB-LEFT) | Scan |
| 6A | 6 | Boot | Application | P-3 (USB-BACK) | P-4 | P-1 (USB-LEFT) | Scan | P-2 (USB-RIGHT) | P-5 |
| 6B | 7 | Boot | Application | Scan | P-3 (USB-LEFT) | P-4 | P-1 (USB-RIGHT) | P-2 (USB-BACK) | P-5 |
| 6C | 8 | Boot | P-3 (USB-RIGHT) | P-4 | P-1 (USB-LEFT) | Application | Scan | P-2 (USB-BACK) | P-5 |
| 6D | 9 | Boot | P-3 (USB-RIGHT) | Application | P-4 | Scan | P-1 (USB-BACK) | P-2 (USB-LEFT) | P-5 |
| 7 | 10 | Boot | P-1 (USB-RIGHT) | Application | P-5 | Scan | P-3 (USB-BACK) | P-2 (USB-LEFT) | P-4 |
| 8A | 11 | Boot | P-4 | P-2 (USB-RIGHT) | P-3 (USB-LEFT) | Application | Scan | P-5 | P-1 (USB-BACK) |
| 8B | 12 | Boot | P-4 | P-2 (USB-RIGHT) | P-3 (USB-BACK) | P-5 | Application | Scan | P-1 (USB-LEFT) |
| 9 | 13 | Boot | Application | P-3 (USB-LEFT) | Scan | P-1 (USB-RIGHT) | P-4 | P-5 | P-2 (USB-BACK) |
| 10A | 14 | Boot | P-2 (USB-BACK) | P-5 | P-4 | P-1 (USB-LEFT) | P-3 (USB-RIGHT) | Application | Scan |
| 10B | 15 | Boot | P-2 (USB-LEFT) | P-5 | P-4 | P-1 (USB-BACK) | Application | Scan | P-3 (USB-RIGHT) |
| 11 | 16 | Boot | P-3 (USB-BACK) | P-1 (USB-RIGHT) | P-4 | P-5 | Application | P-2 (USB-LEFT) | Scan |
| 12A | 17 | Boot | Application | Scan | P-2 (USB-RIGHT) | P-5 | P-4 | P-1 (USB-BACK) | P-3 (USB-LEFT) |
| 12B | 18 | Boot | P-2 (USB-RIGHT) | Application | Scan | P-5 | P-4 | P-1 (USB-LEFT) | P-3 (USB-BACK) |
| NA | 19 | P-5 | P-4 | P-3 (USB-LEFT) | P-2 (USB-RIGHT) | P-1 (USB-BACK) | Boot | Application | Scan |

# 11 Assertion-Based Testing

Built-in self-test is a common feature for integrated circuits in which additional hardware and software functions allow checking for correct operation while the system is running, in contrast with the use of externally applied tests. A similar concept – embedded assertions – has been used for decades in software, and advances in programming languages and processor performance have made this method even more useful and practical. It can be especially effective with combinatorial testing. If the system is fully exercised with *t*-way tests, and assertions are thorough, we can have reasonable confidence that operation will be correct for up to *t*-way combinations of input values. In addition to standard programming language features, a variety of specialized tools have been developed to make this approach easier and more effective.

Many programming languages include an *assert* feature that allows the programmer to specify properties that are assumed true at a particular point in the program. For example, a function that includes a division in which a particular parameter `x` will be used as a divisor may require that this parameter may never be zero. This function may include the C statement `assert(x != 0);` as the first statement executed. Note that the assertion is not the same as an input validity check that issues an error message if input is not acceptable. The assertion specifies conditions that must hold for the function to operate properly, in this case a non-zero divisor. The distinction between assertions and input validation code is that assertions are intended to catch *programming mistakes*, while input validation detects errors in user or file/database input.

*With self-testing through assertions, thousands of tests can often be run at very low cost, allowing high-strength interaction coverage.*

With a sufficient number of assertions derived from a specification, the program can have a self-checking property [73, 0, 123, 109]. The assertions can serve as a sort of embedded proof of important properties, such that if the assertions pass for all executions of the program, then the properties encoded in the assertions can be expected to hold. Then, if the assertions form a chain of logic that implies a formal statement of program properties, the program's correctness with respect to these properties can be proven. We can take advantage of this scheme in combinatorial testing by demonstrating that the assertions hold for all *t*-way combinations of inputs. While this is not the same as a correctness proof, it is an effective way of integrating formal methods for correctness with program testing, and an extensive body of research has developed this idea for practical use (for a survey, see [9]). Modern programming languages, include support for including assertions that encode program properties, and tools such as the Java Modeling Language [102] have been designed to integrate assertions with testing. In many cases, using assertions to self-check important properties makes it practical to run thousands of tests in a fully automated fashion, so high-strength interactions of 4-way and above can be done in reasonable time. Since important properties of the system are checked with every run, by executing the code with all t-way combinations of input we can have high confidence that it works correctly.

## *11.1 Basic Assertions for Testing*

To clarify this somewhat abstract discussion, we will analyze requirements for a small function that handles withdrawal processing for an automated teller machine (ATM). Graphical user interface code for the ATM will not be displayed, as this would vary considerably for different systems. The decision not to include GUI code in this example also illustrates a practical limitation of this type of testing: there are many potential sources of error in a software project, and testing may not deal with all of them at the same time. The GUI code may be analyzed separately, or a more complex verification with

assertions may specify properties of the GUI calls, but in the end some human involvement is needed to ensure that the screen information is properly displayed. However, we can do very thorough testing of the most critical aspects of the withdrawal module.

Requirements for the module are as follows:

1. Some accounts have a minimum balance requirement, indicated by boolean variable `minflag`.
2. The bank allows all customers a basic overdraft protection amount, but for a fee, customers may purchase overdraft protection that exceeds the default.
3. If the account has a minimum balance, the withdrawal cannot reduce account balance below `(minimum balance – overdraft default)` unless overdraft protection is set for this account and the allowed overdraft amount for this account exceeds the default, in which case the balance cannot be reduced below (`minimum balance – overdraft amount).`
4. No withdrawals may exceed the default limit (to keep the ATM from running out of cash), although some customers may have a withdrawal limit below this amount, such as minors who have an account with limits placed by parents.
5. The overdraft privilege can be used only once until the balance is made positive again.
6. Cards flagged as stolen are to be captured and logged in the hot card file. No withdrawal is allowed for a card flagged as stolen.

The module has these inputs from the user after the user is authorized by another module:

```
string num:  the user card number
int amt:  withdrawal amount requested
```

and these inputs from the system:

```
int balance:  user account balance
boolean minflag:  account has minimum balance requirement
int min:  account minimum balance
boolean odflag: account has overdraft protection
int odamt:  overdraft protection amount,
int oddefault: overdraft default
boolean hot:  card flagged as stolen
boolean limflag:  withdrawal limit less than default
int limit:  withdrawal limit for this account
int limdefault:  withdrawal limit default
```

How should these requirements be translated into assertions and used in testing? Consider requirement 1: if `minflag` is set, then the balance before and after the withdrawal must be no less than the minimum balance amount. This could be translated directly into logic for assertions: `minflag => balance >= min.` If the assertion facility does not include logical implication, then the equivalent expression can be used, for example, in C syntax: `!minflag || balance >= min.`

However, we must also consider overdraft protection and withdrawal limits, so the assertion above is not adequate. Collecting conditions, we can develop assertions for each of the eight possible settings of `minflag`, `odflag`, and `limflag`. If there is a minimum balance requirement, no overdraft protection,

and a withdrawal limit below the default, what is the relationship between balance and the other parameters?

```
minflag && !odflag && limflag
     => balance >= min – oddefault && amt <= limit
```

This relation must hold after the withdrawal, so to develop an assertion that must hold immediately before the withdrawal, substitute (balance – amt) for balance in the expression above:

```
balance0 – amt >= min – oddefault && amt <= limit
```

Assertions such as this would be placed *immediately* before the balance is modified, not at the beginning of the code for the withdrawal function. Code prior to the subtraction from balance should have ensured that properties encoded by assertions hold immediately before the subtraction, thus any violation of the assertions indicates an error in the code (or possibly in the assertions!) that must be investigated. This is illustrated in Figure 26, where "wdl_init.c" and "wdl_final.c" are files containing assertions such as developed above.

Including the card number, there are 11 parameters for this module. We need to partition the inputs to determine what values to use in generating a covering array. Partitions should cover valid and invalid values, minimum and maximum for ranges, and values at and on either side of boundaries. The bank uses a check digit scheme for card numbers to detect errors such as digit transposition when numbers are entered manually. A simple partition could be as follows:

```
string acct:  {valid, invalid}
int amt:  {0, divisible by 20, not divisible by 20, max}
int balance:  {0, negative, positive, max int}
int minflag:  {T, F}
int min:  {0, negative, positive, max int}
boolean odflag: {T, F}
int odamt:  {0, negative, positive, max int}
int oddefault: {0, negative, positive, max int}
boolean hot:  {T, F}
int acctlim:  {0, negative, positive, max int}
int lim:  {0, negative, positive, max int}
```

Using the equivalence classes above, this is thus a $2^4 4^7$ system, or 262,144 possible inputs. If values on either side of boundaries are used, the number of possible input combinations will be much larger, but using combinatorial methods we can cover 3-way or 4-way combinations with only a few hundred tests.

### *11.2 Stronger Assertion-based Testing*

*The quality of assertion-based testing with combinatorial methods depends on the strength of assertions, in addition to t-way interaction strength.*

While the method described in the previous section can be very effective in testing, notice that it will be inadequate for many problems, because basic assertion functions such as those provided in the C language library do not support important logic operators such as $\forall$ (*for all*) and $\exists$ (*for some*). Thus expressing simple properties such as *S is sorted in ascending order* = $\forall i : 0 \le i < n-1 : S[i] \le S[i+1]$ cannot be done without a good deal of additional coding. While it would be possible to add code to handle these problems in assertions, a better solution is to use an assertion language that is designed for the purpose and contains all the necessary features.

```
1. while (!valid(acct)) {/* get account number input */}
2. if (amt  > lim ) { return ERROR;  }
3. else {
4.  if (odflag ) {
5.        if (amt  > balance  + odamt )
6.        { return ERROR;  }
7.  }
8. else {
9.  if (amt  > balance  + oddefault )
10.       {return ERROR;  }
11.       else {
12.             if (amt  > lim )
13.             { return ERROR;  }
14.       }
15. #include "wdl_init.c"
16. balance  -= amt ;
17. #include "wdl_final.c"
18. }
19. }
20. }
```

**Figure 26.** Withdrawal function code to be tested.

Tools such as Anna [106] for Ada, the Java Modeling language (JML) [102] and iContract [76] for Java, and APP [150] or Nana [108] for C, can be used to introduce complex assertions, effectively embedding a specification within the code. An example of JML [191] can be seen in Figure 27. The assertions are annotated with "//@", to indicate statements that are input to the pre-processor. JML provides a collection of keywords making it possible to specify the behavior of software and have the specifications checked as the program runs. Other assertion languages may use different keywords, but usually provide similar functionality. The basic run-time assertion checking features illustrated in the example are:

- `//@ requires`: defines a precondition, i.e. a condition that must hold on entry to a module or section of code
- `//@ ensures`: defines a postcondition, i.e. a condition that must hold on exit from a module or section of code
- `//@ public invariant`: defines and invariant, i.e. a condition that must always hold
- `\old`: the value of the variable or expression on entry to the method
- `\result`: the return value of the method

JML and other assertion languages also provide features to make them easy to use for a specific programming language, and additional logic statements, such as the quantifiers `forall`, `exists` (*for some*), and logical implications: `a ==> b, a <== b, a <=> b.`

```
public class BankingExample {

   public static final int MAX_BALANCE = 1000;
   private /*@ spec_public @*/ int balance;
   private /*@ spec_public @*/ boolean isLocked = false;

   //@ public invariant balance >= 0 && balance <= MAX_BALANCE;
```

```
    //@ assignable balance;
    //@ ensures balance == 0;
    public BankingExample() { balance = 0; }

    //@ requires 0 < amount && amount + balance < MAX_BALANCE;
    //@ assignable balance;
    //@ ensures balance == \old(balance + amount);
    public void credit(int amount) { balance += amount; }

    //@ requires 0 < amount && amount <= balance;
    //@ assignable balance;
    //@ ensures balance == \old(balance) - amount;
    public void debit(int amount) { balance -= amount; }

    //@ ensures isLocked == true;
    public void lockAccount() { isLocked = true; }

    //@   requires !isLocked;
    //@   ensures \result == balance;
    //@ also
    //@   requires isLocked;
    //@   signals_only BankingException;
    public /*@ pure @*/ int getBalance() throws BankingException {
        if (!isLocked) { return balance; }
        else { throw new BankingException(); }
    }
}
```

**Figure 27.** Toy Bank Module Example in JML

## *11.3 Cost and Practical Considerations*

Assertions may be a cost-effective approach to test automation because they can be a simple extension of coding. In general, use of assertions is correlated with reduced error rates [100], but a very wide range of effectiveness results from variations in usage. In many applications, assertions are used in a very basic way, such as ensuring that null pointers are not passed to a function that will use them, or that parameters that may be used as divisors are non-zero.

More complex assertions can provide stronger assurance, but there are limits to their effectiveness. For example, invariants (properties that are expected to hold throughout a computation) cannot be assured without placing an assertion for every line of code. Since assertions must be executed to show the presence or absence of a property at some point, errors that prevent the assertion from being reached may not be detected. As an example, consider the code in Figure 26. If a coding error in the first few lines of the function prevents execution the code at of lines 15 and 17, the assertions will not be executed and it may be assumed that the test was passed. In this case, an ERROR return for the particular test case might trigger an investigation that would identify the faulty code, but this may not happen with other applications. Specialized assertion-checking languages such as JML can alleviate many of these problems by providing preprocessor statements to generate code that implements such complex checking without making the program difficult to read.

## *11.4 Chapter Summary*

Assertions are one of the easiest to use and most effective approaches to dealing with the oracle problem. Properties ranging from simple parameter checks to effectively embedded proofs can be encoded in assertions, but special language support is needed for the stronger forms of assurance. This support may be provided as language preprocessors, as in the case of Anna [106] and others. Placement within code is particularly important to assertion effectiveness [0, 183], but if sufficiently strong assertions are embedded, the code becomes self-checking for important properties. With self-checking code, thousands of tests can be run at low cost in most cases, greatly improving the chances that faults will be detected.

# 12 Model-Based Testing

Probably the most time-consuming aspect of testing is the oracle problem: determining the correct results for a given set of inputs. This problem is especially complex when the expected result of a test requires human intervention. Even if the output can be fully processed and verified correct by machine, for every test, the expected output must be determined. At first glance, the problem may seem almost insoluble: how can we check the correctness of complex software without implementing equally complex software, whose correctness must also be checked, leading to an infinite string of verification exercises? Assertions and self-checking software can help (Chapter 11), but they are not always sufficient. In other cases, previous versions of the software may be available to check at least the old functionality, or the code may be implementing a formal standard (e.g. for network protocols or cryptography) and other implementations may exist to compare against. In most cases though, the software is doing something new, and we need to verify that it is working correctly for a large set of possible inputs. The difficulty of devising a set of complete tests with inputs and expected results is one of the reasons why somewhat *ad hoc* approaches such as "use cases" are widespread. Testers use formal or informal requirements to determine anticipated system uses, plus inputs and outputs for each such use, a slow and expensive way to develop a test oracle. To make thorough testing practical, more automated approaches are needed.

*Model-based testing can provide very strong assurance, with a tradeoff of additional up-front time.*

One of the most effective ways to produce test oracles is to use a model of the system under test, and generate complete tests, including both input data and expected results, directly from the model. We use the term model in the same way it would be used in other branches of engineering: the model incorporates aspects of the system that we want to study, but not every detail just as an aircraft model might be used in a wind tunnel to evaluate airflow but not all characteristics of a design. Models in software testing may be used to check calculations or performance, for example, but not other properties such as the location of a particular numeric value on a screen. (If it did include all details, the model would be equivalent to the system itself). This chapter provides a step-by-step introduction to model-based automated generation of tests that provide combinatorial coverage. Procedures introduced in this tutorial will produce a set of complete tests, i.e., input values with the expected output for each set of inputs.

In addition to the ACTS covering array generator, (see **Error! Reference source not found.**), we use NuSMV [41], a variant of the original SMV model checker. NuSMV is freely available and was developed by Carnegie Mellon University, Instituto per la Ricerca Scientifica e Tecnolgica (IRST), U. of Genova, and U. of Trento. NuSMV can be installed on either UNIX/Linux or Windows systems running Cygwin. Links and instructions for downloading NuSMV are found at http://nusmv.fbk.eu/. The methods described in this chapter could of course be used with other model checkers as well, with some adaptation as needed for differences in capabilities of the different tools.

Also needed is a formal or semi-formal specification of the system or subsystem under test (SUT). This can be in the form of a formal logic specification, but state transition tables, decision tables, pseudo-code, or structured natural language can also be used, as long as the rules are unambiguous. The specification will be converted to SMV code, which provides a precise, machine-processable set of rules that can be used to generate tests.

## *12.1 Overview*

To apply combinatorial testing, two tasks must be accomplished:

1. Using ACTS, construct a set of tests that will cover all *t*-way combinations of parameter values. The covering array specifies test data, where each row of the array can be regarded as a set of parameter values for an individual test (see Chapter 4).

2. Determine what output should be produced by the SUT for each set of input parameter values. The test data output from ACTS will be incorporated into SMV specifications that can be processed by the NuSMV model checker for this step. In many cases, the conversion to SMV will be straightforward. The example in Section 12.2 illustrates a simple conversion of rules in the form "if *condition* then *action*" into the syntax used by the model checker. The model checker will instantiate the specification with parameter values from the covering array once for each test in the covering array. Because the model checker works to *disprove* claims, the resulting specification is evaluated against a claim that negates each specified result $R_j$ to produce the expected result as a counterexample. Thus the model checker evaluates claims in the following form: $C_i$ => ~$R_j$, where $C_i$ is a set of parameter values in one row of the covering array in the form $p_1 = v_{i1}$ & $p_2 = v_{i2}$ & ... & $p_n = v_{in}$, and $R_j$ is one of the possible results. The output of this step is a set of counterexamples that show how the SUT can reach the claimed result $R_j$ from a given set of inputs.

The example in the following sections illustrates how these counterexamples are converted into tests. Other approaches to determining the correct output for each test can also be used. For example, in some cases we can run a model checker in simulation mode, producing expected results directly rather than through a counterexample.

The completed tests can be used to validate correct operation of the system for interaction strengths up to some pre-determined level *t*. Depending on the system type and level of effort, we may want to use pairwise (*t*=2) or higher strength, up to *t*=6 way interactions. We do not claim this guarantees correctness of the system, as there may be failures triggered only by interaction strengths greater than *t*. In addition, some of the parameters are likely to have a large number of possible values, requiring that they be abstracted into equivalence classes. If the abstraction does not faithfully represent the range of values for a parameter, some flaws may not be detected by the equivalence class members used.

## *12.2 Access Control System Example*

Here we present a small example of a very simple access control system. The rules of the system are a simplified multi-level security system, given below, followed by a step-by-step construction of tests using a fully automated process.

Each subject (user) has a clearance level *u_l*, and each file has a classification level, *f_l*. Levels are given as 0, 1, or 2, which could represent levels such as Confidential, Secret, and Top Secret. A user *u* can read a file f if $u\_l \geq f\_l$ (the "no read up" rule), or write to a file if $f\_l \geq u\_l$ (the "no write down" rule).

Thus a pseudo-code representation of the access control rules is:

```
if u_l >= f_l & act = rd then GRANT;
else if f_l >= u_l & act = wr then GRANT;
else  DENY;
```

Tests produced will check that these rules are correctly implemented in a system.

*SMV Model*

This system is easily modeled in SMV as a simple two-state finite state machine. The START state merely initializes the system (line 8, Figure 28), with the rule above used to evaluate access as either GRANT or DENY (lines 9-13). For example, line 9 represents the first line of the pseudo-code above: in the current state (always START for this simple model), if $u\_l \geq f\_l$ then the next state is GRANT. Each line of the case statement is examined sequentially, as in a conventional programming language. Line 12 implements the "else DENY" rule, since the predicate "1" is always true. SPEC clauses given at the end of the model are simple "reflections" that duplicate the access control rules as temporal logic statements. They are thus trivially provable, but we are interested in using them to generate tests rather than to prove properties of the system.

```
MODULE main
1.      VAR
--Input parameters
2.      u_l:   0..2;          -- user level
3.      f_l:   0..2;          -- file level
4.      act:  {rd,wr};        -- action

--output parameter
5.      access: {START_, GRANT,DENY};

6.      ASSIGN
7.      init(access) := START_;
--if access is allowed under rules, then next state is GRANT
--else next state is DENY
8.      next(access) := case
9.      u_l >= f_l & act = rd : GRANT;
10.     f_l >= u_l & act = wr : GRANT;
11.     1 : DENY;
12.     esac;
13.     next(u_l) := u_l;
14.     next(f_l) := f_l;
15.     next(act) := act;

-- if user level is at or above file level then read is OK
SPEC AG ((u_l >= f_l & act = rd ) -> AX (access = GRANT));

-- if user level is at or below file level, then write is OK
SPEC AG ((f_l >= u_l & act = wr ) -> AX (access = GRANT));

-- if neither condition above is true, then DENY any action
SPEC AG (!( (u_l >= f_l & act = rd ) | (f_l >= u_l & act = wr ))
          -> AX (access = DENY));
```

**Figure 28.** SMV model of access control rules

Separate documentation on SMV should be consulted to fully understand the syntax used, but specifications of the form "AG ((*predicate 1*) -> AX (*predicate 2*))" indicate essentially that for all paths (the "A" in "AG") for all states globally (the "G"), if *predicate 1* holds then ( "->") for all paths, in the next state (the "X" in "AX") *predicate 2* will hold. In the next section we will see how this specification

can be used to produce complete tests, with test data input and the expected output for each set of input data.

*If a property cannot be proved, the model checker produces a counterexample, giving inputs and paths that lead to the violation.*

Model checkers can be used to perform a variety of valuable functions, because they make it possible to evaluate whether certain properties are true of the system model. Conceptually, the model checker can be viewed as exploring all states of a system model to determine if a property claimed in a SPEC statement is true. If the statement can be proved true for the given model, the model checker reports this fact. What makes a model checker particularly valuable for many applications, though, is that if the statement is false, the model checker not only reports this, but also provides a "counterexample" showing how the claim in the SPEC statement can be shown false. The counterexample will include input data values and a trace of system states that lead to a result contrary to the SPEC claim (Figure 29). In the process described in this section, the input data values will be the covering array generated by ACTS.

For advanced uses in test generation, this counterexample generation capability is very useful for proving properties such as liveness (absence of deadlock) that are difficult to ensure through testing. In this tutorial, however, we will simply use the model checker to determine whether a particular input data set makes a SPEC claim true or false. That is, we will enter claims that particular results can be reached for a given set of input data values, and the model checker will tell us if the claim is true or false. This gives us the ability to match every set of input test data with the result that the system should produce for that set of input data.

The model checker thus automates the work that normally must be done by a human tester – determining what the correct output should be for each set of input data. In some cases, we may have a "reference implementation", that is, an implementation of the functions that we are testing that is assumed to be correct. This happens, for example, in conformance testing for protocols, where many vendors implement their own software for the protocol and submit it to a test lab for comparison with an existing implementation of the protocol. In this case the reference implementation could be used for determining the expected output, instead of the model checker. Of course before this can happen the reference implementation itself must be thoroughly tested before it can be the gold standard for testing other products. The method we describe here may be needed to produce tests for the original reference implementation.

Checking the properties in the SPEC statements shows that they match the access control rules as implemented in the FSM, as expected. In other words, the claims we made about the state machine in the SPEC clauses can be proven. This step is used to check that the SPEC claims are valid for the model defined previously. If NuSMV is unable to prove one of the SPECs, then either the spec or the model is incorrect. This problem must be resolved before continuing with the test generation process. Once the model is correct and SPEC claims have been shown valid for the model, counterexamples can be produced that will be turned into test cases, by which we mean a set of test inputs with the expected result for these inputs. In other words, ACTS is used to generate tests, then the model checker determines expected results for each test.

```
-- specification AG((u_l >= f_l & act = rd) -> AX access = GRANT)
    is true
```

```
-- specification AG((f_l >= u_l & act = wr) -> AX access = GRANT)
    is true
-- specification AG(!((u_l >= f_l & act = rd)|(f_l >= u_l & act = wr))
                                    -> AX access = DENY)  is true
```

**Figure 29.** NuSMV output

*Integrating Combinatorial Tests into the Model*

We will compute covering arrays that give all *t*-way combinations, with interaction strength = 2 for this example. This section describes the use of ACTS as a standalone command line tool, using a text file input (see **Error! Reference source not found.**). The first step is to define the parameters and their values in a system definition file that will be used as input to ACTS. Call this file "in.txt", with the following format:

```
[System]
[Parameter]
   u_l: 0,1,2
   f_l: 0,1,2
   act: rd,wr
[Relation]
[Constraint]
[Misc]
```

For this application, the [Parameter] section of the file is all that is needed. Other tags refer to advanced functions that will be explained in other documents. After the system definition file is saved, run ACTS as shown below:

```
java -Ddoi=2 –jar acts_cmd.jar ActsConsoleManager in.txt out.txt
```

The "-Ddoi=2" argument sets the interaction strength (degree of interaction) for the covering array that we want ACTS to compute. In this case we are using simple 2-way, or pairwise, interactions. (For a system with more parameters we would use a higher strength interaction, but with only three parameters, 3-way interaction would be equivalent to exhaustive testing.) ACTS produces the output shown in Figure 30.

Each test configuration defines a set of values for the input parameters u_l, f_l, and act. The complete test set ensures that all 2-way combinations of parameter values have been covered. If we had a larger number of parameters, we could produce test configurations that cover all 3-way, 4-way, etc. combinations. ACTS may output "don't care" for some parameter values. This means that any legitimate value for that parameter can be used and the full set of configurations will still cover all t-way combinations. Since "don't care" is not normally an acceptable input for programs being tested, a random value for that parameter is substituted before using the covering array to produce tests.

```
Number of parameters: 3
  Maximum number of values per parameter: 3
  Number of configurations: 9
  ------------------------------------
  Configuration #1:
  1 = u_l=0
  2 = f_l=0
  3 = act=rd
  ------------------------------------
  Configuration #2:
  1 = u_l=0
  2 = f_l=1
  3 = act=wr
  ------------------------------------
  Configuration #3:
  1 = u_l=0
  2 = f_l=2
  3 = act=rd
  ------------------------------------
  Configuration #4:
  1 = u_l=1
  2 = f_l=0
  3 = act=wr
  ------------------------------------
  Configuration #5:
  1 = u_l=1
  2 = f_l=1
  3 = act=rd
  ------------------------------------
  Configuration #6:
  1 = u_l=1
  2 = f_l=2
  3 = act=wr
  ------------------------------------
  Configuration #7:
  1 = u_l=2
  2 = f_l=0
  3 = act=rd
  ------------------------------------
  Configuration #8:
  1 = u_l=2
  2 = f_l=1
  3 = act=wr
  ------------------------------------
  Configuration #9:
  1 = u_l=2
  2 = f_l=2
  3 = (don't care)
```

**Figure 30.** ACTS output

The next step is to assign values from the covering array to parameters used in the model. For each test, we claim that the expected result will not occur. The model checker determines combinations that would disprove these claims, outputting these as counterexamples. Each counterexample can then be converted to a test with known expected result. Every test from the ACTS tool is used, with the model

checker supplying expected results for each test. (Note that the trivially provable positive claims have been commented out. Here we are concerned with producing counterexamples.)

Recall the structure introduced in Section 12.1: $C_i => \sim R_j$. Here $C_i$ is the set of parameter values from the covering array. For example, for configuration #1 in Section:

```
u_l = 0 & f_l = 0 & act = rd
```

As can be seen below, for each of the 9 configurations in the covering array we create a SPEC claim of the form:

```
SPEC AG(( <covering array values> ) -> AX !(access = <result>));
```

This process is repeated for each possible result, in this case either “GRANT” or “DENY”, so we have 9 claims for each of the two results. The model checker is able to determine, using the model defined in Section 12.2, which result is the correct one for each set of input values, producing a total of 9 tests.

Excerpt:

```
...
-- reflection of the assign for access
--SPEC AG ((u_l >= f_l & act = rd ) -> AX (access = GRANT));
--SPEC AG ((f_l >= u_l & act = wr ) -> AX (access = GRANT));
--SPEC AG (!((u_l >= f_l & act = rd ) | (f_l >= u_l & act = wr ))
            -> AX (access = DENY));

SPEC AG((u_l = 0 & f_l = 0 & act = rd) -> AX !(access = GRANT));
SPEC AG((u_l = 0 & f_l = 1 & act = wr) -> AX !(access = GRANT));
SPEC AG((u_l = 0 & f_l = 2 & act = rd) -> AX !(access = GRANT));
SPEC AG((u_l = 1 & f_l = 0 & act = wr) -> AX !(access = GRANT));
SPEC AG((u_l = 1 & f_l = 1 & act = rd) -> AX !(access = GRANT));
SPEC AG((u_l = 1 & f_l = 2 & act = wr) -> AX !(access = GRANT));
SPEC AG((u_l = 2 & f_l = 0 & act = rd) -> AX !(access = GRANT));
SPEC AG((u_l = 2 & f_l = 1 & act = wr) -> AX !(access = GRANT));
SPEC AG((u_l = 2 & f_l = 2 & act = rd) -> AX !(access = GRANT));

SPEC AG((u_l = 0 & f_l = 0 & act = rd) -> AX !(access = DENY));
SPEC AG((u_l = 0 & f_l = 1 & act = wr) -> AX !(access = DENY));
SPEC AG((u_l = 0 & f_l = 2 & act = rd) -> AX !(access = DENY));
SPEC AG((u_l = 1 & f_l = 0 & act = wr) -> AX !(access = DENY));
SPEC AG((u_l = 1 & f_l = 1 & act = rd) -> AX !(access = DENY));
SPEC AG((u_l = 1 & f_l = 2 & act = wr) -> AX !(access = DENY));
SPEC AG((u_l = 2 & f_l = 0 & act = rd) -> AX !(access = DENY));
SPEC AG((u_l = 2 & f_l = 1 & act = wr) -> AX !(access = DENY));
SPEC AG((u_l = 2 & f_l = 2 & act = rd) -> AX !(access = DENY));
```

## *12.3 Generating Tests from Counterexamples*

NuSMV produces counterexamples where the input values would disprove the claims specified in the previous

*Counterexamples from the model checker can be post-processed into complete tests, with inputs and expected output for each.*

section. Each of these counterexamples is thus a set of test data that would have the expected result of GRANT or DENY.

For each SPEC claim, if this set of values cannot in fact lead to the particular result $R_j$, the model checker indicates that this is true. For example, for the configuration below, the claim that access will not be granted is true, because the user's clearance level (`u_l = 0`) is below the file's level (`f_l = 2`):

```
-- specification AG (((u_l = 0 & f_l = 2) & act = rd) -> AX !(access =
GRANT))  is true
```

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 system under test.

Excerpt from NuSMV output:

```
-- specification AG (((u_l = 0 & f_l = 0) & act = rd) -> AX
   access = GRANT))  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  u_l = 0
  f_l = 0
  act = rd
  access = START_
-> Input: 1.2 <-
-> State: 1.2 <-
  access = GRANT
```

The model checker finds that 6 of the input parameter configurations produce a result of GRANT and 3 produce a DENY result, so at the completion of this step we have successfully matched up each input parameter configuration with the result that should be produced by the SUT.

We now strip out the parameter names and values, giving tests that can be applied to the system under test. This can be accomplished using a variety of methods; a simple script used in this example is given in the appendix. The test inputs and expected results produced are shown below:

```
u_l = 0 & f_l = 0 & act = rd -> access = GRANT
u_l = 0 & f_l = 1 & act = wr -> access = GRANT
u_l = 1 & f_l = 1 & act = rd -> access = GRANT
u_l = 1 & f_l = 2 & act = wr -> access = GRANT
u_l = 2 & f_l = 0 & act = rd -> access = GRANT
u_l = 2 & f_l = 2 & act = rd -> access = GRANT
u_l = 0 & f_l = 2 & act = rd -> access = DENY
u_l = 1 & f_l = 0 & act = wr -> access = DENY
u_l = 2 & f_l = 1 & act = wr -> access = DENY
```

These test definitions can now be post-processed using simple scripts written in PERL, Python, or similar tool to produce a test harness that will execute the SUT with each input and check the results. While tests for this trivial example could easily have been constructed manually, the procedures introduced in this tutorial can, and have, been used to produce tens of thousands of complete test cases in a few minutes, once the SMV model has been defined for the SUT.

## *12.4 Cost and Practical Considerations*

Model based test generation trades up-front analysis and specification time against the cost of greater human interaction for analyzing test results. The model or formal specification may be costly to produce, but once it is available, large numbers of tests can be generated, executed, and analyzed without human intervention. This can be an enormous cost savings, since testing usually requires 50% or more of the software development budget. For example, suppose a $100,000 development project expects to spend $50,000 on testing, because of the staff time required to code and run tests, and analyze results. If a formal model can be created for $20,000, complete tests generated and analyzed automatically, with another $10,000 for a smaller number of human-involved tests and analysis, then the project will save 20%. One tradeoff for this savings is the requirement for staff with skills in formal methods, but in some cases this approach may be practical and highly cost-effective.

Model-based testing can reduce overall cost because of the tradeoffs involved.

One nice property of the model checking approach described in this chapter is that test case generation can be run in parallel. For each test row of the covering array, we run the model checker to determine the expected results for the inputs given by that row, and model checker runs are independent of each other. Thus this task falls into the class of parallelization problems known as "embarrassingly parallel"; for *N* covering array rows, we can assign up to *N* processors. With the widespread availability of cloud and cluster systems, test generation can run very quickly. In most cases, test execution can be run in parallel also, although we may be limited by practical concerns such as availability of specialized hardware.

## *12.5 Chapter Summary*

1. The oracle problem must be solved for any test methodology, and it is particularly important for thorough testing that produces a large number of test cases. One approach to determining expected results for each test input is to use a model of the system that can be simulated or analyzed to compute output for each input.

2. Model checkers can be used to solve the oracle problem because whenever a specified property for a model does not hold, the model checker generates a counter-example. The counter-example can be post-processed into a complete working test harness that executes all tests from the covering array and checks results.

3. Several approaches are possible for integrating combinatorial testing with model checkers, but some present practical problems. The method reported in this chapter can be used to generate full combinatorial test suites, with expected results for each test, in a cost effective way.

# 13 Fault Localization

Developing dependable software requires preventing as many bugs as possible and detecting, then repairing, those that remain. Testing can identify flaws in software, but after a failed test is discovered, it is necessary to determine what caused the failure. In most cases this may be accomplished for combinatorial testing in the same way as other test methodologies, using a debugger or in-circuit emulator. But one goal of combinatorial testing is to identify the particular *t*-way combination that triggered a failure. The problem of fault localization, identifying such combination(s), is an area of active research, but some basic approaches can be identified. The discussion in this chapter assumes systems are deterministic, such that a particular input always generates the same output.

At first glance, fault localization may not appear to be a difficult problem, and in many cases it will not be, but we want to automate the process as much as possible. To understand the size of the problem, consider a module that has 20 input parameters. A set of 3-way covering tests passes 100%, but several tests derived from a 4-way covering array result in failure. (Therefore, at least four parameter values are involved in triggering the failure. It is possible that a 5-way or higher combination caused the failure, since any set of *t*-way tests also includes (*t*+1)-way and higher strength combinations as well.) A test with 20 input parameters has C(20, 4) = 4,845 4-way combinations, yet presumably only one (or just a few) of these triggered the failure. To determine the combination at fault, a variety of strategies can be used.

## *13.1 Set-theoretic Analysis*

The analysis presented here applies to a deterministic system, in which a particular set of input values always results in the same processing and outputs. Let *P* = {combinations in passing tests} and *F* = {combinations in failing tests} and *C* = {fault-triggering combinations}. Then $F \setminus P$, combinations in failing tests that are not in any passing tests, must contain the fault-triggering combinations *C* because if any of those in *C* were in *P*, then the test would have failed. So in most cases, $C \subseteq F \setminus P$, as shown in Figure 31.

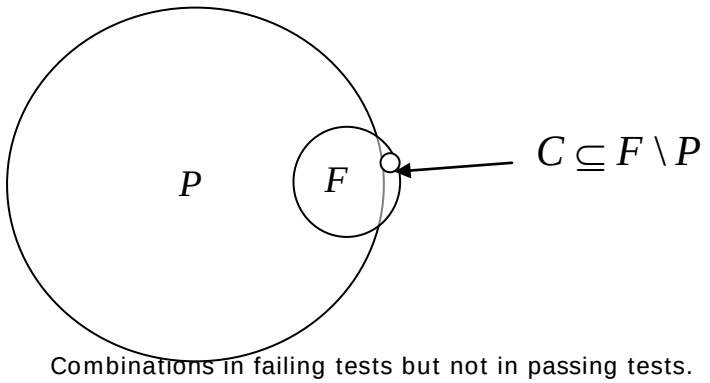



**Figure 31.** Combinations in failing tests but not in passing tests.

Continuing with the analysis in this manner, some properties become apparent. For the discussion below, *Pt* = {combinations in *t*-way passing tests}, with *Ft* and *Ct* defined analogously. Let *Tt* = {*t*-way tests} and *f*(*x*) be a function that indicates whether a test *x* passes or fails for the system under test. Thus *P*4 = {combinations in 4-way passing tests}, *T*5= {5-way tests}, etc.

Suppose that a particular combination *c* triggers or causes a failure if whenever *c* is contained in some test *x*, *f*(*x*) = *fail*. (That is, the system is deterministic and the failure-triggering combination is not masked by

other parameter values.) We can now consolidate these ideas into heuristics for identifying the failure-triggering combination(s) *C*.

- *Elimination*: For a deterministic system, $F \setminus P$ must contain the fault-triggering combinations *C* because if any of those in *C* were in *P*, then the test would have failed.

- *Interaction level lower bound*: If all *t*-way tests pass, then a *t*-way or lower strength combination did not cause the failure. The failure must have been caused by a (*t*+*k*)-way combination, for some $k > t$. Note that the converse is not necessarily true: if some *t*-way test fails, we cannot conclude that a *t*-way test caused the failure, because any *t*-way test set contains some *k*-way combinations, for $k > t$.

- *Interaction continuity*: Now consider *Ct*. Because *t*-way tests cover all combinations of *t*-way or lower strength (e.g., 4-way tests also cover all 3-way combinations), a combination that triggered the failure in *Ft* must also occur in *F*(*t*+1), *F*(*t*+2), etc. Therefore we can further reduce the potential failure-triggering combinations by computing $Ft \bigcap F(t+1) \bigcap ... \bigcap F(t+k)$ for whatever interaction strength *k* we have tests available.

- *Value dependence*: If tests in *Ft* cover all values for a *t*-way parameter combination *c*, then the failure is independent of *c*; i.e., *c* is not a *t*-way failure-triggering combination(s).

**Example:** In the preceding discussion we assumed that a particular combination *c* triggers or causes a failure if whenever *c* is contained in some test *x*, *f*(*x*) = *fail*. However, in many cases the presence of a particular combination may trigger a failure, but is not guaranteed to do so (see discussion of interaction level lower bound above). Consider the following:

```
1. p(int a, int b, int c, int d, int e) {
2. if (a && b)          return 1;
3. else if (c && d)     return 2;
4. else if (e)          return 3;
5. else                 return 4;
6. }
```

If line 3 is incorrectly implemented as “return 7” instead of “return 2”, then p(1,1,1,1,0) = 1 because “a && b” evaluates to 1, but p(0,1,1,1,0) will detect the error. A complete 3-way covering test set will detect the error because it must include at least one test with values 0,1,1,1,. and one with 1,0,1,1,. . Figure 32 shows tests for this example for t = 2, 3, and 4. Failing tests are underlined.

A 2-way test *may* detect the error, since “c && d” is the condition necessary, but this will only occur if line 3 is reached, which requires either a=0 or b=0. In the example test set this occurs with the second test. So in this case, a full 2-way test set has detected the error, and the heuristics above for 2-way combinations will find that tests with *c*=1 and *d*=1 occur in both *P* and *F*. In this case, debugging may identify c=1, d=1 as a combination that triggers the failure, but automated analysis using the heuristics will find two 3-way combinations that occur in failing tests but not passing tests: a=0, c=1, d=1 and b=0, c=1, d=1. As Figure 33 illustrates, in most cases we will find more than one combination identified as possible causes of failure.

| 1 way tests | 2 way tests | 3 way tests | 4 way tests |
|---|---|---|---|
| 0,0,0,0,0 | 0,0,0,0,0 | 0,0,0,0,0 | 0,0,0,0,0 |
| 1,1,1,1,1 | 0,1,1,1,1 | 0,0,1,1,1 | 0,0,0,1,1 |
| | 1,0,1,0,1 | 0,1,0,1,0 | 0,0,1,0,1 |
| | 1,1,0,1,0 | 0,1,1,0,1 | 0,0,1,1,0 |
| | 1,1,1,0,0 | 1,0,0,1,1 | 0,1,0,0,1 |
| | 1,0,0,1,1 | 1,0,1,0,0 | 0,1,0,1,0 |
| | | 1,1,0,0,1 | 0,1,1,0,0 |
| | | 1,1,1,1,0 | 0,1,1,1,1 |
| | | 0,0,1,1,0 | 1,0,0,0,1 |
| | | 1,1,0,0,0 | 1,0,0,1,0 |
| | | 0,0,0,0,1 | 1,0,1,0,0 |
| | | 1,1,1,1,1 | 1,0,1,1,1 |
| | | 0,1,1,1,0 | 1,1,0,0,0 |
| | | | 1,1,0,1,1 |
| | | | 1,1,1,0,1 |
| | | | 1,1,1,1,0 |

**Figure 32.** Tests for fault location example.

The heuristics above can be applied to combinations in the failed tests to identify possible failure-triggering combinations, shown in Figure 33.

- The 1-way tests do not detect any failures, but the 2-way tests do, so *t*=2 is a lower bound for the interaction level needed to detect a failure.

- The value dependence rule applies to combination "be" – since all four possible values for this combination occur in failing tests, failure must be independent of combination be. In other words, we do not consider the pair be to be a cause of failure because it does not matter what value this pair has. Every test must have some value for these parameters.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| *t*=2 | **ab**<br>01<br>00<br>10 | **ac**<br>01<br>11 | **ad**<br>01<br>11 | **ae**<br>01<br>00<br>11 | **bc**<br>11<br>01 | **bd**<br>11<br>01 | **be**<br>11<br>01<br>00<br>10 | **cd**<br>11 | **ce**<br>11<br>10 | **de**<br>11<br>10 |
| *t*=3 | **abc**<br>011<br>001<br>101 | **abd**<br>011<br>001<br>101 | **abe**<br>011<br>001<br>000<br>101<br>010 | **acd**<br>011<br>111 | **ace**<br>011<br>010<br>111 | **ade**<br>011<br>010<br>111 | **bcd**<br>111<br>011 | **bce**<br>111<br>011<br>010<br>110 | **bde**<br>111<br>011<br>010<br>110 | **cde**<br>111<br>110 |
| *t*=4 | **abcd**<br>0111<br>0011<br>1011 | **abce**<br>0111<br>0011<br>0010<br>1011<br>0110 | **abde**<br>0111<br>0011<br>0010<br>1011<br>0110 | **bcde**<br>1111<br>0111<br>0110<br>1110 | | | | | | |

**Figure 33.** Combinations in failing tests.

- The elimination rule can be applied to determine that there are no 1-way or 2-way combinations that do not appear in both passing and failing tests. Results for 3-way and 4-way combinations are shown in Figure 34. These results were produced by an analysis tool which outputs in the

format <test number>:<*t* level> <parameter numbers> = <parameter values>. Two different 3-way combinations are identified: a=0, c=1, d=1 and b=0, c=1, d=1. A large number of 4-way combinations are also identified, but we can use the interaction continuity rule to show that one of the two 3-way combinations occurs in all of the failing 4-way failing tests. Therefore we can conclude that covering all 3-way parameter interactions would detect the error.

| | |
|---|---|
| 1 :3way 0,2,3 = 0,1,1 | 1 :4way 0,1,2,3 = 0,0,1,1 |
| 2 :3way 0,2,3 = 0,1,1 | 2 :4way 0,1,2,3 = 0,0,1,1 |
| 3 :3way 0,2,3 = 0,1,1 | 3 :4way 0,1,2,3 = 0,1,1,1 |
| 4 :3way 0,2,3 = 0,1,1 | 4 :4way 0,1,2,3 = 0,1,1,1 |
| 1 :3way 1,2,3 = 0,1,1 | 5 :4way 0,1,2,3 = 1,0,1,1 |
| 2 :3way 1,2,3 = 0,1,1 | 1 :4way 0,1,2,4 = 0,0,1,0 |
| 5 :3way 1,2,3 = 0,1,1 | 1 :4way 0,1,3,4 = 0,0,1,0 |
| | 4 :4way 0,1,3,4 = 0,1,1,1 |
| | 1 :4way 0,2,3,4 = 0,1,1,0 |
| | 2 :4way 0,2,3,4 = 0,1,1,1 |
| | 3 :4way 0,2,3,4 = 0,1,1,0 |
| | 4 :4way 0,2,3,4 = 0,1,1,1 |
| | 1 :4way 1,2,3,4 = 0,1,1,0 |
| | 2 :4way 1,2,3,4 = 0,1,1,1 |
| | 5 :4way 1,2,3,4 = 0,1,1,1 |

**Figure 34.** 3-way and 4-way combinations in $F \setminus P$

The situation is more complex with continuous variables. If, for example, a failure-related branch is taken any time $x > 100$, $y = 3$, $z < 1000$, there may be many combinations implicated in the failure. Analysis will show that [x = 200, y = 3, z = 120], [x = 201, y = 3, z = 119], [x = 999, y = 3, z = 999], [x = 101, y = 3, z = 0], [x = 200, y = 3, z = 0] are all combinations that trigger the failure. With more than three input parameters, there may be dozens or hundreds of failure-triggering combinations, even though there is most likely a single point in the code that is in error.

## *13.2 Fault Localization Using Fault Identifier Tool*

This section describes a method of software fault localization using the Fault Identifier tool. It shows how, for a failure-triggering fault *x* containing *t* variable values, a small set of possible failure-triggering combinations can be identified with a *t*-way covering array, and how fault *x* can be identified uniquely with a variety of techniques. Simple set operations can identify a small number of suspect *t*-way combinations that must contain the failure-triggering combination, if tests from a full *t*-way covering array are run on the SUT. Once tests have been run, the test set is augmented with additional tests in one of three ways described below.

**Procedure:**

Step 1. *Run tests*. All tests are run against the system under test, and the rows of the array divided into two sets, *F* = tests that produced a failure or other faulty operation, and *P* = tests that did not detect faulty operation.

Step 2. *Compute set difference F\P*, the combinations that occur in failing tests but not in passing tests. The tool identifies the tests in which these occur.

Step 3. *Create augmented passing set*. For each failing test identified in Step 2, create additional tests using either the alternate value or base choice procedures detailed below. Not all failing

tests may be included in the list from Step 2 because combinations that have already been identified will not be repeated. Set $P^+ = P \cup$ {passing base choice tests}.

Step 3 (alternative). Generate a (t+1)-way covering array, and set P+ = (t+1) covering array.

Step 4. *Compute* $F \backslash P^+$ to identify failure triggering combinations. Select 2-way through 6-way analysis, depending on the strength of the covering array used in testing, and the type of analysis desired.

**Test augmentation.** Additional tests are generated according to one of the methods below. Each new test is run and those that pass are added to set *P* to produce set $P^+$. We then compute $F \backslash P^+ = c$, detecting the FT combination.

*Alternate value.* Let $T_f$ be a failing test. For each of the i=1..n parameters, create one new test for each parameter i, with all other parameter values held constant. For example, if we have five binary parameters, Tf = 01011, create 11011, 00011, 01111, 01001, and 01010. This procedure generates *kn* new tests, where *k* = number of tests identified in Step 2; *n* = number of parameters.

*Base choice*: Let $T_f$ be a failing test. For each of the i=1..n parameters, create one new test for each parameter, for each value of parameter i, with all other parameter values held constant. For example, if each parameter has three possible values, 0, 1, and 2, and Tf = 10212, create 20212, 00212, 11212, 12212, etc. The base choice procedure creates $kn(v\text{-}1)$ new tests, where *k* = number of tests identified in Step 2; *n* = number of parameters; v = number of values per parameter. If parameters have different numbers of values, vi values for parameter I, then $\sum_{i=1,n} (v_i\text{-}1)$ new tests are created.

**Example, test augmentation:**
We have binary variables *a* through *e*, and the 2-way combination *a*=0, *b*=1 triggers a failure.
A covering array for this system is:

0,0,0,0,0
0,1,1,1,1
1,0,1,0,1
1,1,0,1,0
1,1,1,0,0
0,0,0,1,1

The row containing a=0, b=1 is 0,1,1,1,1, which becomes set *F*. Set *P* is combinations from:

0,0,0,0,0
1,0,1,0,1
1,1,0,1,0
1,1,1,0,0
0,0,0,1,1

Now generate the base choice tests, as shown below. Note that not all of the base choice tests may be necessary, since we only need to disrupt the FT combination *c* such that *T'* passes, so it is not necessary to run all before recomputing the set difference. An alternative is to generate the supplemental tests one at a time and do the computation F\P after each test. The choice of procedures depends on the tradeoff between test execution time.

The additional tests and results are:

1,1,1,1,1 (pass)
0,0,1,1,1 (pass)
0,1,0,1,1 (fail)
0,1,1,0,1 (fail)
0,1,1,1,0 (fail)

Using the new information from the base choice tests, we have two additional passing tests: 1,1,1,1,1 and 0,0,1,1,1. Adding these to the previous passing set $P$ to produce $P^+$, we have

0,0,1,1,1
0,0,0,0,0
1,0,1,0,1
1,1,0,1,0
1,1,1,0,0
0,0,0,1,1
1,1,1,1,1

Then computing $F \setminus P^+ = c = (ab = 01)$, thus correctly identifying the fault. In this case we detected that $a$ was involved in the failure on the first test, but note that we would need at most $n$-1 new base choice tests to find a parameter involved in failure (at worst, the last two parameters would be the fault-triggering combination). Thus this method requires a total of $\sum_{i=1,n} (v_i - 1)$ new tests to augment the existing set.

**Example, (t+1)-way augmentation:** As above we have binary variables $a$ through $e$, and the 2-way combination $a$=0, $b$=1 triggers a failure.
A covering array for this system is:

0,0,0,0,0
0,1,1,1,1
1,0,1,0,1
1,1,0,1,0
1,1,1,0,0
0,0,0,1,1

The row containing a=0, b=1 is 0,1,1,1,1, which becomes set $F$. Set $P$ is combinations from:

0,0,0,0,0
1,0,1,0,1
1,1,0,1,0
1,1,1,0,0
0,0,0,1,1

A 3-way covering array for $a$-$e$ is below (two rows with $a$=0, $b$=1 are shown at the end of the array):

0,0,0,0,0
0,0,1,1,1
1,0,0,1,1
1,0,1,0,0
1,1,0,0,1
1,1,1,1,0
0,0,1,1,0

1,1,0,0,0
0,0,0,0,1
1,1,1,1,1
0,1,0,1,0
0,1,1,0,1

So *P*+ is the set of combinations from the array above with the last two lines removed. Now any 2-way combination that can be made from either *a* or *b* and some other variable in 0,1,1,1,1, must be duplicated in *P*+. If one of these six contains *a*=0, then it must not contain *b*=1, because otherwise it would be in *F*. So it must be in *P*+. If it does not contain either *a* or *b*, then there must be more than one (three) 3-way combinations containing it. Not all of these can be in rows of *F* because *F* contains all failing tests. (end of example)

**Tool Example.** A hypothetical application has 20 variables with 3 values each, and two failure triggering faults: v0, v1, v2 = 1,1,2, and v4, v5, v6 = 2,1,1. Tests are run using 3-way and 4-way covering arrays. Figure 1 shows the tool output for the 3-way test array, and Figure 2 for the 4-way test array. Rows of the failing test file are shown in the top panel, and 2-way, 3-way, or 4-way combinations that may trigger the fault are shown in the lower panels. In Fig. 1, there are 138 possible combinations that could have triggered the failure, i.e., combinations that occurred in 3-way failing tests but not in 3-way passing tests. Running 3-way analysis with passing tests from a 4-way covering array, shown in Fig. 2, identifies the two failure-triggering combinations: v0=1, v1=1, v2=2, and v4=2, v5=1, v6=1. In the Background section below, it is shown how this approach is able to identify failure-triggering combinations exactly.

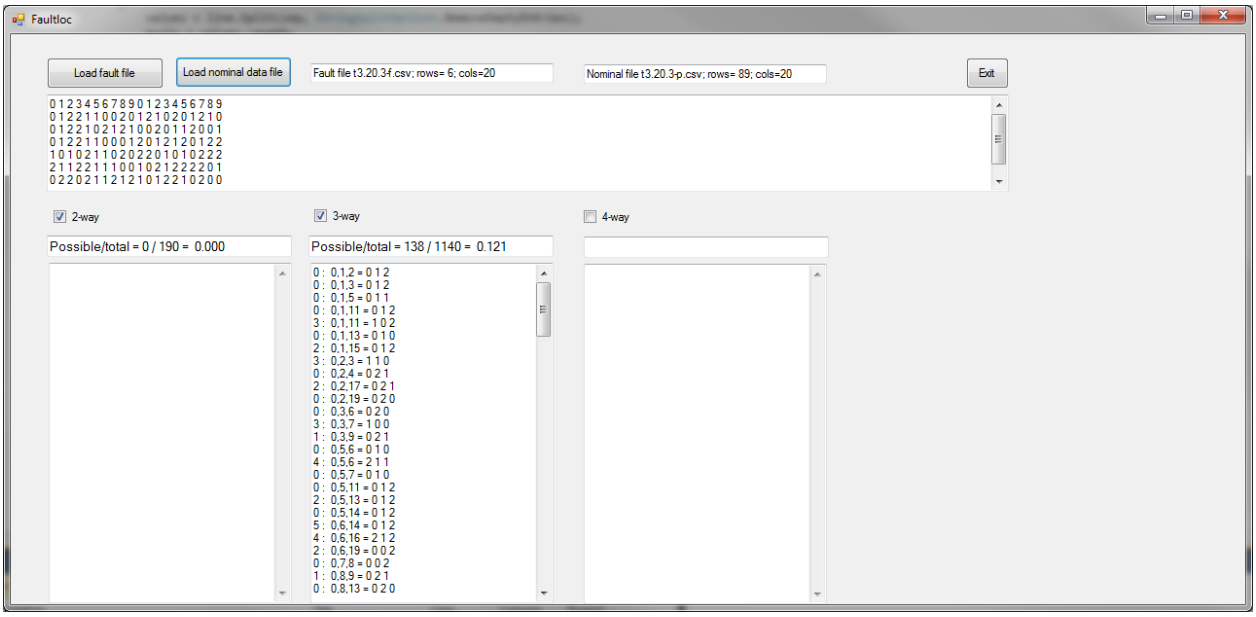



**Figure 1.** Tool output for 3-way combinations in the 3-way test array.

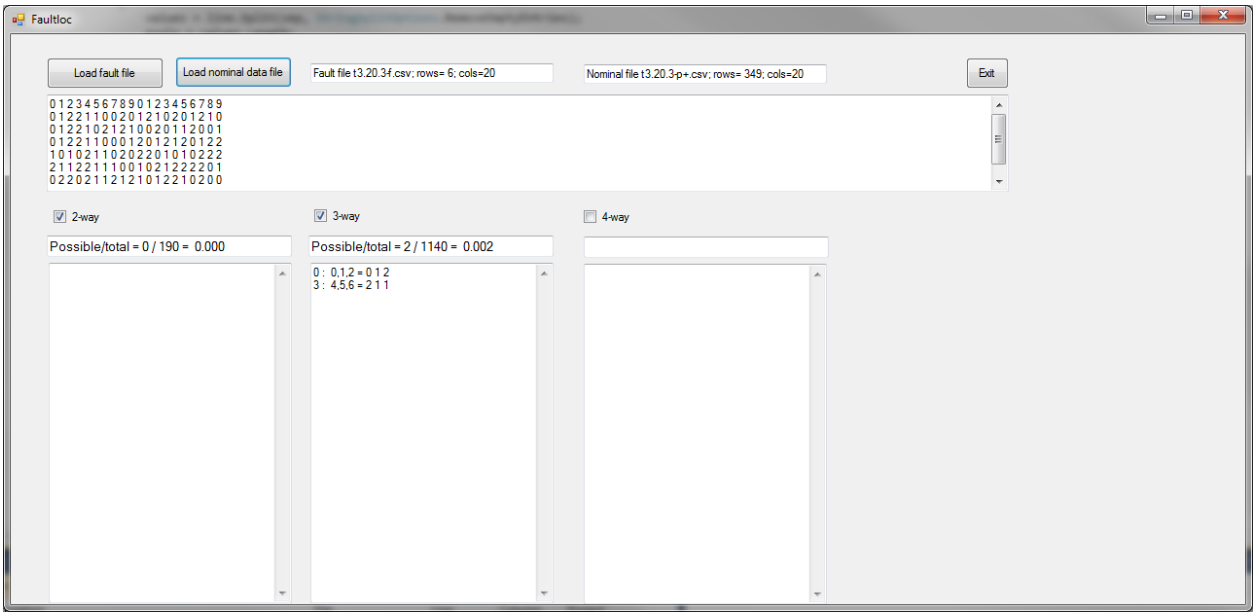



**Figure 2.** Tool output for 3-way combinations in the 4-way test array.

Given these sets, $F \backslash P$ will produce a small set of suspect combinations. Examples are given in Table 1. Loading $F$ with the "Load fault file" and $P$ with "Load nominal file" will display suspect combinations from $F \backslash P$ in the panels for 2-way through 4-way combinations, as selected on the screen. Loading $P+$ instead of $P$ will result in the computation of $F \backslash P+ = c$. This works because $c$ is a $t$-way combination that triggers the failure, $c$ is in $F$ but is not in $P$ or $P+$. For $F \backslash P+ = c$, $P+$ must contain at least one of all $t$-way combinations except $c$. To see that all combinations but $c$ are in $P+$ consider that $c$ contains $t$ values among the $n$ variables. Any $t$-way combination $d \neq c$ must have at least one value different from $c$. We know that $A+$ contains all $(t+1)$-way combinations, and therefore also all $t$-way combinations. For $c$ to be unique in $F \backslash P+$, we need to show that every combination $d \neq c$ in $F$ also has a copy in $P+$. $A+$ contains all $(t+1)$-way combinations, so $P+$ contains all rows that do not contain $c$. Any $t$-way combination $d \neq c$ in $F$ must be included in at least $n$-$t$ $(t+1)$-way combinations, because $d$'s variables can be joined with any of the other variables not in $d$ in order to produce a $(t+1)$-way combination. Since $d$ differs from $c$ in at least one variable, there is a $(t+1)$-way combination in $A+$ that includes a variable from $c$ (with a different value for the same variable in $c$) plus the variables of $d$. Because it differs from $c$ in at least one variable value, it is not in $F$, so must be in $P+$. This procedure works for up to $v$-1 faults.

## *13.3 Cost and Practical Considerations*

As shown in the example above, it is a non-trivial matter to determine the failure-triggering combination(s) from test results alone. When source code is available, the methods described in this section are probably unnecessary, and can be replaced with conventional debugging techniques. In black-box testing situations where there is no source code, these methods may be useful in narrowing the search for failure-triggering combinations. Tools to implement these methods have been developed and are available from the ACTS project site.

Determining the approach to fault location in black-box test situations also depends on cost. Clearly, if faults are detected at one level of interaction strength, there may be additional faults, including some that are more complex and only detectable with higher strength test sets. Thus it will usually be desirable to run a ($t$+1)-way test set when faults are detected at level $t$. But going from a $t$-way array to a ($t$+1)-way array requires a much larger set of tests, which may not be practical from a time or cost standpoint. In these cases, base choice augmentation can be a highly cost-effective alternative for fault location.

## *13.4 Chapter Summary*

When source code is available, the best way to identify the cause of a failure is with conventional debugging techniques, since the error must be fixed in code anyway. With pure black-box testing and no access to source code, the heuristics discussed in this chapter may help to narrow down possible causes. Usually there will be many combinations identified as possible causes, so substantial additional testing may be needed to determine the exact cause.

## Appendix A - REFERENCES


1. P. Ammann, P.E. Black, Abstracting Formal Specifications to Generate Software Tests via Model Checking, Proc. *18th Digital Avionics Systems Conference*, Oct. 1999, IEEE, vol. 2. pp. 10.A.6.1-10

2. P. E. Ammann, Knight, J. C. Data diversity: An approach to software fault tolerance. IEEE Transactions on Computers, 37(4), (1988), 418–425.

3. Ammann, P. E. & Offutt, A. J. (1994). Using formal methods to derive test frames in category-partition testing, Proceedings of the Ninth Annual Conference on Computer Assurance (COMPASS'94),Gaithersburg MD, IEEE Computer Society Press, pp. 69-80.

4. P. Ammann, J. Offutt, *Introduction to Software Testing*, Cambridge University Press, New York, 2008.

5. Apilli, B. S., L. Richardson, C. Alexander, Fault-based combinatorial testing of web services. In *Proc. 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications* (Orlando, October 25 - 29, 2009)

6. A. Arcuri, L. Briand, *Adaptive Random Testing: An Illusion of Effectiveness*, International Symposium on Software Testing and Analysis 2011

7. A. Arcuri, L. Briand, "Formal Analysis of the Probability of Interaction Fault Detection Using Random Testing," *IEEE Transactions on Software Engineering*, 18 Aug. 2011. IEEE Computer Society, http://doi.ieeecomputersociety.org/10.1109/TSE.2011.85

8. J. Bach, P. Shroeder, Pairwise Testing - A Best Practice That Isn't. Proceedings of *22nd Pacific Northwest Software Quality Conference*, 2004, pp. 180-196

9. W.A. Ballance, S. Vilkomir, W. Jenkins, Effectiveness of Pair-wise Testing for Software With Boolean Inputs, Workshop on Combinatorial Testing (CT), April 17, 2012, International Conference on Software Testing, (ICST 2012, April 17-21)Montreal, Canada.

10. W. Ballance, W. Jenkins, and S. Vilkomir, "Probabilistic Assessment of Effectiveness of Software Testing for Safety-Critical Systems," *Proceedings of the 10th International Probabilistic Safety Assessment & Management Conference (PSAM 10)*, Seattle, Washington, USA, 7-11 June (2010).

11. M. Banbara, N. Tamura, and K. Inoue, Generating event-sequence test cases by answer set programming with the incidence matrix, in Technical Communications of the 28th International Conference on Logic Programming (ICLP12), 2012, pp. 86-97.

12. L. Baresi, M. Young, Test Oracles, Dept. of Computer and Information Science, Univ. of Oregon, 2001. http://www.cs.uoregon.edu/michal/pubs/oracles.html

13. B. Beizer, Software Testing Techniques, Van Nostrand Reinhold, New York, 2nd edition, 1990.

14. K. Z. Bell and Mladen A. Vouk. On effectiveness of pairwise methodology for testing network-centric software. *Proceedings of the ITI Third IEEE International Conference on Information & Communications Technology*, pages 221–235, Cairo, Egypt, December 2005.

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

16. P. G. Bishop, The variation of software survival times for different operational input profiles. In Proceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS-23) (pp. 98–107). IEEE Computer Society Press.

17. P. E. Black, V. Okun, Y. Yesha, "Testing with Model Checkers: Insuring Fault Visibility", *WSEAS Trans. Sys*., 2 (1): 77-82, Jan. 2003.

18. P. E. Black, V. Okun, Y. Yesha, "Mutation Operators for Specfications", *Automated Software Engineering*, 2000

19. B.W. Boehm, *Software Engineering Economics*, Prentice Hall, 1981.

20. George E. P. Box, W. G. Hunter, and J. Stuart Hunter (1978) Statistics for Experimenters, New York: Wiley

21. R. Brownlie, J. Prowse, and M. S. Phadke (1992) "Robust testing of AT&T PMX/Starmail using OATS," AT&T Technical Journal, 71, pp 41-47

22. R. Bryce, C.J. Colbourn. The Density Algorithm for Pairwise Interaction Testing, *Journal of Software Testing, Verification and Reliability*, August 2007

23. Bryce, R. C.J. Colbourn, M.B. Cohen. *A Framework of Greedy Methods for Constructing Interaction Tests.* The 27th International Conference on Software Engineering (ICSE), St. Louis, Missouri, pages 146-155. (May 2005).

24. [Bryce IST06] R. Bryce and C. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. Journal of Information and Software Technology, 48(10):960–970, 2006.

25. R. Bryce, A. Rajan, M.P.E. Heimdahl, Interaction Testing in Model Based Development: Effect on Model Coverage, *IEEE, 13th Asia Pacific Software Engineering Conference (APSEC'06)* pp. 259-268.

26. R. Bryce, S. Sampath, A. Memon. Developing a Single Model and Test Prioritization Strategies for Event-Driven Software, Transactions on Software Engineering, (January 2011), 37(1):48-64.

27. Renee C. Bryce and Atif M. Memon, Test Suite Prioritization by Interaction Coverage. In Proceedings of The Workshop on Domain-Specific Approaches to Software Test Automation (DoSTA 2007); co-located with the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, (Dubrovnik, Croatia), pp 1-7, September 2007.

28. R. Bryce, S. Sampath, J. Pedersen, S. Manchester. Test Suite Prioritization by Cost-based Combinatorial Interaction Coverage, International Journal on Systems Assurance Engineering and Management (Springer), (April 2011), 2(2): 126-134.

29. R. Bryce, Y. Lei, D.R. Kuhn, R. Kacker, "Combinatorial Testing", Chap. 14, *Handbook of Research on Software Engineering and Productivity Technologies: Implications of Globalization*, Ramachandran, ed. , IGI Global, 2009.

30. Burners-Lee, T. (1994). *Uniform resource locators (URL)*. Retrieved June 2012, from IETF: http://www.ietf.org/rfc/rfc1738.txt

31. K. Burr and W. Young Combinatorial Test Techniques: Table-Based Automation, Test Generation, and Test Coverage, *International Conference on Software Testing, Analysis, and Review (STAR), San Diego, CA, October, 1998*.

32. K. Burroughs, A. Jain, and R. L. Erickson. Improved quality of protocol testing through techniques of experimental design. In *Proceedings of the IEEE International Conference on Communications (Supercomm/ICC'94)*, May 1-5, New Orleans, Louisiana, USA. IEEE, May 1994, pp. 745-752

33. K. A. Bush (1952) "Orthogonal arrays of index unity," Annals of Mathematical Statistics, 23, pp 426-434

34. T. Y. Chen, H. Leung, and I. K. Mak. Adaptive random testing. *Advances in Computer Science*, pages 320–329, 2004.

35. T.Y. Chen, Adaptive Random Testing, Eighth Intl. Conf. on Quality Software, 2008. IEEE, pp. 443.

36. T. Y. Chen, F.C. Kuo, R. G. Merkel, T.H. Tse. Adaptive random testing: The art of test case diversity. Journal of Systems and Software (JSS), 2010. (also HKU-CS Tech Rpt.TR-2009-07.)

37. B. Chen, J. Yan, J. Zhang, Combinatorial Testing with Shielded Parameters, APSEC '10 Proceedings of the 2010 Asia Pacific Software Engineering Conference, IEEE Computer Society, pp. 280-289.

38. B. Chen, J. Zhang, Tuple Density: A New Metric for Combinatorial Test Suites, Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011. ACM 2011, ISBN 978-1-4503-0445-0

39. J. J. Chilenski, An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion, Report DOT/FAA/AR-01/18, April 2001, 214 pp.

40. A. Cimatti, E. Clarke, F. Giunchiglia and M. Roveri. NuSMV: a new symbolic model verifier. In N. Halbwachs and D. Peled, editors. *Proceeding of International Conference on Computer-Aided Verification* (CAV'99). In *Lecture Notes in Computer Science*, no. 1633, pp. 495-499, Trento, Italy, July 1999. Springer Verlag.

41. I. Ciupa, A. Leitner, M. Oriol and B. Meyer: *ARTOO: Adaptive Random Testing for Object-Oriented Software*, in *ICSE 2008: Proceedings of 30th International Conference on Software Engineering*, Leipzig, 10-18 May 2008, IEEE Computer Society Press, 2008,

42. E. M. Clarke, K. L. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 419-422, New Brunswick, NJ, USA, July/August 1996. Springer Verlag.

43. L. Clarke, H. Hassell, and D. Richardson. "A Close Look at Domain Testing." *IEEE Transactions on Software Engineering*, 1982: 380–392.

44. William G. Cochran and M. G. Cox (1950) Experimental Designs, New York: Wiley

45. [Codd 1972] E.F. Codd. Further normalization of the data base relational model. R. Rustin, editor, *Data Base Systems*, pages 33-64, Prentice-Hall, 1972.

46. M.B. Cohen, J. Snyder, G. Rothermel. Testing Across Configurations: Implications for Combinatorial Testing, *Workshop on Advances in Model-based Software Testing*, Raleigh, Nov. 2006, pp. 1-9

47. D. M. Cohen, S. R. Dalal, J. Parelius, G. C. Patton The Combinatorial Design Approach to Automatic Test Generation, *IEEE Software*, Vol. 13, No. 5, pp. 83-87, September 1996

48. L. Copeland, *A Practitioner's Guide to Software Test Design*, Artech House Publishers, Boston, 2004.

49. M. Cohen, M. Dwyer, J. Shi. Interaction testing of highly-configurable sys-tems in the presence of constraints. *International Symposium on Software Testing and Analysis*, vol 4961/2008, pp 129–139, 2007.

50. M. Cohen, M. Dwyer, J. Shi. Coverage and adequacy in software product line testing. *International Symposium on Software Testing and Analysis*, pp 53–63, 2006.

51. D.M. Cohen, S.R. Dalal, J. Parelius, and G.C. Patton, *"The Combinatorial Design Approach to Automatic Test Generation"*, IEEE Software, Vol. 13, No. 5, pp. 83-87, (1996).

52. D.M. Cohen, S.R. Dalal, M.L. Fredman, and G.C. Patton, *"The AETG System: An Approach to Testing Based on Combinatorial Design"*, IEEE Transactions on Software Engineering, Vol. 23, No. 7, pp. 437–444, (1997).

53. Jacek Czerwonka (webpage) http://www.pairwise.org/

54. J. Czerwonka, *"Pairwise testing in real world: Practical extensions to test case generator"*, Proceedings of 24th Pacific Northwest Software Quality Conference, October 9–11, 2006, Portland, Oregon, USA, pp. 419–430, (2006).

55. Dalal, S.R., C.L. Mallows, Factor-covering Designs for Testing Software, *Technometrics*, v. 40, 1998, pp. 234-243.

56. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, A. Iannino. Applying design of experiments to software testing, *Proceedings of the Intl. Conf. on Software Engineering, (ICSE '97)*, 1997, pp. 205-215, New York

57. L. du Bousquet, Y. Ledru, O. Maury, C. Oriat, J.-L. Lanet, A case study in JML-based software validation. *Proceedings of 19th Int. IEEE Conf. on Automated Sofware Engineering*, pp. 294-297, Linz, Sep. 2004

58. E. Erdem, K. Inoue, J. Oetsch, J. Puhrer, H. Tompits, C. Yilmaz, Answer Set Programming as a new Approach to Event Sequence Testing, VALID 2011, Third Intl. Conference on Advances in System Testing and Validation Lifecycle, IARIA, Oct. 23, 2011, pp. 25-34.

59. R. A. Fisher (1925) Statistical Methods for Research Workers, Edinburgh: Oliver and Boyd

60. R. A. Fisher (1935) The Design of Experiments, Edinburgh: Oliver and Boyd

61. G. B. Finelli, NASA software failure characterization experiments. Reliability Engineering and System Safety, 32(1–2), (1991). pp. 155–169.

62. Michael Forbes, J. Lawrence, Yu Lei, R. N. Kacker, D. R. Kuhn (2008) "Refining the In-Parameter-Order strategy for constructing covering arrays," Journal of Research of NIST, 113, pp 287-297

63. Michael Forbes (webpage) http://math.nist.gov/coveringarrays/

64. Angelo Gargantini and Paolo Vavassori, *CitLab: a Laboratory for Combinatorial Interaction Testing* in *Workshop on Combinatorial Testing (CT) International Conference on Software Testing* (ICST 2012, April 17-21)IEEE Computer Society (2012): 559-568 ISBN 978-0-7695-4670-4

65. D. Giannakopoulou, D.H. Bushnell, J. Schumann, H. Erzberger, K. Heere, "Formal Testing for Separation Assurance", *Ann. Math. Artif. Intell.*, 2011. DOI: 10.1007/s10472-011-9224-3

66. M. Grindal, J. Offutt, S.F. Andler, Combination Testing Strategies: a Survey, *Software Testing, Verification, and Reliability*, v. 15, 2005, pp. 167-199.

67. M. Grindal, J. Offutt, J. Mellin. "Managing Conflicts when Using Combination Strategies to Test Software." *Proceedings of the 2007 Australian Software Engineering Conference (ASWEC'07).* IEEE, 2007.

68. Grochtmann, M. "Test Case Design Using Classification Trees." *STAR'94.* Washington, 1994.

69. M. GROCHTMANN, K. GRIMM. "Classification Trees for Partition Testing." *SOFTWARE TESTING, VERIFICATION AND RELIABILITY*, 1993: VOL. 3, 63-82.

70. Guo, Y. a. (2008). Web application fault classification-an exploratory study. *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement* (pp. 303-305).

71. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

72. A. Hartman. Software and hardware testing using combinatorial covering suites. *Graph Theory, Combinatorics and Algorithms: Interdisciplinary Applications*, pp. 327–266, 2005.

73. A. Hartman and L. Raskin, Problems and algorithms for covering arrays. Discrete Math. 284 (2004), 149–156.

74. A. S. Hedayat, N. J. A. Sloan, and J. Stufken (1999) Orthogonal Arrays: Theory and Applications, New York: Springer

75. C.A.R. Hoare, "Assertions, a Personal Perspective", IEEE Annals of the History of Computing, vol. 25, no. 2, pp. 14-25, 2003.

76. Ari Jaaksi. Developing mobile browsers in a product line. *IEEE Software*, 19(4):73–80, July/August 2002.

77. Raine Kauppinen and Juha Taina. Rita environment for testing framework-based software product lines. In *Proceedings of the Eighth Symposium on Programming Languages and Software Tools*, pages 58–69, June 2003.

78. Chang Hwan Peter Kim , Don S. Batory , Sarfraz Khurshid, Reducing combinatorics in testing product lines, *Proc. 10th international conference on Aspect-oriented software development*, March 21-25, 2011, Porto de Galinhas, Brazil

79. Kamsties, E. & Lott, C. (1995a). An Empirical Evaluation of Three Defect Detection Techniques, Technical Report ISERN 95-02, Dept of Computer Science, University of Kaiserslauten.

80. Kamsties, E. & Lott, C. (1995b). An empirical evaluation of three defect detection techniques, Proceedings of the 5th European Software Engineering Conference (ESEC95), Barcelona, Spain, September 25-28, 1995.

81. Oscar Kempthorne (1952) Design and Analysis of Experiments, New York: Wiley

82. Daniel J. Kleitman, and J. Spencer (1973) "Families of k-independent sets," Discrete Mathematics, 6, pp 255-262

83. R. Kramer, "iContract – The Java Design by Contract Tool". In *Proceedings of TOOLS26*: *Technology of Object-Oriented Languages and Systems*, pp. 295-307, IEEE, 1998.

84. V. Hu, D.R. Kuhn, T. Xie, "Property Verification for Generic Access Control Models", *IEEE/IFIP International Symposium on Trust, Security, and Privacy for Pervasive Applications*, Shanghai, China, Dec. 17-20, 2008.

85. Institute of Electrical and Electronics Engineers, *IEEE Standard Glossary of Software Engineering Terminology*, ANSI/IEEE Std. 729-1983.

86. C. W. Krueger. New Methods in Software Product Line Practice. Communications of the ACM, 49(12):37–40, 2006.

87. P.M. Kruse, J. Wegener, Test Sequence Generation from Classification Trees. *2012 IEEE Fifth International Conference Software Testing, Verification and Validation* (ICST),. 17-21 April 2012, pp. 539-548

88. D.R. Kuhn, "Fault Classes and Error Detection Capability of Specification Based Testing," *ACM Transactions on Software Engineering and Methodology,* Vol. 8, No. 4 (October,1999).

89. R. Kuhn, R. Kacker, Y. Lei, J. Hunter, "Combinatorial Software Testing", *IEEE Computer*, vol. 42, no. 8 (August 2009).

90. D.R. Kuhn, R. Kacker, Y. Lei, "Automated Combinatorial Test Methods: Beyond Pairwise Testing", *Crosstalk, Journal of Defense Software Engineering*, vol. 21, no. 6, June 2008

91. D.R. Kuhn and V. Okun, "Pseudo-exhaustive Testing for Software," *Proceedings of 30th NASA/IEEE Software Engineering Workshop*, pp. 153-158, 2006

92. D.R. Kuhn, M.J. Reilly, An Investigation of the Applicability of Design of Experiments to Software Testing, *27th NASA/IEEE Software Engineering Workshop*, NASA Goddard Space Flight Center, 4-6 December, 2002 .

93. D.R. Kuhn, D.R. Wallace, and A. Gallo, "Software Fault Interactions and Implications for Software Testing," *IEEE Transactions on Software Engineering*, 30(6): 418-421, 2004

94. D.R. Kuhn, R. Kacker, Y.Lei, "Random vs. Combinatorial Methods for Discrete Event Simulation of a Grid Computer Network", *Proceedings, Mod Sim World 2009*, Oct. 14-17 2009, Virginia Beach, pp. 83-88, NASA CP-2010-216205, National Aeronautics and Space Administration.

95. D.R. Kuhn, R. Kacker, Y. Lei, "Combinatorial and Random Testing Effectiveness for a Grid Computer Simulator" NIST Tech. Rpt. 24 Oct 2008.

96. D.R. Kuhn, R. Kacker, Y.Lei, *Practical Combinatorial Testing*, NIST SP 800-142, October, 2010.

97. D. R. Kuhn, J. M. Higdon, J. F. Lawrence, R. N. Kacker, and Y. Lei, Combinatorial methods for event sequence testing, in IEEE Fifth International Conference on Software Testing, Veri_cation and Validation (ICST), 2012, pp. 601-609.

98. D. R. Kuhn, J. M. Higdon, J. F. Lawrence, R. N. Kacker, and Y. Lei, Combinatorial methods for event sequence testing, CrossTalk: The Journal of Defense Software Engineering, 25 (2012), pp. 15-18.

99. D.R. Kuhn, J.M. Higdon, Testing Event Sequences, http://csrc.nist.gov/groups/SNS/acts/sequence_cov_arrays.html Oct., 2009.

100. D.R. Kuhn, Combinatorial Measurement Tool User Guide, Available online at http://csrc.nist.gov/groups/SNS/acts/documents/ComCoverage110130.pdf, Published on January 30, 2011 and last accessed on May 14, 2012.

101. G. Kundrajavets, N. Nagappan, T. Ball, Assessing the Relationship between Software Assertions and Faults: an Empirical Investigation, *Proceedings of 17th International Symposium on Software Reliability Engineering*, IEEE, pp. 204-212, Raleigh, 2006.

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

103. Y. Lei, R. Kacker, D.R. Kuhn, V. Okun, J. Lawrence, "IPOG/IPOG-D: Efficient Test Generation for Multi-Way Combinatorial Testing", *Software Testing, Verification, and Reliability*.

104. Y. Lei and K.C. Tai, *"In-Parameter-Order: A Test Generation Strategy for Pairwise Testing"*, Proceedings of the 3rd IEEE international Symposium on High-Assurance Systems Engineering, November 13-14, 1998, pp. 254-261, (1998)

105. J. Leyden, "Symantec update killed biz PCs in three-way software prang", The Register, July 16, 2012, http://www.theregister.co.uk/2012/07/16/symantec_update_snafu/.

106. D.C. Luckham, F.W. von Henke. "Overview of Anna, a Specification Language for Ada", *IEEE Software*, vol. 2, no. 2, pp. 9-22, March 1985.

107. M. Lyu, ed. *Software Reliability Engineering*, McGraw Hill, 1996.

108. P.J. Maker, *GNU Nana – User's Guide* (version 2.4). Technical report, School of Information Technology – Northern Territory Univ., July 1998.

109. B.A. Malloy, J.M. Voas, "Programming with Assertions – a Prospectus", IEEE IT Professional, vol. 6, no. 5, pp. 53-59, Sept./Oct. 2004.

110. Robert Mandl (1985) "Orthogonal Latin squares: an application of experiment design to compiler testing," Communications of the ACM, 28, pp 1054-1058

111. B. Marick, *The Craft of Software Testing*, Simon & Schuster, 1995.

112. Marick, B. "Test Requirement Catalog: Generic Clues, Developer Version." *www.exampler.com.* 1995. http://www.exampler.com/testing-com/writings/catalog.pdf (accessed Sept. 2012).

113. A.P. Mathur, *Foundations of Software Testing*, Addison-Wesley, New York, 2008.

114. Maughan, C. (2012). *TEST CASE GENERATION USING COMBINATORIAL BASED COVERAGE FOR RICH WEB APPLICATIONS* . Logan, UT: Utah State University.

115. J.R. Maximoff, M.D. Trela, D.R. Kuhn, R. Kacker, "A Method for Analyzing System State-space Coverage within a t-Wise Testing Framework", *IEEE International Systems Conference 2010*, Apr. 4-11, 2010, San Diego.

116. E.J. McCluskey, S. Bozorgui-Nesbat, "Design for Autonomous Test", IEEE Trans. On Computers, vol. C-30, no. 11, Nov. 1981, pp. 866-875.

117. M. Memon and Q. Xie. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Trans. Softw. Eng.*, 31(10):884–896, 2005.

118. C. Montanez, D.R. Kuhn, M. Brady, R.M. Rivello, J. Reyes, and M.K. Powers. "Evaluation of Fault Detection Effectiveness for Combinatorial and Exhaustive Selection of Discretized Test Inputs". *Software Quality Professional* , Volume 14, Issue 3, p. 32-38 (June 2012).

119. Douglas C. Montgomery (2004) Design and Analysis of Experiments, 4-th edition, New York: Wiley

120. J. Musa, G. G. Fuoco, N. Irving, D. Kropfl, B. Jublin, The Operational Profile, Chapter 5 in Handbook of Software Reliability Engineering, M.R. Lyu, ed. McGraw-Hill, 1996.

121. J. Musa, A. Iannino, K. Okumoto, Software Reliability – Measurement, Prediction, Application, McGraw-Hill, 1987.

122. B. Meyer, *Object-Oriented Software Construction*, Second Edition, Prentice Hall, 1997, ISBN 0-13-629155-4

123. B. Meyer, I. Ciupa, A. Leitner, A. Fiva, Y. Wei and E. Stapf: Programs that Test Themselves, IEEE *Computer*, vol. 42, no. 9, pages 46-55, September 2009,

124. G. Myers, *The Art of Software Testing*, John Wiley and Sons, New York, 1979.

125. National Aeronautics and Space Administration, Meteorological Measurement System, http://geo.arc.nasa.gov/sgg/mms/Integration/dc8/mission_manager.htm

126. National Institute of Standards and Technology. Dictionary of Algorithms and Data Structures. http://xlinux.nist.gov/dads/HTML/greedyalgo.html

127. National Institute of Standards and Technology. Test Accelerator. . http://www.itl.nist.gov/div897/docs/testacc.html

128. National Institute of Standards and Technology. Number of interactions involved in software failures – empirical data. http://csrc.nist.gov/groups/SNS/acts/ftfi.html

129. Linda M. Northrop. SEI's software product line tenets. *IEEE Software*, 19(4):32–40, July/August 2002.

130. K. Nurmela (2004) "Upper bounds for covering arrays by tabu search," Discrete Applied Mathematics, 138, pp 143–152

131. K. Nurmela (webpage) http://www.tcs.hut.fi/~kjnu/covarr.html

132. Ocariza Jr, F. P. (2011). JavaScript errors in the wild: An empirical study. *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium* (pp. 100–109).

133. V. Okun, P. E. Black, "Issues in Software Testing with Model Checkers", *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2003)*, June 2003

134. V. Okun, "Specification Mutation for Test Generation and Analysis", PhD Dissertation, U of Maryland Baltimore Co., 2004

135. [OMG 2010] Object Management Group. *OMG Unified Modeling Language (OMG UML), Superstructure Version 2.3.*

136. Sebastian Oster , Florian Markert , Philipp Ritter, Automated incremental pairwise testing of software product lines, *Proceedings of the 14th international conference on Software product lines: going beyond*, September 13-17, 2010, Jeju Island, South Korea

137. T.J. Ostrand, M.J. Balcer, "The Category Partition Method for Specifying and Generating Functional Tests", *Communications of the ACM*, vol. 31, no. 6 (June, 1988), pp. 676-686.

138. M. Pezze, M. Young. *Software Testing and Analysis—Process, Principles and Techniques.* John Wiley & Sons, 2008.

139. M.S. Phadke. Quality Engineering Using Robust Design. Prentice-Hall Inc., New Jersey, 1989.

140. Alexander Pretschner, Tejeddine Mouelhi, Yves Le Traon. Model Based Tests for Access Control Policies, *2008 International Conference on Software Testing, Verification, and Validation* pp. 338-347

141. Damaraju Raghavarao (1971) Constructions and Combinatorial Problems in Design of Experiments, Dover: New York

142. C. R. Rao (1947) "Factorial experiments derivable from combinatorial arrangements of arrays," Journal of Royal Statistical Society (Supplement), 9, pp 128-139

143. Alfred Renyi (1971) Foundations of Probability, New York: Wiley

144. G. Roux (1987) "*k*–propriétés dans les tableaux de *n* colonnes: cas particulier de la *k*–surjectivité et de la k–permutivité," Unpublished PhD dissertation, University of Paris

145. X. Qu, M. B. Cohen, and K. M. Woolf. Combinatorial interaction regression testing: A study of test case generation and prioritization. In Intl. Conference on Software Maintenance, pages 255–264, Oct. 2007.

146. Rudolf Ramler, Theodorich Kopetzky and Wolfgang Platz, Combinatorial Test Design in the TOSCA Testsuite: Lessons Learned and Practical Implications, *Workshop on Combinatorial Testing (CT) International Conference on Software Testing* (ICST 2012, April 17-21)IEEE Computer Society (2012)

147. E. Reisner, C. Song, K.K. Ma, J.S. Foster, A. Porter, Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems, Proceeding ICSE '10 Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 Pages 445-454.

148. Andreas Reuys, Sacha Reis, Erik Kamsties, and Klaus Pohl. Derivation of domain test scenarios from activity diagrams. In *Proceedings of the International Workshop on Product Line Engineering The Early Steps: Planning, Modeling, and Managing (PLEES'03)*, 2003.

149. D. Richardson, L. Clarke, "A Partition Analysis Method to Increase Program Reliability", *Proc. 5th Intl. Conf. Software Eng.*, IEEE, Mar. 9-12, 1981, pp. 244-253.

150. D.S. Rosenblum. A Practical Approach to Programming with Assertions, *IEEE Trans. on Software Eng*., vol. 21, no. 11, pp. 777-793, Jan. 1995.

151. G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. IEEE Trans. on Software Engineering, (October 2011), 27(10):929–948.

152. Sampath, S. S. (2007). Applying concept analysis to user-session-based testing of web applications. *Transactions on Software Engineering. 33(10)*, pp. 643-658.

153. Sreedevi Sampath, Renée Bryce, Gokulanand Viswanath, Vani Kandimalla, A. Günes Koru, "Prioritizing User-Session-Based Test Cases for Web Application Testing", International Conference on Software Testing, Verification, and Validation (ICST) (April 2008), pp..141-150.

154. S. Sampath, R. Bryce, S. Jain, S. Manchester. A Tool for Combinatorial-based Prioritization and Reduction of User-Session-Based Test Suites, Proceedings of the International Conference on Software Maintenance (ICSM) - Tool Demonstration Track, Williamsburg, VA (September 2011), pp. 574-577.

155. S. Sampath, R. Bryce. Improving the effectiveness of test suite reduction for user-session-based testing of web applications, Information and Software Technology Journal (IST, Elsevier), (July 2012), 54(7): 724-738.

156. Shinobu Sato and H. Shimokawa (1984) “Methods for setting software test parameters using the design of experiments method (in Japanese),” Proceedings of the 4-th Symposium on Quality Control in Software, Japanese Union of Scientists and Engineers (JUSE), pp 1-8

157. Patrick J. Schroeder, Pankaj Bolaki, and Vijayram Gopu. Comparing the fault detection effectiveness of n-way and random test suites. In *Proceedings of the IEEE International Symposium on Empirical Software Engineering*, pages 49–59, 2004.

158. I. Segall, R. Tzoref-Brill, A. Zlotnick, Simplified Modeling of Combinatorial Test Spaces, *Workshop on Combinatorial Testing (CT) International Conference on Software Testing* (ICST 2012, April 17-21)IEEE Computer Society (2012):

159. G. Sherwood. Efficient testing of factor combinations. Proc. Third International Conf.Software Testing, Analysis, and Review (Jacksonville FL), 1994.

160. [Sherwood 2011] George B. Sherwood. *Getting the Most from Pairwise Testing: A Guide for Practicing Software Engineers.* CreateSpace, 2011.

161. N. Sloane. Covering arrays and intersecting codes. Journal of Combinatorial Designs, 1(1):51–63, 1993.

162. Hiroki Shimokawa (1985) “Method of generating software test cases using the experimental design (in Japanese),” Report on Software Engineering SIG, Information Processing Society of Japan (IPSJ) No.1984-SE-040

163. Neil J. A. Sloan (webpage) http://www2.research.att.com/~njas/oadir/

164. George W. Snedecor, and W. G. Cochran (1967) Statistical Methods, Iowa State University Press

165. Software Engineering Institute. Catalog of Software Product Lines. http://www.sei.cmu.edu/productlines/casestudies/catalog/?location=tertiary-nav&source=10755

166. Software Engineering Institute. Software Product Lines, http://www.sei.cmu.edu/productlines/. 19 July 2012.

167. Charles Song, Adam Porter, Jeffrey S. Foster: iTree: Efficiently discovering high-coverage configurations using interaction trees. 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland. IEEE 2012, ISBN 978-1-4673-1067-3 pp. 903-913

168. M. Sutton, A. Greene, P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*, Addison-Wesley, 2007

169. Genichi Taguchi (1986) Introduction to Quality Engineering, White Plains New York: UNIPUB, Kraus International

170. Genichi Taguchi (1987) System of Experimental Design, Vol. 1 and Vol. 2, White Plains New York: UNIPUB, Kraus International (English translations of the 3-rd edition of Jikken Keikakuho (Japanese) published in 1977 and 1978 by Maruzen)

171. Genichi Taguchi (1993) Taguchi on Robust Technology Development, New York: ASME Press

172. K. C. Tai and Yu Lei (2002) "A test generation strategy for pairwise testing," IEEE Transactions on Software Engineering, 28, pp 109-111

173. Keizo Tatsumi (1987) "Test-case design support system" Proceedings of the International Conference on Quality Control (ICQC 87), Tokyo, 20-23 October 1987, pp 615-620

174. Keizo Tatsumi, S. Watanabe, Y. Takeuchi, and H. Shimokawa (1987) "Conceptual support for test case design," Proceedings of 11-th IEEE Computer Software and Applications Conference (COMPSAC 87), Tokyo, 7-9 October 1987, pp 285-290

175. [Tatsumi 1987] K. Tatsumi. Test case design support system. *Proceedings of the International Conference on Quality Control (ICQC), Tokyo, 1987*, pages 615–620, 1987.

176. [Testcover.com 2012] www.testcover.com.

177. Jose Torres-Jimenez and E. Rodriguez-Tello (2012) "New bounds for binary covering arrays using simulated annealing," Information Sciences, 185, pp 137-152

178. Jose Torres-Jimenez (webpage) http://www.tamps.cinvestav.mx/~jtj/CA.php

179. J.G. Udell, E.J. McCluskey, "Efficient Circuit Segmentation for Pseudoexhaustive Test", Proc. Intl. Conf. on Computer-Aided Design, 1987, pp. 148-151.

180. University of Nebraska Lincoln. Software Artifact Infrastructure Repository. Siemens Traffic Collision Avoidance System code. http://sir.unl.edu/portal/bios/tcas.php

181. S. Vilkomir, O. Starov, R. Bhambroo, Evaluation of t-wise Approach for Testing Logical Expressions in Software, (submitted for publication).

182. J.M. Voas, K.W. Miller, "Putting Assertions in their Place", *Proceedings of International Symposium on Software Reliability Engineering*, IEEE, pp. 152-157, 1994.

183. J. Voas, Schatz, M., Schmid, M., "A Testability-based Assertion Placement Tool for Object-Oriented Software," National Institute for Standards and Technology NIST GCR 98-735, 1998.

184. Wenhua Wang, Yu Lei, D. Liu, D. Kung, C. Csallner, D. Zhang, R. N. Kacker and D. R. Kuhn (2011) "A combinatorial approach to detecting buffer overflow vulnerabilities," Proceedings of 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Hong Kong, 27-30 June 2011, pp 269-278

185. Wenhua Wang, S. Sampath, Yu Lei, and R. N. Kacker (2008) "An interaction-based test sequence generation approach for testing web applications," Proceedings of 11-th IEEE International Conference on High Assurance Systems Engineering, Nanjing China, 3-5 December 2008, pp 209-218

186. Wenhua Wang, Yu Lei, S. Sampath, R. N. Kacker, D. R. Kuhn, and J. Lawrence (2009) "A combinatorial approach to building navigation graphs for dynamic web applications," Proceedings of 25th IEEE International Conference on Software Maintenance, Edmonton Canada, 20-26 September 2009, pp 211-220

187. Wenhua Wang, Yu Lei, D. Liu, D. Kung, C. Csallner, D. Zhang, R. N. Kacker and D. R. Kuhn (2011) "A combinatorial approach to detecting buffer overflow vulnerabilities," Proceedings of 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Hong Kong, 27-30 June 2011, pp 269-278

188. X. Yuan, M.B. Cohen, A. Memon, "Covering Array Sampling of Input Event Sequences for Automated GUI Testing", November 2007 ASE '07: *Proceedings of the 22nd IEEE/ACM Intl. Conf. Automated Software Engineering*, pp. 405-408.

189. X. Yuan and A. M. Memon. Using GUI run-time state as feedback to generate test cases. In ICSE'07, *Proceedings of the 29th International Conference on Software Engineering*, pages 396–405, Minneapolis, MN, USA, May 23–25, 2007.

190. D.R. Wallace, D.R. Kuhn, Failure Modes in Medical Device Software: an Analysis of 15 Years of Recall Data, *International Journal of Reliability, Quality, and Safety Engineering*, Vol. 8, No. 4, 2001.

191. E. Weyuker, using Failure Cost Information for Testing and Reliability Assessment, *ACM Trans. on Software Engineering and Methodology*, v. 5, n. 2, apr. 1996, pp. 87-90.

192. E. Weyuker, Testing Component-Based Software: a Cautionary Tale, *IEEE Software*, Sept./Oct. 1998, IEEE.

193. E. Weyuker, T. Goradia, and A. Singh, *"Automatically generating test data from a Boolean specification"*, IEEE Transactions on Software Engineering, Vol. 20, No. 5, pp. 353–363, (1994).

194. L. White, E. Cohen. "A domain strategy for computer program testing." *IEEE Transactions on Software Engineering* (IEEE Transactions on Software Engineering), 1980: 247-257

195. Wikipedia. Java Modeling Language, http://en.wikipedia.org/wiki/Java_Modeling_Language.

196. A.W. Williams, R.L. Probert. A practical strategy for testing pair-wise coverage of network interfaces *The Seventh International Symposium on Software Reliability Engineering (ISSRE '96)* p. 246

197. World Wide Web Consoritum, DOM Level 3 Events Specification, 8 Sept 2009. http://www.w3.org/TR/DOM-Level-3-Events/

198. World Wide Web Consoritum, Document Object Model Conformance Test Suites. http://www.w3.org/DOM/Test/

199. Zhang, Z., X. Liu, and J. Zhang, Combinatorial Testing on ID3v2 Tags of MP3 Files, Workshop on Combinatorial Testing, at Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference, 17-21 April 2012, pp. 587-590.