NIST Special Publication 800-142

National Institute of
Standards and Technology
Technology Administration
U.S. Department of Commerce

INFORMATION SECURITY

PRACTICAL COMBINATORIAL TESTING

D. Richard Kuhn, Raghu N. Kacker, Yu Lei

October, 2010

U.S. Department of Commerce
Gary Locke, Secretary

National Institute of Standards and Technology
Patrick Gallagher, Director

Reports on Computer Systems Technology

The Information Technology Laboratory (ITL) at the National Institute of Standards and Technology (NIST) promotes the U.S. economy and public welfare by providing technical leadership for the Nation's measurement and standards infrastructure. ITL develops tests, test methods, reference data, proof of concept implementations, and technical analyses to advance the development and productive use of information technology. ITL's responsibilities include the development of technical, physical, administrative, and management standards and guidelines for the cost-effective security and privacy of sensitive unclassified information in Federal computer systems. This Special Publication 800-series reports on ITL's research, guidance, and outreach efforts in computer security, and its collaborative activities with industry, government, and academic organizations.

U.S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 2010

For sale by the Superintendent of Documents, U.S. Government Printing Office
Internet: bookstore.gpo.gov — Phone: (202) 512-1800 — Fax: (202) 512-2250
Mail: Stop SSOP, Washington, DC 20402-0001Note to Readers

Note to Readers

This document is a publication of the National Institute of Standards and Technology (NIST) and is not subject to U.S. copyright. Certain commercial entities, equipment, or materials may be identified in this document in order to describe an experimental procedure or concept adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the entities, materials, or equipment are necessarily the best available for the purpose.

For questions or comments on this document, contact Rick Kuhn, kuhn@nist.gov or 301-975-3337.

Acknowledgements

Special thanks are due to Tim Grance, Jim Higdon, Eduardo Miranda, and Tom Wissink for early support and evangelism of this work, and especially Jim Lawrence who has been an integral part of the team since the beginning. We have benefitted tremendously from interactions with researchers and practitioners including Renee Bryce, Myra Cohen, Charles Colbourn, Mike Ellims, Vincent Hu, Justin Hunter, Aditya Mathur, Josh Maximoff, Carmelo Montanez-Rivera, Jenise Reyes Rodriguez, Rick Rivello, Sreedevi Sampath, Mike Trela, and Tao Xie. We also gratefully acknowledge NIST SURF students Michael Forbes, William Goh, Evan Hartig, Menal Modha, Kimberley O'Brien-Applegate, Michael Reilly, Malcolm Taylor and Bryan Wilkinson who contributed to the software and methods described in this document.

Table of Contents

Executive Summary

Software implementation errors are one of the most significant contributors to information system security vulnerabilities, making software testing an essential part of system assurance. In 2003 NIST published a widely cited report which estimated that inadequate software testing costs the US economy $59.5 billion per year, even though 50% to 80% of development budgets go toward testing. Exhaustive testing – testing all possible combinations of inputs and execution paths – is impossible for real-world software, so high assurance software is tested using methods that require extensive staff time and thus have enormous cost. For less critical software, budget constraints often limit the amount of testing that can be accomplished, increasing the risk of residual errors that lead to system failures and security weaknesses.

Combinatorial testing is a method that can reduce cost and increase the effectiveness of software testing for many applications. The key insight underlying this form of testing is that not every parameter contributes to every failure and most failures are caused by interactions between relatively few parameters. Empirical data gathered by NIST and others suggest that software failures are triggered by only a few variables interacting (6 or fewer). This finding has important implications for testing because it suggests that testing combinations of parameters can provide highly effective fault detection. Pairwise (2-way combinations) testing is sometimes used to obtain reasonably good results at low cost, but pairwise testing may miss 10% to 40% or more of system bugs, and is thus not sufficient for mission-critical software. Combinatorial testing beyond 2-way has been limited, primarily due to a lack of good algorithms for higher interaction levels such as 4-way to 6-way testing. New algorithms, however, have made combinatorial testing beyond pairwise practical for industrial use.

This publication provides a self-contained tutorial on using combinatorial testing for real-world software. It introduces the key concepts and methods, explains use of software tools for generating combinatorial tests (freely available on the NIST web site csrc.nist.gov/acts), and discusses advanced topics such as the use of formal models of software to determine the expected results for each set of test inputs. With each topic, a section on costs and practical considerations explains tradeoffs and limitations that may impact resources or funding. The material is accessible to an undergraduate student of computer science or engineering, and includes an extensive set of references to papers that provide more depth on each topic.

1 INTRODUCTION

Software implementation errors are one of the most significant contributors to information system security vulnerabilities, making software testing an essential part of system assurance. Combinatorial methods can help reduce the cost and increase the effectiveness of software testing for many applications. This publication provides a self-contained tutorial on using combinatorial testing for real-world software. It introduces the key concepts and methods, explains use of software tools for generating combinatorial tests (freely available on the NIST web site csrc.nist.gov/acts), and discusses advanced topics such as the use of formal models of software to determine the expected results for each possible set of test inputs. The material is accessible to an undergraduate student of computer science or engineering, and includes an extensive set of references to papers that provide more depth on each topic.

1.1 Authority

The National Institute of Standards and Technology (NIST) developed this document in furtherance of its statutory responsibilities under the Federal Information Security Management Act (FISMA) of 2002, Public Law 107-347.

NIST is responsible for developing standards and guidelines, including minimum requirements, for providing adequate information security for all agency operations and assets, but such standards and guidelines shall not apply to national security systems. This guideline is consistent with the requirements of the Office of Management and Budget (OMB) Circular A-130, Section 8b(3), "Securing Agency Information Systems," as analyzed in A-130, Appendix IV: Analysis of Key Sections. Supplemental information is provided in A-130, Appendix III.

This guideline has been prepared for use by Federal agencies. It may be used by nongovernmental organizations on a voluntary basis and is not subject to copyright, though attribution is desired.

Nothing in this document should be taken to contradict standards and guidelines made mandatory and binding on Federal agencies by the Secretary of Commerce under statutory authority, nor should these guidelines be interpreted as altering or superseding the existing authorities of the Secretary of Commerce, Director of the OMB, or any other Federal official.

1.2 Document Scope and Purpose

This publication introduces combinatorial testing and explains how to use it effectively for system and software assurance.

1.3 Audience and Assumptions

This document assumes that the readers have experience with software development and testing, some familiarity with scripting languages, and basic knowledge of programming, logic, and discrete mathematics equivalent to what would be acquired in an undergraduate computer science or engineering program. Most of the material should be readily understood by an undergraduate student with some programming experience. Because of the constantly changing nature of the information technology industry, readers are strongly encouraged to take advantage of other resources (including those listed in this document) for more current and detailed information.

1.4 Organization: How to use this Document

The document is divided into chapters, with background material covered in appendices. Because it is intended to be self-contained, each chapter provides material that will be used in later topics. Chapters 2, 3, and 4 will be needed by most testers, while the material in later chapters is specialized for various topics. Appendices include a review of basic combinatorics and a discussion of empirical data on software failures.

Readers new to combinatorial testing may want to review the basics of combinatorics in Appendix A and read chapters 2, 3, and 4. Other sections of the publication can be reserved for later use as needed.

2 COMBINATORIAL METHODS IN TESTING

Developers of large data-intensive software often notice an interesting—though not surprising—phenomenon: When usage of an application jumps dramatically, components that have operated for months without trouble suddenly develop previously undetected errors. For example, the application may have been installed on a different OS-hardware-DBMS-networking platform, or newly added customers may have account records with an oddball combination 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.

Combinatorial testing can help detect problems like this 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). To detect 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 software failures often involve interactions between parameters. For example, a router may be observed to fail only for a particular protocol when packet volume exceeds a certain rate, a 2-way interaction between protocol type and packet rate. Figure 1 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.

```
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 failure triggered only 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, has come to be

accepted as the common approach to combinatorial testing because it is computationally tractable and reasonably effective.

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.

But what if some failure is triggered only by a very unusual combination of 3, 4, or more sensor 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? 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. Results showed that across a variety of domains, all failures could be triggered by a maximum of 4-way to 6-way interactions [34, 35, 36, 65]. 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 [6, 7, 26] have been consistent with these results.

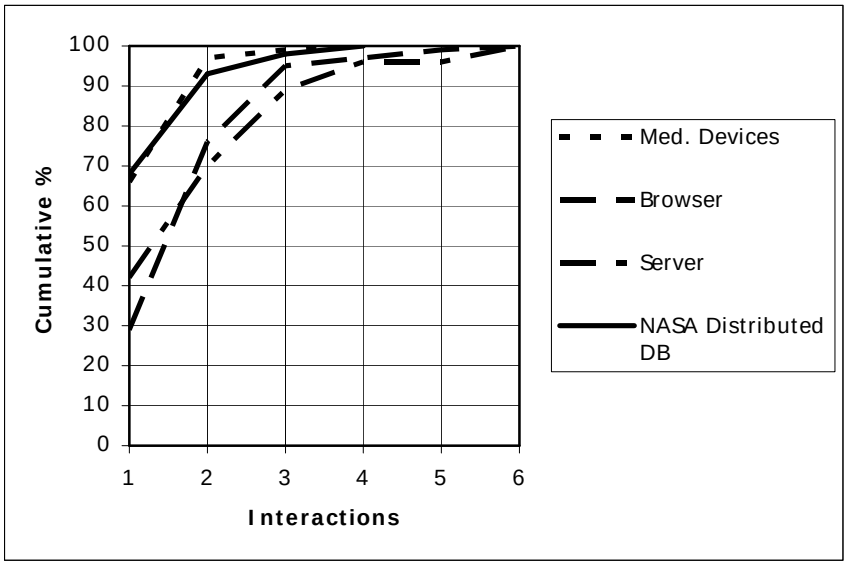


Figure 2. Error detection rates for interaction strengths 1 to 6

While not conclusive, 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 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.***

Testing all 4-way to 6-way combinations may therefore provide reasonably 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. In addition, most parameters are continuous variables which have possible values in a very large range (+/-2^{32} 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 combination covering test generation, 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 publication 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.

A note on terminology: we use the definitions below, following the Institute of Electrical and Electronics Engineers [30]. 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.

2.1 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, etc., 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 configuration parameter values. (Note, the term *variable* is often used interchangeably with *parameter* to refer to inputs to a function.)

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

In the second approach, we select combinations of *input data* values, which then become part of complete test cases, creating a test suite for the application. In this case combinatorial coverage of input data values is required for tests constructed. A typical ad hoc approach to testing involves subject matter experts setting up use 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 Appendix C), which is available freely from NIST.

2.1.1 Configuration Testing

Many, if not most, software systems have a large number of configuration parameters. Many of the earliest applications of combinatorial testing were in testing all pairs of system configurations. For example, telecommunications software may be configured to work with different types of call (local, long distance, international), billing (caller, phone card, 800), access (ISDN, VOIP, PBX), and server for billing (Windows Server, Linux/MySQL, Oracle). The software must work correctly with all combinations of these, so a single test suite could be applied to all pairwise combinations of these four major configuration items. Any system with a variety of configuration options is a suitable candidate for this type of testing.

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 $3 \cdot 2 \cdot 2 \cdot 2 \cdot 3 = 72$ 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.

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

2.1.2 Input Parameter 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 may 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, we create a test suite to cover all 3-way combinations (known as a *covering array*) [12, 14, 23, 26, 30, 43, 63].

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 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.

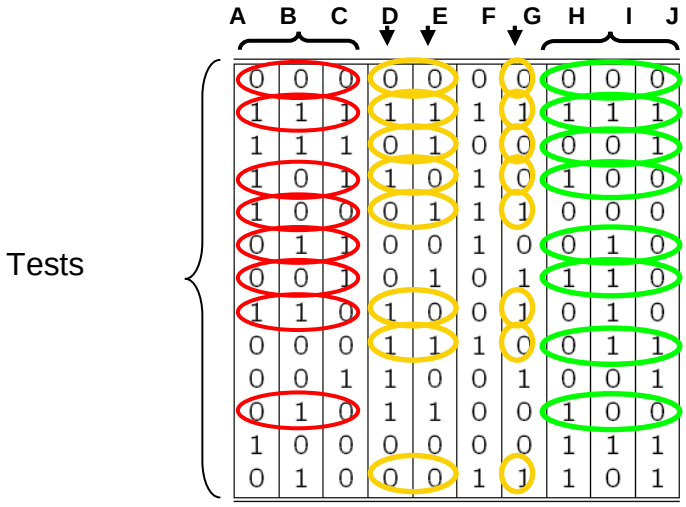


Figure 3. 3-way covering array

Similar arrays can be generated to cover up to all 6-way combinations. In general, the number of *t*-way combinatorial tests that will be required is proportional to $v^t \log n$, for n parameters with v possible values each.

Figure 4 contrasts these two approaches. 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 (approach 2) run against all the configuration combinations (approach 1).

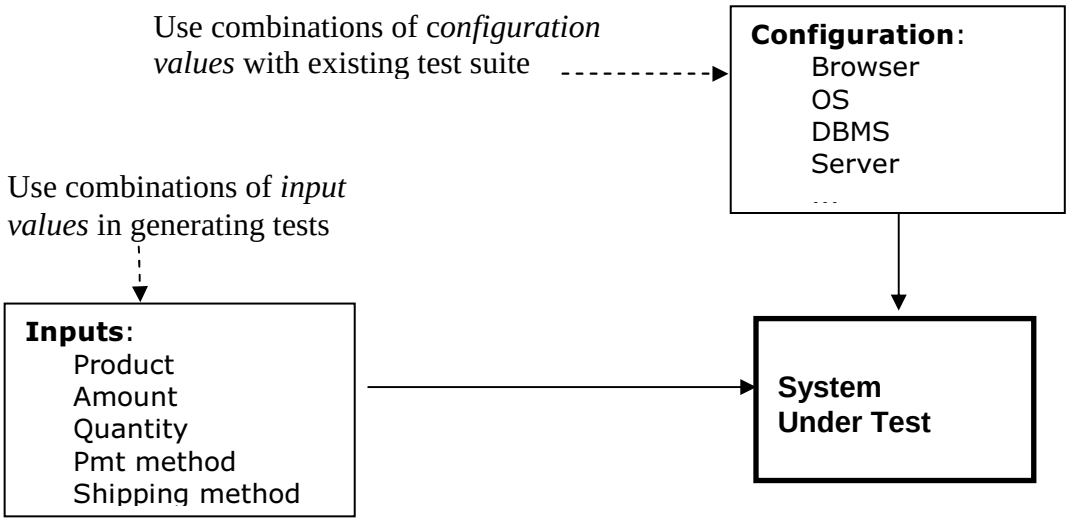


Figure 4. Two ways of using combinatorial testing

2.2 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. Approaches to solving the oracle problem for combinatorial testing include:

Crash testing: the easiest and least expensive approach is to simply run tests against the system under test (SUT) to check whether any unusual combination of input values causes a crash or other easily detectable failure. This is essentially the same procedure used in "fuzz testing", which sends random values against the SUT. This form of combinatorial testing could be regarded as a disciplined form of fuzz testing [59]. 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. Purely random generation requires over 900 tests to provide full 3-way coverage.

Embedded assertions: An increasingly popular "light-weight formal methods" technique is to embed assertions within code to ensure proper relationships between data, for example as preconditions, postconditions, or input value 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 [25]. Results showed that 80% - 90% of errors could be found in this way.

Model based test generation uses a mathematical model of the SUT and a simulator or model checker to generate expected results for each input [1,8,9,52,55]. 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. Later chapters develop detailed procedures for applying each of these testing approaches.

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

2.3 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.
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. Combinatorial methods can be applied to configurations or to input parameters, or in some cases both.
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 are available to solve the oracle problem.

3 CONFIGURATION TESTING

This chapter presents worked examples illustrating development of test configurations. As will be seen, the advantages of combinatorial testing increase with the size of the problem.

3.1 Simple Application Platform Example

Returning to the simple example introduced in Chapter 2, 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 $3 \cdot 2 \cdot 2 \cdot 2 \cdot 3 = 72$ configurations. The convention for describing the variables and values in combinatorial testing is $v_1^{n1} v_2^{n2}$ ... where the v_i are number of variable values and n_i are number of occurrences. Thus this configuration is designated $2^3 3^2$. 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.

We can now generate test configurations using the ACTS tool. For simplicity of presentation we illustrate usage of the command line version of ACTS, but an intuitive GUI version is available that may be more convenient. This tool is summarized in Appendix C and 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 in a file for input to ACTS, as shown in Figure 5:

```
[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 5. Simple example input file for ACTS.

Note that most of the bracketed tags in the input file are optional, and not filled in for this example. The essential part of the file is the [Parameter] specification, in the format <parameter name> (<type>): <values>, where one or more values are listed separated by commas. The tool can then be run at the command line:

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

A variety of options can be specified, but for this example we only use the "degree of interaction" option to specify 2-way, 3-way, etc. coverage. Output can be created in a convenient form shown below, or as a matrix of numbers, comma separated value, or Excel spreadsheet form. If the output will be used by human testers rather than as input for further machine processing, the format in Figure 6 is useful:

```
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
. . .
```

Figure 6. Excerpt of test configuration output covering all 2-way combinations.

The complete test set for 2-way combinations is shown in Table 1 in Section 2.1.1. 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 Smart Phone Application Example

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.

Figure 7 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 7. 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. Android configuration options.

Using Table 4, we can now calculate the total number of configurations: $3 \cdot 3 \cdot 4 \cdot 3 \cdot 5 \cdot 4 \cdot 4 \cdot 5 \cdot 4$ = 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. How can we 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.

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. Number of combinatorial tests for Android example.

3.3 Cost and Practical Considerations

3.3.1 Invalid Combinations and Constraints

The system described in Section 3.1 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 2.1.1 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:

```
(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 8 below. 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 help 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.

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 8. Test configurations for simple example with constraint.

3.3.2 Cost Factors

Using combinatorial methods to design test configurations is probably the most widely used combinatorial approach because it is quick and easy to do and typically delivers significant improvements to testing. Combinatorial testing for input parameters can provide better test coverage at lower cost than conventional tests, and can be extended to high strength coverage to provide much better assurance.

3.4 Chapter Summary

1. 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, unless configurable attributes have more than 8 or 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.

2. Because many systems have certain configurations that may not be of interest (such as 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.

4 INPUT PARAMETER TESTING

As noted in the introduction, 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. 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. In this chapter we work through a small example to illustrate the use of these methods.

4.1 Example Access Control Module

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 9:

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

Figure 9. 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 3.1 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 10. Several

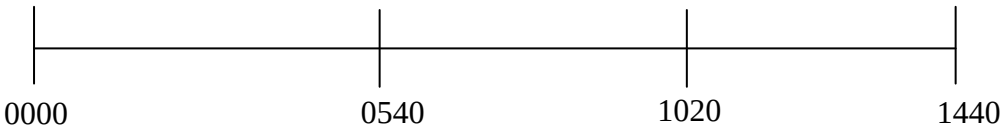
parameters are boolean, and we will use 0 and 1 for false and true values respectively. For day of the week, there are only seven values, so these can all be used. However, hour of the day presents a problem. Recall that the number of tests generated for *n* parameters is proportional to v^t, where *v* is the number of values and *t* is the interaction level (2-way to 6-way). For all boolean values and 4-way testing, therefore, the number of tests will be some multiple of 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 $24^4 = 331{,}736$. For 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 two of these: *equivalence partitioning* and *boundary value analysis*. Additional background on these methods can be found in software testing texts such as Ammann and Offutt [2], Beizer [4], Copeland [21], Mathur [45], and Myers [52].

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

Figure 10. Parameters and values for access control example.

Both of these intuitively obvious 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. Some 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 chose; 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 empirical 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 $2 \cdot 8 \cdot 7 \cdot 2 \cdot 2 = 448$.

Generating covering arrays for $t = 2$ through 6, as detailed in Section 3.1 results in the following number of tests:

t	# Tests
2	56
3	112
4	224

Figure 11. Number of tests for access control example.

4.2 Real-world Systems

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

As with the previous example, the advantage over exhaustive testing is not large, because of the small number of parameters. With 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 shown in Figure 12. There are 34 switches, which can each be either on or off, for a total of 2^{34}

= 1.7 x 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 12. Panel with 34 switches.

4.3 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 [30]. 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 [10, 45]. 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 [34] shown in Figure 13. While the number of tests will be different (probably much smaller than in Figure 13) depending on the system under test, the magnitude of difference between levels of *t* will be similar to Figure 13, 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.

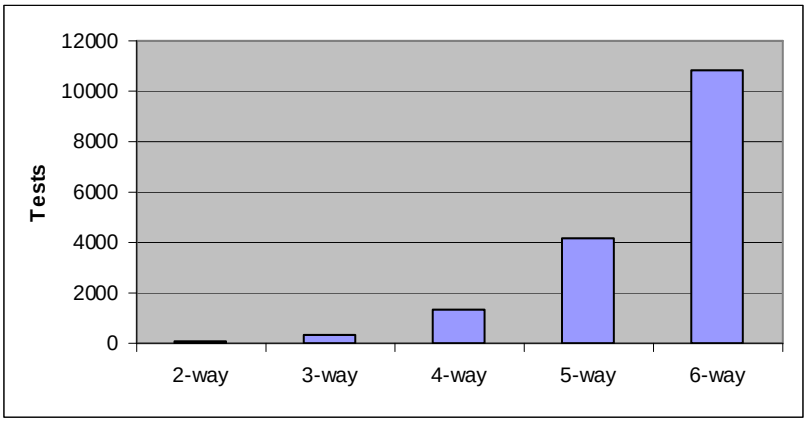


Figure 13. Number of tests for avionics example.

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 (section 4.1), it is best to keep the number of values per parameter below 8 or 10 if possible.

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.

4.4 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 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, not necessarily if they are back to back. This situation reflects the fact that in many cases, a particular state must be reached before a particular failure can be triggered. For example, a failure might occur when connecting device A only if device B is already connected. The methods described in this chapter were developed to solve a real problem in interoperability test and evaluation, using combinatorial methods to provide efficient testing. Sequence covering arrays, as defined here, ensure that any *t* events will be tested in every possible *t*-way order.

In many systems, the order of inputs is important.

For this problem we can define a sequence-covering array [39, 40], 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 6.

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 Table 8).

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 6. System events

Definition. We define a sequence covering array, SCA(*N, S, t*) as 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, 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 Table 7).

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 7. Tests for four events.

5.1 Constructing Sequence Covering Arrays

A 2-way sequence covering array can be constructed 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 the procedure in Example 2 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.

For *t-way sequence* test generation, where t > 2, we 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,

5.2 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 (Figure 14) 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 9.

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 14. Seven-event test from pre-computed test library.

5.3 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’. The algorithm we have developed 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.

5.4 Chapter Summary

1. Sequence covering arrays are a new application of combinatorial methods, developed by NIST to solve problems with interoperability testing. A sequence-covering array 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.

2. 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.

3. 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 NIST has 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.

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
90	44	
100	44	

Table 8. Number of tests for combinatorial 3-way and 4-way sequences.

Table 9. 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

6 MEASURING COMBINATORIAL COVERAGE

Since it is nearly always impossible to test all possible combinations, combinatorial testing is a reasonable alternative. For some value of *t*, testing all *t*-way interactions among *n* parameters will detect nearly all errors. It is possible that $t = n$, but recalling the empirical data on failures, we would expect *t* to be relatively small. Determining the level of input or configuration state space coverage can help in understanding the degree of risk that remains after testing. If 90% - 100% of the state space has been covered, then presumably the risk is small, but if coverage is much smaller, then the risk may be substantial. This chapter describes some measures of combinatorial coverage that can be helpful in estimating this risk that we have applied to tests for spacecraft software [50] but have general application to any combinatorial coverage problem.

6.1 Software Test Coverage

Test coverage is one of the most important topics in software assurance. Users would like some quantitative measure to judge the risk in using a product. For a given test set, what can we say about the combinatorial coverage it provides? With physical products, such as light bulbs or motors, reliability engineers can provide a probability of failure within a particular time frame. This is possible because the failures in physical products are typically the result of natural processes, such as metal fatigue.

Commonly used coverage measures do not apply well to combinatorial testing.

With software the situation is more complex, and many different approaches have been devised for determining software test coverage. With millions of lines of code, or only with a few thousand, the number of paths through a program is so large that it is impossible to test all paths. For each *if* statement, there are two possible branches, so a sequence of *n if* statements will result in 2^n possible paths. Thus even a small program with only 270 *if* statements in an execution trace may have more possible paths than there are atoms in the universe, which is on the order of 10^{80}. With loops (*while* statements) the number of possible paths is literally infinite. Thus a variety of measures have been developed to gauge the degree of test coverage. The following are some of the better-known coverage metrics:

- **Statement coverage:** This is the simplest of coverage criteria – the percentage of statements exercised by the test set. While it may seem at first that 100% statement coverage should provide good confidence in the program, in practice, statement coverage is a relatively weak criterion. At best, statement coverage represents a sanity check: unless statement coverage is close to 100%, the test set is probably inadequate.

- **Decision or branch coverage:** The percentage of branches that have been evaluated to both *true* and *false* by the test set.

- **Condition coverage:** The percentage of conditions within decision expressions that have been evaluated to both true and false. Note that 100% condition coverage does not guarantee 100% decision coverage. For example, "`if (A || B) {do something} else {do something else}`" is tested with [0 1], [1 0], then A and B will both have been evaluated to 0 and 1, but the *else* branch will not be taken because neither test leaves both A and B false.

- **Modified condition decision coverage (MCDC):** This is a strong coverage criterion that is required by the US Federal Aviation Administration for Level A (catastrophic failure consequence) software; i.e., software whose failure could lead to complete loss of life. It requires that every condition in a decision in the program has taken on all possible outcomes at least once, and each condition has been shown to independently affect the decision outcome, and that each entry and exit point have been invoked at least once.

6.2 Combinatorial Coverage

Note that the coverage measures above depend on access to program source code. Combinatorial testing, in contrast, is a black box technique. Inputs are specified and expected results determined from some form of specification. The program is then treated as simply a processor that accepts inputs and produces outputs, with no knowledge expected of its inner workings.

Even in the absence of knowledge about a program's inner structure, we can apply combinatorial methods to produce precise and useful measures. In this case, we measure the state space of inputs. Suppose we have a program that accepts two inputs, *x* and *y*, with 10 values each. Then the input state space consists of the $10^2 = 100$ pairs of *x* and *y* values, which can be pictured as a checkerboard square of 10 rows by 10 columns. With three inputs, *x*, *y*, and *z*, we would have a cube with $10^3 = 1{,}000$ points in its input state space. Extending the example to n inputs we would have a (hard to visualize) hypercube of *n* dimensions with 10^n points. Exhaustive testing would require inputs of all 10^n combinations, but combinatorial testing could be used to reduce the size of the test set.

How should state space coverage be measured? Looking closely at the nature of combinatorial testing leads to several measures that are useful. We begin by introducing what will be called a *variable-value configuration*.

Definition. For a set of *t* variables, a variable-value configuration is a set of *t* valid values, one for each of the variables.

Example. Given four binary variables, *a*, *b*, *c*, and *d*, *a*=0, *c*=1, *d*=0 is a variable-value configuration, and *a*=1, *c*=1, *d*=0 is a different variable-value configuration for the same three variables *a*, *c*, and *d*.

6.2.1 Simple t -way combination coverage

Of the total number of *t*-way combinations for a given collection of variables, what percentage will be covered by the test set? If the test set is a covering array, then coverage

is 100%, by definition, but many test sets not based on covering arrays may still provide significant *t*-way coverage. If the test set is large, but not designed as a covering array, it is very possible that it provides 2-way coverage or better. For example, random input generation may have been used to produce the tests, and good branch or condition coverage may have been achieved. In addition to the structural coverage figure, for software assurance it would be helpful to know what percentage of 2-way, 3-way, etc. coverage has been obtained.

Definition: For a given test set for *n* variables, simple *t*-way combination coverage is the proportion of *t*-way combinations of *n* variables for which all variable-values configurations are fully covered.

Example. Figure 15 shows an example with four binary variables, *a*, *b*, *c*, and *d*, where each row represents a test. Of the six 2-way combinations, *ab, ac, ad, bc, bd, cd*, only *bd* and *cd* have all four binary values covered, so simple 2-way coverage for the four tests in Figure 15 is 1/3 = 33.3%. There are four 3-way combinations, *abc, abd, acd, bcd*, each with eight possible configurations: 000, 001, 010, 011, 100, 101, 110, 111. Of the four combinations, none has all eight configurations covered, so simple 3-way coverage for this test set is 0%.

a	b	c	d
0	0	0	0
0	1	1	0
1	0	0	1
0	1	1	1

Figure 15. An example test array for a system with four binary components

6.2.2 (t + k)-way combination coverage

A test set that provides full combinatorial coverage for *t*-way combinations will also provide some degree of coverage for (*t*+1)-way combinations, (*t*+2)-way combinations, etc. This statistic may be useful for comparing two combinatorial test sets. For example, different algorithms may be used to generate 3-way covering arrays. They both achieve 100% 3-way coverage, but if one provides better 4-way and 5-way coverage, then it can be considered to provide more software testing assurance.

A test set for t-way interactions will also cover some higher strength interactions at t+1, t+2, etc.

Definition. For a given test set for *n* variables, (*t*+*k*)-way combination coverage is the proportion of (*t*+*k*)-way combinations of *n* variables for which all variable-values configurations are fully covered. (Note that this measure would normally be applied only to a *t*-way covering array, as a measure of coverage beyond *t*).

Example. If the test set in Figure 15 is extended as shown in Figure 16, we can extend 3-way coverage. For this test set, *bcd* is covered, out of the four 3-way combinations, so 2-way coverage is 100%, and (2+1)-way = 3-way coverage is 25%.

a	b	c	d
0	0	0	0
0	1	1	0
1	0	0	1
0	1	1	1
0	1	0	1
1	0	1	1
1	0	1	0
0	1	0	0

Figure 16. Eight tests for four binary variables.

6.2.3 Variable-Value Configuration coverage

So far we have only considered measures of the proportion of combinations for which all configurations of *t* variables are fully covered. But when *t* variables with *v* values each are considered, each *t*-tuple has v^t configurations. For example, in pairwise (2-way) coverage of binary variables, every 2-way combination has four configurations: 00, 01, 10, 11. We can define two measures with respect to configurations:

Definition. For a given combination of *t* variables, variable-value configuration coverage is the proportion of variable-value configurations that are covered.

Definition. For a given set of *n* variables, (*p*, *t*)-completeness is the proportion of the C(*n*, *t*) combinations that have configuration coverage of at least *p* [50].

Example. For Figure 16 above, there are C(4, 2) = 6 possible variable combinations and $C(4,2)2^2$ = 24 possible variable-value configurations. Of these, 19 variable-value configurations are covered and the only ones missing are *ab*=11, *ac*=11, *ad*=10, *bc*=01, *bc*=10. But only two, *bd* and *cd*, are covered with all 4 value pairs. So for the basic definition of simple *t*-way coverage, we have only 33% (2/6) coverage, but 79% (19/24) for the configuration coverage metric. For a better understanding of this test set, we can compute the configuration coverage for each of the six variable combinations, as shown in Figure 17. So for this test set, one of the combinations (*bc*) is covered at the 50% level, three (*ab*, *ac*, *ad*) are covered at the 75% level, and two (*bd*, *cd*) are covered at the 100% level. And, as noted above, for the whole set of tests, 79% of variable-value configurations are covered. All 2-way combinations have at least 50% configuration coverage, so (.50, 2)-completeness for this set of tests is 100%.

Although the example in Figure 17 uses variables with the same number of values, this is not essential for the measurement. Coverage measurement tools that we have developed compute coverage for test sets in which parameters have differing numbers of values, as shown in Figure 18 and Figure 19.

Vars	Configurations covered	Config coverage
a b	00, 01, 10	.75
a c	00, 01, 10	.75
a d	00, 01, 11	.75
b c	00, 11	.50
b d	00, 01, 10, 11	1.0
c d	00, 01, 10, 11	1.0

- *total 2-way cove*rage = 19/24 = .79167
- *(.50, 2)-completeness* = 6/6 = 1.0
- *(.75, 2)-completeness* = 5/6 = 0.83333
- *(1.0, 2)-completeness* = 2/6 = 0.33333

Figure 17. The test array covers all possible 2-way combinations of a, b, c, and d to different levels.

Figure 18 is an example of coverage for a $2^{87}3^{2}4^{5}$ set (87 binary, two 3-value, and five 4-value) of input variables (blue=2-way, pink=3-way, yellow=4-way). This particular test set was not a covering array, but pairwise coverage is still quite good, with about 95% of the variables having all possible 2-way configurations covered. Even for 4-way combinations we see that all variables have at least 28% of their configurations covered, and about 25% of them have about 98% or more of 4-way configurations covered. Figure 19 shows a similar plot for a $2^{79}3^{1}4^{1}6^{1}9^{1}$ configuration.

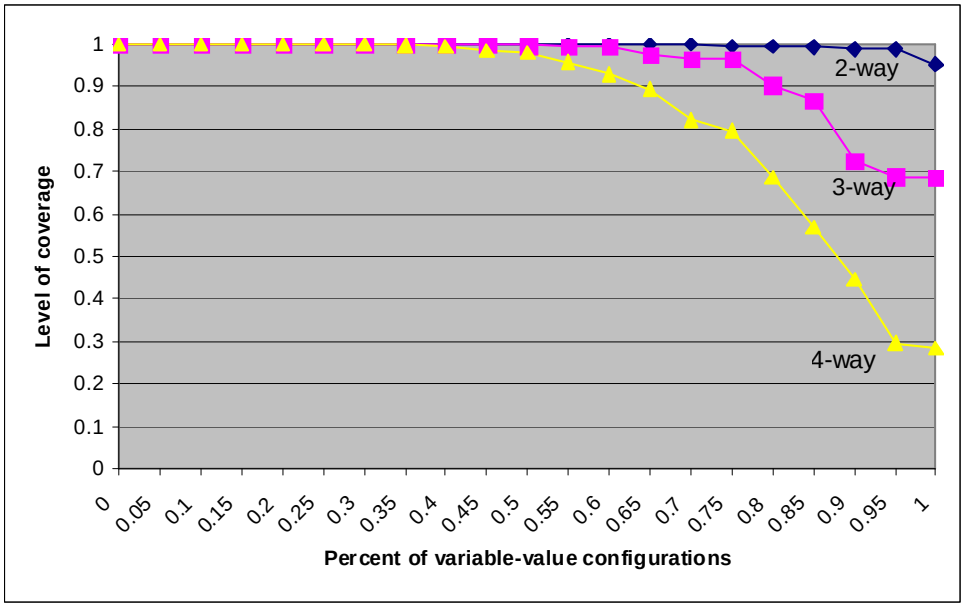


Figure 18. Configuration coverage for $2^{87}3^{2}4^{5}$ inputs.

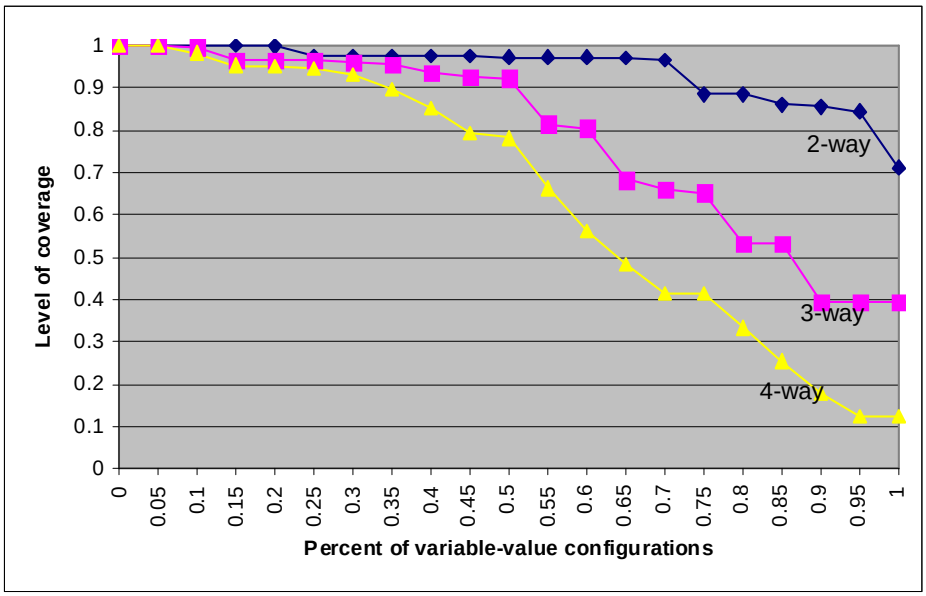


Figure 19. Configuration coverage for $2^{79}3^{1}4^{1}6^{1}9^{1}$ inputs.

6.3 Cost and Practical Considerations

An important cost advantage introduced by coverage measurement is the ability to use existing test sets, identify particular combinations that may be missing, and supplement existing tests. In some cases, as in the example of Figure 18, it may be discovered that the existing test set is already strong with respect to a particular strength *t* (in this case 2-way), and tests for *t*+1 generated. The tradeoff in cost of applying coverage measurement is the need to map existing tests into discrete numerical values that can be analyzed by the coverage measurement tools (see Appendix C). For example, the days of the week in the example of Figure 10 would have to be mapped into 0 - 6 or 1 - 7. Future versions of the coverage measurement tools may include more flexibility in handling parameter values.

6.4 Chapter Summary

1. Many coverage measures have been devised for code coverage, including statement, branch or decision, condition, and modified condition decision coverage. These measures are based on aspects of source code and are not suitable for combinatorial coverage measurement.

2. Measuring configuration-spanning coverage can be helpful in understanding state space coverage. If we do use combinatorial testing, then configuration-spanning coverage will be 100% for the level of *t* that was selected, but we may still want to investigate the coverage our test set provides for *t*+1 or *t*+2. Calculating this statistic can help in choosing between t-way covering arrays generated by different algorithms. As seen in the examples above, it may be relatively easy to produce tests that provide a high degree of spanning coverage, even if not 100%. In many cases it may be possible to generate additional tests to boost the coverage of a test set.

7 COMBINATORIAL AND RANDOM TESTING

For testing to be most efficient and effective, we need an understanding of when a particular test method is most appropriate. That is, what characteristics of a problem lead us to use one form of testing over another, and what are the tradeoffs with respect to cost and effectiveness? Some studies have compared the effectiveness of combinatorial and random approaches to testing, but have reached conflicting results [3, 4, 56, 58]. This chapter presents an analysis [37, 38] of these two methods and discusses how random testing may complement combinatorial methods.

7.1 Coverage of Random Tests

Because a significant percentage of failures can only be triggered by the interaction of two or more variables, one consideration in comparing random and combinatorial testing is the degree to which random testing covers particular *t*-way combinations. Table 10 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 20 through Figure 24 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.

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 11 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 combinatorial test set 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 if the goal is 100% combination coverage for a given value of *t*.

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

Vars	Values	2-way Tests			3-way Tests			4-way Tests		
		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 11. 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 12. Average ratio of random/ACTS for covering arrays by values per variable, variables = 10, 15, 20, 25

7.2 Comparing Random and Combinatorial Coverage

The comparisons between random and 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 combinatorial testing.

- *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*. Combinatorial testing for a module with approximately 10 variables should be significantly more effective than random testing, while the difference between the two test methods should be less for modules with 20 or more variables.

- *For 100% combination coverage, the efficiency advantage of combinatorial testing varies directly with the number of values per variable and inversely with the interaction strength t.* Figure 25 illustrates how these factors (interaction strength *t* and values per variable *v*) combine: the ratio of random/combinatorial 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 than combinatorial testing, requiring 2 to nearly 5 times as many tests as a covering array generated by ACTS. Thus if 100% combination coverage is desired, combinatorial testing should be significantly less expensive than random test generation.

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. It is not uncommon for 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.

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 13 and Table 14 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 7.1 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.

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

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 13. 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 14. Higher interaction coverage of random tests

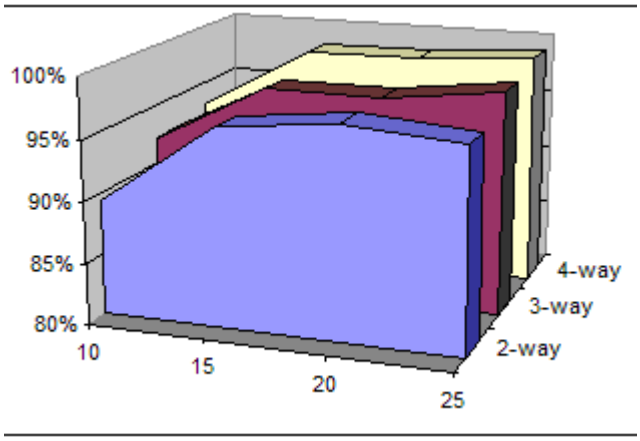


Figure 20. Percent coverage of *t*-way combinations for v=2.

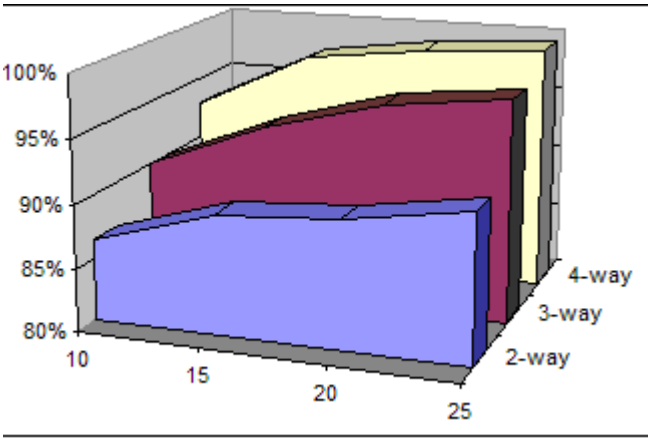


Figure 21. Percent coverage of *t*-way combinations for v=4.

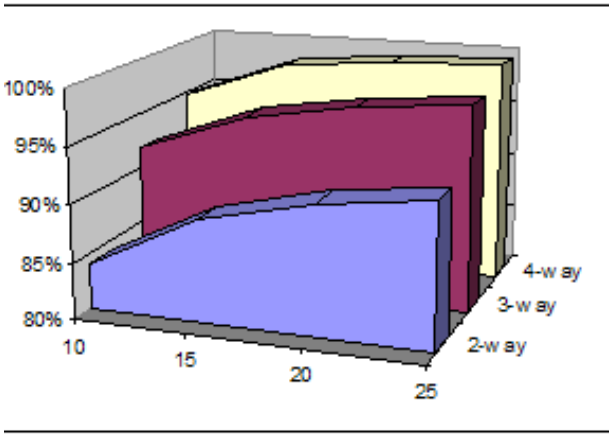


Figure 22. Percent coverage of *t*-way combinations for v=6.

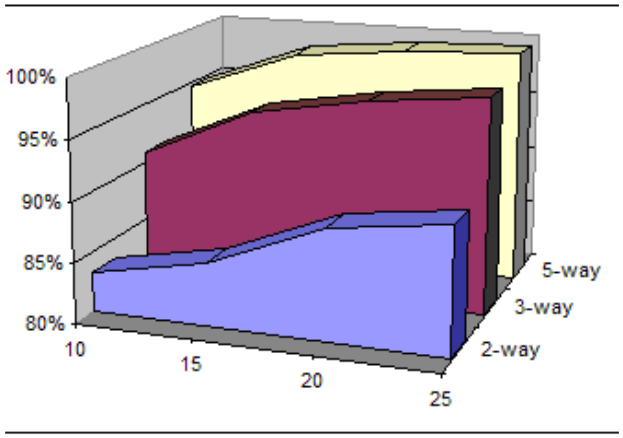


Figure 23. Percent coverage of *t*-way combinations for v=8.

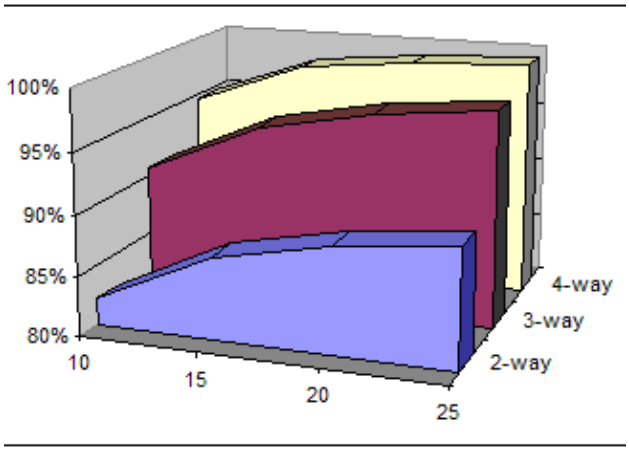


Figure 24. Percent coverage of *t*-way combinations for v=10

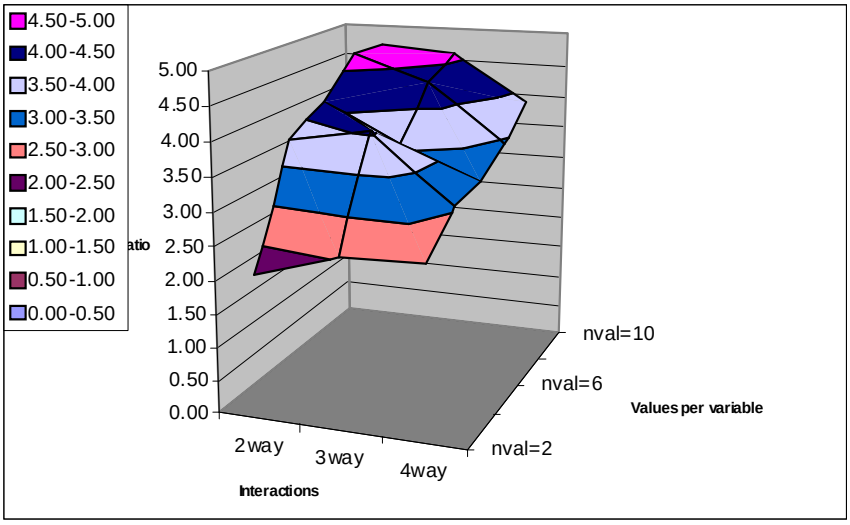


Figure 25. Average ratio of random/ACTS for covering arrays by values per variable

7.3 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 6), then adding tests as needed to provide full combinatorial coverage. Notice the last column of Table 10. 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. This is an option that may be cost effective even for smaller problems, and should be kept in mind for test planning.

7.4 Chapter Summary

1. 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.

2. While the analysis reported here does 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 the same. 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.

8 ASSERTION-BASED TEST ORACLES

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 gives conditions that must hold for the function to operate properly, in this case a non-zero divisor. It is the responsibility of the programmer to ensure that a zero divisor is never passed to the function. 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-checking 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 [27, 60, 47]. The assertions can serve as a sort of embedded proof of important properties, such that if the assertions hold for all executions of the program, then the properties encoded in the assertions are guaranteed 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 [4]). Some modern programming languages, such as Eiffel [52], include extensive support for including assertions that encode program properties, and tools such as the Java Modeling Language [42] 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.

8.1 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.

```
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.

8.2 Stronger Assertion-based Testing

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 in 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.

Tools such as Anna [44] for Ada, the Java Modeling language (JML) [42] and iContract [28] for Java, and APP [57] or Nana [46] for C, can be used to introduce 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 [25]. Results showed that 80% - 90% of errors could be found in this way.

8.3 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 [41], 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.

8.4 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 [44] and others. Placement within code is particularly important to assertion effectiveness [60, 61], 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.

9 MODEL-BASED TEST ORACLES

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. The model in this case is exactly what the name implies: it incorporates the most important aspects of the system, but not every detail such as the location of an amount on a screen (if it did include all details, it 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 Appendix C), we use NuSMV [18], 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 included in the appendix.

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.

9.1 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 9.2.1 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. The resulting specification is evaluated against a claim that negates each specified result R_j using a model checker, so that the model checker evaluates claims in the following form: $C_i => \sim R_j$, 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 equivalence classes used.

9.2 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.

9.2.1 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 27), 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.

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

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

7.      ASSIGN
8.      init(access) := START_;
--if access is allowed under rules, then next state is GRANT
--else next state is DENY
9.      next(access) := case
10.     u_l >= f_l & act = rd : GRANT;
11.     f_l >= u_l & act = wr : GRANT;
12.     1 : DENY;
13.     esac;
14.     next(u_l) := u_l;
15.     next(f_l) := f_l;
16.     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 27. 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.

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 28). 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 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 used as the gold standard for testing other products, so 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 28. NuSMV output

9.2.2 Integrating Combinatorial Tests into the Model

We will compute covering arrays that give all *t*-way combinations, with degree of interaction coverage = 2 for this example. This section describes the use of ACTS as a standalone command line tool, using a text file input (see Section 3.1). 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 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 29.

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 29. 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 9.1: C_i => ~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 9.2.1, 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));
```

9.2.3 Generating Tests from Counterexamples

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

9.3 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.

9.4 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.

10 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.

10.1 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 30.

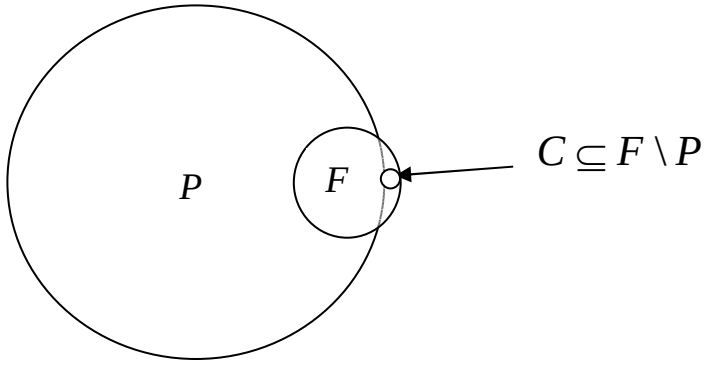


Figure 30. Combinations in failing tests but not in passing tests.

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 31 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 32 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 31. 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 32.

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

Figure 32. 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 33. 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
2 :3way 0,2,3 = 0,1,1
3 :3way 0,2,3 = 0,1,1
4 :3way 0,2,3 = 0,1,1
1 :3way 1,2,3 = 0,1,1
2 :3way 1,2,3 = 0,1,1
5 :3way 1,2,3 = 0,1,1
```

```
1 :4way 0,1,2,3 =   0,0,1,1
2 :4way 0,1,2,3 =   0,0,1,1
3 :4way 0,1,2,3 =   0,1,1,1
4 :4way 0,1,2,3 =   0,1,1,1
5 :4way 0,1,2,3 =   1,0,1,1
1 :4way 0,1,2,4 =   0,0,1,0
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 33. 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.

10.2 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.

10.3 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 – MATHEMATICS REVIEW

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

Combinatorics

Permutations and Combinations

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

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

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

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

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

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

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

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

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

Orthogonal Arrays

Many software testing problems can be solved with an orthogonal array, a structure that has been used for combinatorial testing in fields other than software for decades. An

orthogonal array, $OA_\lambda(N;t,k,v)$ is an N x k array. In every N x t subarray, each t-tuple occurs exactly λ times. We refer to t as the *strength* of the coverage of interactions, k as the number of parameters or components (degree), and v as the number of possible values for each parameter or component (order).

Example. Suppose we have a system with three on-off switches, controlled by an embedded processor. The following table tests all pairs of switch settings exactly once each. Thus $t = 2$, $\lambda = 1$, $v = 2$. Note that there are $v^t = 2^2$ possible combinations of values for each pair: 00, 01, 10, 11. There are C(3,2) = 3 ways to select switch pairs: (1,2), (1,3), and (2,3), and each test covers three pairs, so the four tests cover a total of 12 combinations which implies that each combination is covered exactly once. As one might suspect, it can be very challenging to fit all combinations to be covered into a set of tests exactly the same number of times.

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

Covering Arrays

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

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

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

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

Number of Tests Required

The challenge in computing covering arrays is to find the smallest possible array that covers all configurations of t variables. If every new test generated covered all previously uncovered combinations, then the number of tests needed would be

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

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

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

for n variables with v values each.

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

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

Table 1. Growth of v^t

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

Regular Expressions

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

Expression Operators

Basic elements of regular expressions include:

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

Combining Operators

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

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

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

Appendix B - Empirical Data on Software Failures

One of the most important questions in software testing is "how much is enough"? For combinatorial testing, this question includes determining the appropriate level of interaction that should be tested. That is, if some failure is triggered only by an unusual combination of more than two values, how many testing combinations are enough to detect all errors? What degree of interaction occurs in real system failures? This section summarizes what is known about these questions based on research by NIST and others [4, 7, 34, 35, 36, 65].

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

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

Table 1. Number of variables involved in triggering software failures

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

Table 2. System characteristics

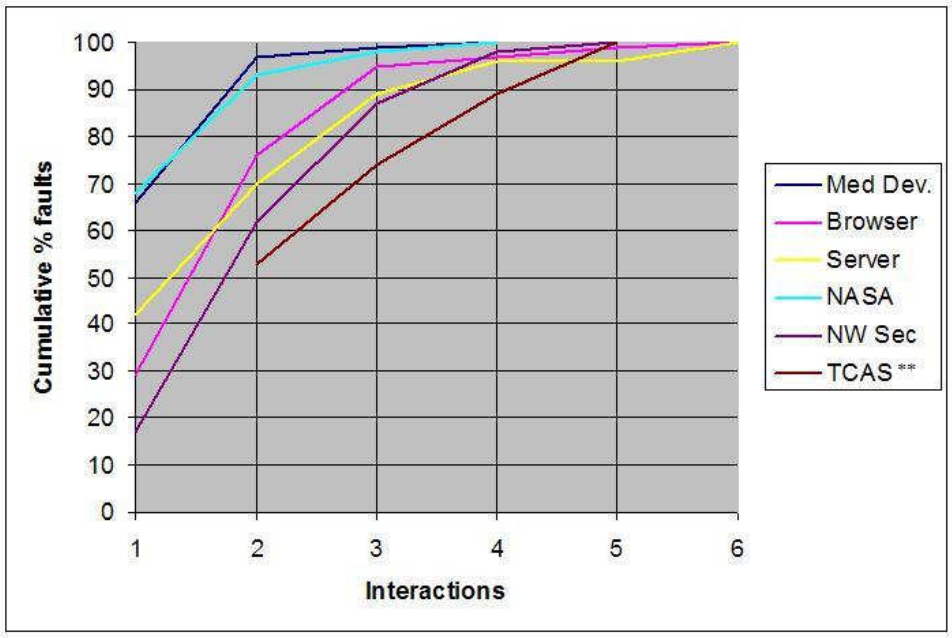


Figure 1. Cumulative percentage of failures triggered by *t*-way interactions.

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

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

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

Why do the failure detection curves look this way? That is, why does the error rate tail off so rapidly with more variables interacting? One possibility is that there are simply few complex interactions in branching points in software. If few branches involve 4-way, 5-way, or 6-way interactions among variables, then this degree of interaction could be rare for failures as well. The table below (Table 4 and Fig. 2) gives the number and percentage of branches in avionics code triggered by one to 19 variables. This distribution was

developed by analyzing data in a report on the use of MCDC testing in avionics software [16], which contains 20,256 logic expressions in five different airborne systems in two different airplane models. The table below includes all 7,685 expressions from *if* and *while* statements; expressions from assignment (:=) statements were excluded.

Table 4. Number of variables in avionics software branches

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

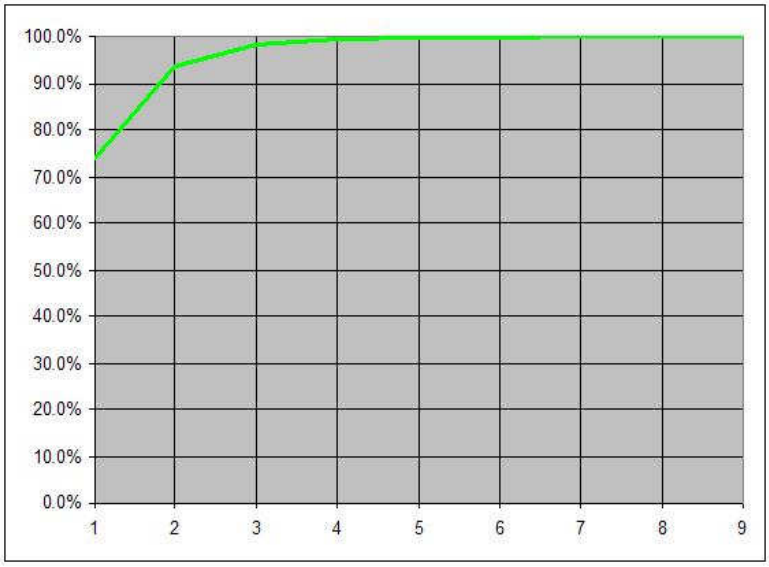


Figure 2. Cumulative percentage of branches containing *n* variables.

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

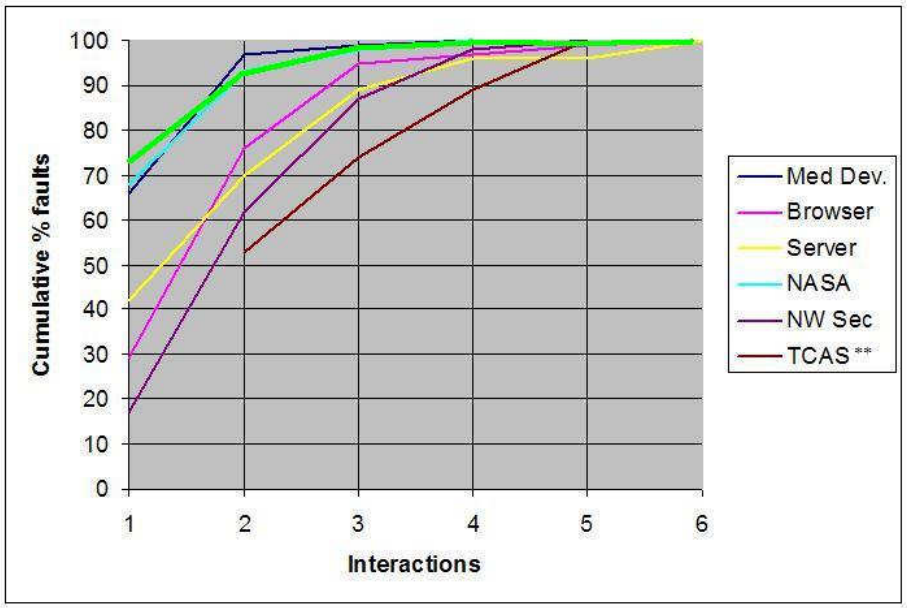


Figure 3. Branch distribution (green) superimposed on Fig. 1.

Appendix C - TOOLS FOR COMBINATORIAL TESTING

A variety of software tools are available to assist with combinatorial testing projects. Here we summarize those available from the NIST ACTS project. The ACTS covering array generator is generally faster and produces smaller test arrays than others, based on comparisons we have done in 2009. The other tools, to the best of our knowledge, have functions that are not available elsewhere.

- ACTS covering array generator – produces compact arrays that will cover 2-way through 6-way combinations. It also supports constraints that can make some values dependent on others, and mixed level covering arrays which offer different strength coverage for subsets of the parameters (e.g., 2-way coverage for one subset but 4-way for another subset of parameters). Output can be exported in a variety of formats, including human-readable, numeric, and spreadsheet. Either "don't care" or randomized output can be specified for tests that include combinations already fully covered by previous tests.

- Coverage measurement tool – produces a comprehensive set of data on the combinatorial coverage of an existing set of tests, as explained in Chapter 6. Output can be generated in spreadsheet format to allow easy processing and graphing.

- Sequence covering array generator – produces sequence covering arrays as defined in Chapter 5. It includes an option for constraints in the form of prohibited sequences.

To obtain any of these, see the ACTS web site at csrc.nist.gov/acts.

Appendix D - 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. Ammann, J. Offutt, *Introduction to Software Testing*, Cambridge University Press, New York, 2008.

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

4. 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

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

6. 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.

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

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

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

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

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

12. 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.

13. 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.

14. 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*.

15. 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

16. 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.

17. 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.

18. 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.

19. 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

20. 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

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

22. 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)

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

24. 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

25. 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

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

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

28. 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.

29. 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.

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

31. 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).

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

33. 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

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

35. 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 .

36. 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

37. 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.

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

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

40. D.R. Kuhn, J.M. Higdon, J.F. Lawrence, R. Kacker, Y. Lei, "Combinatorial Methods for Event Sequence Testing", (to appear).

41. 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.

42. 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

43. 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.*

44. 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.

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

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

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

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

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

50. 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.

51. 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.

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

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

54. 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

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

56. 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

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

58. 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.

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

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

61. 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.

62. W. Wang Sampath, S. Yu Lei Kacker, R., An Interaction-Based Test Sequence Generation Approach for Testing Web Applications, *High Assurance Systems Engineering Symposium*, 2008. HASE 2008. Nanjing, 3-5 Dec. 2008 pp. 209-218.

63. 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.

64. 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.

65. 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.

66. 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