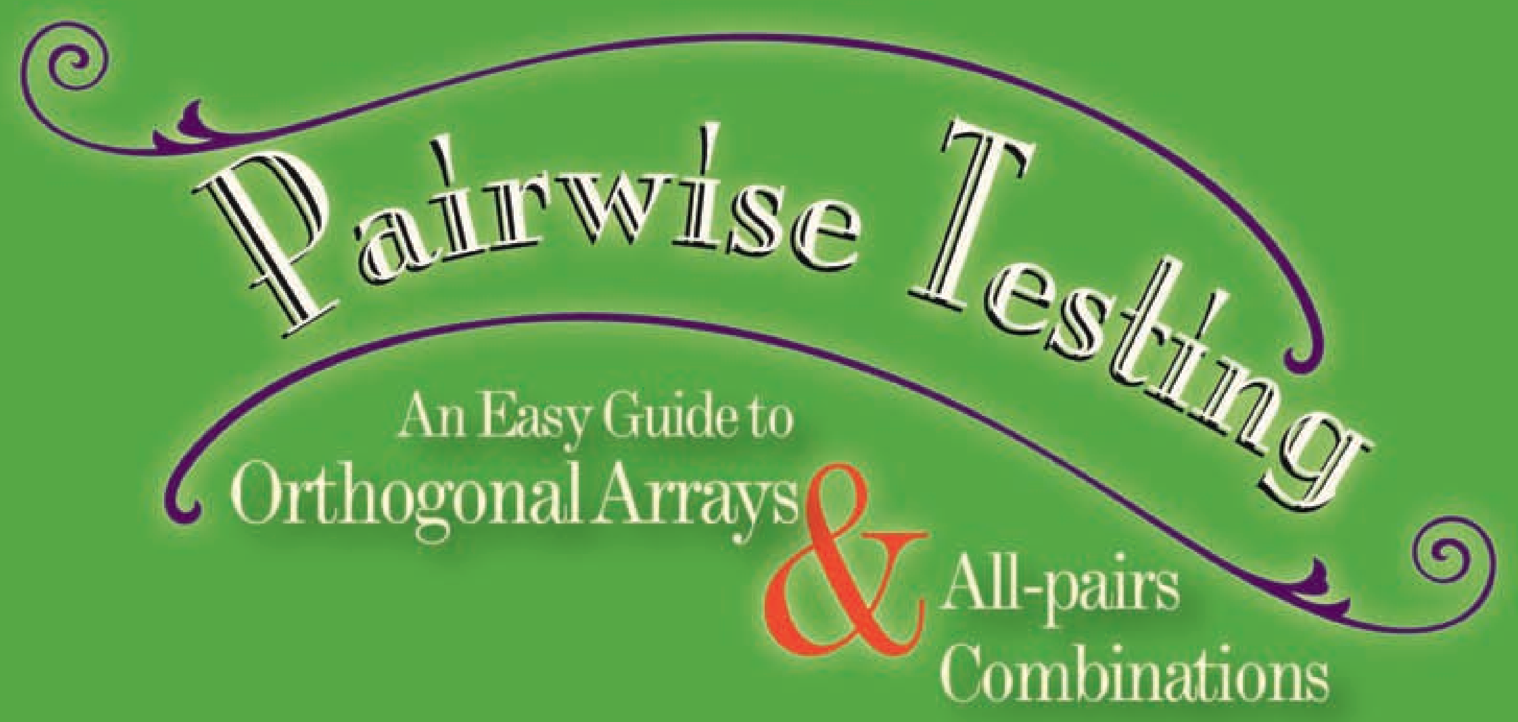
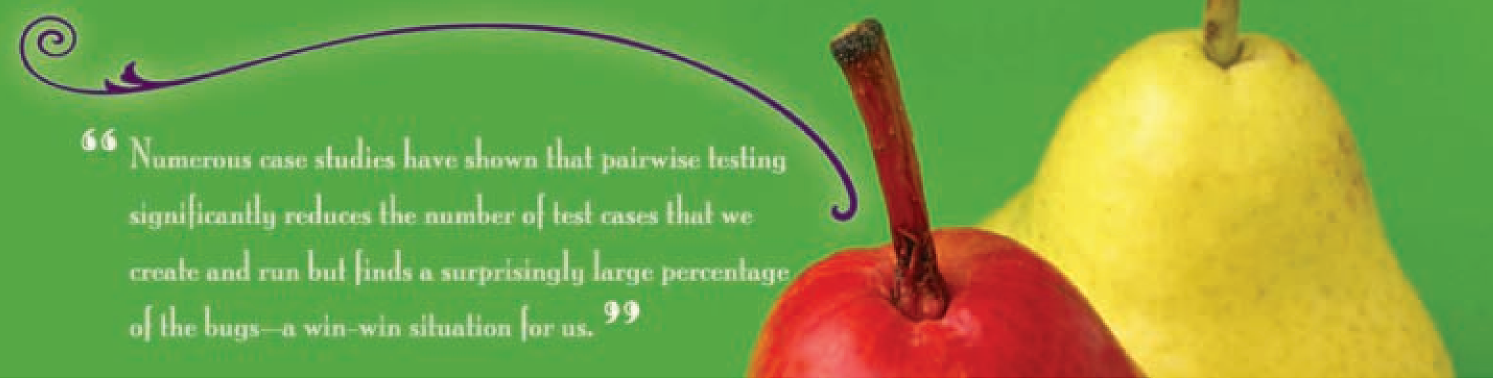


BY LLOYD RODEN

ISTOCKPHOTO



I remember the first time I heard the phrase "orthogonal arrays"—it was in 2000 when I attended a tutorial on performance testing by Ross Collard. During the tutorial, Ross mentioned how different combinations of configurations of machines could be tested effectively and ultra-efficiently using a technique called "orthogonal arrays."

At the time, I had no idea what Ross was talking about. I had studied mathematics at university. I knew orthogonal arrays were based on certain mathematical properties, so you might have expected me to rush out, research the subject, and grasp the technique with both hands—but I didn't! Probably because the word "orthogonal" has more than one syllable, I stayed far away from the concept. But finally, after years of ignorance, I set off on a quest to learn how this technique actually works and how orthogonal arrays along with other pairwise techniques can benefit testers.

The Combinatorial Testing Problem

One of the challenges we face during testing is managing the large number of test cases we need to create and execute. Consider this problem on a small scale with an example of a simplified mobile phone:

- Three call types (free, chargeable national, chargeable international)
- Three call methods (phone book, voice activated, quick dial)
- Two account types (pay as you go, contract)

In order to test all combinations we would need 3 x 3 x 2 = 18 test cases.

In this example, generating and running eighteen test cases is not a big problem—but most of our applications are not this simple. The problems we face within our organizations involve much more complex combinations and ultimately vast numbers of test cases.

Take, for instance, a car insurance quotation:

- Three policy types (third party; third party, fire, theft; fully comprehensive)
- Three storage modes (garage, driveway, road)
- Four no-claims-discount types (NCD) (0 years, 1 year, 2 years, and 3+ years)
- Two license types (provisional, full)
- Five age categories (17-21, 22-30, 31-40, 41-50, 50+)
- Five engine sizes (<1,001 cc, 1,001 cc-1,600 cc, 1,601 cc-2,000 cc, 2,001 cc-2,999 cc, 3,000 cc+)

To test all combinations we would need: 3 x 3 x 4 x 2 x 5 x 5 = 1,800 test cases.

Let us estimate that each test case takes half a day to design, set up, run, and check. This would equate to 900 days testing—that's almost four years!

Therefore, we are faced with a dilemma. On one hand, exhaustive testing is often impractical and in most instances impossible to achieve in the time allocated. But, if we were to ignore certain combinations, there is a risk of missing important bugs. On the other hand, if we were to test all combinations, we could find ourselves executing tests for infeasible combinations. For example, a quote for a thirty-five-year-old with four years' NCD and a 1,600 cc car garaged with a full license is probably a "typical" quote. But a seventeen-year-old with five years' NCD and a 4,000 cc Ferrari on the road with a provisional license is not that likely!

So how do we select the best combinations? We could start by selecting our favorite tests, the easy tests, or the first few tests that come to mind—and hope for the best. Or we could use a little magic called pairwise testing, in which we don't test all combinations of inputs, but we do test "all pairs" of possible inputs.

Why Use Pairwise Testing?

So how does this magic work? Numerous case studies have shown that pairwise testing significantly reduces the number of test cases that we create and run but finds a surprisingly large percentage of the bugs—a win-win situation for us (see the StickyNotes for references). One case study by D. Richard Kuhn and Michael J. Reilly showed that there were diminishing returns when the number of combinations tested was increased; 70 percent of the bugs were found with two or fewer conditions. There also have been case studies that show pairwise testing achieves a very high level of branch coverage (testing all branches within the code).

One of the reasons that pairwise testing can find a large number of bugs is that the bugs are either:

- *Single Modal*—in which something works or it fails
- *Dual Modal*—in which even though two things work by themselves, they fail when paired (connected) together
- *Multi Modal*—in which three or more things in combination don't work together

With case studies showing a vast percentage of bugs being found using pairwise testing, it appears that the majority of software bugs are either single or dual modal. Pairwise testing defines from all possible combinations a minimal set of tests that guides us to test all single and dual modal bugs. But it's important to note that pairwise testing tests some but not all combinations; therefore, we have not tested everything.

There are two methods for generating all-pairs combinations: orthogonal arrays and the all-pairs algorithm.

Orthogonal Arrays Explained

Having studied mathematics as part of my degree course, I can readily appreciate the power and wonder of tried-and-tested mathematical formulas. So those of you who usually switch off at the thought of using mathematics, please read on. You will be amazed by how this technique can help generate test cases.

	1	2	3
1	1	1	1
2	1	2	2
3	2	1	2
4	2	2	1

Table 1

Columns 1 & 2		Columns 1 & 3		Columns 2 & 3	
1	1	1	1	1	1
1	2	1	2	2	2
2	1	2	2	1	2
2	2	2	1	2	1

Figure 1

Orthogonal arrays have unique properties that allow them to be applied to a systematic way of testing. Originating from Euler, a mathematician who generated Latin squares, orthogonal arrays are a series of two-dimensional arrays based upon the following notation:

$L_x(n^y)$

Where x = number of rows

y = number of columns

n = maximum number choices within each category/variable

	1	2	3	4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Table 2

Please don't stop reading—you will grasp the concept and will be able to apply this technique, I promise!

The Unique Properties of Orthogonal Arrays

Consider the numbers 1 and 2. The total possible pair combinations are: (1, 1); (1, 2); (2, 1); and (2, 2).

Table 1 shows the $L_4(2^3)$ orthogonal array.

If we pick any two columns, we can see that every pair combination has been covered, as shown in figure 1.

Let's now consider the numbers 1, 2, and 3. There are nine pair combinations: (1, 1); (1, 2); (1, 3); (2, 1); (2, 2); (2, 3); (3, 1); (3, 2); and (3,3).

Let us examine the $L_9(3^4)$ orthogonal array shown in table 2.

Pick any two columns, and again ask the question, "Has every pair combination been exercised?" The answer is yes. But notice that not every possible combination has been covered, for example (1, 1, 2) has not been covered anywhere in this orthogonal array.

It is not necessary to create orthogonal arrays for your testing. You can buy books full of these arrays (a very good read whilst on holiday—if you suffer from insomnia), or you can download them from the various Web sites listed in the StickyNotes.

Let us follow a simple five-step guide to generating test cases with orthogonal arrays using our mobile phone example.

	1	2	3
1	1	1	1
2	1	2	2
3	1	3	3
4	2	1	2
5	2	2	3
6	2	3	1
7	3	1	3
8	3	2	1
9	3	3	2

Table 3

Step 1—Identify each variable within our scenario/requirements: call type, call method, account type.

Step 2—Determine the number of choices we have for each category/variable:

Call type = 3 (free, chargeable national, chargeable international)

Call method = 3 (phone book, voice activated, quick dial)

Account type = 2 (pay as you go, contract)

Step 3—Choose the best-fit orthogonal array using $L_x(n^y)$, where:

x = number of rows = number of test cases ultimately required (determined by the orthogonal array chosen)

y = number of columns = number of categories/variables

n = maximum number of choices for each category/variables

Ideally we would like a $2^1 3^2$ array (one variable with two choices and two variables with three choices), but unfortunately

it does not exist. The best-fit array would be a 3^3 array, and there is an $L_9 3^3$, which is shown in table 3.

	Call Type	Call Method	Account Type
1	Free	PB	PAYG
2	Free	VA	Cont
3	Free	QD	~
4	Nat	PB	Cont
5	Nat	VA	~
6	Nat	QD	PAYG
7	Int	PB	~
8	Int	VA	PAYG
9	Int	QD	Cont

Table 4

Step 4—Superimpose the variables and choices onto the array:

Column 1 = Call type
- Number 1 = Free
- Number 2 = Chargeable national (Nat)
- Number 3 = Chargeable international (Int)

Column 2 = Call method
- Number 1 = Phone book (PB)
- Number 2 = Voice activated (VA)
- Number 3 = Quick dial (QD)

Column 3 = Account type
- Number 1 = Pay as you go (PAYG)
- Number 2 = Contract (Cont)
- Number 3 = Don't care—choose any (~)

Step 5—Each row becomes a test case, as shown in table 4.

Test Case 1 = Free call using the phone book, and it is a pay-as-you-go account

Test Case 2 = Free call using voice activation, and it is a contract account

Etc.

So, we have reduced the eighteen possible test cases that test all combinations to nine test cases while testing all the pair combinations. This is probably not regarded as a huge saving, but now let's look at the car insurance example from earlier.

To adequately test all combinations we would need: 3 x 3 x 4 x 2 x 5 x 5 = 1,800 test cases.

There are six variables: policy types, storage modes, no-claims types, license types, age category, and engine sizes. Ideally we would like a $2^1 3^2 4^1 5^2$ array, but it doesn't exist. The maximum number of choices we have is five, and the number of variables we have is six. So we should look for a 5^6 array. The number of rows should be a minimum of (number of variables - 1) multiplied by the maximum number of choices. This is because we are comparing each variable with a different one in order to obtain the pairs.

In this example, number of rows = (6 - 1) x 5 = 5 x 5 = 25, and there is an $L_{25} 5_6$ array. Therefore we have: maximum number of rows = 25 = the number of test cases needed to test all pair combinations. This is a significant reduction, as we started with 1,800 possible test cases.

All-pairs Explained

Some people claim that it is not necessary to understand orthogonal arrays when an algorithm exists that is faster and easier to use. While there is some merit in this argument, I believe that by understanding orthogonal arrays and how they work, we become better able to justify our choice of test design approaches rather than "blindly" following an algorithm or tool. Once we understand the principles and theories, we can use tools and algorithms with knowledge and expertise.

An alternative method of testing all-pairs combinations is by using an "all-pairs algorithm" or utility. The following are some sources for all-pairs algorithm examples (see the StickyNotes for links):

- "Allpairs" by James Bach
- "pict" by Jacek Czerwoner at Microsoft
- "Classification Tree Editor CTE"

The advantage of using these utilities is that they do not need to search out the orthogonal array, so they can be faster to use. However, there are utilities that use orthogonal arrays rather than all-pairs algorithms, such as "ReducedArray2" by Gregory T. Daich.

Like orthogonal arrays, these utilities identify all the pair combinations, but they do not test all the combinations, and some combinations that are generated might be infeasible in the real world.

One warning in using any utility: Remember this is software, and like any software it will have bugs in it!

Mobile Phone Example Using James Bach's All-pairs Algorithm

I particularly like the fact that this utility tells you how many unique pairs you have in each test case. For example, as table 5 shows, test case 1 has three unique pairs: (Free, PB); (Free,

	Call Type	Call Method	Account Type	Unique Pairing
1	Free	PB	PAYG	3
2	Free	VA	Cont	3
3	Nat	PB	Cont	3
4	Nat	VA	PAYG	3
5	Int	QD	PAYG	3
6	Int	PB	Cont	2
7	Free	QD	Cont	2
8	Nat	QD	~PAYG	1
9	Int	VA	~PAYG	1

Table 5

Variable	Values	Choices
Browsers	IE5, IE6, IE7, Netscape 6.0, Netscape 6.1, Netscape 7.0, Mozilla 1.1, and Opera 7	8
Plug-ins	None, RealPlayer, MediaPlayer	3
Operating System	WME, W2000, NT, XP, XP-Pro, and Vista	6
Server	IIS, Apache, WebLogic	3
Server O/S	NT, XP, Linux	3

Table 6

PAYG); and (PB, PAYG). However, test case 9 has only one unique pair: (Int, VA), as (Int, PAYG) has already been covered in test case 5 and (VA, PAYG) has been covered in test case 4.

This would provide a useful coverage measure if you wanted to run the test cases in the order they have been presented. For example, if you only have the chance to test five test cases and choose the top five in the table, then you have tested fifteen unique pairs (out of a possible twenty-one). I would like to highlight the following potential danger in using these utilities: The test cases are produced automatically and, like any tool support for testing, should not replace the human mind. In the example above, it could be that a national, quick-dial call is the most important test—but it is test case 8.

The utility also has generated an actual account type for the cells where it does not matter. This could be a useful feature in that it saves you from thinking about what to choose, but it could also be viewed as a bad thing, as the coverage of account types in this example is weighted towards PAYG.

So What's the Difference?

As shown in our mobile phone example, in some instances orthogonal arrays and the all-pairs algorithm produce the same number of test cases (nine). But in other situations the number of test cases might differ.

Where the All-Pairs Algorithm Produces Fewer Test Cases

Consider table 6, taken from Lee Copeland's *A Practitioner's Guide to Software Test Design*.

The number of possible combinations = 8 x 3 x 6 x 3 x 3 = 1,296. Using orthogonal arrays we would need the $L_{64}8^2 4^3$ array, which would require sixty-four test cases. However, using the all-pairs algorithm generates forty-eight test cases.

Where the Orthogonal Arrays Produce Fewer Test Cases

Again, recall the car insurance example we looked at earlier. Using orthogonal arrays we would need the $L_{25}5^6$ array, which would require twenty-five test cases. Using the all-pairs algorithm generates twenty-eight test cases.

Preferences

There are differing schools of thought about which method is better. Some people view orthogonal arrays as better because they are based on tried-and-tested mathematical principles and more complex faults can be found. Others suggest that if testing all-pairs combinations is our goal, then the all-pairs algorithm can generate possible test cases far more quickly than orthogonal arrays.

As for which method I prefer, that is a difficult question. If I were given plenty of time to generate test cases, then I would prefer orthogonal arrays, as I like to see the test cases being formed as well as using arrays that have "magical" properties. This is probably because I loved mathematics at school.

However, producing the test cases manually using orthogonal arrays can be time consuming, and most of us do not have the luxury of time. In these instances I would choose to use any utility that supports pairwise testing. The utility should also use a common interface, such as Excel, as this avoids additional learning curves.

By understanding and using orthogonal arrays, I am better equipped to explain how the maximum number of test cases to test all combinations can be so dramatically reduced. This could be vital if you find yourself challenged within your organization.

My advice would be to practice with some exercises using orthogonal arrays, and learn and understand the principles of pairwise testing before using algorithms. Once you have mastered this, you will be able to download these free utilities and use them with confidence.

A Final Word of Warning

Some important points to keep in mind when implementing pairwise testing

- All-pairs and orthogonal arrays do not test all combinations.
- All-pairs and orthogonal arrays may not produce the most commonly used combinations.
- No priority is given to the tests.
- Some combinations produced might be infeasible in the real world.

You must use your testing skills, expertise, and experience to improve the test cases produced by these methods. It is worth starting with the set produced with these methods and then adding any "more important" combinations or even removing some combinations that are considered infeasible. Pairwise testing is a tool to be used to enhance our testing skills—not to replace them. **{end}**

Lloyd Roden has been involved in the software industry since 1980. He started his career as a developer and after five years moved into test analysis and test management working through key issues such as: testing to pre-defined deadlines, managing a test team, successfully implementing and using test automation tools, and building quality into the testing process. He joined Grove Consultants in April 1999.

Sticky Notes

For more on the following topics go to www.StickyMinds.com/bettersoftware.

- References
- Orthogonal array downloads
- Utility examples