Blog: Pairwise Testing

Version1.5, November, 2007
From Michael Bolton at Developsense.
Original post at http://www.developsense.com/pairwiseTesting.html

Maybe we can save time and effort and find bugs efficiently by a technique for testing variables and values in combination.

"Order one or more magazines by checking the appropriate boxes on this form"; "Choose one from Column A and one from Column B"; colours and options when we're purchasing an automobile; when we choose things from a set, we create combinations with our choices. In formal mathematics, the study of combinations is called "combinatorics"; combinations are, formally, "selections of a number of different items from a set of distinguishable items when the order of selection is ignored" [Penguin1998].

In computer systems, defects usually involve a single condition, independent of any other condition in the system. If there's a problem with a device or a variable or a setting, the problem is usually in that device or variable or setting alone. As an example, in a word processing program, a given font may be rendered improperly, irrespective of the printer, printer connection, or version of Windows. In some cases, however, there are two or more bad dancers at the party; normally everything is all right, but if one encounters the other while in a certain state, they trip over each other's feet. In testing, we want to be sure that we don't miss problems based on conflicts between two or more conditions, variables, or configurations, so we often test in combinations in order to find defects most efficiently. All printers except one might render a given font perfectly, and that printer might render all fonts perfectly except for that one. How can we maximize the chance of finding such a problem in the limited time we have to test?

When we consider such a task, we begin almost immediately to run into the problem of "complete testing". Suppose that a defect depends upon every variable in the system to be in one specific state. If only we could try all of the possible combinations by checking each aspect of the system in each of its states, combined with all of the other aspects in each of their possible states, we would thereby test every state and every combination of states in the system. With an effective test of this nature, we'd be guaranteed to find any defect that depended on the state of the system.

That guarantee of finding any state-based defect is very desirable. Alas, it's completely infeasible; even for a small program, with a relatively simple set of variables and relatively few possible states per variable, the total number of possible valid states in combination is intractably large. To arrive at that number, we consider each variable, and count the number of valid states for it. We then multiply the numbers of all those valid states for each variable together.

Suppose that there are five variables, each represented by a letter of the alphabet. And suppose that each variable can contain a value from one to five. Let's set variables A, B, C, and D all equal to 1. If those values are fixed, variable E can have a value from one to five—so five combinations there. We'll keep track of the total number of combinations in our leftmost column; our first five combinations will be numbered from 1 to 5.

Table 1: Varying column E only

Combination Number	A	B	C	D	E
1	1	1	1	1	1
2	1	1	1	1	2
3	1	1	1	1	3
4	1	1	1	1	4
5	1	1	1	1	5

Let's set variable D to two instead of one, and keep A, B, and C at one. Five more combinations, numbered 6-10.

Table 2: Varying Column E, having changed D

Combination Number	A	B	C	D	E
6	1	1	1	2	1
7	1	1	1	2	2
8	1	1	1	2	3
9	1	1	1	2	4
10	1	1	1	2	5

Then we'll set D to 3, then to 4, and then to 5, setting E to values from 1 to 5 each time. We'll have to go through 25 combinations before we exhaust all of the combinations for D and E, and only then can we change C to from 1 to 2. The columns on the right will roll over relatively quickly, but as we move leftwards, columns A, B, and C will change much less often, like the odometer on a car.

In fact, if there are only five variables in the program, each with five possible states, the program would be absurdly simple, yet we'd have to test 3125 combinations—five combinations for the first variable, times five for the second, times five for the third, times five for the fourth, times five for the fifth—5 x 5 x 5 x 5 x 5. If we could test one combination per minute, complete testing would take seven and a half days of continuous, seven-hour-a-day tests—assuming that we didn't die of boredom first. There's got to be a way to reduce the number of tests into something that we can handle.

To start figuring out how to cut down the number of tests, let's assume a simple program—an even simpler program than the one above, in which we have only two checkboxes. A checkbox can be in one of two states—checked or unchecked. Imagine that our defect depends upon checkbox A being cleared (that is, unchecked) and checkbox B being set (checked). If we try all of the possible settings in combination with one another, we'll find the defect. We'll set up a table, with a row for each variable. Within each cell, we'll put one of the variable's possible states—set or cleared. In the leftmost column, we'll put an index number so that we can refer to each combination by number.

Table 3: Possible states for two checkboxes

Combination Number	**Box A**	**Box B**
1	Set	Set
2 (defect!)	Cleared	Set
3	Set	Cleared
4	Cleared	Cleared

Combinations 1, 3, and 4 work all right, but Combination 2 exhibits the problem. We'd require four tests to make sure that we had covered all of the combinations in which A and B could be found. Two variables, and two settings for each; four combinations.

Now consider something only a little more complex: a black-box test of a dialog with three sets of radio buttons on it, representing travel needs in North America. The first set of radio buttons specifies a destination, by country; the second notes the choice of an aisle or window seat; and the third affords the choice of coach, business, or economy class. When the user makes her selections and presses "OK" to close the dialog, the application will take on a different path of execution for each combination of given options in given states.

As usual, the simplest kind of problem to detect is one that is triggered by a single variable in a single state. This is called a single-mode fault [Phadke1997]. For example, consider an application that contains a defect causes it to behave bizarrely when the "destination" option is set to "Canada". Naturally, the way to find such a defect would be to ensure that we run at least one test with "destination" set to "Canada".

Assume now that there is a defect that depends upon two conditions�for example, the destination field being set to "Mexico" and the service class being set to "Business" at the same time. This is known as a double-mode fault. It should be clear if we run at least one test with "destination" set to "Mexico" and the "Class" field to "Business", we'll encounter the defect.

With two objects and a two options for each, we could compile a relatively straightforward table. However, with any additional complexity, the required testing effort grows very quickly. For two radio buttons (each with three possible states) and one checkbox, there are 3 x 3 x 2, or 18 possible values to test. Here's a table that shows each possible combination:

Table 4: Possible states for two radio buttons and one checkbox

Test	Destination	Class	Seat Preference
1	Canada	Coach	Aisle
2	Mexico	Coach	Aisle
3 (defect!)	USA	Coach	Aisle
4	Canada	Business Class	Aisle
5	Mexico	Business Class	Aisle
6	USA	Business Class	Aisle
7	Canada	First Class	Aisle
8	Mexico	First Class	Aisle
9	USA	First Class	Aisle
10	Canada	Coach	Window
11	Mexico	Coach	Window
12 (defect!)	USA	Coach	Window
13	Canada	Business Class	Window
14	Mexico	Business Class	Window
15	USA	Business Class	Window
16	Canada	First Class	Window
17	Mexico	First Class	Window
18	USA	First Class	Window

We invent tests in response to risks. There's a risk that the developer has written his code with a single-mode fault, so we do want to be sure that we test every variable in each of its states if we can. There is a lessened risk that the developer has written in a double-mode fault, so ideally we'll want to produce at least one test of each variable in each of its states with every other variable in each of its states. If our theory of error is that a developer has made a double-mode fault mistake (say, if there is a conflict between one country, Mexico, and one service class, Business), we need to run at least one test that includes the combination of "Mexico" and "Business". On the other hand, if we run a number of equivalent tests on Mexico/Business, we could be duplicating effort and wasting time.

The most difficult kind of problem to find by black-box testing is one in which several variables are involved, and each must be in a specific, unique state to trigger the problem. For example, when the country field is set to "USA", the seat choice to "aisle", and the service class to "First", then and only then will the bug be triggered. This is known as a triple-mode fault, or more generally as a multi-mode fault, which describes faults associated with three or more parameters. For testers, finding this kind of problem depends on one of two approaches: testing all combinations of variables and states in the program; or some other technique combined with... well, luck. The first option is flatly impossible for anything but the most trivial program. The key to the second option is optimize the tests so to be as lucky as possible.

In 1997, researchers at Telcordia Technologies (formerly Bell Communications Research, or Bellcore) published a paper by Siddharta Dalal et. al., "The Combinatorial Design Approach to Automatic Test Generation" [Telcordia 1997]. Telcodia's studies suggest that "most field faults were caused by either incorrect single values or by an interaction of pairs of values." If that's generally correct, we ought to focus our testing on the risk of single-mode and double-mode faults. The Telcordia paper suggests that we can get excellent coverage by choosing tests such that 1) each state of each variable is tested, and 2) each variable in each of its states is tested in a pair with every other variable in each of its states. This is called pairwise testing or all-pairs testing.

Focusing on single- and double-mode faults reduces dramatically the number of combinations that we have to run. If all of the the pairs in a given combination exist in other combinations, we can drop that combination. For example, consider combination 14 from Table 4 above: the Mexico/Business pair also appears in combination 5, the Business/Window pair appears in combination 13, and the Mexico/Window combination appears in combination 11. We can drop combination 14 without leaving any of its pairs untested.

Here's a table, based on Table 4 above, that tests all of pairs of values:

Table 5: Possible paired states for two radio buttons and one checkbox

Number	Destination	Class	Seat Preference
1	Canada	Coach	Aisle
3 (defect!)	USA	Coach	Aisle
5	Mexico	Business Class	Aisle
8	Mexico	First Class	Aisle
9	USA	First Class	Aisle
11	Mexico	Coach	Window
13	Canada	Business Class	Window

15	USA	Business Class	Window
16	Canada	First Class	Window

This table makes sure that Canada is tested at least once with each of Coach, Business Class, and First Class, and with the checkbox in the Aisle state and the Window state. Similarly, every option is tested with every other option. We capture the double-mode fault exposed in combination number 3; we need not test combination 12, since its pairs are covered by combination 15 (which tests Destination=USA and Seat=Window), combination 11 (Class=Coach and Seat=Window) and combination 3 (which tests Destination=USA and Class=Coach, and exposes the conflict between them). Using the all-pairs theory as the basis for selecting tests, in this case we cut in half the number of test requirements, from 18 to nine.

Now, how did we manage to winnow out the duplicates? In this case, the table of all possible combinations is 18 entries. The easiest thing to do is to select lines, starting at the bottom of the table, and if all of the pairs for the line are duplicated somewhere higher up in the table, we can remove the line.

Line 18 contains USA, First Class, and Window.

the USA-First Class pair appears in line 9

the First Class-Window pair appears in lines 16 and 17

the USA-Window pair appears in line 12 and line 15

So we can drop line 18.

Line 17 contains Mexico, First Class, and Window

the Mexico-First Class pair appears in line 8

the First-Class Window pair appears in line 16

the Mexico-Window pair appears in lines 14 and 11

We can drop line 17 too.

Line 16 contains Canada, First Class, and Window

Canada-First Class appears in line 7

Canada-Window appears in line 13

First-Class-Window doesn't appear elsewhere in the table.

We must keep line 16 to preserve the First Class-Window pair.

This is a reasonable, albeit tedious, approach when the main table is very small, but a selection set with a trivial number of options increases complexity enormously. As we add variables or values, three things happen. First, the size of the all-combinations table expands exponentially. Second, the effort associated with sifting out the duplicated pairs becomes enormous. On the third point, there is some good news: the all-pairs selection cuts the number of required combinations much more dramatically than in our example above. Assume a program that has 75 binary options; the table of all possible combinations of states would extend to 2^75 entries. (Just for fun, that number happens to be 37,778,931,862,957,161,709,568). Telcordia's paper notes that 75 parameters of two states each can be pair-checked with only 28 combinations! [Telcordia, 1997]

So all-pairs testing is a very powerful technique for reducing the number of tests to be run, but there are two problems to consider.

The first problem is that if the defect in are example requires three variables to be just so (Destination=Canada, Class=Business, and Seat Preference=Aisle), our smaller table won't trigger the defect. Still, simply bashing away at random will be even less efficient. One workaround is to complement pairwise testing with other testing and QA techniques�code inspections, coverage tools, boundary tests and risk-based tests, to name but a few.

The second problem is that an all-pairs table takes a significant time to construct and to check for anything but a small number of variables and more possible states for each variable. If you want to test for more than pairs--triples, or n-tuples--the problem quickly becomes intractable for a human test planner. Fortunately, there are a few solutions to this problem.

Telcordia has created a Web-based tool called the Automatic Efficient Test Generator (AETG). You can find more information about the AETG Web service here; you can also find a paper which describes the principles here. The paper is tough but worthwhile reading. AETG-based testing isn't for everyone, but the principles behind it are worth considering, and the service may be useful for your project.

James Bach has written a tool called ALLPAIRS, a command-line PERL script that will create all-pairs tables. James' site is http://www.satisfice.com/. ALLPAIRS is not as efficient as AETG, but as he says, "Telcordia Technologies...has a web-based tool that does a somewhat better job than Allpairs...However, their tool costs $6000 a seat. Compare that to FREE."

At this point, you should have a sufficient understanding of pairwise testing to understand its value (and if you don't, I'd like you to let me know). If you're interested in more formal math surrounding pairwise testing, read on.

About Orthogonal Arrays

Pairwise testing is strongly influenced by a mathematical construct called an orthogonal array, or an OA. Orthogonal arrays are used in a variety of disciplines, including medical research, manufacturing, metallurgy, polling, and other fields that require testing and statistical sampling.

An orthogonal array has specific properties. First, an OA is a rectangular array or table of values, presented in rows and columns, like a database or spreadsheet. In this spreadsheet, each column represents a variable or parameter. Here are some column headers for a compatibility test matrix.

Table 6: Column Headers for a Test Matrix

Combination Number	Display Resolution	Operating System	Printer

The value of each variable is chosen from a set known as an alphabet. This alphabet doesn't have to be composed of letters�it's more abstract than that; consider the alphabet to be "available choices" or "possible values". A specific value, represented by a symbol within an alphabet is formally called a level. That said, we often use letters to represent those levels; we can use numbers, words, or any other symbol. As an example, think of levels in terms of a variable that has Low, Medium, and High settings. Represent those settings in our table using the letters A, B, and C. This gives us a three-letter, or three-level alphabet.

At an intersection of each row and column, we have a cell. Each cell contains a variable set to a certain level. Thus in our table, each row represents a possible combination of variables and values, as in Table 4 above.

Imagine a table that contains combinations of display settings, operating systems, and printers. One column represents display resolution (800 x 600 or Low; 1024 x 768, or Medium; and 1600 x 1200, High); a second column represents operating systems (Windows 98, Windows 2000, and Windows XP); and a third column represents printer types (PostScript, LaserJet, and BubbleJet). We want to make sure to test each operating system at each display resolution; each printer with operating system; and each display resolution with each printer. We'd need a table with 3 x 3 x 3 rows—27 rows—to represent all of the possible combinations; here are the first five rows in such a table:

Table 7: Compatibility Test Matrix (first five rows of 27)

Combination Number	Display Resolution	Operating System	Printer

1	**Low**	**Win98**	**PostScript**
2	**Low**	**Win98**	**LaserJet**
3	**Low**	**Win98**	**BubbleJet**
4	**Low**	**Win2000**	**PostScript**
5	**Low**	**Win2000**	**LaserJet**

Writing all those options out takes time; let's use shorthand. All we need is a legend or mapping to associated each variable with a letter of the alphabet: Low=A, Medium=B, High=C; Windows 98=A, Windows 2000=B, and Windows XP=C; PostScript=A, LaserJet=B, and BubbleJet=C. Again, this is only the first five rows in the table.

Table 8: Compatibility Test Matrix (first five rows of 27)

Combination Number	Display Resolution	Operating System	Printer
1	**A**	**A**	**A**
2	**A**	**A**	**B**
3	**A**	**A**	**C**
4	**A**	**B**	**A**
5	**A**	**B**	**B**

Note that the meaning of the letter depends on the column it's in. In fact, the table can mean whatever we like; we can construct very general OAs using letters, instead of specific values or levels. When it comes time to test something, we can associate new meanings with each column and with each letter. Again, the first five lines:

Table 9: Compatibility Test Matrix (first five rows of 27)

Combination Number	Variable 1	Variable 2	Variable 3
1	**A**	**A**	**A**
2	**A**	**A**	**B**
3	**A**	**A**	**C**
4	**A**	**B**	**A**
5	**A**	**B**	**B**

While we can replace the letters with specific values, a particular benefit of this approach is that we can broaden the power of combination testing by replacing the letters with specific classes of values. One risk associated with free-form text fields is that they might not be checked properly for length; the program could accept more data than expected, copy that data to a location (or "buffer") in memory, and thereby overwrite memory beyond the expected end of the buffer. Another risk is that the field might accept an empty or null value even though some data is required. So instead of associating specific text strings with the levels in column 3, we could instead map A to "no data", B to "data from 1 to 20 characters", and C to "data from 21 to 65,336 characters". By doing this, we can test for risks broader than those associated with specific enumerated values.

For example, suppose that we are testing an insurance application. Assume that Variable 1 represents a tri-state checkbox (unchecked = A = no children, checked = B = dependent children, and greyed out = C = adult children). Next assume that Variable 2 represents a set of radio buttons (labeled single, married, or divorced). Finally assume that Variable 3 represents a free-form text field of up to 20 characters for the spouse's first name. Suppose further the text field is properly range checked to make sure that it's 20 characters or fewer. However, suppose that a bug exists wherein the application produces a garbled record when the spouse's name is empty, but only when the "married" radio button is selected. By using classes of data (null, valid, excessive) in Column 3, pairwise testing can help us find the bug.

Back to orthogonal arrays. There are two kinds of orthogonal arrays: those which use the same-sized alphabet over all of the columns, and those which use a larger alphabet in at least one column. The latter type is called a "mixed-alphabet" orthogonal array, or simply a "mixed orthogonal array". If we decide to add another display resolution, we need to add another letter (D) to track it; we we would then have a mixed orthogonal array.

A regular (that is, non-mixed) orthogonal array has two other properties: strength and index.

Formally, an orthogonal array of strength S and index I over an alphabet A is a rectangular array with elements from A having the property that, given any S columns of the array, and given any S elements of A (equal or not), there are exactly I rows of the array where those elements appear in those columns.

Only a mathematician could appreciate that definition. Let's figure out what it means in practical terms.

If we're using an OA to test for faults, strength essentially refers to the mode of the fault for which we're checking. While checking for a double-mode fault, we consider pairs of columns and pairs of letters from the alphabet; we would need a table with a strength of 2. We select a strength by choosing a certain number of columns from the table and the same number of values from the alphabet; we'll call that number S. The array is orthogonal if in our S columns, each combination of S symbols appears the same number of times; or to put it another way, when S = 2, each pair of symbols must appear the same number of times. Thus we are performing pairwise or all-pairs testing when we create test conditions using orthogonal arrays of strength 2.

Were we checking for a triple-mode fault—a fault that depends on three valuables set to a certain value, we would need to look at three columns at a time, and combinations of three letters from the alphabet—a table with strength of 3. In general, strength determines how many variables we want to test in combination with each other.

An index is a more complicated concept. The orthogonal array has an index I if there are exactly I rows in which the S values from the alphabet appear. This effectively means that OAs that have an index must either have the same alphabet, or that all the columns must contain alphabets of equal size, which amounts to the same thing. An mixed-alphabet orthogonal array therefore cannot have an index.

The index is important when you want to make sure not only that each combination is tested, but that each combination is tested the same number of times. That's important in manufacturing and product safety tests, because it allows us to account for wear or friction between components. When we test combinations of components using orthogonal arrays, we find not only which combination of components will break, but also which combination will break first.

This leads us to the major difference between orthogonal array testing and all-pairs testing. If we are searching for simple conflicts between variables, we would presume that we'll expose the defect on the first test of a conflicting value pair. If we stick to that presumption, it is not very useful to test the same combination of paired values more than once, so when testers use pairwise testing, they use orthogonal arrays of strength 2 to produce pairwise test conditions, and, to save time, they'll tend to stick to an index of 1, to avoid duplication of effort. In cases where the alphabets for each column are of mixed size, we'll take the hit and test some pairs once and some pairs more than once.

Moreover, unless we specifically plan for it, it's not very likely that the variables in a piece of software or the parameters in configuration testing will have the same number of levels per variable. If we adjust all of our columns such that they have the same-sized alphabets, our array grows. This gives us an orthogonal array that is balanced, such that each combination appears the same number of times, but at a cost: the array is much larger than neccessary for all-pairs. In all-pairs testing, the combinations are pairs of variables in states such that each variable in each of its states is paired at least once with some other variable in each of its states. In strictly orthogonal arrays, if one pair is tested three times, all pairs have to be tested three times, whether the extra tests reveal more information or not.

A strictly orthogonal array will have a certain number of rows for which the value of one or more of the variables is irrelevant. This isn't necessary as long we've tested each pair against the other once; thus in practical terms, a test suite based on an orthogonal array has some wasted tests. Consquently, in pairwise testing, our arrays will tend not to have an index, and will be "nearly orthogonal".

So let's construct an orthogonal array. We have three columns, representing three variables. We'll choose an alphabet of Red, Green, and Blue--that's a three-level alphabet. Then we'll arrange things into a table for all of the possible combinations:

Table 10: All Combinations for Three Variables of Three Levels Each

	A	B	C
1	Red	Red	Red
2	Red	Red	Green
3	Red	Red	Blue
4	Red	Green	Red
5	Red	Green	Green
6	Red	Green	Blue
7	Red	Blue	Red
8	Red	Blue	Green
9	Red	Blue	Blue
10	Blue	Red	Red
11	Blue	Red	Green
12	Blue	Red	Blue
13	Blue	Green	Red
14	Blue	Green	Green
15	Blue	Green	Blue
16	Blue	Blue	Red
17	Blue	Blue	Green
18	Blue	Blue	Blue
19	Green	Red	Red
20	Green	Red	Green
21	Green	Red	Blue
22	Green	Green	Red
23	Green	Green	Green
24	Green	Green	Blue
25	Green	Blue	Red
26	Green	Blue	Green
27	Green	Blue	Blue

Now, for each pair of columns, AB, AC, and BC, each pair of colours appears exactly three times. Our table is an orthogonal array of level 3, strength 2, and index 3. In order to save testing effort, let's reduce the apperance of each pair to once.

Table 11: All-Pairs Array, Three Variables of Three Levels Each

	A	B	C
2	Red	Red	Green
4	Red	Green	Red
9	Red	Blue	Blue
12	Blue	Red	Blue
14	Blue	Green	Green
16	Blue	Blue	Red
19	Green	Red	Red
24	Green	Green	Blue
26	Green	Blue	Green

How did I happen to chose these nine specific combinations? That was the easy part: I cheated. I tried trial and error, and couldn't get it down to fewer than 12 entries without spending more time than I felt it was worth. The AllPairs tool mentioned above got me down to 10 combinations. However, on the Web, there are precalculated orthogonal array tables for certain numbers of variables and alphabets; one is at http://www.freequality.org/beta%20freequal/fq%20web%20site/Tools/Tagarray_files/tamatrix.htm; that's the one that I used here.

Mixed-alphabet orthogonal arrays permit alphabets of different sizes in the columns. The requirement is that the number of rows where given elements occur in a given number of prescribed columns is constant over all choices of the elements, but is allowed to vary for different sets of columns. Such an orthogonal array does not have an index, and is known as a "nearly orthogonal array." The airline example above is of this "mixed alphabet" type. The consequence is that some combinations will be tested more often than others. Note that in Table 5, there are two instances of Canada as the destination and Window as the seating preference, but each test is significant with respect to the class column. In cases where a single column has a larger alphabet than all the others, we'll test all pairs, but some pairs may be duplicated or may have variables that can be set to "don't care" values. If there's an opportunity to broaden the testing in a "don't care" value—for example by trying a variety of representatives of a class that might be otherwise be considered equivalent, we can broaden the coverage of the tests somewhat.

Once again, the goal of using orthogonal arrays in testing is to get the biggest bang for the testing buck, performing the smallest number of tests that are likely to expose defects. To keep the numbers low, the desired combinations are pairs of variables in each of their possible states, rather than combinations of three or more variables. This makes pairwise testing a kind of subset of orthogonal array testing. An orthogonal array isn't restricted to pairs (strength 2); it can be triples�strength 3, or combinations of three variables; or n-tuples�strength n, or combinations of n variables. However, orthogonal arrays with a strength greater than 2 are large, complex to build, and generally return too many elements to make them practicable. In addition, in the case of mixed alphabets, strictly orthogonal arrays produce more pairwise tests than we need to do a good job, so all-pairs testing uses nearly orthogonal arrays, which represents reasonable coverage and reduced risk at an impressive savings of time and effort.

Postscript 2007

I wrote this paper in 2004 (and earlier), and now, in 2007, it's time for an update.

First, there appears to be great confusion in the world between orthogonal arrays and pairwise testing. People use the terms interchangeably, but there is a clear and significant difference. I hope this article provides the distinction in a clear and useful way. If we're going to talk about these things we might as well get them right, so if I'm wrong, I urge you to disabuse me.

Second, I'm no longer convinced of the virtues of either orthogonal arrays or pairwise testing, at least not in the pat and surfacey way that I talked about them above.

An on-the-job experience provided a tremor to remind me to be skeptical. The project was already a year behind schedule (for an 18-month project), and in dire shape. Pretty much everyone knew it, so the goal became plausible deniability—or, less formally, ass-covering. One of the senior project manager looked over my carefully constructed pairwise table, and said "Hey... this looks really good—this will impress the auditors." He didn't have other questions, and he seemed not to be concerned about the state of the project. Impressing the auditors was all that mattered.

This gave me pause, because it suddenly felt as though my work was being used to help fool people. I wondered if I was fooling myself too. Until that moment, I had taken some pride in the work that I had been doing. Figuring out the variables to be tested had taken a long time, and preparing the tables had taken quite a while too. Was the value of my work matching the cost? I suddenly realized that I hadn't interacted with the product at all. When I finally got around to it, I discovered that the number, scope, and severity of problems in the product were such that the pairwise tests were, for that time, not only unnecessary but a serious waste of time. The product simply wasn't stable enough to use them. Perhaps much later, after those problems had been fixed, and after I had learned a lot more about the product, I could have done a far better job of creating pairwise tables—but by then I might have found that pairwise tables wouldn't have shed light on the things that mattered to the project. At the point that I was engaged in this elaborate exercise, I now believe that I should have been operating and observing the product, rather than planning to test a product that desperately needed testing immediately.

My test manager, for whom I have great respect, disappeared from that project due to differences with the project managers, and I was encouraged to disappear myself a week or two later. The project had been scheduled to deploy about six weeks after that. It didn't. It eventually got released four months later, was pulled from production, and then re-released about six months after that.

A year or so later, there was an earthquake-level jolt of skepticism in the form of this paper by James Bach and Pat Schroeder. If you want to understand a much more nuanced and coherent story about pairwise testing than the one that I prepared in 2004, look there.

I was seduced. Pairwise testing is very seductive. It provides us with a plausible story to tell about one form of test coverage, it's dressed up in fancy mathematical clothing, and it looks like it might reduce our workload. Does it provide the kind of coverage the kind that's most important to the project? Is reducing the number of tests we run a goal, or is it a form of goal displacement? Might we be fooling ourselves? Maybe, or maybe not, but I think we should ask. I should have.

In a blog post response, Cem Kaner added this, to remind me not to let the pendulum swing too far the other way. I think his comments warrant attention.

All-pairs is a coverage criterion, like all-triples, orthogonal arrays, all branches, all-fixed-bugs-retested, etc.

Set aside questions of whether it's good for impressing the auditors. I think it's good for at least three other purposes:

(1) Sometimes, there is a genuine (or perceived) risk of interaction among variables. An all-pairs tool gives you a simply-structured vehicle for designing tests that explore that risk. My primary concern about using all-pairs in this case (any form of combination testing that isn't carefully thought out) is that people often go through the motions of entering the variables' values into the input fields but they don't then continue with the program to see how it uses those values, checking the decisions or calculations that might actually be affected by the combination.(Similarly for combination tests used for configuration testing, you have to use the configured system enough to find out if there is a problem.) I think all-pairs is at least as good a start for the design as any other combination heuristic. Note that I am suggesting that all-pairs is a vehicle for EXPLORING the risk. Once you have a better idea of what variables interact and how, other tests become more interesting.

(2) Sometimes, there is no reason to know in advance whether there are interactions, but you have to do some budget negotiations. The all-pairs criterion is a way of stating a scope of preliminary combination testing. If you find bugs with these tests, you continue with more tests that are better tailored to the risks you now understand. On the other hand, if you don't find bugs with these tests, you stop at the agreed stopping point, having spent the expected cost.

(3) All-pairs provides a structure for considering what variables are worth combining and what values of those variables are worth combining. Not everyone needs that structure, and this structure doesn't work for every test designer, but it's a tool in the belt.

One way to think about this is to distinguish between three types of combination test design: (a) mechanical combination test design (all-pairs is an example, random combinations put together algorithmically with a random number generator is another) (b) scenario-based design, in which you consider how the product is likely to be used or installed and (c) risk-based design in which you consider specifically how these combinations might create a failure. Mechanically-oriented designs are not optimized for real-life emulation or risk. They are just mechanical.

A second distinction is between using combination testing in an exploratory way or a scripted-regression way. I've described using all-pairs in an exploratory way at the system level.

At the subsystem level, I'd be more likely to use all-pairs in a regression suite. Consider protocol testing -- testing communications between two applications (probably two that are authored and maintained separately). The protocol specifies how they communicate. Over time, one program might be revised in accordance with an upgraded protocol (or just be revised incorrectly), causing bad interoperation with the other application. A regression suite of test messages, where the risk over time is very general: a well-formed message (which combines values of many variables) will be misunderstood or a badly-formed-under-this-protocol message will be accepted. As with automated unit tests, I would expect these types of tests to be cheap to create and run and useful over time as refactoring aids, more than as system test aids.

This reminds me, as James Bach says in a forthcoming book, that skepticism isn't the rejection of belief, but the rejection of certainty.

References and Bibliograpy

One of the leading exponents of the Robust Testing method is Madhav S. Phadke. His paper, Planning Efficient Software Tests, describes orthogonal arrays (of strength 2, mostly) and their application in software testing.

Elfriede Dustin wrote an article called "Orthogonally Speaking" in the September/October 2001 issue of STQE Magazine.

[Phadke, 1997] Phadke, Madhav S., Planning Efficient Software Tests. Crosstalk, The Journal of Defense Software Engineering, October 1997. http://www.stsc.hill.af.mil/crosstalk/1997/10/planning.asp

[Telcordia, 1997] Cohen, D. M., et. al. The AETG System: An Approach to Testing Based on Combinatorial Design. IEEE Transactions On Software Engineering, July 1997 (Vol. 23, No. 7). http://www.argreenhouse.com/papers/gcp/AETGieee97.shtml

Hedayat, A.S., et. al. Orthogonal Arrays: Theory and Applications. Springer Verlag, August 1999.

Montgomery, Douglas C.Design and Analysis of Experiments, 5th Edition. Wiley Text Books, June 2000.
