Efficient Testing with All-Pairs

Prepared for STAREast 2003 International Conference on Software Testing
Bernie Berger

Acknowledgements: This presentation draws upon material from various sources, especially Lessons Learned in Software Testing by Kaner, Bach and Pettichord. Many thanks to them for their time and advice. Thanks to the AETG team who developed the all-pairs approach. Thanks to the reviewers of this paper for their insightful feedback: Lawrence Nuanez, Michael Steinhart, Dmitry Shchelokov

If you're a software tester who's been in the field for a few years, you may have found yourself in one of the following situations:

- You're working as hard as you can to find bugs in a huge system and you can't get to everything within the deadline. You've already stumbled across some good bugs, and you think there are more in there, but the deadline comes and the software is released. A week later, a major client finds a serious problem with the new release and lets everyone know about it in an industry press release. You begin thinking about what went wrong and how you can improve your testing coverage.

- You're on a job interview, and the person across the desk asks you how to test a product, especially when there's too much to do in so little time.

- Your management has an elementary understanding of software testing, and as a result, sets unrealistic expectations for you to "test everything." They demand 100% coverage, including testing all possible inputs, from all possible interfaces, into all possible system paths, into all possible outputs. You know these are ridiculous demands, and you start thinking about alternate testing methods. (and/or alternate employment opportunities).

Well, I'm able to cite these examples because I've been in each of these situations myself. A few years ago, I started thinking about the Coverage question of testing software – how can you "test everything" without really testing everything? How can you test efficiently: to minimize testing efforts but maximize testing results?

I found a method that I enjoy so much that I use it and talk about it as often as possible. I've seen this technique referred to as "Pairwise Testing," "Combinatorial Method," and "Orthogonal Arrays" (actually, each of these is similar but different), but I'll use the term "All-Pairs."

In All-Pairs test design, we are concerned with *variables* of a system, and the possible *values* each variable could take. We will generate testcases by pairing values of different variables. Don't re-read that last sentence -- generating testcases using All-Pairs is easier than it sounds. It's like learning a new card game – at first you have to learn the object of the game, all the rules, and all the exceptions to the rules, and then the tips and strategies. But after you've played a few times, it seems naturally easy to play. It's the same here. The best way to explain how it works is with an example, so keep reading…

Cartesian Products

A good starting point for a discussion about all-pairs is with Cartesian Products. A Cartesian product is a scenario in which every unit of a group is matched with every unit of every other group, so that all combinations of units are achieved across all groups.

For example, imagine the following simple software application: A one-screen GUI, with two dropdown lists and an ‘OK’ button. List1 contains three values, ‘0’, ‘1’, and ‘2’; List2 contains two values, ‘10’, and ‘20’. The user selects one value from each list, and after clicking OK the product of the two values is displayed on the screen. The variables and their values look like this:

List1	**List2**
0	10
1	20
2	

How would you test this program? How many input combinations are there? How long will it take to test all input combinations?

There are 3x2= 6 possible combinations. The 6 resulting combinations are a result of the Cartesian product of the two inputs, and would look like this:

List1	**List2**
0	10
0	20
1	10
1	20
2	10
2	20

Each value of each variable is matched so that you have achieved all combinations. You can execute these tests manually in less than two minutes. Let’s add a little more complexity; Version 2.0 of our program has some new features:

List1 now contains integer values 0 through 9, and List2 was changed to an input Textbox, allowing all integers between 1 and 99. Additionally, there are some new checkboxes. When checked, checkbox1 multiplies the product by -1 (makes it negative). Checkbox2 will multiply the product by itself (gives the product’s square). The variables and values now look like this:

List	**Textbox**	**Checkbox1 (-*x*)**	**Checkbox2 (x^2)**
0	1	On	On
1	2	Off	Off
2	3		
3	4		
4	…		
5	96		
6	97		
7	98		
8	99		
9			

Now I ask the same questions: How would you test this program? How many combinations are there? How long will it take to test all combinations?

Now there are 99x10x2x2=3,960 possible valid input combinations. There are also a host of invalid ones. (Note that the Textbox introduces a new concept – the possibility of invalid input. We will discuss that soon.) What to do?

Don't Use All Combinations

At the heart of all-pairs test design is the idea that you don't need to achieve all combination testing. Let's think about all combinations for a minute, in terms of finding bugs. In the previous example, let's say there is a bug where the program freezes when List=0 and Checkbox1 is on. That is, the program gets confused if it has to multiply zero by -1. That's not an unrealistic possibility, right? Now, if you were testing all combinations, how many test cases would fail because of this one bug?

99! That seems like a huge waste of effort, no? The fact is, the value in the Textbox is irrelevant to finding this bug. You could have found this bug with only one testcase – one that pairs a List=0 with a Checkbox=On – you don't need the extra 98 cases to find it.

What are you saying when you're testing all combinations? You're saying that you're looking for a bug that will only appear when one particular set of values across all your variables fail. Let me say that again in a different way: Each and every variable in the application has to be set to a particular value setting for the bug to appear. If even one value was changed, you wouldn't get that same bug. You're looking for a very specific set of conditions. Even in this nonsense calculator example, you have a 1:3,960 chance of failure. Will there be a bug that occurs once (and *only* once) out of 3,960 distinct input combinations? (And if by chance, there happens to be one, how low a priority do you think it would be? Picture the bug report…If there were 20 variables in your application, a bug description would look something like: "When *amount* is 3, and *security level* is set to high, and *color* is green, and *filter* is on, and *day* is Tuesday, and, and, and…<repeat 15 more times>… then the calculated output is incorrect")

In reality, all combinations is usually overkill testing. I should say though, that in mission-critical or safety-critical applications, all combinations might be a rational approach, such as pharmaceuticals or military defense systems, but that discussion is outside our scope.

Let's not be concerned with attempting to test combinations of all variables. Most bugs are found when only two variable values conflict, not when all conflict at the same time. In a recent NIST analysis of medical software device failures, only three of 109 failure reports indicated that more than two conditions were required to cause the failure (Wallace 2000). This is the main idea of all-pairs. It is more likely that you will find your bugs as a result of two values conflicting. It is far less likely that you need ALL variables in a particular value formation. Don't test all combinations. Test all-pairs.

OK, so that's the theory. How do you put it into practice? How do you figure out what pairs of variable values you have, and how do you fit them into actionable test scenarios?

Again, this is most easily explained with an example.

All-Pairs Working Example

First, figure out what your parameters (variables) are, and what each variable's possible values could be. Organize this information in a spreadsheet.

Next, simplify your values. Group them. Use boundaries. For example, in the Textbox, instead of listing every valid value between 1 and 99, choose a smart representative sample. (As a rule of thumb, unless you have other specific information, use min-1, min, min+1, max-1, max, max+1.) [For more information about this technique, see the material on Equivalence Class Partitioning in Testing Computer Software]

In our example, the user has more freedom to enter invalid choices in the Textbox than in the dropdown list. Realize that when input is non-discrete, you can still group it into values. For example, value categories for freeform text might be: all alpha chars, all numeric chars, uppercase, lowercase, words in single quotes, keywords, etc.

In our example, to keep it simple for now, let's reduce the 99 valid values plus the infinite number of invalid values down to three: **any valid integer**, **any invalid integer**, and **any alpha chars** (which would be invalid).

Let's also reduce the 10 values in the dropdown list to two: **0**, and **any other** value.
Now our variables and values look like this:

List	**Textbox**	**Checkbox1 (-*x*)**	**Checkbox2 (x^2)**
0	Valid int	On	On
Any other	Invalid int	Off	Off
	Alpha		

Next, put your variables across the top header row in a table. Order your variables so that the one with the most number of values is first and the least is last. Here, we put Textbox first because it has 3 values. The other variables each have 2 values.

Textbox (3)	**List (2)**	**Negative (2)**	**Square (2)**

Now we will start filling in the table. Each row of the table will represent a unique test case/scenario. We will fill the table column by column. Look at how many values there are in column 2. Here, we see that the List column has 2 values. That's how many times you will need to insert the values of the first column, Textbox. So we begin:

Textbox (3)	**List (2)**	**Negative (2)**	**Square (2)**
Valid integer			
Valid integer			
Invalid Integer			
Invalid Integer			
Alpha			
Alpha			

We inserted six rows. The three values of Textbox, each repeated twice. Also notice that we skipped a row between each set of values. This is important – we will get to that soon.

Now, we fill in column 2. For each set of values in column 1, we will put both values of column 2 like so:

Textbox (3)	List (2)	Negative (2)	Square (2)
Valid integer	0		
Valid integer	Other		
Invalid Integer	0		
Invalid Integer	Other		
Alpha	0		
Alpha	Other		

So far, so good. We have paired values across our two first variables. We can do a quick check… Valid and 0, Valid and Other. Invalid and 0, Invalid and Other. Alpha and 0, Alpha and Other. It's all good.

Let's continue on to the third variable. Column three is the checkbox that determines whether you want to multiply the product by -1. There are two values, on and off. Let's put in the ons and offs in column 3 and see what happens.

Textbox (3)	List (2)	Negative (2)	Square (2)
Valid integer	0	On	
Valid integer	Other	Off	
Invalid Integer	0	On	
Invalid Integer	Other	Off	
Alpha	0	On	
Alpha	Other	Off	

Let's check to make sure we have all our pairs between column 3 and column 2. We have a 0 and On, but wait – there's no 0 and Off. We have an Other and Off, but there's no Other and On. Let's swap around the values in the second set in the third column:

Textbox (3)	List (2)	Negative (2)	Square (2)
Valid integer	0	On	
Valid integer	Other	Off	
Invalid Integer	0	Off	
Invalid Integer	Other	On	
Alpha	0	*On*	
Alpha	Other	*Off*	

There. Much better. We have a 0/On, 0/Off, Other/On, and Other/Off. Notice that the last set on and off are arbitrary – we already have our pairs – and we don't care if the order is on/off or off/on.

Let's continue to the fourth column. This is the checkbox that multiplies the product by itself. There are also two settings, checked and unchecked. (I will call the values of this checkbox checked and unchecked so that they're different from the *on* and *off* in column 3, just for the example. You can have values of different variables called the same thing). We have to enter values in such a way that we get all our pairs. Let's give it a try:

Textbox (3)	List (2)	Negative (2)	Square (2)
Valid integer	0	On	Checked
Valid integer	Other	Off	Unchecked

Invalid Integer	0	Off	Checked
Invalid Integer	Other	On	Unchecked
Alpha	0	*On*	Unchecked
Alpha	Other	*Off*	Checked

Once again, let's check our pairs. We have a 0/Checked and 0/Unchecked. We have Other/Checked and Other/Unchecked. Good, now let's take a look at columns 3 and 4. We have On/Checked and On/Unchecked, and Off/Checked and Off/Unchecked. Not bad.

See, we fit every pair of values into six cases. If we were testing all combs, we would have 3x2x2x2=24. And if you consider that we reduced 99 to 3 in the Textbox, and 10 down to two in the dropdown list, that's an even bigger savings.

Remember when I said that skipping lines is important? Well, it is. Let's say Version 3.0 of our multiplier adds two more checkboxes. Checkbox3 will give the factorial value of the output, and Checkbox4 will convert the output into Hexadecimal notation. So we have to add two more columns to our table and enter our values. Let's continue with Checkbox3 in Column 5:

Textbox (3)	**List (2)**	**Negative (2)**	**Square (2)**	**Factorial (2)**	**Hex (2)**
Valid integer	0	On	Checked	Yes	
Valid integer	Other	Off	Unchecked	No	
Invalid Integer	0	Off	Checked	No	
Invalid Integer	Other	On	Unchecked	Yes	
Alpha	0	On	Unchecked	No	
Alpha	Other	Off	Checked	Yes	

Let's make sure each column has at least one pair with our newly added fifth column. Looks like we did OK: Column 2 is OK: (0/Yes 0/No, Other/Yes, Other/No), Column 3 is OK: (On/Yes, On/No, Off/Yes, Off/No), and Column 4 is OK: (Checked/Yes, Checked/No, Unchecked/Yes, Unchecked/No). We're golden. Notice that the on off sequence in the last set in the third column is no longer arbitrary as it was when we had only three columns filled in. We need it in that order now for our new value pairs.

Here we go one more time with the last column:

Textbox (3)	**List (2)**	**Negative (2)**	**Square (2)**	**Factorial (2)**	**Hex (2)**
Valid integer	0	On	Checked	Yes	Dec
Valid integer	Other	Off	Unchecked	No	Hex
Invalid Integer	0	Off	Checked	No	Hex
Invalid Integer	Other	On	Unchecked	Yes	Dec
Alpha	0	On	Unchecked	No	Dec
Alpha	Other	Off	Checked	Yes	Hex

And let's see how we did:
Column 2 is OK. We have a 0/Dec 0/Hex Other/Dec Other/Hex.
Column 3 is problematic: We do have an On/Dec and Off/Hex, but we're missing On/Hex and Off/Dec pairs.
Column 4 is OK: Checked/Dec, Checked/Hex, Unchecked/Dec, Unchecked/Hex
Column 5 is OK: We have a Yes/Dec, Yes/Hex, No/Dec, and No/Hex.

This time, we can't fit in our missing pairs (On/Hex and Off/Dec) by swapping around values. If we did, then other pairs would get out of whack. Instead, we simply add two more testcases that contain these pairs. Hence, the blank rows.

Textbox (3)	List (2)	Negative (2)	Square (2)	Factorial (2)	Hex (2)
Valid integer	0	On	Checked	Yes	Dec
Valid integer	Other	Off	Unchecked	No	Hex
		On			**Hex**
Invalid Integer	0	Off	Checked	No	Hex
Invalid Integer	Other	On	Unchecked	Yes	Dec
		Off			**Dec**
Alpha	0	On	Unchecked	No	Dec
Alpha	Other	Off	Checked	Yes	Hex

The other variable values are purely arbitrary. They need to be filled in with some value, but we don't care which value, because we already have all our pairs. Go ahead and fill in the empty cells as you desire, and there you have it -- all-pairs in eight cases instead of all combinations in 96!

Textbox (3)	List (2)	Negative (2)	Square (2)	Factorial (2)	Hex (2)
Valid integer	0	On	Checked	Yes	Dec
Valid integer	Other	Off	Unchecked	No	Hex
Valid integer	*Other*	On	*Unchecked*	*No*	Hex
Invalid Integer	0	Off	Checked	No	Hex
Invalid Integer	Other	On	Unchecked	Yes	Dec
Invalid Integer	*Other*	Off	*Unchecked*	*Yes*	Dec
Alpha	0	On	Unchecked	No	Dec
Alpha	Other	Off	Checked	Yes	Hex

I bet you're saying, this is really great, but do I have to go through this lengthy exercise with all this checking and swapping whenever I need an all-pairs analysis? This example is not nearly as complicated as the project I'm testing back home. It will take forever to figure it out with all my variables and values. Is there a way to automate these calculations? If only there were a free downloadable script that calculated all-pair combinations…

Introducing James Bach's All-Pairs Tool

Fortunately for you, James Bach has already developed an all-pairs calculator. And because James is such a nice guy, it's free to download from his website at www.satisfice.com He even gives you an instruction manual, a sample example, and the Perl source code. Let's take a quick look at it, and this time, let's use a real-life example.

You're testing an online mortgage application system. Using a web browser, users surf to the site and enter personal data into a series of forms. The system processes the data and, based on the data entered and the business logic programmed into the application, the system tells users what kind of mortgage products they qualify for and what their interest rate will be. Here's what the variable values look like:

Region[1]	**Tier**[2]	**Property**	**Credit**[3]	**Residence**[4]	**LTV**[5]	**NIV**[6]	**NAV**[7]	**Refinance**	**CC**[8]	**Intro Rate**[9]	**Emp. D**[10]
NY	L	1 fam	A+	Pri	80%	Yes	Yes	Yes	Cust	Yes	Yes
NJ	M	2 fam	A	Vac	90%	No	No	No	Bank	No	No
FL	H	3 fam	A-	Inv	100%						
TX	H+1	4 fam	B								
CA	H+2	Coop	<B								
DC	H+3	Condo									
Other											

There are 7x6x6x5x3x3x2x2x2x2x2x2 = 725,760 valid combinations. All-pairs does it in 50. Here's what to do:

1. Download and unzip ALLPAIRS from www.saticefice.com
2. Organize your variable and value matrix the way we did here and save it in tab-delimited text format. You can use Microsoft Excel and save as a .txt file. Remember to keep the formatting simple, and don't use spaces.
3. From a DOS command line, call the ALLPAIRS executable with the name of the .txt file as an argument. Redirect the output to an output file. For example,

```
ALLPAIRS.EXE MORTGAGE_IN.TXT > MORTGAGE_OUT.XLS
```

That's all there is to it.

Additional Considerations

Despite its ease of use, don't think that this tool is the silver bullet that will magically fix all your testing problems. You still need to think about how and when using all-pairs as a test technique is appropriate. We already discussed briefly that it may not be an appropriate method for all situations. Here are some more points to consider:

[1] Location of property by US State
[2] Amount borrowed.
[3] Credit rating of applicant
[4] Type of residence: Primary, Vacation, Investment
[5] Loan to Value: amount of loan as percentage of value of property
[6] No Income Verification
[7] No Asset Verification
[8] Closing Cost paid by Bank or Customer
[9] Lower rate for first year of loan, followed by higher rate for subsequent years
[10] Applicant is an employee of bank and gets a discount

Don’t be tempted to find all-pairs of all variables, just because you can.
Sometimes, a particular variable will or will not exist, depending on the value of another variable. For example, let’s say the same system that processes mortgages will also handle two other home finance products: home equity loans, and home equity lines of credit. There might be different business rules for processing line-of-credit applications. Maybe the different products are not offered in the same group of Regions, or perhaps there are different categories for LTV among the different home finance products.

It wouldn’t make sense to add a “product” variable with values “Standard Mortgage”, “Home Equity Loan” and “Line of Credit” into our all-pairs matrix because some of the resulting pairs (and therefore, testcases) would be undefined. If you did, you might generate a testcase that pairs a line-of-credit application with values that are not available for that product. This is different than negative testing, where you would be trying an option that the system doesn’t expect. Here, you couldn’t try that option at all because it doesn’t exist. You would be creating buggy testcases.

One solution is to create separate all-pairs matrixes for the individual products. In this example, you can create individual all-pairs matrixes because the business rules are so different for each Home Equity product.

You won’t find all your bugs by using this technique exclusively.
Remember, all-pairs only tests to see whether any two variables conflict, not if three or more conflict. Also, if you forget to include a variable, or you decide to exclude one from the matrix, its values won’t be meshed with the rest of the variables. Lastly, reducing the number of values per variable (as we did with the Textbox in the first example) could cause an important pair to be missed.

In Lessons Learned in Software Testing, Kaner, Bach, and Pettichord suggest to add additional cases, especially if you know of a specific combination that is widely used or likely to be troublesome. Additionally, try to introduce the all-pairs technique to your current test process gently. If your current testing process isn’t awful, then don’t end it suddenly and replace it only with All-Pairs. All-Pairs is a great method to *add* to your testing toolbox.

Don’t limit use of all-pairs to input variables
Throughout this presentation, we talked about variables as input to a system. Remember, variables can also mean test environments, paths through a system, and outputs. A common usage of all-pairs with non-input variables is in setting up test environments for Internet applications. Often, web apps need to be tested on a host of OS/Browser combinations. Nguyen illustrates this with an all-pairs example in Testing Applications on the Web.

Conclusion

Rooted in mathematical theory, the all-pairs technique is a thoughtful method when test planning. Download the all-pairs calculator and try it out. Using it, you can quickly generate test cases that have a good chance of finding bugs, instead of mindlessly copying and pasting text into testcase templates. The next time someone asks you to test “everything,” or you’re at an interview and you want to talk about test efficiency, remember this presentation and tell them about the benefits of the all-pairs approach.

References

Lessons Learned in Software Testing, Kaner, Bach, Pettichord

Testing Applications on the Web, Nguyen

Testing Computer Software, Kaner, Nguyen, Falk

"*The Combinatorial Design Approach to Automatic Test Generation*", Cohen, Dalal, Parelius, and Patton

"The AETG System: An Approach to Testing Based on Combinatorial Design" Cohen, Dalal, Fredman, Patton

"*Orthogonally Speaking*", Dustin, STQE Magazine 2001

"*Orthogonal Array Testing Strategy (OATS) Technique*", Harrell

"*Converting System Failure Histories into Future Win Situations*", Wallace and Kuhn

"*ALLPAIRS*", Bach

"A Dimensionality Model Approach to Testing and Imporving Software Robustness", Koopman, Pan, and Siewiorek