Pairwise Testing: A Best Practice That Isn't

James Bach
Principal, Satisfice, Inc.
james@satisfice.com

Patrick J. Schroeder
Professor, Department of EE/CS
Milwaukee School of Engineering
patrick.schroeder@msoe.edu

James Bach (http://www.satisfice.com) is a pioneer in the discipline of exploratory software testing and a founding member of the Context-Driven School of Test Methodology. He is the author (with Kaner and Pettichord) of Lessons Learned in Software Testing: A Context-Driven Approach. Starting as a programmer in 1983, James turned to testing in 1987 at Apple Computer, going on to work at several market-driven software companies and testing companies that follow the Silicon Valley tradition of high innovation and agility. James founded Satisfice, Inc. in 1999, a test training and outsourcing company based in Front Royal, Virginia.

Patrick J. Schroeder is a professor in the Electrical Engineering and Computer Science Department at the Milwaukee School of Engineering in Milwaukee, Wisconsin. Dr. Schroeder teaches courses in software testing, software engineering, and software project management. Dr. Schroeder is the current President of the Association of Software Testing, and newly formed organization dedicated to improving the practice of software testing by advancing the science of testing and its application. Prior to joining academia, Dr. Schroeder spent 14 years in the software industry, including seven years of at AT&T Bell Laboratories.

Abstract

Pairwise testing is a wildly popular approach to combinatorial testing problems. The number of articles and textbooks covering the topic continues to grow, as do the number of commercial and academic courses that teach the technique. Despite the technique's popularity and its reputation as a best practice, we find the technique to be over promoted and poorly understood. In this paper, we define pairwise testing and review many of the studies conducted using pairwise testing. Based on these studies and our experience with pairwise testing, we discuss weaknesses we perceive in pairwise testing. Knowledge of the weaknesses of the pairwise testing technique, or of any testing technique, is essential if we are to apply the technique wisely. We conclude by re-stating the story of pairwise testing and by warning testers against blindly accepting best practices.

No Simple Formulas for Good Testing

A commonly cited rule of thumb among test managers is that testing accounts for half the budget of a typical complex commercial software project. But testing isn't just expensive, it's arbitrarily expensive. That's because there are more distinct imaginable test cases for even a simple software product then can be performed in the natural lifetime of any tester [1]. Pragmatic software testing, therefore, requires that we take shortcuts that keep costs down. Each shortcut has its pitfalls.

We absolutely need shortcuts. But we also need to choose them and manage them wisely. Contrary to the fondest wishes of management, there are no pat formulas that dictate the best course of testing. In testing software, there are no "best practices" that we simply must "follow" in order to achieve success.

Take *domain partitioning* as an example. In this technique, instead of testing with every possible value of every possible variable, the tester divides test or test conditions into different sets wherein each member of each set is more or less equivalent to any other member of the same set for the purposes of discovering defects. These are called equivalence classes. For example, instead of testing with every single model of printer, the tester might treat all Hewlett-Packard inkjet printers as roughly equivalent. He might therefore test with only one of them as a representative of that entire set. This method can save us a vast amount of time and energy without risking too much of our confidence in our test coverage— as long as we can tell what is equivalent to what. This turns out to be quite difficult, in many cases. And if we get it wrong, our test coverage won't be as good as we think it is [2].

Despite the problem that it's not obvious what tests or test data are actually equivalent among the many possibilities, domain partitioning is touted as a technique we should be using [3]. But the instruction "do domain testing" is almost useless, unless the tester is well versed in the technology to be tested and proficient in the analysis required for domain partitioning. It trivializes the testing craft to promote a practice without promoting the skill and knowledge needed to do it well.

This article is about another apparent best practice that turns out to be less than it seems: pairwise testing. Pairwise testing can be helpful, or it can create false confidence. On the whole, we believe that this technique is over promoted and poorly understood. To apply it wisely, we think it's important to put pairwise testing into a sensible perspective.

Combinations are Hard to Test

Combinatorial testing is a problem that faces us whenever we have a product that processes multiple variables that may interact. The variables may come from a variety of sources, such the user interface, the operating system, peripherals, a database, or from across a network. The task in combinatorial testing goes beyond testing individual variables (although that must also be

done, as well). In combinatorial testing the task is to verify that different combinations of variables are handled correctly by the system.

An example of a combinatorial testing problem is the options dialog of Microsoft Word. Consider just one sub-section of one panel (fig. 1). Maybe when the status bar is turned off, and the vertical scroll bar is also turned off, the product will crash. Or maybe it will crash only when the status bar is on and the vertical bar is off. If you consider any of those conditions to represent interesting risks, you will want to test them.

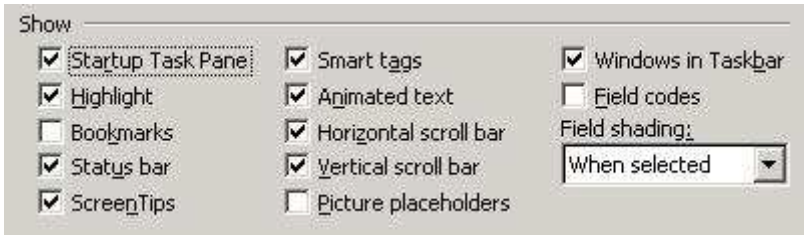

Figure 1. Portion of Options Dialog in MS Word

Combinatorial testing is difficult because of the large number of possible test cases (a result of the "combinatorial explosion" of selected test data values for the system's input variables). For example, in the Word options dialog box example, there are 12,288 combinations (2^{12} times 3 for the Field shading menu, which contains three items). Running all possible combinatorial test cases is generally not possible due to the large amount of time and resources required.

Pairwise Testing is a Combinatorial Testing Shortcut

Pairwise testing is an economical alternative to testing all possible combinations of a set of variables. In pairwise testing a set of test cases is generated that covers all combinations of the selected test data values for each *pair* of variables. Pairwise testing is also referred to as *all-pairs* testing and *2-way* testing. It is also possible to do all triples (3-way) or all quadruples (4-way) testing, of course, but the size of the higher order test sets grows very rapidly.

Pairwise testing normally begins by selecting values for the system's input variables. These individual values are often selected using domain partitioning. The values are then permuted to achieve coverage of all the pairings. This is very tedious to do by hand. Practical techniques used to create pairwise test sets include Orthogonal Arrays, a technique borrowed from the design of statistical experiments [4-6], and algorithmic approaches such as the *greedy* algorithm, presented by Cohen, et al. [7]. A free, open source tool to produce pairwise test cases, written by one of the authors (Bach), is available from Satisfice, Inc. [http://www.satisfice.com/tools/pairs.zip].

For a simple example of pairwise testing, consider the system in Figure 2. System S has three input variables X, Y, and Z. Assume that set *D*, a set of test data values, has been selected for each of the input variables such that $D(X) = \{1, 2\}$; $D(Y) = \{Q, R\}$; and $D(Z) = \{5, 6\}$.

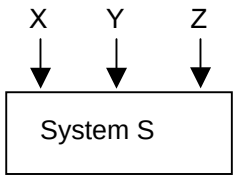

Figure 2. System S with 3 input variables

The total number of possible test cases is 2×2×2 = 8 test cases. The pairwise test set has a size of only 4 test cases and is shown in Table 1.

Table 1: Pairwise test cases for System S

Test ID	Input X	Input Y	Input Z
TC_1	1	Q	5
TC_2	1	R	6
TC_3	2	Q	6
TC_4	2	R	5

In this example, the pairwise test set of size 4 is a 50% reduction from the full combinatorial test set of size 8. You can see from the table that every pair of values is represented in at least one of the rows. If the number of variables and values per variable were to grow, the reduction in size of the test set would be more pronounced. Cohen, et al. [8] present examples where a full combinatorial test set of size 2^{120} is reduced to 10 pairwise test cases, and a full combinatorial test set of size 10^{29} is reduced to 28 test cases. In the case of the MS Word options dialog box, 12,288 potential tests are reduced to 10, when using the Satisfice Allpairs tool.

Pairwise testing is a wildly popular topic. It is often used in configuration testing [9, 10] where combinatorial explosion is a clear and present issue. It is the subject of over 40 conference and journal papers, and topic in almost all new books on software testing (e.g., [10-14]). The topic is so popular as to be taught in commercial and university testing courses (such as those of the authors), and testing conferences continue to solicit for more papers on the topic (www.sqe.com/starwest/speak.asp). Pairwise testing recently became much more accessible to testing practitioners through a number of open source and free software tools that generate sets of pairwise test cases.

In discussing a table representation of a pairwise test set created from an orthogonal array, Dalal and Mallows comment that the set "protects against any incorrect implementation of the code involving any pairwise interaction, and also whatever other higher order interactions happen to be represented in the tables [15]."

The pairwise approach is the popular because it generates small test sets that are relatively easy to manage and execute. But it is not just the significant reduction in test set size that makes pairwise testing appealing. It is also thought to focus testing on a richer source of more important bugs. It is believed by many, for instance, that most faults are not due to complex interactions, but rather, "most defects arise from simple pairwise interactions [6]." This is idea is supported by an internal memorandum at Bellcore indicating that "most field faults are caused by interactions of 1 or 2 fields [16]." Furthermore, it is argued that the probability of more than two particular values coinciding cannot be greater than and is probably a lot less than the probability

of only two coinciding values. Perhaps a problem that only occurs when, say, seven variable values line up in a perfect conjunction may not *ever* occur except in a test lab under artificial conditions.

This then is the pairwise testing story. See how nicely it fits together:

1) Pairwise testing protects against pairwise bugs
2) *while* dramatically reducing the number tests to perform,
3) *which is especially cool because* pairwise bugs represent the majority of combinatoric bugs,
4) *and* such bugs are a lot more likely to happen than ones that only happen with more variables.
5) *Plus,* the availability of tools means you no longer need to create these tests by hand.

The story is commonly believed to be grounded in practitioner experience and empirical studies. In referring to pairwise testing, Copeland states: "The success of this technique on many projects, both documented and undocumented, is a great motivation for its use [11]." Lei states that: "Empirical results show that pairwise testing is practical and effective for various types of software systems [17]," and refers the readers to several papers, including [7, 8, 16].

Well, let's look at the empirical evidence for pairwise testing.

What Do Empirical Studies Say About Pairwise Testing?

Early work by Mandl [4] described the use of orthogonal Latin Squares in the validation of an Ada compiler. The test objective was to verify that "ordering operators on enumeration values are evaluated correctly even when these enumeration values are ASCII characters." For the test data values selected, 256 test cases were possible; Mandl employed a Greco-Latin square to achieve pairwise testing resulting in 16 test cases. Mandl has great confidence that these 16 test cases provide as much useful information all 256 test cases. He has this confidence because he analyzed the software being tested, considered how it could fail, and determined that pairwise testing was appropriate in his situation. This analysis is enormously important! It is this analysis that we find missing in almost all other pairwise testing case studies. Instead of blindly applying pairwise testing as a "best practice" in all combinatorial testing situations, we should be analyzing the software being tested, determining how inputs are combined to create outputs and how these operations could fail, and applying an appropriate combinatorial testing strategy. Unfortunately, this lesson is lost early in the history of pairwise testing.

As commercial tools became available to generate pairwise and n-way test suites, case studies were executed using these tools. These studies are frequently conducted by employees of the pairwise tool vendor or by consultants that provide pairwise testing services.

In a study presented by Brownlie, Prowse, and Phadke [18], a technique referred to as Robust Testing™ (Robust Testing is a registered trademark of Phadke Associates, Inc.) is used. The Robust Testing strategy uses the OATS (Orthogonal Array Testing System) to produce pairwise

test sets in the system level testing of AT&T's PMX/StarMAIL product. In the study, a "hypothetical" conventional test plan is evaluated. It is determined that 1500 test cases are need, although time and resources are available for only 1000 test cases. Instead of executing the conventional test plan, the Robust Testing technique is applied to generate pairwise test set that combines both hardware configuration parameters and the software's input variables. The results of applying the technique are: 1) the number of test cases is reduced from 1000 to 422; 2) the testing effort is halved; and 3) the defect detection rate improves, with the conventional test plan finding 22% fewer faults than the Robust Testing technique. The problem with the study is that the conventional test plan is never executed. Important information, such as the number of faults detected by executing the conventional test plan, is estimated, not directly measured. The use of such estimates threatens the validity of the results and the conclusions that can be drawn from them.

Another commercial tool that can generate pairwise and n-way test suites is the Automatic Efficient Test Generator or AETG™ (AETG is a registered trademark of Telcordia Technologies, Inc., formerly Bellcore, the research arm of the Bell Operating Companies). Many industrial case studies have been conducted using the AETG. Cohen, et al. present three studies that apply pairwise testing to a Bellcore inventory database systems [7, 8, 16]. Software faults found during these studies lead the authors to conclude that the AETG coverage seems to be better than the *previous* test coverage. Dalal, et al. [19] presents two case studies using the ATEG on Bellcore's Integrated Services Control Point (ISCP). Similar studies by Dalal, et al. are presented in [20]. In all of these studies the pairwise testing technique appears to be effective in finding software faults. Some studies compare the fault detection of the pairwise testing with a less effective concurrently executed testing process. (Note that the AETG™ is a sophisticated combinatorial testing tool that provides complete n-way testing capabilities that can accommodate any combinatorial testing strategy; however, in most reported case studies it is used to generate pairwise test sets.)

The results of these case studies are difficult to interpret. In most of the studies, pairwise testing performs better than *conventional*, *concurrent*, or *traditional* testing practices; however, these practices are never described in enough detail to understand the significance of these results (some studies seem to be comparing pairwise testing to 1-way or *all values* testing). Additionally, little is known about the characteristics of software systems used in the studies (e.g., how much combinatorial processing do the systems perform?), and no information is supplied about the skill and motivation of the testers, nor about the kinds of bugs that are considered important. Clearly such factors have a bearing on the comparability of test projects.

A great deal has been written about achieving coverage using pairwise testing [7, 8, 15, 20-24]. "Coverage" in the majority of the studies refers to code coverage, or the measurement of the number of lines, branches, decisions or paths of code executed by a particular test suite. High code coverage has been correlated with high fault detection in some studies [25]. In general, the conclusion of these studies is that testing *efficiency* (i.e., amount time and resources required to conduct testing) is improved because the much smaller pairwise test suite achieves the same level of coverage as larger combinatorial test suites. While testing efficiency improves, we do not know the exact impact on the defect removal efficiency of the testing process. When executing combinatorial testing, one believes that executing different combinations of test data

values may expose faults. Therefore, one is not satisfied with a single execution of a hypothetical path through the code (as one is in code coverage testing), but rather may execute the same path many times with different data value combinations in order to expose combinatorial faults. Consequently, we find the use of code coverage to measure the efficiency of pairwise testing to be confusing. If one believes that code coverage is adequate for testing a product, one would not execute combinatorial testing. One would manually construct a white box test set that is almost certainly guaranteed to be smaller than the pairwise test set.

What do we know about the defect removal efficiency of pairwise testing? Not a great deal. Jones states that in the U.S., on average, the defect removal efficiency of our software processes is 85% [26]. This means that the combinations of all fault detection techniques, including reviews, inspections, walkthroughs, and various forms of testing remove 85% of the faults in software before it is released.

In a study performed by Wallace and Kuhn [27], 15 years of failure data from recalled medical devices is analyzed. They conclude that 98% of the failures could have been detected in testing *if* all pairs of parameters had been tested (they didn't execute pairwise testing, they analyzed failure data and speculate about the type of testing that would have detected the defects). In this case, it appears as if adding pairwise testing to the current medical device testing processes could improve its defect removal efficiency to a "best in class" status, as determined by Jones [26].

On the other hand, Smith, et al. [28] present their experience with pairwise testing of the Remote Agent Experiment (RAX) software used to control NASA spacecraft. Their analysis indicates that pairwise testing detected 88% of the faults classified as correctness and convergence faults, but only 50% of the interface and engine faults. In this study, pairwise testing apparently needs to be augmented with other types of testing to improve the defect removal efficiency, especially in the project context of a NASA spacecraft. Detecting only 50% of the interface and engine faults is well below the 85% U.S. average and presumably intolerable under NASA standards. The lesson here seems to be that one cannot blindly apply pairwise testing and expect high defect removal efficiency. Defect removal efficiency depends not only on the testing technique, but also on the characteristics of the software under test. As Mandl [4] has shown us, analyzing the software under test is an important step in determining if pairwise testing is appropriate; it is also an important step in determining what additional testing technique should be used in a specific testing situation.

Given that we know little about the defect removal efficiency of pairwise testing when applied in real software projects, we would at least like to know that it performs better than random testing. That is, our well planned, well reasoned best practice will remove more faults from software than just randomly selecting test cases. Unfortunately, we do not even know this for certain. In a controlled study in an academic environment, Schroeder, et al. [29] (referred to in this paper as Schroeder's study) found *no significant difference* in the detection removal efficiency of pairwise test sets and same-size randomly selected test sets. There was also no difference in the defect removal efficiency of 3-way and 4-way test sets when compared with random test sets (random test sets in this study where generated by randomly selecting from the full combinatorial test set without replacement; the same test data values were used to create the n-way and random test sets).

Schroeder's study is only a single study and the study was conducted using fault injected from a fault model (rather than real faults made by developers), but the results are distressing. Once again our best testing techniques appear to work no better than random selection [30]. The mystery disappears when we look closely at the random and pairwise test set. It turns out that random test sets are very similar to pairwise test sets! A pairwise test set covers 100% of the combinations of the selected values for each pair of input variables. For the two systems used in the study, random combinatorial test sets, on average, covered 94.5% and 99.6% of test data combination pairs [29].

Dalal and Cohen [15] present similar results in the analysis of pairwise and random test data sets generated for several hypothetical cases in which they varied both the number of input parameters and the number of selected test data values. For small test sets, they found that random sets covered a substantial proportion of pairs. For a test set of size 4, the random set covered on 68.4% of the pairs of data combinations; for a test set of size 9 random covered 65.4% of the pairs of data combinations. For 18 other test sets ranging in size from 10 to 40 test cases, the coverage of pairs by random sets ranged from 82.9% to 99.9% with an average coverage of 95.1%. White [23] provides a counter example that suggests that for very large test data sets, pairwise sets of size 121, 127, and 163, randomly selected test sets would not cover a high percentage of pairs of data combinations.

When one of the authors of this paper (Schroeder) confronted the other (Bach) with this data, Bach didn't initially believe it. "I had, after all, written a program to do pairwise testing, and promoted the technique in my testing classes," says Bach. Bach continues:

> *It just stands to reason that it would outperform mere random testing. So, with Schroeder driving me to back the airport after giving me a tour of his lab in Milwaukee, and the guy going on and on about pairwise testing being less useful than it appears, I pulled out my computer and modified my pairwise tool to choose tests randomly. To my surprise, the random selections covered the pairs quite well. As I expected, many more random tests were needed to cover all the pairs, but the random tests very quickly covered most of them. Schroeder seemed enormously pleased. I was shocked.*

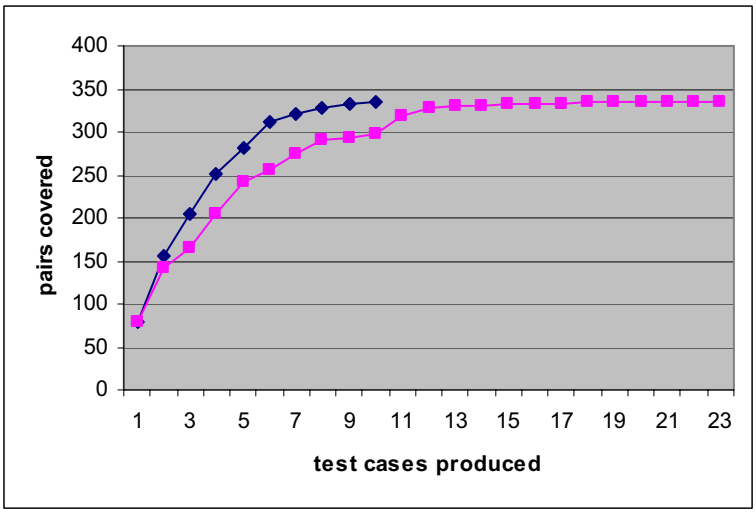

Figure 3. Pairwise vs. Random for the MS Word Options example

Figure 3 shows a comparison of pairwise and random testing in terms of covering pairs over a set of test cases. This is the MS Word Options dialog example that requires 12,288 cases to cover all combinations. The shorter curve is pairwise. In both cases we see a cumulative exponential distribution. The pairwise example is a steeper, shorter curve, but notice how even though the random selection requires double the test cases to complete, that's mainly due to the difficulty of finding the last few pairings. At the time the pairwise test set is complete, the random set has already covered 88% of the pairs. When you look at the coverage growth curve instead of merely the number of test cases to cover 100% of the pairs, it becomes clear why pairwise coverage algorithms aren't necessarily much better than simple random testing.

In additional research conducted by Bach, the same basic pattern in eight different and widely varying scenarios for test sets ranging up to 1200 test cases.

How Pairwise Testing Fails

Based on our review the literature and additional work in combinatorial testing, we feel the pairwise "story" needs to be re-written.

In Schroeder's study, two software systems (the Loan Arranger System (LAS) and the Data Management and Analysis System (DMAS)) were injected with faults. Each system was then tested with n-way test sets where n = 2, 3, and 4. Ten n-way test set of each type were generated using a greedy algorithm similar to that presented by Cohn, et al. [7].

Table 2. Fault classification for injected faults

Fault Type	LAS	DMAS
2-way	30	29
3-way	4	12
4-way	7	1
> 4-way	7	3
Not Found	34	43

Table 2 categorizes the faults in Schroeder's study, based on the n-way test set that exposed the fault. Faults in the "> 4-way" category were found by some 4-way test suites, indicating that data combinations of 5 or more values may be required to ensure the fault is detected.

Pairwise testing fails when you don't select the right values to test with.

The faults in the "not found" category in Table 2 are particularly interesting. Additional analysis of these faults shows that the majority of them are one-way faults. Since pairwise testing subsumes one-way testing, why weren't they found? Not only were they not detected by the pairwise tests, they weren't detected by any of the n-way test sets and would not be detected even if the full combinatorial test set was executed on the particular values we chose for those variables. This is because the test data value required to expose the fault was not selected. Similarly, a number of "not found" faults were 2-way faults that were not detected because a particular combination of data values had not been selected. Ironically, the statement that "most field faults are caused by the interaction of 1 or 2 fields [16]" is true; however, in this study it is true before *and after* pairwise testing is executed.

It is not true that a pairwise test set "protects against any incorrect implementation of the code involving any pairwise interaction … [15];" pairwise testing can only protect against pairwise interactions of *the input values you happen to select for testing*. The selected values are normally an extremely small subset of the actual number of possible values. And if you select values individually using domain analysis, you might miss productive test cases that contain special-case data combinations and error-prone combinations. The weaknesses of domain partitioning are carried over into pairwise testing and magnified.

Pairwise testing fails when you don't have a good enough oracle.

The selection problem is not an issue for the MS Word Options example. In that case we select all the values for testing, leaving out none. Does that mean pairwise testing will discover the important combinatorial bugs? The answer is still no, because of the oracle problem. The "oracle problem" is the challenge of discovering whether a particular test has or has not revealed a bug. In the Options case, we set the options a certain way, but we then have to use Word for a while to see not only whether all the options are behaving correctly, but also whether any other aspect of Word has failed. There's no sure way to know that some subtle problem, say a progressive corruption of the document, is not occurring. We simply do a little tour of Word and

hope for the best. Maybe among the pillars of some testing Olympus it is possible to specify coverage conditions and assume that any failures that happen will be immediately obvious, but on Earth it doesn't work that way. The pairwise test sets produced by our tools or orthogonal arrays can't help us solve that particular problem.

Pairwise testing fails when highly probable combinations get too little attention.

Another issue that is evident in the Options example is the non-uniformity of input distribution. Each set of options is not equally likely to be selected by users. Some are more popular. While we can't know the exact probability, perhaps there is one particular set of options, out of the 12,288 possibilities, that is far more likely to occur than any of the others: the default set. If the percentage of people who never change their options isn't 95%, it must be something close to that, considering that most people are not professional desktop publishers. Therefore, it might be more profitable to cluster our tests near that default. If we tested the default option set, then changed only one variable at a time from the default, that would be 14 tests. If we changed two variables at a time, that would be something on the order of 100 tests. But those combinations would be more realistic than ones that ignored the existence of default input. Of course, we can do both pairwise testing and this default input mutation testing. It's not one or the other, but the point is that pairwise testing might *systematically* cause us to ignore highly popular combinations.

Pairwise testing fails when you don't know how the variables interact.

A key factor in the success or failure of pairwise testing is the degree of interaction among the input variables in creating outputs. A testing practitioner may apply the pairwise technique to two different programs and have wildly different outcomes because of the combinatorial characteristics of the software being testing. Some software programs combine only a small number of input variables in creating outputs; some software programs combine a large number of input variables (e.g., 8 to 12) in creating outputs. For example, Figure 4 displays a program that has three input variables (X, Y, Z) and three outputs numbered 1 through 3. The arrows in the diagram indicate how input variables are combined to create program outputs. For example, inputs X and Y are combined in creating output #1 and inputs X and Z are combined in creating output #2. For this program, pairwise testing seems to be a highly appropriate choice. One would expect that executing pairwise testing on this program to be effective, and that use of this program in a case study of the effectiveness of pairwise testing would yield a very positive result.

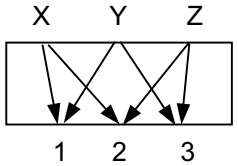

Figure 4. A program with 3 inputs and 3 outputs

What happens when outputs are created by the interaction of more the 2 input variables? Is pairwise testing still the best strategy? Figure 5 presents the interaction of input variables in creating output values for the DMAS system used in Schroeder's study. For the combinatorial operation being tested, DMAS had 42 output fields (numbered along the Y axis in Figure 5). The X axis in Figure 5 indicates how many input variables influence, or interact to create, each output value. The figure indicates that output #1, for example, is created using the value of only 1 input variable, and that output #42 is created by combining 10 different input variables.

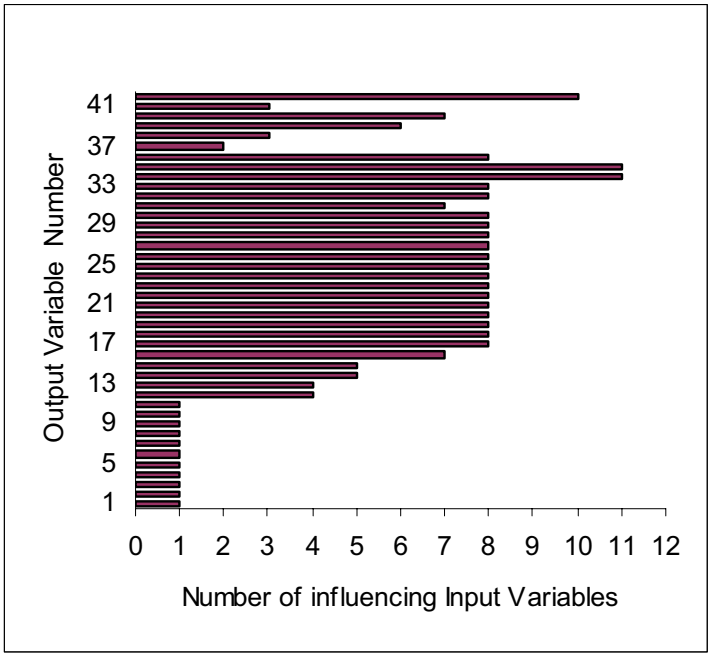

Figure 5. Number of inputs influencing an output for DMAS

Figure 5 indicates that DMAS has many outputs that are created by combining more than 2 input variables. Twenty of the program's output values are created by combining eight or more of the programs input variables. When an output is created from more than 2 input variables, it becomes possible for developers to create faults that may not be exposed in pairwise testing, but may be exposed only by higher-order test data combinations. For example, when testing DMAS in Schroeder's study, twelve 3-way faults, one 4-way fault, and three faults requiring combinations of 5 or above were detected. Given this understanding of how DMAS' input variables interact, some testers may feel that pairwise testing does not adequately address that risk inherent in the software and may focus on high-order interactions of the program's input variables.

The problem, as we see it, is that the key concept of how program inputs variables interact to create outputs is missing from the pairwise testing discussion. The pairwise testing technique does not even consider outputs of the program (i.e., the definition and application of the pairwise technique is accomplished without any mention of the program's outputs). In some instances, blindly applying pairwise testing to combinatorial testing problems may increase the risk of delivering faulty software. While the statement "most field faults are caused by the interaction of 1 or 2 fields [16]" may be true, testing practitioners must be cautioned that in the case of

pairwise testing, detecting "most" faults may result in a defect removal efficiency closer to the 50% experienced by Smith, et al. [28] rather than to the 98% projected by Wallace and Kuhn [27].

If pairwise testing is anointed as a best practice, it seems to us to become easier for novice testers, and experienced testers alike, to blindly apply it to every combinatorial testing problem, rather than doing the hard work of figuring out how the software's inputs interact in creating outputs. Gaining a black-box understanding of a program's interactions for a given set of test data values is not an easy task. Testers currently do this analysis in the process of determining the expected results for their test cases. That is, to correctly determine an expected output of a test case one must know which inputs are used in creating the output. This is an error prone process and automation can sometimes be used to determine how a program's inputs interact. The display in Figure 5, for example, was created using a technique called input-output analysis [31], which is an automated approach to determining how a program's input variables influence the creation of the program's outputs.

Conclusion

The typical pairwise testing story goes like this:

1) Pairwise testing protects against pairwise bugs
2) *while* dramatically reducing the number tests to perform,
3) *which is especially cool because* pairwise bugs represent the majority of combinatoric bugs,
4) *and* such bugs are a lot more likely to happen than ones that only happen with more variables.
5) *Plus,* you no longer need to create these tests by hand.

Critical thinking and empirical analysis requires us to change the story:

1) Pairwise testing **might find some** pairwise bugs
2) *while* dramatically reducing the number tests to perform, **compared to testing all combinations, but not necessarily compared to testing just the combinations that matter.**
3) *which is especially cool because* pairwise bugs **might** represent the majority of combinatoric bugs, **or might not, depending on the actual dependencies among variables in the product.**
4) *and* **some** such bugs are more likely to happen than ones that only happen with more variables, **or less likely to happen, because user inputs are not randomly distributed.**
5) *Plus,* you no longer need to create these tests by hand, **except for the work of analyzing the product, selecting variables and values, actually configuring and performing the test, and analyzing the results.**

The new story may not be as marketable as the original, but it considerably more accurate. Sorting it out, there are at least seven factors that strongly influence the outcome of your pairwise testing:

1) The actual interdependencies among variables in the product under test.
2) The probability of any given combination of variables occurring in the field.
3) The severity of any given problem that may be triggered by a particular combination of variables.
4) The particular variables you decide to combine.
5) The particular values of each variable you decide to test with.
6) The combinations of values you actually test.
7) Your ability to detect a problem if it occurs.

Items 1 through 3 are often beyond your control. You may or may not have enough technical knowledge to evaluate them in detail. If you don't understand enough about how variables might interact, or you test with combinations that are extremely unlikely to occur in the field, or if the problems that occur are unimportant, then your testing will not be effective.

Items 4 and 5 make a huge difference in the outcome of the testing. Item 4 is directly dependent on item 1. Item 5 is also dependent on item 1, but it also relates to your ability to find relevant equivalence classes. Similarly, item 7 depends on your ability to determine if the software has indeed failed.

The pairwise testing process, as described in the literature, addresses only item 6. That is not much help! Item 6 is solved by the pairwise testing tool you are using. Because of the availability of tools, this is the only one of the seven factors that is no longer much of a problem.

Pairwise testing permeates the testing world. It has invaded our conferences, journals, and testing courses. The pairwise story is one that all testers would like to believe: a small cleverly constructed set of test cases can do the work of thousands or millions of test cases. As appealing as the story is, it is not going to be true in many situations. Practitioners must realize that pairwise testing does not protect them from all faults caused by the interaction of 1 or 2 fields. Practitioners must realize that blindly applying pairwise testing may increase the risk profile of the project.

Should you adopt pairwise testing? If you can reduce your expectations of the technique from their currently lofty level the answer is *yes*. Pairwise testing should be one tool in a toolbox of combinatorial testing techniques. In the best of all possible worlds, a tester selects a testing strategy based on the testing problem at hand, rather than fitting the problem at hand to a pre-existing test strategy [10].

Develop Skill, Take Ownership, but Do Not Trust "Best Practice"

All useful and practical test techniques are heuristic: they are shortcuts that might help, but they do not guarantee success. Because test techniques are heuristic, there can be no pat formulas for great testing. No single test technique will serve; we are obliged to use a diversified strategy. And to be consistently successful requires skill and judgment in the selection and application of those techniques.

We, the authors, both began with the assumption, based on a good-sounding story, that pairwise testing is obviously a good practice. By paying attention to our experiences, we discovered that our first impression of pairwise testing was naïve.

Because test techniques are heuristic, focus not on the technique, but your skills as a tester. Develop your skills.

We recommend that you refuse to follow any technique that you don't yet understand, except as an experiment to further your education. You cannot be a responsible tester and at the same time do a technique just because some perceived authority says you should, unless that authority is personally taking responsibility for the quality of your work. Instead, relentlessly question the justification for using each and any test technique, and try to imagine how that technique can fail. This is what we have done with pairwise testing.

Don't follow techniques, own them.

Achieving excellence in testing is therefore a long-term learning process. It's learning for each of us, personally, and it's also learning on the scale of the entire craft of testing. We the authors consider pairwise testing to be a valuable tool, but the important part of our story is how we came to be wiser in our understanding of the pairs technique, and came to see its popularity in a new light.

We believe testers have a professional responsibility to subject every technique they use to this sort of critical thinking. Only in that way can our craft take its place among other respectable engineering disciplines.

References

[1] C. Kaner, J. L. Falk, and H. Q. Nguyen, *Testing Computer Software*, 2nd ed. New York: Van Nostrand Reinhold, 1993.

[2] C. Kaner, "Teaching Domain Testing: A Status Report," presented at Conference on Software Engineering Education & Training, Norfolk, Virginia, 2004.

[3] British Computer Society, "Information Systems Examinations Board Practitioner Certificate in Software Testing Guidelines and Syllabus, Ver. 1.1," http://www.bcs.org/BCS/Products/Qualifications/ISEB/Areas/SoftTest/Syllabus.htm.

[4] R. Mandl, "Orthogonal Latin Squares: An Application of Experiment Design to Compiler Testing," *Communication of the ACM*, vol. 28, no. 10, pp. 1054-1058, 1985.

[5] A. S. Hedayat, N. J. A. Sloane, and J. Stufken, *Orthogonal Arrays: Theory and Applications*. New York: Springer-Verlag, 1999.

[6] J. M. Harrel, "Orthogonal Array Testing Strategy (OATS) Technique," http://www.seilevel.com/OATS.html, 05/28/2004.

[7] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG System: An Approach to Testing Based on Combinatorial Design," *IEEE Transactions on Software Engineering*, vol. 23, no. 7, pp. 437-444, 1997.

[8] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, "The Combinatorial Design Approach to Automatic Test Generation," *IEEE Software*, vol. 13, no. 5, pp. 83-88, 1996.

[9] A. L. Williams and R. L. Probert, "A practical strategy for testing pairwise coverage at network interfaces," in *Proc. IEEE ISSRE: Int'l Symp. Softw. Reliability Eng*. White Plains, NY, 1996.

[10] R. D. Craig and S. P. Jaskiel, *Systematic Software Testing*. Boston: Artech House, 2002.

[11] L. Copeland, *A Practitioner's Guide to Software Test Design*. Boston, MA: Artech House Publishers, 2003.

[12] C. Kaner, J. Bach, and B. Pettichord, *Lessons Learned in Software Testing: A Context Driven Approach*. New York: John Wiley & Sons, Inc., 2002.

[13] S. Splaine and S. P. Jaskiel, *The Web Testing Handbook*. Orange Park, FL: STQE Publishing, 2001.

[14] J. D. McGregor and D. A. Sykes, *A Practical Guide to Testing Object-Oriented Software*. Boston, MA: Addison-Wesley, 2001.

[15] S. Dalal and C. L. Mallows, "Factor-Covering Designs for Testing Software," *Technometrics*, vol. 50, no. 3, pp. 234-243, 1998.

[16] D. M. Cohen, S. R. Dalal, A. Kajla, and G. C. Patton, "The Automatic Efficient Test Generator (AETG) System," in *Proc. of the 5th Int'l Symposium on Software Reliability Engineering*: IEEE Computer Society Press, 1994, pp. 303-309.

[17] Y. Lei and K. C. Tai, "In-Parameter-Order: A Test Generation Strategy for Pairwise Testing," in *Proc. of the 3rd IEEE High-Assurance Systems Engineering Symposium*, 1998, pp. 254-261.

[18] R. Brownlie, J. Prowse, and M. Phadke, "Robust Testing of AT&T PMX/StarMail Using OATS," *AT&T Technical Journal*, vol. 71, no. 3, pp. 41-47, 1992.

[19] S. Dalal, A. Jain, N. Karunanithi, J. Leaton, and C. Lott, "Model-Based Testing of a Highly Programmable System," in *Proceedings of the Nineth International Symposium on Software Reliability Engineering (ISSRE 98)*. Paderborn, Germany, 1998, pp. 174-178.

[20] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. Lott, G. C. Patton, and B. M. Horowitz, "Model-Based Testing in Practice," in *Proceedings of the International Conference on Software Engineering*. Los Angeles, CA, 1999, pp. 285-294.

[21] K. Burroughs, A. Jain, and R. L. Erickson, "Improved Quality of Protocol Testing Through Techniques of Experimental Design," in *Proc. Supercomm./IEEE International Conference on Communications*, 1994, pp. 745-752.

[22] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino, "Applying Design of Experiments to Software Testing," in *Proceedings of the Nineteenth International Conference on Software Engineering*. Boston, MA, 1997, pp. 205-215.

[23] L. J. White, "Regression Testing of GUI Event Interactions," in *Proceedings of the International Conference on Software Maintenance*. Washington, 1996, pp. 350-358.

[24] H. Yin, Z. Lebne-Dengel, and Y. K. Malaiya, "Automatic Test Generation using Checkpoint Encoding and Antirandom Testing," in *Proceedings of the Eighth International Symposium on Software Reliability Engineering (ISSRE '97)*. Albuquerque, NM, 1997, pp. 84-95.

[25] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of Test Set Size and Block Coverage on Fault Detection Effectiveness," in *Proceedings of the Fifth IEEE International Symposium on Software Reliability Engineering*. Monterey, CA, 1994, pp. 230-238.

[26] C. Jones, *Sofware Assessments, Benchmarks, and Best Practices*. Boston, MA: Addison Wesley Longman, 2000.

[27] D. R. Wallace and D. R. Kuhn, "Failure Modes in Medical Device Software: An Analysis of 15 Years of Recall Data," *Int'l Jour. of Reliability, Quality and Safety Engineering*, vol. 8, no. 4, pp. 351-371, 2001.

[28] B. Smith, M. S. Feather, and N. Muscettola, "Challenges and Methods in Testing the Remote Agent Planner," in *Proc. 5th Int'l Conf. on Artificial Intelligence Planning and Scheduling (AIPS 2000)*, 2000, pp. 254-263.

[29] P. J. Schroeder, P. Bolaki, and V. Gopu, "Comparing the Fault Detection Effectiveness of N-way and Random Test Suites," in *Proc. of the Int'l Symposium on Empirical Software Engineering*. Redondo Beach, CA, 2004.

[30] D. Hamlet and R. Taylor, "Partition Testing Does Not Inspire Confidence," *IEEE Transactions on Software Engineering*, vol. 19, no. 3, pp. 202-213, 1990.

[31] P. J. Schroeder, B. Korel, and P. Faherty, "Generating Expected Results for Automated Black-Box Testing," in *Proc. of the Int'l Conf. on Automated Software Engineering (ASE2002)*. Edinburgh, UK, 2002, pp. 139-48.