December 2014

testing experience

The Magazine for Professional Testers

The Three Pillars of Agile Quality and Testing
by Robert Galen

Testing the Internet of Things – The Future is Here
by Venkat Ramesh Atigadda

... and many more

printed in Germany
print version 8,00 €
free digital version
www.testingexperience.com
ISSN 1866-5705

By Danilo Berta

Combinatorial Testing Tools

When a software application accepts several inputs, each of which can assume different values, it is impossible – in general – to test all combinations of values of input variables, simply because they are too many. Let's take an example – consider a software feature that accepts as inputs three possible values A, B, and C. These values can be chosen arbitrarily from the following table:

A	B	C
A1	B1	C1
A2	B2	C2
A3	B3	
A4		
# Values		
4	3	2

Table 1. Variables and Values

The total number of possible combinations of variables (A, B, C) is equal to 4•3•2=24; in practice, in order to ensure trying all possible combinations of the values of the variables (A, B, C) at least once, 24 test cases must be carried out. Such combinations are the following:

1–4	5–8	9–12	13–16	17–20	21–24
A1;B1;C1	A1;B3;C1	A2;B2;C1	A3;B1;C1	A3;B3;C1	A4;B2;C1
A1;B1;C2	A1;B3;C2	A2;B2;C2	A3;B1;C2	A3;B3;C2	A4;B2;C2
A1;B2;C1	A2;B1;C1	A2;B3;C1	A3;B2;C1	A4;B1;C1	A4;B3;C1
A1;B2;C2	A2;B1;C2	A2;B3;C2	A3;B2;C2	A4;B1;C2	A4;B3;C2

Table 2. Combinations of Variables for A, B, and C Values

Now, in this particular case, such a number of tests can still be affordable. However, if we consider the general case of N variables $X_1, X_2, ..., X_k$, the first accepting n_1 possible values, the second n_2 possible values, the k-th that assumes n_k possible values, the total number of combinations is equal to: $n_1 \cdot n_2 \cdot ... \cdot n_k$. Such a value, even for low values of n_1, n_2, ..., n_k is a high number. For example, if k = 5 and ($n_1=3$; $n_2=4$; $n_3=2$; $n_4=2$; $n_5=3$) we get a number of combinations equal to 3•4•2•2•3=144. That is quite a large number of tests to perform if you want to ensure complete coverage of all combinations.

In real software applications, the number of values that can assume ni variables is high and it is easy to reach the hundreds of thousands of combinations, making it impossible to perform comprehensive tests on all combinations.

How can we carry out an effective test when the number of variables and values is so high as to make it impossible to exhaustively test all combinations? What reduction techniques apply?

1-wise testing

When the number of combinations is high, it is possible at least verify that – at least once – each individual value of the variables is given as input to the program to be tested. In other words, if the variable A can take the values A1, A2, A3, at least a first test must be executed in which the variable A = A1, a second test in which A = A2, and a third test in which the variable A = A3; the same goes for the other variables. This type of test provides a so-called wise-1 cover, and we will see shortly the meaning. In practice, we have the following table:

# TEST	A	B	C
1	A1	*	*
2	A2	*	*
3	A3	*	*
4	A4	*	*
5	*	B1	*
6	*	B2	*
7	*	B3	*
8	*	*	C1
9	*	*	C2

Table 3. Max Wise-1 Test Set

A possible first reduction is to set a value for the first variable and assign a random (but permitted) value to the other variables (stated with * in Table 3) and proceed in this way for all the variables and values. In this way, we reduce the test cases from 24 to just 9. It is still possible to further reduce the number of test cases, considering that instead of * you can put a value of the variable which can then be excluded from the subsequent test cases.

Put into practice, for test case #1 in place of B = * put B = B1, instead of C = * put C = C1 and remove test case #5 and test case #8, which are now both covered by test case #1.

Test case #2: in place of B = * put B = B2, and in place of C = * put C = C2, and erase test cases #6 and #9, both of which are now covered by test case #2.

Test case #3: instead of B = * put B = B3, and in place of C = * insert any value C1 or C2, considering that the values of the variable C equal to C1 and C2 have already been covered by test cases #1 and #2; we can let C = * and postpone the choice of whether to enter C1 or C2. Now, remove test case #7, since B = B3 is now covered by test case #3.

Having understood the mechanism, there remains only test case #4, which covers A = A4; we can let B = * and C = *, postponing the choice of what to actually select when we will really perform the test.

The symbol * represents "don't care"; we can put any value in it and the coverage of the test set does not change, and the values of all variables will be used at least once. Those with "*" value should be covered more than once.

The final minimized test set for wise-1 coverage is the following:

# TEST	A	B	C
1	A1	B1	C1
2	A2	B2	C2
3	A3	B3	*
4	A4	*	*

Table 4. Minimized Wise-1 Test Set

Table 4 is drawn from Table 3, moving up the columns of the variables B and C to fill the values with * with specific values; the * values "stagnate" in the lines that cannot be covered from specific values (row 3 variable C and row 4 variable B), because the number of values of the variables are different (for example, variable B has just 3 values, while variable A has 4 values; the missing B value – with respect to A – is replaced by *).

Therefore saying that a test set, such as that reported in Table 4, provides wise-1 coverage is equivalent to saying that each individual value of each variable is covered at least once.

The general wise-1 rule we can deduce is the following:

> "N variables $X_1, X_2, ..., X_k$, the first assuming n_1 possible values, the second n_2 possible values, the k-th n_k possible values, the maximum number of tests that provide coverage wise-1 is equal to $n_1+n_2+...+n_k$, while the minimum number of tests is equal to the maximum value among $\{ n_1, n_2, ..., n_k \}$."

In real cases, what is of interest is always the test set with the minimum number of test cases that ensures the chosen coverage (and this is for obvious reasons).

2-wise testing or pairwise testing

If the wise-1 test guarantees coverage of every single value for each variable, it is easy to see that a wise-2 test set ensures that all pairs of values of the variables are covered at least once. In the case of the variables listed in Table 1, *all pairs of variables* are as follows: {(A, B), (A, C), (B, C)}. In fact, the combinatorial calculation shows that the number of combinations of N values taken K to K (with $N \geq K$) is equal to:

$$\binom{N}{K} = \frac{N!}{K! \cdot (N-K)!}$$

In our example we have three variables (N = 3) taken two to two (K = 2), and applying the combinatorial formula above we get $\binom{3}{2} = \frac{3!}{2! \cdot (3-2)!} = 3$; the three pairs are {(A, B), (A, C), (B, C)}.

Wanting to compute all possible pairs *of variable values*, we need to consider the following table:

PAIRS	# VARIABLES VALUES			TOTAL
	A	B	C	
(A,B)	4	3		4•3=12
(A,C)	4		2	4•2=8
(B,C)		3	2	3•2=6
GRAND TOTAL				12+8+6=26

Table 5. Counting the Pairs of Values of the Variables A, B, and C

Hence, the total of all the pairs of values of the variables A, B, and C whose values are reported in Table 1 is equal to 26 and they are all printed in the following table:

#	# of pairs of values		
	A, B	A, C	B, C
1	A1,B1	A1,C1	B1, C1
2	A1,B2	A1,C2	B1, C2
3	A1,B3	A2,C1	B2, C1
4	A2,B1	A2,C2	B2, C2
5	A2,B2	A3,C1	B3, C1
6	A2,B3	A3,C2	B3, C2
7	A3,B1	A4,C1	
8	A3,B2	A4,C2	
9	A3,B3		
10	A4,B1		
11	A4,B2		
12	A4,B3		
# PAIRS	12	8	6
TOTAL	12+8+6=26		

Table 6. Pairs of Values of the Variables A, B, and C

Why should you consider a test set to cover wise-2? Is it not enough to consider a test set with 1-wise coverage? Here we enter into a thorny issue, in which opinions are different, concordant, and discordant.

Below is the "incipit" from the site *www.pairwise.org* [1]:

> "Pairwise (a.k.a. all-pairs) testing is an effective test case generation technique that is based on the observation that most faults are caused by interactions of at most two factors. Pairwise-generated test suites cover all combinations of two, therefore are much smaller than exhaustive ones yet still very effective in finding defects."

We mention also the opinion of James Bach and Patrick J. Schroeder about the pairwise method: *"Pairwise Testing: A Best Practice That Is Not"* from James Bach, Patrick J. Schroeder available from *http://www.testingeducation.org/wtst5/PairwisePNSQC2004.pdf* [2]:

> "What do we know about the defect removal efficiency of pairwise testing? Not a great deal. Jones states that in the US, 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 did not execute pairwise testing, they analyzed failure data and speculated 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 % US 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 whether pairwise testing is appropriate; it is also an important step in determining what additional testing technique should be used in a specific testing situation."

[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.

[26] Jones, Software 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

To clarify, we can say that the pairwise or 2-wise test method ensures that all combinations of pairs of values of the variables are tested, "theoretically ensuring" the maximization of the anomalies found, with percentages ranging from 50 % to 98 % according to the studies. In fact, no test can ever guarantee a defined percentage removal of defects (which can *only* be calculated *ex post* for the *specific project*). Let's say – trying to be realistic – that pairwise achieves a valid agree-

testing experience

GET YOUR PRINTED COPY

Order all issues in our shop!

www.testingexperience-shop.com

ment between the number of tests to be performed and the anomalies detected, when the number of variables and their values is so high that it is not possible to test all the combinations (the so called all-wise testing or N-wise testing, where N is the number of variables we are playing with).

In the case of a test set covering wise-2 level, it is very simple to know the *maximum number* of tests that provides coverage of all pairs of values of the variables. This value is equal to the number of pairs of values of the variables themselves. In our example of three variables A, B and C in Table 1, this number is 26 (calculated as described in Table 6). The real problem – still unsolved – is to discover the *minimum number* of tests that guarantees wise-2 coverage, although there are a variety of methods and algorithms that approximate this value for a problem with an arbitrary number of variables and values. Examples of tools that use these algorithms are:

1. Microsoft Pairwise Independent Combinatorial Testing tool (PICT), downloadable from *http://download.microsoft.com/download/f/5/5/f55484df-8494-48fa-8dbd-8c6f76cc014b/pict33.msi*
2. NIST Tools: *http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html*
3. James Bach AllPairs downloadable from *http://www.satisfice.com/tools.shtml*
4. Other tools here: *http://www.pairwise.org/tools.asp*

n-wise Testing with n > 2

It is now easy to extend the concept of pairwise or 2-wise testing to the generic case of n-wise, with n > 2. The generic test set provides n-wise coverage if it is able to cover all the n-tuples (3-tuples if n = 3, 4-tuples if n = 4 and so on). As in the pairwise, is always possible to know the size of the maximum test set, equal to the number n-tuples of values of the variables, but there is no way to know – in the general case – the size of the minimum test set that guarantees coverage n-wise.

Using NIST Tools (ACTS) or Microsoft PICT (or other similar tools), it is possible to extract a test set that approximates as closely as possible the minimum test set. It is clear that, given a set of N variables, the maximum level of wise which you can have is equal to the number of variables. So, if we have four variables, a 4-wise test set coincides with all possible combinations, while a 5-wise test set or higher makes no sense.

The combinatorial testing techniques that we discussed in the first part of the article are intended to solve a basic problem, which we have already discussed and we rephrase as follows:

Direct Combinatorial Test Problem: *"In a software system accepting N input variables, each of which can take on different values, find the test set with the smallest number of test cases that guarantees (at least) coverage of all combinations (2-tuples) of all the values of the variables involved."*

The Pairwise technique and a large number of support tools have been developed to solve this problem. Once such a test set (the smallest possible) has been generated, you should run the test cases and detect (if present) all the defects that arise in the software under test.

There is also a second problem, maybe less "popular" compared to the previous, as follows:

Reverse Combinatorial Test Problem: *"Given a Test Set for which you do not know the method of generation (if any), calculate what percentage of nwise coverage the test set ensures, with nwise between 1 and the number variables in the test set".*

An example: tests generated by automatic tools for which you have low or almost zero process control, or when the "test cases" are generated by the automatic flows that feed interfaces between different systems (think of a system that transmits accounting data from a system A to B); test data are – in general – excerpts from historical series over which you have no control.

For test scenarios in some way related to a combinatorial inverse problem, it is not easy to find support tools as such tools are not readily available. The only tool I found is NIST CCM, in alpha-phase at the time I am writing this article. If you like, you can request a copy of the software: go to *http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html* as previously reported [3].

In the following we describe a set of tools called *"Combinatorial Testing Tools"* (executable under Windows, but, if needed, not difficult to port under Unix/Linux) that enables the (quasi)minimal test set to be extracted and the coverage of a generic test set to be calculated, using systematic algorithms calculation coverage and starting from all n-tuples of the variables' values. Such algorithms should be categorized as "brute force algorithms" and should be used (on a normal supermarket-bought PC) if the number of variables and values is not too high.

Overview of CTT

Tools in the Combinatorial Testing Tools product try to provide support to help solve both problems of combinatorial testing previously stated.

> Combinatorial Testing Tools *do not intend to compete* with the existing tools aimed at solving the direct problem of combinatorial testing, such as Microsoft PICT, AllPair J. Bach, NIST, or several other commercial and non-commercial tools already present on the market. These tools implement algorithms definitely more effectively than CTT and *therefore should be favorite* – I repeat – to solve the direct problem.

Regarding the reverse problem of combinatorial tests, to my knowledge there are no tools on the market (except NIST CCM in alpha release) and the CTT then attempt to provide a *very first* solution to the reverse problem, to be surely improved over time, when we will better understand the logic and the rules that are behind combinatorial testing and the minimal determination of test sets related to it.

We would like to remember that:

a. Solving the direct problem means determining the smallest possible test set with a level of WISE coverage agreed (usually WISE = 2) from the set of variables and values.

b. Solving the reverse problem means determining the level of coverage of a given test set with respect to a reference WISE level (also here usually WISE = 2)

The tools should be categorized as follows:

a. **First level tools:** batch DOS scripts providing an immediate response to standard scenarios that typically occur in test projects requiring combinatorial techniques.

b. **Second level tools:** executable (C++/Perl development language) and more versatile, giving response to problems that may be less common, but sometimes occur in test projects requiring combinatorial techniques.

First level scripts were thought of as the "wrapper" around the second level executables, in order to "simplify end user life" with a set of simple commands that enable you to quickly get a number of items of "standard information". The following table maps the two categories of tools and the tool with the kind of information it supplies.

In this article we cannot go in details on the behavior of the tools; what follows is a short description of all of the first level tools. More information can be found in the tool's user manual which is downloadable (with the scripts) from: *http://www.opensource-combinatorial-soft-test-tools.net/index.php/download*

Anyway, here is a useful summary table that links together first level and second level tools.

First Level Tools	Second Level												
	1	2	3	4	5	6	7	8	9	10	11	12	13
runW			×	×	×	×	×			×			
runCC	×			×	×								
runsCC		×		×	×								
runT				×	×			×					
runsT				×	×				×				
runTS				×	×			×					
runsTS				×	×				×				
runTSF	×			×	×			×					
runsTSF	×			×	×				×				
runC												×	
runR				×	×						×		

Table 7. Mapping of First Level vs. Second Level Tools (Wrapping Map)

Below is the list of second level executables:

1. calcolaCopertura.exe
2. calcolaCoperturaSlow.exe
3. Combinazioni_n_k.exe
4. contrLimWise.exe
5. count_line.exe
6. creaFileProdCart.exe
7. creaStringa.exe
8. generaTestSet.exe
9. generaTestSetSlow.exe
10. ProdCart.exe
11. reduceNple.exe
12. runConstrains.pl
13. uniqueRowFile.exe

First Level Tools – Batch DOS Script

Tool runW

This is the first tool that is mandatory to run before all the other tools. It generates all the n-tuples corresponding to the value of the Wise past input (runW = run **W**ise)

Tool runCC and runsCC

Computes the input test set coverage in respect of the WISE passed as input (runCC = run **C**irculate **C**overage)

Tools runT and runsT

Get the minimal test set with guaranteed coverage equal to the Wise passed in input (runT = run **T**est), extracting the test cases from the file of the WISE_MAX-tuples (all combinations).

Tools runTS and runsTS

Get the minimal test set with guaranteed coverage equal to the Wise passed in input or equal to the coverage of the test set passed in input if less than WISE, extracting the test cases from the file of the test set passed in input (runTS = run **T**est **S**et)

Tools runTSF and runsTSF

Get the minimal test set with guaranteed coverage equal to the Wise passed in input or equal to the coverage of the test set passed in input if less than WISE, extracting the test cases from the file of the test set passed in input, excluding n-tuples already covered by the partial test set input file (runTSF = run **T**est **S**et **F**orbidden).

Tool runR

Extracts a *non-minimal* test set but still smaller than the maximum test set with guaranteed coverage equal to the Wise passed as input (runR = run **R**educe).

Tool runC

Applies constraints to n-tuple file (or test set file) passed as input.

Second Level Tools – Executable

In the following we describes the second level tools, a little more hard to use but more versatile; may be useful to experienced users for managing more complex scenarios.

Executable calcolacopertura.exe and calcolaCoperturaSlow.exe

Performs the coverage calculation of the input test set in respect of the input WISE.

Executable Combinazioni_n_k

Extracts all K by K combinations of a string of length N passed as input.

Executable generaTestSet.exe and generaTestSetSlow.exe

Gets the minimal test set with guaranteed coverage equal to the Wise passed in input or equal to the coverage of the test set passed in input if less than WISE, extracting the test cases from the file of the test set passed in input, excluding n-tuples already covered by the partial test set input file.

Executable ProdCart.exe

Generates all possible combinations of the values of variables as defined in the input file.

Executable reduceNple.exe

"Squeezes" as many n-tuples as possible contained in the input file, replacing the values "*" with specific values of the variables and thus creating a test set from the file of n-tuples. While not the test set minimum, it is reduced compared to the test set maximum (coincident with all n-tuples). The number of records depends on the sorting of the n-tuples input file, in an unknown way. There is definitely a sorting of the files row to which the test set output contains a minimum number of test cases with guaranteed WISE-coverage, but finding this sort is not feasible from a computational point of view, as it is too onerous.

There are six other executables that do not provide direct support to the generation and/or operation of the test ssets, but are predominantly used by DOS batch tools to perform secondary operations that it is impossible or – at least – very complex to do directly from DOS. These utilities may also be of some use, even if they are not to be considered "tout cours" test tools. We have not described those utilities in this article.

Conclusion

In the article we gave an overview of a test methodology that uses combinatorial calculus to find test cases for a software component, knowing the inputs of the same. Generally speaking a combinatorial technique like this generates too many test cases, so we need to define a so-called "N-wise coverage" (with N from 1 to the number of input variables), select a value of N (usually N=2, pairwise testing) and extract a subsystem of test cases with the guarantee of N-wise coverage.

This is the **"Direct Combinatorial Test Problem"** and there are a lot of wonderful tools on the market that solve the problem very quickly.

We then dealt with the **Reverse Combinatorial Test Problem**: if you have a test set build upon the N inputs of a software component about which you know nothing, what percentage of N-wise coverage does the test set ensure? On the market I just found one tool that addresses this problem: NIST CCM, which is still in alpha phase at the time I am writing this article. In the article I give an overview of the CTT (Combinatorial Testing Tools) I developed in C++ that, using a "brute-force" approach, try to give a very first response to the "Reverse Combinatorial Test Problem".

But there is (at least) a problem whose solution is not still known. For a software with N inputs, what is the minimum test set (if it exists) that guarantees the N-level coverage? The solution exists just for a trivial case: for 1-wise coverage is always equal to the number of values of the variable with a maximum number of values, while N-wise coverage coincides with all the variable combinations of the value. And what if we also include the outputs? This could be the material for another article in the future ... ■

Notes of Appreciation

[1] Many thanks to Jacek Czerwonka, owner of the web site *www.pairwise.org*, who allowed me to reprint the "incipit" of the same. By the way, he wrote to me about some evolution on the subject of pairwise vs. random testing that you can find in the article *"An Empirical Comparison of Combinatorial and Random Testing"* available at the link: *http://www.nist.gov/customcf/get_pdf.cfm?pub_id=915439* written by: Laleh Sh. Ghandehari, Jacek Czerwonka, Yu Lei, Soheil Shafiee, Raghu Kacker, and Richard Kuhn.

[2] Many thanks to James Bach, owner of the website *www.satisfice.com* who allowed me to reprint part of the article *"Pairwise Testing: A Best Practice That Is Not"* from James Bach, Patrick J. Schroeder, available from *http://www.testingeducation.org/wtst5/PairwisePNSQC2004.pdf*.

[3] Many thanks to Dr. Richard Kuhn from NIST who kindly sent me a copy of the NIST CCM tools. We would like to remind you that you should request a copy: go to *http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html* as previously reported.

> about the author

Danilo Berta graduated in Physics at Turin University (Italy) in 1993 and first started work as a Laboratory Technician, but he soon switched to the software field as an employee of a consultancy company working for a large number of customers. Throughout his career, he has worked in banks as a Cobol developer and analyst, for the Italian treasury, the Italian railways (structural and functional software testing), as well as for an Italian automotive company, the European Space Agency (creation and testing of scientific data files for the SMART mission), telecommunication companies, the national Italian television company, Italian editorial companies, and more. This involved work on different kinds of automation test projects (HP Tools), software analysis, and development projects. With his passion for software development and analysis – which is not just limited to testing – he has written some articles and a software course for Italian specialist magazines. His work enables him to deal mainly with software testing; he holds both the ISEB/ISTQB Foundation Certificate in Software Testing and the ISEB Practitioner Certificate; he is a regular member of the British Computer Society. Currently, he lives in Switzerland and works for a Swiss company.
Website: *www.bertadanilo.name*