Pairwise Testing in the Real World: Practical Extensions to Test-Case Scenarios

Jacek Czerwonka

Microsoft Corporation

February 2008

Summary: Pairwise testing has become an indispensable tool in a software tester's toolbox. This article pays special attention to usability of the pairwise-testing technique. In particular, it focuses on ways in which the pure pairwise-testing approach must be modified to become practically applicable, and on the features that tools must offer to support the tester who is trying to use pairwise testing in practice. (15 printed pages)

Contents

Overview

Introduction

Advanced Features

Future Work

Conclusion

Acknowledgments

References

Overview

Pairwise testing has become an indispensable tool in a software tester's toolbox. The technique has been known for almost 20 years [22], but it is only in the last five years that we have seen a tremendous increase in its popularity.

So far, information on at least 20 tools that can generate pairwise test cases has been published [1]. Most tools, however, lack practical features that are necessary for them to be used in the industry.

This article pays special attention to usability of the pairwise-testing technique. In particular, it does not describe any radically new method of efficient generation of pairwise test suites—a topic that has already been researched extensively—neither does it refer to any specific case studies or results that have been obtained through this method of test-case generation. It does focus on ways in which the pure pairwise-testing approach

must be modified to become practically applicable, and on the features that tools must offer to support the tester who is trying to use pairwise testing in practice.

This article makes frequent references to PICT, an existing and publicly available tool that is built on top of a flexible combinatorial test-case–generation engine, which implements several of the concepts that are described herein.

A set of possible inputs for any nontrivial piece of software is too large to be tested exhaustively. Techniques such as equivalence partitioning and boundary-value analysis [17] help convert even a large number of test levels into a much smaller set with comparable defect-detection power. Still, if software under test (SUT) can be influenced by a number of such factors, exhaustive testing again becomes impractical.

Over the years, a number of combinatorial strategies have been devised to help testers choose subsets of input combinations that would maximize the probability of detecting defects: random testing [16]; each-choice and base choice [2]; antirandom [15]; and, finally, *t*-wise testing strategies, with pairwise testing being the most prominent among these. (See Figure 1.)

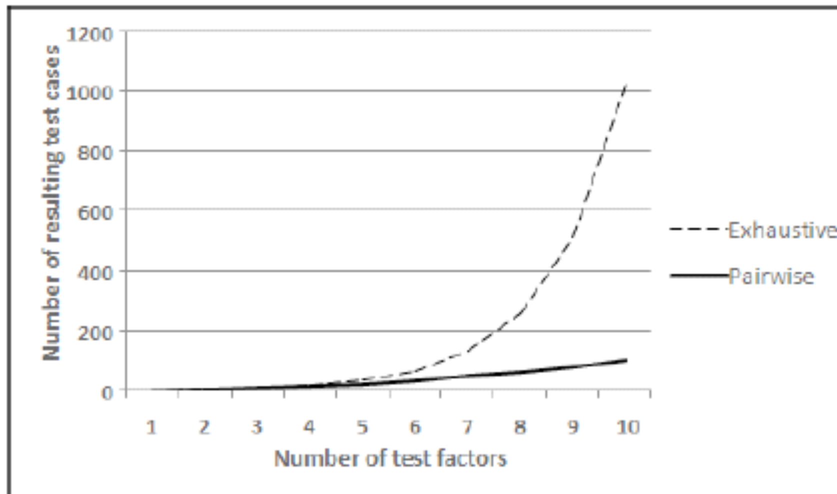


Figure 1. Increase in number of exhaustive and pairwise tests with number of test levels

Pairwise-testing strategy is defined as follows:

Given a set of N independent test factors—f_1, f_2, ..., f_N—with each factor f_i having L_i possible levels—$f_i = \{l_{i,1}, ..., l_{i,L_i}\}$—a set of tests R is produced. Each test in R contains N test levels, one for each test-factor f_i; and, collectively, all tests in R cover all possible pairs of test-factor levels (belonging to different parameters); in other words, for each pair of factor levels $l_{i,p}$ and $l_{j,q}$—where $1 \le p \le L_i$, $1 \le q \le L_j$, and $i \ne j$—there exists at least one test in R that contains both $l_{i,p}$ and $l_{j,q}$.

This concept can easily be extended from covering all pairs to covering any t-wise combinations in which $1 \le t \le N$. When $t = 1$, the strategy is equivalent to *each-choice*; if $t = N$, the resulting test suite is exhaustive. Covering all pairs of tested factor levels has

been extensively studied. Mandl described using orthogonal arrays in the testing of a compiler [16]. Tatsumi, in his paper on the Test Case Design Support System (used in Fujitsu Ltd [22]), talks about two standards for creating test arrays: one with all combinations covered exactly the same number of times (orthogonal arrays), and one with all combinations covered at least once. When making that crucial distinction, he references an earlier paper by Shimokawa and Satoh [19].

Over the years, pairwise testing was shown to be an efficient and effective strategy of choosing tests [4, 5, 6, 10, 13, 23]. However, as shown by Smith et al. [20] and, later, by Bach and Shroeder [3], pairwise—like any technique—must be used appropriately and with caution.

Because the problem of finding a minimal array that covers all pairwise combinations of a given set of test factors is NPcomplete [14], a considerable amount of research has understandably gone into efficient creation of such arrays. Several strategies were proposed in an attempt to minimize the number of tests that were produced [11].

Authors of these combinatorial test-case–generation strategies often describe additional considerations that must be taken into account before their solutions become practical. In many cases, they propose methods of handling these in the context of their generation strategies. Tatsumi [22] mentions constraints as a way of specifying unwanted combinations (or, more generally, dependencies among test factors). Sherwood [18] explores adapting conventional *t*-wise strategy to invalid testing and the problem of preventing input masking. Cohen et al. [6] describe seeds that allow specifying combinations that must appear in the output and covering combinations with mixed-strength arrays, as a way of putting more emphasis on interactions of certain test factors.

This article describes PICT, a test-case–generation tool that has been in use at Microsoft Corporation since 2000, which implements both the *t*-wise–testing strategy and features that make the strategy feasible in the practice of software testing.

Introduction

Models

PICT was designed with three principles in mind: (1) speed of test generation, (2) ease of use, and (3) extensibility of the core engine. Although the ability to create the smallest possible covering array was given less emphasis, the efficiency of PICT's core algorithm is comparable with other known test-generation strategies. (See Figure 2.)

Task	AETG [14]	PairTest [21]	TConfig [24]	CTS [12]	Jenny [1]	DDA [8]	AllPairs [1]	PICT
3^4	9	9	9	9	11	?	9	9
3^{13}	15	17	15	15	18	18	17	18
$4^{15}3^{17}2^{29}$	41	34	40	39	38	35	34	37
$4^1 3^{39} 2^{35}$	28	26	30	29	28	27	26	27
2^{100}	10	15	14	10	16	15	14	15
10^{20}	180	212	231	210	193	201	197	210

Figure 2. Comparison of PICT's generation efficiency with other known tools

Input to PICT is a plain-text file (model) in which a tester specifies test factors (referred to as test parameters, later in this article) and test-factor levels (referred to as values of a

parameter). Figure 3 shows an example of a simple model that is used to produce test cases for volume creation and formatting.

```
Type:            Single, Spanned, Striped, Mirror, RAID-5
Size:            10, 100, 1000, 10000, 40000
Format method:   Quick, Slow
File system:     FAT, FAT32, NTFS
Cluster size:    512, 1024, 2048, 4096, 8192, 16384
Compression:     On, Off
```

Figure 3. Parameters for volume creation and formatting

By default, the tool produces a pairwise test array (t = 2). It is possible, however, to specify a different order of combinations. In fact, any t is allowed only if $1 \le t \le N$.

Test-Case-Generation Engine

The test-generation process in PICT comprises two main phases: preparation and generation.

In the preparation phase, PICT computes all of the information that is necessary for the generation phase. This includes the set P of all parameter interactions to be covered. Each combination of values to be covered is reflected in a parameter-interaction structure.

For example, given three parameters A, B (two values each), and C (three values), and pair-wise generation, three parameter-interaction structures are set up: AB, AC, and BC. Each of these has a number of slots that correspond to possible value combinations that participate in a particular parameter interaction—four slots for AB, and six slots each for AC and BC. (See Figure 4.)

```
                                AB   AC   BC
                                --------------
                                00   00   00
A: 0, 1                         01   01   01
B: 0, 1        translates to    10   02   02
C: 0, 1, 2                      11   10   10
                                     11   11
                                     12   12
```

Figure 4. Parameter-interaction structures

Each slot can be marked uncovered, covered, or excluded. All of the uncovered slots in all parameter interactions constitute the set of combinations to be covered. If any constraints were defined in a model, they are converted into a set of exclusions—value combinations that must not appear in the final output. Corresponding slots are then marked excluded in parameter-interaction structures and, therefore, removed from the combinations to be covered. The slot becomes covered when the algorithm produces a test case that satisfies that particular combination. The algorithm terminates when there are no uncovered slots.

The core generation algorithm is a greedy heuristic. (See Figure 5.) It builds one test case at a time, locally optimizing the solution. It is similar to the algorithm that is used in AETG

[6], with the key differences being that the PICT algorithm is deterministic and that it does not produce candidate tests.

Note PICT does make pseudo-random choices; however, unless a user specifies otherwise, the pseudo-random generator is always initialized with the same seed value. Therefore, two executions on the same input produce the same output.

```
# Assume test cases r1,...,ri-1 are already produced
# Slots in P reflecting combinations selected by r1,...,ri-1 are set to covered

If there are any unused seed combinations not violating any exclusions
    Add a seed combination to ri
    Mark all slots in P covered by the seed combination as covered

While there are parameters with no values in ri

    If ri is empty
        Choose a parameter interaction p from P with most uncovered slots
        Pick the first uncovered combination from p
    Else
        # Assume values l1,...,lk-1 have already been chosen and added to ri

        Look at subset Q of P that covers at least one parameter with no
         representation in l1,...,lk-1

        Look at slots in Q which values are consistent with already chosen values in l1,...,lk-1

        If there exist uncovered combinations
            Pick a slot with values which when added to ri would cover the most uncovered
             combinations with l1,...,lk-1 and the resulting partial test case ri would not
             contain an excluded combination
        Else
            Pick randomly a covered combination which when added to l1,...,lk-1 would not
             contain an excluded combination

    Add values of this combination to ri
    Mark the chosen combination in P as covered
```

Figure 5. PICT heuristic algorithm

The generation algorithm does not assume anything about the combinations to be covered. It operates on a list of combinations that is produced in the preparation phase.

This flexibility of the generation algorithm allows for adding interesting new features easily. The algorithm is also quite effective. It can compute test suites that are comparable in size to other tools that exist in the field, and it is fast enough for all practical purposes. (For instance, for 50 parameters with 20 values each (2050), PICT generates a pairwise test suite in under 20 seconds on an Intel Pentium M 1.8GHz computer that is running Microsoft Windows XP SP2.)

Advanced Features

Mixed-Strength Generation

Most commonly, when *t*-wise testing is discussed, it is assumed that all parameter interactions have a fixed-order *t*. In other words, if t = 3 is requested, all triplets of parameter values will be covered. It is sometimes useful, however, to be able to define different orders of combinations for different subsets of parameters. For example, interactions of parameters B, C, and D might require better coverage than interactions of A or E. (See Figure 6.) We should be able to generate all possible triplets of B, C, and D, and cover all pairs of all other parameter interactions.

The importance of this feature stems from the fact that certain parameter interactions often seem to be more "sensitive" than others. Possibly, experience has shown that interactions of these parameters are at the root of proportionally more defects than other interactions; therefore, they should be tested more thoroughly. On the other hand, setting a higher *t* on the entire set of test parameters could produce too many test cases. Using mixed-strength generation might be a way to achieve higher coverage where necessary, without incurring the penalty of having too many test cases.

```
A: 0, 1
B: 0, 1
C: 0, 1
D: 0, 1
E: 0, 1
```

translates to

AB	AC	AD	AE	BC	BD	BE	CD	CE	DE
00	00	00	00	00	00	00	00	00	00
01	01	01	01	01	01	01	01	01	01
10	10	10	10	10	10	10	10	10	10
11	11	11	11	11	11	11	11	11	11

```
A:      0, 1
B @ 3:  0, 1
C @ 3:  0, 1
D @ 3:  0, 1
E:      0, 1
```

translates to

AB	AC	AD	AE	BCD	BE	CE	DE
00	00	00	00	000	00	00	00
01	01	01	01	001	01	01	01
10	10	10	10	010	10	10	10
11	11	11	11	011	11	11	11
				100			
				101			
				110			
				111			

Figure 6. Fixed-strength and mixed-strength generation

Cohen et al. describe the concept of subrelations as a way of getting an output with varying levels of interactions between parameters [6]. AETG actually uses seeding to achieve this. In PICT, because the generation phase operates solely on parameter-interaction structures, they can be manipulated to reflect the need for higher-order interactions of certain parameters.

Creating a Parameter Hierarchy

To complement the mixed-strength generation, PICT allows a user to create a hierarchy of test parameters. This scheme allows for certain parameters to be *t*-wise–combined first, and then that product is used for creating combinations with parameters on upper levels of the hierarchy. This is a useful technique that can be used to (1) model test domains with a clear hierarchy of test parameters—that is, API functions taking structures as arguments and user-interface (UI) windows with additional dialog boxes—or (2) to limit the combinatorial explosion of certain parameter interactions; (1) is intuitive, and (2) requires explanation.

When describing the process of analyzing test parameters [22], Tatsumi distinguishes between *input* parameters, which are direct inputs to the SUT, and *environmental* parameters, which constitute the environment in which the SUT operates. Typically, input parameters can be controlled and set much easier than environmental ones (compare supplying an API function with different values for its arguments with calling the same function on different operating systems). Because of that, it is sometimes better to constrain the number of environments to the absolute minimum.

Consider the example that is shown in Figure 7, which contains the same test parameters

as Figure 3, but with hardware specification added. To cover all pairwise combinations of all nine parameters, PICT generated 31 test cases, which included 17 different combinations of the hardware-related parameters: platform, CPUs, and RAM.

Test domain consisting of 'input' and 'environment' parameters:

```
# Input parameters

Type:           Single, Spanned, Striped, Mirror, RAID-5
Size:           10, 100, 1000, 10000, 40000
Format method:  Quick, Slow
File system:    FAT, FAT32, NTFS
Cluster size:   512, 1024, 2048, 4096, 8192, 16384
Compression:    On, Off

# Environment parameters

Platform:  x86, x64, ia64
CPUs:      1, 2
RAM:       1GB, 4GB, 64GB

# Environment parameters will form a sub-model

{ PLATFORM, CPUS, RAM } @ 2
```

Hierarchy of test parameters:

```
- Type
- Size
- Format method
- File system
- Cluster size
- Compression
- <CompoundParameter> (t=2)
   - Platform
   - CPUs
   - RAM
```

Figure 7. Two-level hierarchy of test parameters

Instead, hardware parameters can be designated as a *sub-model* and pairwise-combined first. The result of this generation is then used to create the final output, in which six individual input parameters and one compound environment parameter take part. The result is a larger test suite (54 tests), but it contains only nine unique combinations of the hardware parameters. Users of this feature have to be cautious, however, and understand that not all *t*-wise combinations of all nine parameters will be covered in this scheme.

If the goal is to achieve low volatility of a certain subset of parameters, one might implement an even better solution. Namely, generate all required *t*-wise combinations at the lower level (platform, CPUs, and RAM), and use them for the higher-level combinations (all nine parameters), with the requirement that, in any test case, a combination of platform, CPUs, and RAM must come from the result of the lower-level generation. In this case, one would achieve low volatility and not lose *t*-wise coverage. This feature has yet to be implemented in PICT.

Excluding Unwanted Combinations

The definition of pairwise testing that is given in the “Overview” section talks about test factors (parameters) being independent. However, in practice, this is rarely the case. That is why constraints are an indispensable feature of a test-case generator. They describe *limitations* of the test domain—that is, combinations that are impossible to be successfully executed in the context of given SUT. Going back to the example in Figure 3, the FAT file system cannot actually be applied to volumes that are larger than 4 gigabytes (GB). Any test case that asks for FAT and a volume that is larger than 4 GB will fail to execute properly. One might think that removing such test cases from the resulting test suite would solve the problem. However, such a test case might cover other, valid combinations (for example, [FAT, RAID5]) that are not covered elsewhere in the test suite.

Researchers recognized this problem very early. Tatsumi describes the concept of constraints and proposes special handling of those by marking test cases with excluded combinations as *errors* [22]. Later, several different ways in which constraints can be handled were proposed. The simplest methods involve asking a user to manipulate the definition of test parameters—either by splitting parameter definitions onto disjoint subsets [18] or by creating hybrid parameters [25]—so that unwanted combinations cannot possibly be chosen. Other methods could involve post-processing of resulting test suites and modifying test cases that violate one or more constraints so that the violation is avoided.

Dalal et al. describe AETGSpec, a test-domain modeling language that includes specifying constraints in the form of propositional formulas [9]. PICT uses a similar language of constraint rules. In Figure 8, three **IF-THEN** statements describe *limitations* of a particular test domain.

```
Type:           Single, Spanned, Striped, Mirror, RAID-5
Size:           10, 100, 1000, 10000, 40000
Format method:  Quick, Slow
File system:    FAT, FAT32, NTFS
Cluster size:   512, 1024, 2048, 4096, 8192, 16384
Compression:    On, Off

# There are limitations on volume size

IF [File system] = "FAT"   THEN [Size] <= 4096;
IF [File system] = "FAT32" THEN [Size] <= 32000;

# And not all file systems support compression

IF [File system] <> "NTFS" or
  ([File system] =  "NTFS" and [Cluster size] > 4096)
THEN [Compression] = "Off";
```

Figure 8. Parameters for volume creation and formatting augmented with constraints

PICT internally translates constraints into a set of combinations that are called *exclusions* and uses those to mark appropriate slots as excluded in parameter-interaction structures. This method poses two practical problems:

1. How to ensure that all combinations that must be excluded are, in fact, marked excluded
2. How to handle exclusions that are more granular than the corresponding parameter-interaction structure—that is, they refer to a larger number of parameters than there are in the parameter-interaction structure

The first problem can be resolved by calculating dependent exclusions. Consider the example that is shown in Figure 9. Constraints on that model create a circular dependency loop between values *A:0 → B:0 → C:0 → A:1* which results in a contradiction: If *A:0* is chosen, only *A:1* can be chosen. In the end, if the generation is to proceed, we must ensure that we do not pick *A:0* at all. Instead of the initial three, five combinations must be excluded—among them, all combinations of *A:0*.

Input:

```
A: 0, 1
B: 0, 1
C: 0, 1

IF [A] = 0 THEN [B] = 0;
IF [B] = 0 THEN [C] = 0;
IF [C] = 0 THEN [A] = 1;
```

Before and after calculating dependent exclusions:

```
A:0, B:1          A:0
B:0, C:1    ⇒     B:0, C:1
A:0, C:0
```

Before and after excluding dependent combinations:

AB	AC	BC
00	~~00~~	00
~~01~~	01	~~01~~
10	10	10
11	11	11

⇒

AB	AC	BC
~~00~~	~~00~~	00
~~01~~	~~01~~	~~01~~
10	10	10
11	11	11

Figure 9. Calculating dependent exclusions

The second problem is a case in which directly marking combinations as excluded in parameter-interaction structures is impossible. Consider the example that is shown in Figure 10, in which three-element exclusions are created, but parameter-interaction structures refer only to two parameters at a time. In other words, there is not a parameter-interaction structure ABC that can be used to mark the excluded combination; AB, AC, or BC is not granular enough. In such a case, one more parameter-interaction structure ABC is set up. Appropriate combinations are marked as excluded, and the rest of them are marked as covered. The generation algorithm will ensure that all possible combinations of AB, AC, and BC will be covered without actually picking the A:0, B:0, C:1 combination.

Input:

```
A: 0, 1
B: 0, 1
C: 0, 1
D: 0, 1

IF [A] = 0 AND [B] = 0 THEN [C] = 0;
```

Exclusions:

```
A:0, B:0, C:1
```

Before and after adding new parameter interaction ABC:

AB	AC	AD	BC	BD	CD
00	00	00	00	00	00
01	01	01	01	01	01
10	10	10	10	10	10
11	11	11	11	11	11

⇒

AB	AC	AD	BC	BD	CD	ABC
00	00	00	00	00	00	000✓
01	01	01	01	01	01	~~001~~
10	10	10	10	10	10	010✓
11	11	11	11	11	11	011✓
						100✓
						101✓
						110✓
						111✓

Figure 10. Handling exclusions that are more granular than parameter-interaction structures

Both steps—expanding the set of exclusions to cover all dependent exclusions and adding auxiliary parameter-interaction structures—happen in the preparation phase. After that, produced test cases are guaranteed not to violate any exclusion. Because there is no need for validity verification or post-processing of produced results, the generation phase can be very fast.

Seeding

Cohen et al. use the term *seeds* to describe test cases that must appear in the generated test suite [6]. Seeding has two practical applications. Specifically, seeding:

1. Allows explicit specification of important combinations.
2. Can be used to minimize change in the output when the test-domain description is modified and the resulting test suite is regenerated.

The first application is intuitive. If a tester is aware of combinations that are likely to be used in the field, the resulting test suite should contain them. All *t*-wise combinations that are covered by these seeds will be considered covered, and only incremental test cases will be generated and added.

The need for the second application stems from the fact that even a small modification of the test-domain description, such as adding parameters or parameter values, might cause big changes in the resulting test suite. Containing those changes can be an important cost-saving option, especially when hardware parameters are part of the test domain.

It happens often that the initial test-domain specification is incomplete; however, the test suite that it produces is a basis for the first set of configurations on which to run tests. For example, when a test case specifies a computer with two AMD64 CPUs, a SCSI hard drive, exactly 1 GB of RAM, Windows XP with Service Pack 2, and a certain version of Microsoft Internet Explorer, such a computer must be assembled, and all of the necessary software must be installed. Later, when a modification to the model of SUT is required, it might perturb the resulting test cases enough to invalidate some (if not all) of the already prepared configurations. Seeding allows for reuse of old test cases in newly generated test suites.

Note Certain precautions must be taken in cases that involve removal of parameter values, removal of entire parameters, or addition of new constraints.

In PICT, these seeding combinations can be full combinations (with values for all test parameters specified) or partial combinations. Figure 11 shows two seeding combinations: one full and one partial. The former will become the first test case in the resulting suite. The latter will initialize the second test case with values for Type, File system, and Format type. The actual values of Size, Cluster size, and Compression will be left for the tool to determine.

Test parameters:

```
Type:           Single, Spanned, Striped, Mirror, RAID-5
Size:           10, 100, 1000, 10000, 40000
Format method:  Quick, Slow
File system:    FAT, FAT32, NTFS
Cluster size:   512, 1024, 2048, 4096, 8192, 16384
Compression:    On, Off
```

Seeding file:

```
Type    Size  Format method  File system  Cluster size  Compression
Single  100   Quick          NTFS         4096          Off
Mirror        Slow           FAT32
```

Figure 11. Seeding

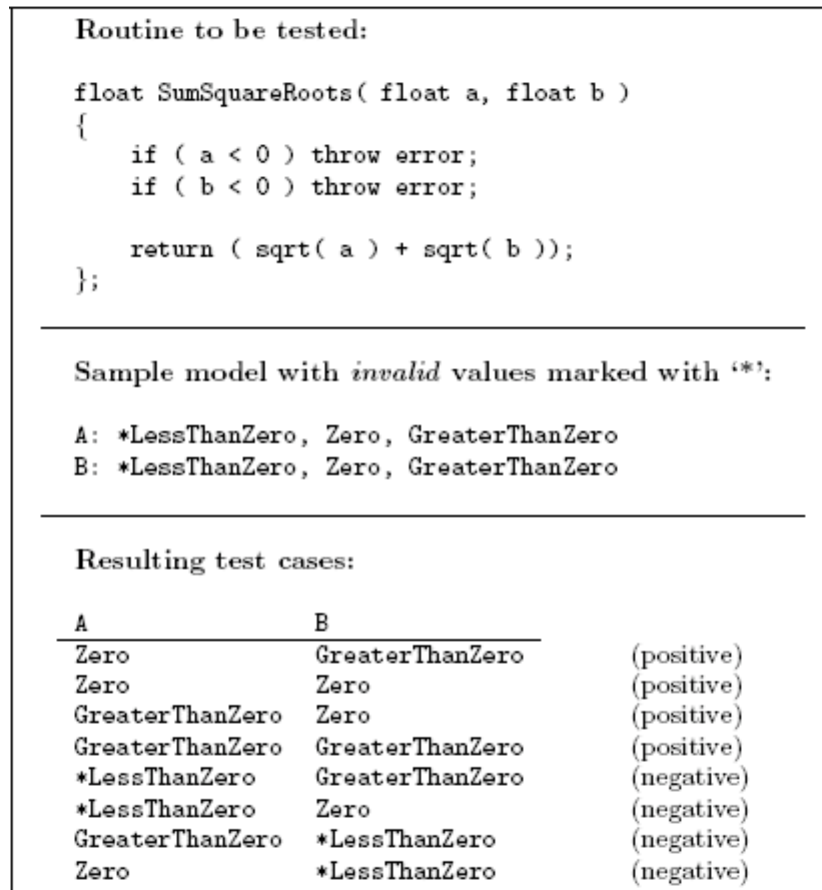
Testing with Negative Values

Often, in addition to testing all valid combinations, you might want to test by using values that are outside the allowable range, to ensure that the SUT handles error conditions properly. This *negative testing* should be conducted so that only one invalid value is present in any test case [7, 17, 18]. This is due to the way in which typical applications are written—namely, to take some failure action upon the first error that is detected. For this reason, a problem that is known as input masking—in which one invalid input prevents another invalid input from being tested—can occur.

For instance, the routine that is shown in Figure 12 can be called with any a or b that is a valid float. However, it makes sense to do the calculation only when a _ 0 and b _ 0. For that reason, the routine verifies a and b before any calculation is carried out. Assume that a test [a = −1; b = −1] was used to test for values that are outside of the valid range. Here, a actually masks b, and the verification of b being a nonnegative float value would

never get executed; and, if it were absent from the implementation, this fact would go unnoticed.

Note You might want to test more than one invalid value in a test case, but you should do so in addition to covering each invalid value separately. PICT can easily handle both cases.



Routine to be tested:

```
float SumSquareRoots( float a, float b )
{
    if ( a < 0 ) throw error;
    if ( b < 0 ) throw error;

    return ( sqrt( a ) + sqrt( b ));
};
```

Sample model with *invalid* values marked with '*':

```
A: *LessThanZero, Zero, GreaterThanZero
B: *LessThanZero, Zero, GreaterThanZero
```

Resulting test cases:

A	B	
Zero	GreaterThanZero	(positive)
Zero	Zero	(positive)
GreaterThanZero	Zero	(positive)
GreaterThanZero	GreaterThanZero	(positive)
*LessThanZero	GreaterThanZero	(negative)
*LessThanZero	Zero	(negative)
GreaterThanZero	*LessThanZero	(negative)
Zero	*LessThanZero	(negative)

Figure 12. Avoiding input masking

PICT allows marking values as invalid. (See Figure 12.) The output of such a model has the following properties:

1. All valid values are *t*-wise–combined with all other valid values in positive test cases.
2. If a test case contains an invalid value, there is only one such value.
3. All invalid values are *t*-wise–combined with all valid values in negative test cases.

The actual implementation of negative testing in PICT uses two generation runs: first, on test parameters with invalid values removed (positive test cases); and, second, on all values augmented with extra exclusions to disallow two invalid values from coexisting in one test case (negative test cases). Even without this feature implemented, a user could achieve the same results by modifying models and running the generator twice. In fact, this feature is not a part of the core generation engine; it is implemented in a higher layer of PICT and is there only for the convenience of users.

In practice, this concept can be extended from disallowing two values to disallowing any

two or more combinations from coexisting in one test case. Because this situation can be handled also with appropriately crafted constraints and it occurs less frequently than the need for handling individual invalid values, the author was never compelled to implement it.

Specifying Expected Results

Having a simple way of defining expected results for test cases is another useful feature.

If there are only two possibilities—test cases with valid values must always succeed, and test cases with invalid values should always fail—the task of specifying expected results is straightforward, and it does not require any support from the engine. Frequently, however, the rules for deciding the outcome of a test are more complex than checking for existence of an invalid value in the input data.

Traditionally, for the kind of output that PICT produces, either manual evaluation and assignment of expected test results or automated post-processing of test cases is used. Both are labor-intensive. The former is very hard to maintain when the input model changes, and the latter is typically implemented as a set of single-purpose scripts that have to be rewritten each time that a new test domain is modeled. To simplify this task, PICT reuses its existing artifacts—namely, parameters and constraints—and allows for the defining of expected results within the test model itself.

Defining expected results requires (1) specifying possible result outcomes in the form of result parameters and (2) defining the rules for assigning result values to test inputs. (See Figure 13.)

Sample model for int Sum(int[] Array, int Start, int Count) with expected results specified:

```
# Input parameters:

Array:  *Null, Empty, Valid
Start:  *TooLow, InRange, *TooHigh
Count:  *TooFew, Some, All, *TooMany

# Result parameters:

$Result:  Pass, OutOfBounds, InvalidPointer

# Result rules:

IF [Array] IN {"Empty", "Valid"} AND
   [Start] IN {"InRange"}        AND
   [Count] IN {"Some", "All"}
THEN [$Result] = "Pass";

IF [Array] = "Null"
THEN [$Result] = "InvalidPointer";

IF [Start] IN {"TooLow", "TooHigh"} OR
   [Count] IN {"TooFew", "TooMany"}
THEN [$Result] = "OutOfBounds";
```

Test cases contain input data and expected results:

Array	Start	Count	$Result
Empty	InRange	All	Pass
Valid	InRange	Some	Pass
Valid	InRange	All	Pass
Empty	InRange	Some	Pass
Empty	InRange	*TooMany	OutOfBounds
Empty	*TooHigh	All	OutOfBounds
Valid	InRange	*TooMany	OutOfBounds
Empty	InRange	*TooFew	OutOfBounds
*Null	InRange	All	InvalidPointer
Empty	*TooLow	Some	OutOfBounds
Valid	InRange	*TooFew	OutOfBounds
*Null	InRange	Some	InvalidPointer
Valid	*TooHigh	Some	OutOfBounds
Valid	*TooLow	All	OutOfBounds

Figure 13. Specifying expected results

Defining result parameters is as straightforward as defining input parameters. Syntactically, result rules are the same as constraints, which makes them easy to use. Semantically, however, rules must be both complete and consistent in the context of the values of a result parameter, which was not a requirement for constraints. Completeness and consistency of result rules are required to ensure that one, and only one, result value can be assigned to each possible combination of input parameters.

To deal with result parameters, PICT uses the same procedure that handles constraints. It employs its mixed-strength generation capability to combine the result parameters, which always have the order of generation t set to 1, with the input parameters. It also adds pre-processing and post-processing steps to ensure consistency of expected results. At this time, there is no verification of the completeness of result rules. Users must be careful to define result rules that assign at least one result value to each possible combination of input values. To allow the tool to distinguish between input and result parameters and apply additional processing steps to the latter group, the names of result parameters in PICT are prefixed with a “$” (dollar sign), by convention.

Assigning Weights to Values

In practical applications of automated test generation, it frequently happens that certain parameter values are presumed more *important* than others. For instance, a certain value

among others could be a default choice in the SUT; therefore, the likelihood of a user choosing it is greater than the likelihood of the user choosing other values. The weighting feature in PICT allows the putting of more emphasis on certain parameter values. Figure 14 shows how to set weights on values.

```
Type:           Single (5), Spanned (2), Striped (2), Mirror (2), RAID-5 (1)
Size:           10, 100, 1000, 10000, 40000
Format method:  Quick (5), Slow (1)
File system:    FAT (1), FAT32 (1), NTFS (5)
Cluster size:   512, 1024, 2048, 4096, 8192, 16384
Compression:    On (1), Off (10)
```

Figure 14. Value weights

An ideal weighting mechanism would allow the user to specify proportions of values and actually deliver a test suite that follows them exactly. However, this cannot be guaranteed for strategies whose primary purpose is to minimize the number of test cases that cover all *t*-wise combinations. PICT uses value weights only if two value choices are identical with regard to the covering of still unsatisfied combinations. In fact, in the ideal test generation —which is run when, at each step, there always exists a value that wins over others (in terms of combination coverage)—weights will not be honored at all.

In practice, users might not want to define precise likelihoods of choosing values, and they frequently are satisfied with a mechanism that allows them only to pick certain values more often than others. PICT satisfies that requirement very well.

Future Work

Although PICT already has a reasonably rich set of features, further improvements are necessary—especially, in the area of sub-modeling, which at this time allows for defining only one level of sub-models. It is actually a limitation of the UI; the underlying engine is able to handle any number of model levels, and it should be considerably straightforward to enable it also in the UI. An entirely new and better sub-modeling schema (described in the "Creating a Parameter Hierarchy" section), which is aimed at achieving the low volatility of certain parameters, could also be added.

Another refinement is necessary in the area of the handling of result rules—namely, automatic verification of result-rules completeness. This would remove the burden of manual work from users and fully ensure correctness of the result definitions.

Conclusion

PICT has been in use at Microsoft Corporation since 2000. It was designed with usability, flexibility, and speed in mind, and it employs a simple (yet effective) core generation algorithm that has separate preparation and generation phases. This flexibility allowed for the implementation of several features of practical importance. PICT gives testers a lot of control over the way in which tests are generated. Additionally, it raises the level of modeling abstraction, and it makes pairwise generation both convenient and usable.

Acknowledgments

Special recognition goes to David Erb, who architected and implemented the original generation algorithm, and whose excellent design allowed for many of the advanced features to appear in later versions of PICT. Thanks to Keith Stobie, Noel Nyman, and John Lambert of Microsoft Corporation; Keizo Tatsumi of Fujitsu Ltd; and Richard Vireday of Intel Corporation for their insightful perusal of the first draft.

References

- [1] Czerwonka, J. “Pairwise Testing: Available Tools.” Available at:
- [2] Ammann P. E., and A. J. Offutt. “Using Formal Methods to Derive Test Frames in Category-Partition Testing.” In Ninth Annual Conference on Computer Assurance (COMPASS’94), Gaithersburg, MD, pages 69–80, 1994.
- [3] Bach, J., and P. Shroeder. “Pairwise Testing: A Best Practice that Isn’t.” In Proceedings of the 22nd Pacific Northwest Software Quality Conference, pages 180–196, 2004.
- [4] Burr, K., and W. Young. “Combinatorial Test Techniques: Table-Based Automation, Test Generation, and Test Coverage.” In Proceedings of the International Conference on Software Testing, Analysis, and Review (STAR), San Diego, CA, 1998.
- [5] Burroughs, K., A. Jain, and R. L. Erickson. “Improved Quality of Protocol Testing Through Techniques of Experimental Design.” In Proceedings of the IEEE International Conference on Communications (Supercomm/ICC’94), May 1–5, New Orleans, LA, pages 745–752, 1994.
- [6] Cohen, D. M., 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, 23(7), 1997.
  AETG is a trademark of Telecordia Technologies.
- [7] Cohen, D. M., S. R. Dalal, J. Parelius, and G. C. Patton. “The Combinatorial Design Approach to Automatic Test Generation.” IEEE Software, 13(5):83–87, 1996.
- [8] Colbourn, C. J., M. B. Cohen, and R. C. Turban. “A Deterministic Density Algorithm for Pairwise Interaction Coverage.” In Proceedings of the IASTED International Conference on Software Engineering, 2004.
- [9] Dalal, S. R., A. J. N. Karunanithi, J. M. L. Leaton, G. C. P. Patton, and B. M. Horowitz. “Model-Based Testing in Practice.” In Proceedings of the International Conference on Software Engineering (ICSE 99), New York, pages 285–294, 1999.
- [10] Dunietz, I. S., W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino. “Applying Design of Experiments to Software Testing.” In Proceedings of the International Conference on Software Engineering (ICSE 97), New York, pages 205–215, 1997.
- [11] Grindal, M., J. Offutt, and S. F. Andler. “Combination Testing Strategies: A survey.” GMU Technical Report, 2004.
- [12] Hartman, A., and L. Raskin. “Problems and Algorithms for Covering Arrays.” *Discrete Mathematics*, 284(1–3):149–56, 2004.
- [13] Kuhn, R., and M. J. Reilly. “An Investigation of the Applicability of Design of Experiments to Software Testing.” In Proceedings of the 27th NASA/IEEE Software Engineering Workshop, NASA Goddard Space Flight Center, 2002.

- [14] Lei, Y., and K. C. Tai. "In-Parameter-Order: A Test-Generation Strategy for Pairwise Testing." In Proceedings of the Third IEEE International High-Assurance Systems Engineering Symposium, pages 254–261, 1998.
- [15] Malaiya, Y. K. "Antirandom Testing: Getting the Most Out of Black-Box Testing." In Sixth International Symposium on Software Reliability Engineering, Oct. 24–27, 1995, pages 86–95, 1996.
- [16] Mandl, R. "Orthogonal Latin Squares: An Application of Experiment Design to Compiler Testing." Communications of the ACM, 28(10):1054–1058, 1985.
- [17] Myers, G. J. *The Art of Software Testing*. New York: John Wiley and Sons, 1979.
- [18] Sherwood, G. "Effective Testing of Factor Combinations." In Proceedings of the Third International Conference on Software Testing, Analysis and Review, Washington, DC, pages 133–166, 1994.
- [19] Shimokawa, H., and S. Satoh. "Method of Setting Software Test Cases Using the Experimental Design Approach." In Proceedings of the Fourth Symposium on Quality Control in Software Production, Federation of Japanese Science and Technology, pages 1–8, 1984.
- [20] Smith, B., M. S. Feather, and N. Muscettola. "Challenges and Methods in Testing the Remote Agent Planner." In Proceedings of AIPS, 2000.
- [21] Tai, K. C., and Y. Lei. "A Test-Generation Strategy for Pairwise Testing." IEEE Transactions of Software Engineering, 28(1), 2002.
- [22] Tatsumi, K. "Test-Case–Design Support System." In Proceedings of the International Conference on Quality Control (ICQC), Tokyo, 1987, pages 615–620, 1987.
- [23] Wallace, D. R., and D. R. Kuhn. "Failure Modes in Medical Device Software: An Analysis of 15 Years of Recall Data. *International Journal of Reliability, Quality and Safety Engineering*, 8(4), 2001.
- [24] Williams, A. W. "Determination of Test Configurations for Pairwise Interaction Coverage." In Proceedings of the 13th International Conference on Testing Communicating Systems (Test-Com 2000), pages 59–74, 2000.
- [25] Williams, A. W., and R. L. Probert. "A Practical Strategy for Testing Pairwise Coverage of Network Interfaces." In Proceedings of the Seventh International Symposium on Software Reliability Engineering (ISSRE '96), page 246, 1996.

About the author

Jacek Czerwonka works for Microsoft Corporation in one of its test organizations. For the last few years, he has been involved in the design and implementation of pairwise-related tools, and the evangelization of pairwise testing at Microsoft. You can reach Jacek at jacekcz@microsoft.com.

© 2011 Microsoft. All rights reserved. Used with permission from Microsoft Corporation. This is a copy of the original found at http://msdn.microsoft.com/en-us/library/cc150619.aspx. Please check that document for the latest and most correct version.

Community Content