Pairwise Testing in Real World

Practical Extensions to Test Case Generators

Jacek Czerwonka
Microsoft Corp.
One Microsoft Way
Redmond, WA 98052
jacekcz@microsoft.com

ABSTRACT

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

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

This paper 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 obtained through this method of test case generation. It does focus on ways in which pure pairwise testing approach needs to be modified to become practically applicable and on features tools need to offer to support a tester trying to use pairwise in practice.

The paper makes frequent references to PICT, an existing and publicly available tool built on top of a flexible combinatorial test case generation engine, which implements several of the concepts described herein.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*

General Terms

Verification, Design

Keywords

Pairwise testing, combinatorial testing, test case generation, test case design

1. BACKGROUND

A set of possible inputs for any nontrivial piece of software is too large to be tested exhaustively. Techniques like *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], *anti-random* [15] and finally t-wise testing strategies with pairwise testing being the most prominent among these (see Figure 1).

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) i.e. for each pair of factor levels $l_{i,p}$ and $l_{j,q}$, where $1 \leq p \leq L_i$, $1 \leq q \leq L_j$, and $i \neq j$ 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 where $1 \leq t \leq N$. 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 testing of a compiler [16]. Tatsumi, in his paper on Test Case Design Support System used in Fujitsu Ltd [22], talks about two standards for creating test arrays: (1) with all combinations covered exactly the same number of times (orthogonal arrays) and (2) 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, needs to be used appropriately and with caution.

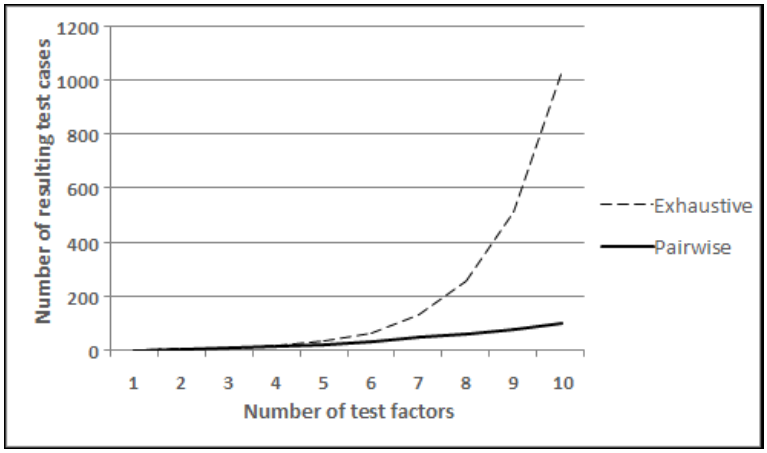

Figure 1: Increase in number of exhaustive and pairwise tests with number of test levels

Since the problem of finding a minimal array covering all pairwise combinations of a given set of test factors is NP-complete [14], understandably a considerable amount of research has gone into efficient creation of such arrays. Several strategies were proposed in an attempt to minimize number of tests 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 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* which allow specifying combinations that need to 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 paper describes PICT, a test case generation tool that has been in use at Microsoft since 2000, which implements the t-wise testing strategy and features making the strategy feasible in practice of software testing.

2. INTRODUCTION

2.1 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. Even though the ability to create the smallest possible covering array was given less emphasis, efficiency of PICT's core algorithm is comparable with other known test generation strategies (see Figure 2).

Input to PICT is a plain-text file (model) in which a tester specifies test factors (referred to as test *parameters* later in this paper) and test factor levels (referred to as *values* of a parameter). Figure 3 shows an example of a simple model used to produce test cases 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 if only $1 \leq t \leq N$.

2.2 Test Case Generation Engine

The test generation process in PICT is comprised of two main phases: (1) preparation and (2) generation.

In the preparation phase, PICT computes all information 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 corresponding to possible value combinations participating in a particular parameter interaction (4 slots for AB, 6 slots for AC and BC). See Figure 4.

Each slot can be marked *uncovered*, *covered*, or *excluded*. All 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 combinations to be covered. The slot becomes *covered* when the algorithm produces a test case satisfying 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 used in AETG[1] [6] with key differences being that PICT algorithm is deterministic[2] and it does not produce candidate tests.

[1]AETG is a trademark of Telecordia Technologies.

[2]PICT does make pseudo-random choices but 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.

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

```
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

```
A: 0, 1
B: 0, 1        translates to
C: 0, 1, 2
```

AB	AC	BC
00	00	00
01	01	01
10	02	02
11	10	10
	11	11
	12	12

Figure 4: Parameter interaction structures

```
# Assume test cases r_1, ..., r_{i-1} are already produced
# Slots in P reflecting combinations selected by r_1, ..., r_{i-1} are set to covered

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

While there are parameters with no values in r_i

    If r_i is empty
        Choose a parameter interaction p from P with most uncovered slots
        Pick the first uncovered combination from p
    Else
        # Assume values l_1, ..., l_{k-1} have already been chosen and added to r_i

        Look at subset Q of P that covers at least one parameter with no
         representation in l_1, ..., l_{k-1}

        Look at slots in Q which values are consistent with already chosen values in l_1, ..., l_{k-1}

        If there exist uncovered combinations
            Pick a slot with values which when added to r_i would cover the most uncovered
             combinations with l_1, ..., l_{k-1} and the resulting partial test case r_i would not
             contain an excluded combination
        Else
            Pick randomly a covered combination which when added to l_1, ..., l_{k-1} would not
             contain an excluded combination

    Add values of this combination to r_i
    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 is able to compute test suites comparable in size to other tools existing in the field and it is fast enough for all practical purposes.[3]

3. ADVANCED FEATURES

3.1 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 (see Figure 6), interactions of parameters B, C, and D might require better coverage than interactions of A or E. We should be able to generate all possible triplets of B, C, and D and cover all pairs of all other parameter interactions. Importance of this feature stems from the fact that often certain parameter interactions seem to be more 'sensitive' than others. Possibly, experience had shown that interactions of these parameters are at the root of proportionally more defects than other interactions. Therefore they should be tested more throughly. 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.

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, since the generation phase operates solely on parameter interaction structures, they can be manipulated to reflect the need for higher-order interactions of certain parameters.

3.2 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 that product is then used for creating combinations with parameters on upper levels of the hierarchy. This is a useful technique which can be used to (1) model test domains with a clear hierarchy of test parameters i.e. API functions taking structures as arguments and UI windows with additional dialogs or (2) to limit the combinatorial explosion of certain parameter interactions. (1) is intuitive, (2) requires explanation.

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

Consider the example 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 9 parameters, PICT generated 31 test cases and they included 17 different combinations of the hardware-related parameters: *Platform*, *CPUs*, and *RAM*.

Instead, hardware parameters can be designated as a 'submodel' and pairwise-combined first. The result of this generation is then used to create the final output in which 6 individual input parameters and 1 compound environment parameter take part. The result is a larger test suite (54 tests) but it contains only 9 unique combinations of the hardware parameters. Users of this feature have to be cautious, however, and understand that in this scheme not all t-wise combinations of all 9 parameters will be covered.

If the goal is to achieve low volatility of a certain subset of parameters, an even better solution can be implemented. Namely, generate all required t-wise combinations at the lower level (*Platform*, *CPUs*, and *RAM*) and use them for the higher-level combinations (all 9 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, we would achieve low volatility and would not lose t-wise coverage. This feature has not been implemented in PICT yet.

3.3 Excluding Unwanted Combinations

The definition of pairwise testing given in section 1 talks about test factors (parameters) being independent. This is, however, rarely the case in practice. That is why constraints are an indispensable feature of a test case generator. They describe 'limitations' of the test domain i.e. combinations that are impossible to be successfully executed in the context of given SUT. Going back to the example in Figure 3, FAT file system cannot actually be applied onto volumes larger than 4 Gigabytes. Any test case that asks for *FAT* and *volume 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 e.g. [*FAT*, *RAID5*], 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 in such a way that unwanted combinations cannot possibly be chosen; either by splitting parameter definitions onto disjoint subsets [18] or by creating hybrid parameters [25]. Other methods could involve post-processing of resulting test suites and modifying test cases that violate one or more constraints in a way that the violation is avoided.

[3]For instance, for 50 parameters with 20 values each (20^{50}), PICT generates a pairwise test suite in under 20 seconds on a Intel Pentium M 1.8GHz machine running Windows XP SP2.

		AB	AC	AD	AE	BC	BD	BE	CD	CE	DE
A: 0, 1		00	00	00	00	00	00	00	00	00	00
B: 0, 1		01	01	01	01	01	01	01	01	01	01
C: 0, 1	**translates to**	10	10	10	10	10	10	10	10	10	10
D: 0, 1		11	11	11	11	11	11	11	11	11	11
E: 0, 1											

			AB	AC	AD	AE	BCD	BE	CE	DE
			00	00	00	00	000	00	00	00
A:	0, 1		01	01	01	01	001	01	01	01
B @ 3:	0, 1		10	10	10	10	010	10	10	10
C @ 3:	0, 1	**translates to**	11	11	11	11	011	11	11	11
D @ 3:	0, 1						100			
E:	0, 1						101			
							110			
							111			

Figure 6: Fixed- and mixed-strength generation

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

Dalal et al. describe *AETGSpec*, a test domain modeling language which 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.

PICT internally translates constraints into a set of combinations 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 need to be excluded are in fact, marked *excluded*.
2. How to handle exclusions that are more granular than the corresponding parameter interaction structure; i.e. 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 depicted in Figure 9. Constraints on that model create a circular dependency loop between values $A{:}0 \Rightarrow B{:}0 \Rightarrow C{:}0 \Rightarrow A{:}1$ 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 have to ensure that we do not pick *A:0* at all. Instead of the initial three, five combinations need to be excluded, among them all combinations of *A:0*.

The second, is a case where directly marking combinations as *excluded* in parameter interaction structures is impossible. Consider the example shown in Figure 10 in which 3-element exclusions are created but parameter interaction structures only refer to two parameters at a time. In other words, there is not a parameter interaction structure *ABC* which can be used to mark the excluded combination; *AB*, *AC* or *BC* are not granular enough. In such a case, one more parameter interaction structure *ABC* is set up. Appropriate combinations are marked *excluded* and the rest of them are marked *covered*. The generation algorithm will ensure that all possible combinations of *AB*, *AC*, and *BC* will be covered without actually picking *A:0, B:0, C:1* combination.

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 exclusions. Since there is no need for any validity verification or post-processing of produced results, the generation phase can be very fast.

3.4 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:

1. It allows explicit specification of 'important' combinations.
2. It can be used to minimize change in the output when the test domain description is modified and resulting test suite 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 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 small modification of the test domain description, like 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 the test domain.

It happens often that the initial test domain specification is incomplete, however, the test suite it produces is a basis for the first set of configurations to run tests on. For example, when a test case specifies a machine with two AMD64 CPUs, a SCSI hard drive, exactly 1 GB of RAM, Windows XP with Service Pack 2, and a certain version of Internet Explorer, such a machine must be assembled and all necessary software 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 already prepared configurations. Seeding allows for re-use of old test cases in newly generated test suites.[4]

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 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.

3.5 Testing with Negative Values

In addition to testing all valid combinations, it is often desirable to test using values outside the allowable range to make sure the SUT handles error conditions properly. This 'negative testing' should be conducted such 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 detected. For this reason a problem known as *input masking*, in which one invalid input prevents another invalid input from being tested, can occur.

For instance, the routine in Figure 12 can be called with any a or b that is a valid *float*. However, it only makes sense to do the calculation when $a \geq 0$ and $b \geq 0$. For that reason, the routine verifies a and b before any calculation is carried out. Assume a test [$a = -1$; $b = -1$] was used to test for values outside of the valid range. Here, a actually masks b and the verification of b being a non-negative *float* value would never get executed and if it is absent from the implementation this fact would go unnoticed.[5]

[4] Certain precautions must be taken in cases involving removal of parameter values, removal of entire parameters, or addition of new constraints.

[5] It might be desirable to test more than one invalid value in a test case but it should be done in addition to covering each invalid value separately. Both cases can easily be handled by PICT.

```
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

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

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

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

$\Rightarrow$

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

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

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* (Figure 12). The output of such 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 to coexist 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 and is implemented in a higher layer of PICT and is there for users' convenience only.

In practice, this concept can be extended from disallowing two *values* to disallowing any two or more *combinations* to coexist in one test case. Since this situation also can be handled with appropriately crafted constraints and occurs less frequently than the need for handling individual invalid values, the author never felt compelled to implement it.

3.6 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 does not require any support from the engine. Frequently, however, rules of 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 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 input model changes, the latter is typically implemented as a set of single-purpose scripts which have to be rewritten each time a new test domain is modeled. To simplify this task, PICT re-uses its existing artifacts, namely parameters and constraints, and allows for defining expected results within the test model itself.

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

Defining result parameters is as straightforward as defining input parameters. Result rules are syntactically the same as constraints which makes them easy to use. Semantically, however, rules must be both *complete* and *consistent* in the

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

context of 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 order of generation t set to 1, with the input parameters. It also adds pre- and post-processing steps to ensure consistency of expected results. At this time, there is no verification of completeness of result rules. Users must be careful to define result rules which 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, names of result parameters in PICT are, by convention, prefixed with a '$'.

3.7 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 it being chosen by a user is greater than her picking other values. Weighting feature in PICT allows putting more emphasis on certain parameter values. Figure 14 shows how to set weights on values.

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 covering all t-wise combinations. PICT uses value weights only if two value choices are identical with regards to covering still unsatisfied combinations. In fact, in the ideal test generation 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 may not want to define precise likelihoods of choosing values and frequently are satisfied with a mechanism that only allows them to pick certain values more often than others. PICT satisfies that requirement very well.

4. FUTURE WORK

Although PICT already has a reasonably rich set of features, further improvements are needed. Especially in the area of sub-modeling which at this time only allows for defining one level of sub-models. It is actually a limitation of the user interface; the underlying engine is able to handle any number of model levels and it should be considerably straightforward to enable it in the user interface as well. An entirely new and better sub-modeling schema, described in section 3.2, aimed at achieving low volatility of certain parameters could also be added.

Another refinement is required in the area of handling of result rules. Namely, automatic verification of result rules completeness is needed. It would remove the burden of manual work from users and fully ensure correctness of the result definitions.

5. SUMMARY

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

6. ACKNOWLEDGMENTS

Special recognition 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 Corp., Keizo Tatsumi of Fujitsu Ltd, and Richard Vireday or Intel Corp. for their insightful perusal of the first draft.

7. REFERENCES

[1] http://www.pairwise.org/tools.asp.

[2] P. E. Ammann 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] J. Bach 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] K. Burr 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] K. Burroughs, 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, Louisiana*, pages 745–752, 1994.

[6] 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*, 23(7), 1997.

[7] D. M. Cohen, 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] C. J. Colbourn, 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] S. R. Dalal, 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*

```
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

Conference on Software Engineering (ICSE 99), New York, pages 285–294, 1999.

[10] 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 International Conference on Software Engineering (ICSE 97), New York*, pages 205–215, 1997.

[11] M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies - a survey. *GMU Technical Report*, 2004.

[12] A. Hartman and L. Raskin. Problems and algorithms for covering arrays. *Discrete Mathematics*, 284(1-3):149–56, 2004.

[13] R. Kuhn 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] Y. Lei 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] Y. K. Malaiya. 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] R. Mandl. Orthogonal latin squares: an application of experiment design to compiler testing. *Communications of the ACM*, 28(10):1054–1058, 1985.

[17] G. J. Myers. *The Art of Software Testing*. John Wiley and Sons, 1978.

[18] G. Sherwood. 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] H. Shimokawa 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] B. Smith, M. S. Feather, and N. Muscettola. Challenges and methods in testing the remote agent planner. In *Proceedings of AIPS*, 2000.

[21] K. C. Tai and Y. Lei. A test generation strategy for pairwise testing. *IEEE Transactions of Software Engineering*, 28(1), 2002.

[22] K. Tatsumi. Test case design support system. In *Proceedings of the International Conference on Quality Control (ICQC), Tokyo, 1987*, pages 615–620, 1987.

[23] D. R. Wallace 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] A. W. Williams. Determination of test configurations for pair-wise interaction coverage. In *Proceedings of the 13th International Conference on Testing Communicating Systems (Test-Com 2000)*, pages 59–74, 2000.

[25] A. W. Williams and R. L. Probert. A practical strategy for testing pair-wise coverage of network interfaces. In *Proceedings of the Seventh International Symposium on Software Reliability Engineering (ISSRE '96)*, page 246, 1996.