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 19 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 describes PICT, a tool built on top of a flexible combinatorial test case generation engine, which implements several concepts of practical importance.

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.

Pairwise testing strategy is defined as follows:
Given a set of N independent test factors: f₁, f₂, ..., fₙ, with each factor fᵢ having Lᵢ possible levels: fᵢ = {lᵢ₁, ..., lᵢₗᵢ}, a set of tests R is produced. Each test in R contains N test levels, one for each test factor fᵢ, and R covers all possible pairs of test factors i.e. for each pair of factor levels lᵢ,p and lⱼ,q, where i ≠ j, 1 ≤ p ≤ Lᵢ, and 1 ≤ q ≤ Lⱼ, there exists at least one test in R that contains both lᵢ,p and lⱼ,q.

This concept can easily be extended from covering all pairs to covering any t-wise combinations where 1 ≤ t ≤ 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], like any technique pairwise needs to be used appropriately and with caution.

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.

1In Japanese.
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. The ability to create the smallest possible covering array was given less emphasis. However, efficiency of PICT's core algorithm is comparable with other known test generation strategies (see Figure 1).

Input to PICT is a plain-text file (model) in which a tester specifies test factors (parameters) and test factor levels (values of a parameter). Figure 2 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 3.

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 4). It builds one test case at a time, locally optimizing the solution. It is similar to the algorithm used in AETG \(^{2}\) [6] with key differences being that PICT algorithm is deterministic \(^{3}\) and it does not produce candidate tests.

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. \(^{4}\)

3. ADVANCED FEATURES

3.1 Mixed-strength Generation

Most commonly, when t-wise testing is discussed it is assumed that all test factor interactions have a fixed order \(t\). In other words, if \(t = 3\) is requested, all triplets of test factor values will be covered. It is sometimes useful, however, to be able to define different orders of combinations for different subsets of test factors. For example (see Figure 5), 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 test factor interactions seem to be more ‘sensitive’ than others. Possibly, experience had shown that interactions of these factors are at the root of proportionally more defects than other factor 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.

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 test factors.

3.2 Creating a Test Factor Hierarchy

To complement the mixed-strength generation, PICT allows a user to create a hierarchy of test factors. This scheme allows for certain factors to be t-wise combined first and that

\(^{2}\) AETG is a trademark of Telecordia Technologies.

\(^{3}\) 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.

\(^{4}\) For instance, for 50 parameters with 20 values each (20\(^{50}\)), PICT generates a pairwise test suite in under 20 seconds on an Intel Pentium M 1.8GHz machine running Windows XP SP2.
**Table:** Comparison of PICT generation efficiency with other known tools

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$3^4$</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>11</td>
<td>?</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>$3^{13}$</td>
<td>15</td>
<td>17</td>
<td>15</td>
<td>15</td>
<td>18</td>
<td>18</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>$4_{15}^{17}2_{29}$</td>
<td>41</td>
<td>34</td>
<td>40</td>
<td>39</td>
<td>38</td>
<td>35</td>
<td>34</td>
<td>37</td>
</tr>
<tr>
<td>$4_{3}^{35}2_{35}$</td>
<td>28</td>
<td>26</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>$2_{100}$</td>
<td>10</td>
<td>15</td>
<td>14</td>
<td>10</td>
<td>16</td>
<td>15</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>$10_{20}$</td>
<td>180</td>
<td>212</td>
<td>231</td>
<td>210</td>
<td>193</td>
<td>201</td>
<td>197</td>
<td>210</td>
</tr>
</tbody>
</table>

**Figure 1:** Comparison of PICT 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 2:** Parameters for volume creation and formatting

<table>
<thead>
<tr>
<th>AB</th>
<th>AC</th>
<th>BC</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>10</td>
<td>02</td>
<td>02</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
</tbody>
</table>

**Figure 3:** Parameter interaction structures

Assume test cases $r_1, ..., r_{i-1}$ are already selected
They 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 4:** PICT heuristic algorithm
product is then used for creating combinations with factors on upper levels of the hierarchy. This is a useful technique which can be used to (1) model test domains with clear structure/substructure hierarchy of test factors i.e. API functions 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 test factors analysis [22], distinguishes between ‘input factors’ which are direct inputs to the SUT and ‘environmental factors’ which constitute the environment the SUT operates in. Typically, input factors can be controlled and set much easier than environmental ones (compare supplying SUT with different parameters to running the SUT 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 6 which contains the same test factors as Figure 2 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 ‘sub-model’ and pairwise-combined first. The result of this generation is then used to generate 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 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 certain parameters, a 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 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 executed in the context of given SUT. Going back to the example in Figure 2, 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’ and ‘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 factors in such a way that unwanted combinations cannot possibly be chosen; either by splitting test factor 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.

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 7, three IF-THEN statements describe ‘limitations’ of a particular test domain.

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

<table>
<thead>
<tr>
<th>A: 0, 1</th>
<th>AB</th>
<th>AC</th>
<th>AD</th>
<th>AE</th>
<th>BC</th>
<th>BD</th>
<th>BE</th>
<th>CD</th>
<th>CE</th>
<th>DE</th>
</tr>
</thead>
<tbody>
<tr>
<td>B: 0, 1</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>C: 0, 1</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>D: 0, 1</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>E: 0, 1</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

**Figure 5:** Fixed- and mixed-strength generation
Test domain consisting of ‘input’ and ‘environment’ factors:

# Input factors

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 factors

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

# Environment factors will form a sub-model

\{ \text{PLATFORM, CPUS, RAM} \} @ 2

Hierarchy of test factors:

- Type
- Size
- Format method
- File system
- Cluster size
- Compression
- \(<\text{CompoundFactor}> \ (t=2)\)
  - Platform
  - CPUs
  - RAM

Figure 6: Two-level hierarchy of test factors

| 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

\[
\text{IF } [\text{File system}] = "\text{FAT}" \text{ THEN } [\text{Size}] \leq 4096; \\
\text{IF } [\text{File system}] = "\text{FAT32}" \text{ THEN } [\text{Size}] \leq 32000; \\
\]

# And not all file systems support compression

\[
\text{IF } [\text{File system}] \neq "\text{NTFS}" \text{ or } ( [\text{File system}] = "\text{NTFS}" \text{ and } [\text{Cluster size}] > 4096) \text{ THEN } [\text{Compression}] = "\text{Off}"; \\
\]

Figure 7: Parameters for volume creation and formatting augmented with constraints
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 8. 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 9 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 specifying ‘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 test factors or test factor values, might cause big changes in the resulting test suite. Containing those changes can be an important cost-saving option especially when hardware factors 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.¹

In PICT, these seeding combinations can be full combinations (with values for all test factors specified) or partial combinations. Figure 10 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 11 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.²

PICT allows marking values invalid (Figure 11). 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.

---

¹ Certain precautions must be taken in cases involving removal of test factor values, removal of entire test factors, or addition of new constraints.

² 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.
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:

<table>
<thead>
<tr>
<th>AB</th>
<th>AC</th>
<th>BC</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

⇒

<table>
<thead>
<tr>
<th>AB</th>
<th>AC</th>
<th>AD</th>
<th>BC</th>
<th>BD</th>
<th>CD</th>
<th>ABC</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>000</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>001</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>010</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>011</td>
</tr>
</tbody>
</table>

Figure 8: 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:

<table>
<thead>
<tr>
<th>AB</th>
<th>AC</th>
<th>AD</th>
<th>BC</th>
<th>BD</th>
<th>CD</th>
<th>ABC</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>000</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>001</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>010</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>011</td>
</tr>
<tr>
<td>100</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 9: Handling exclusions that are more granular than parameter interaction structures
Test factors:

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:

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Format method</th>
<th>File system</th>
<th>Cluster size</th>
<th>Compression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>100</td>
<td>Quick</td>
<td>NTFS</td>
<td>4096</td>
<td>Off</td>
</tr>
<tr>
<td>Mirror</td>
<td>Slow</td>
<td>FAT32</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 10: Seeding

Routine to be tested:

```c
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:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>Zero</td>
<td>GreaterThanZero (positive)</td>
</tr>
<tr>
<td>Zero</td>
<td>Zero (positive)</td>
</tr>
<tr>
<td>GreaterThanZero</td>
<td>Zero (positive)</td>
</tr>
<tr>
<td>GreaterThanZero</td>
<td>GreaterThanZero (positive)</td>
</tr>
<tr>
<td>*LessThanZero</td>
<td>GreaterThanZero (negative)</td>
</tr>
<tr>
<td>*LessThanZero</td>
<td>Zero (negative)</td>
</tr>
<tr>
<td>GreaterThanZero</td>
<td>*LessThanZero (negative)</td>
</tr>
<tr>
<td>Zero</td>
<td>*LessThanZero (negative)</td>
</tr>
</tbody>
</table>

Figure 11: Avoiding input masking
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 factors 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 Result

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 input arrays 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, parameters and constraints, and allows for defining expected results within the test model.

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 12).

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 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 test factor 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 test factor values. Figure 13 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
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:

<table>
<thead>
<tr>
<th>Array</th>
<th>Start</th>
<th>Count</th>
<th>$Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>InRange</td>
<td>All</td>
<td>Pass</td>
</tr>
<tr>
<td>Valid</td>
<td>InRange</td>
<td>Some</td>
<td>Pass</td>
</tr>
<tr>
<td>Valid</td>
<td>InRange</td>
<td>All</td>
<td>Pass</td>
</tr>
<tr>
<td>Empty</td>
<td>InRange</td>
<td>Some</td>
<td>Pass</td>
</tr>
<tr>
<td>Empty</td>
<td>InRange</td>
<td>*TooMany</td>
<td>OutOfBounds</td>
</tr>
<tr>
<td>Empty</td>
<td>*TooHigh</td>
<td>All</td>
<td>OutOfBounds</td>
</tr>
<tr>
<td>Valid</td>
<td>InRange</td>
<td>*TooMany</td>
<td>OutOfBounds</td>
</tr>
<tr>
<td>Empty</td>
<td>InRange</td>
<td>*TooFew</td>
<td>OutOfBounds</td>
</tr>
<tr>
<td>*Null</td>
<td>InRange</td>
<td>All</td>
<td>InvalidPointer</td>
</tr>
<tr>
<td>Empty</td>
<td>*TooLow</td>
<td>Some</td>
<td>OutOfBounds</td>
</tr>
<tr>
<td>Valid</td>
<td>InRange</td>
<td>*TooFew</td>
<td>OutOfBounds</td>
</tr>
<tr>
<td>*Null</td>
<td>InRange</td>
<td>Some</td>
<td>InvalidPointer</td>
</tr>
<tr>
<td>Valid</td>
<td>*TooHigh</td>
<td>Some</td>
<td>OutOfBounds</td>
</tr>
<tr>
<td>Valid</td>
<td>*TooLow</td>
<td>All</td>
<td>OutOfBounds</td>
</tr>
</tbody>
</table>

Figure 12: Specifying expected results

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 13: Value weights
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 John Lambert, Noel Nyman and Keith Stobie of Microsoft Corp. and Keizo Tatsumi of Fujitsu Ltd for encouraging me to write this paper and their insightful perusal of the first draft.

7. REFERENCES


