In-Parameter-Order: A Test Generation Strategy for Pairwise Testing

Jeff Lei
Department of Computer Science and Engineering
The Univ. of Texas at Arlington
6/21/2005
Outline

- Introduction
- The IPO Strategy
- Related Work
- 3-Way Testing and Beyond
- Conclusion
Why Testing?

- Modern society is increasingly dependent on the quality of software systems.
- Software failure can cause severe consequences, including loss of human life.
- **Testing** is the most widely used approach to ensuring software quality.
The Testing Process

The testing process consists of three stages:

- **Test Generation** - Generate test data inputs
- **Test Execution** - Test setup and the actual test runs
- **Test Results Evaluation** - Check if the output is in line with expectations
The Challenge

1. Testing is labor intensive and can be very costly
   - often estimated to consume more than 50% of the development cost

2. Exhaustive testing is impractical due to resource constraints

3. How to make a good trade-off between test effort and quality assurance?
Pairwise Testing

q Given any pair of input parameters of a system, every combination of valid values of the two parameters be covered by at least one test

q A special case of combinatorial testing that requires $n$-way combinations be tested

\[ n \text{ can be } 1, 2, \ldots, \text{ or the total number of parameters in the system} \]

q Based on simple specifications, and does not need to look into the implementation details
### Example (1)

<table>
<thead>
<tr>
<th>Component</th>
<th>Operating System</th>
<th>Connection Type</th>
<th>Printer Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Netscape</td>
<td>Windows</td>
<td>LAN</td>
<td>Local</td>
</tr>
<tr>
<td>IE</td>
<td>Macintosh</td>
<td>PPP</td>
<td>Networked</td>
</tr>
<tr>
<td>Mozilla</td>
<td>Linux</td>
<td>ISDN</td>
<td>Screen</td>
</tr>
</tbody>
</table>

**TABLE 1**

**Four Components, Each With Three Settings**

Exhaustive testing requires 81 tests = $3 \times 3 \times 3 \times 3$. 
## Example (2)

<table>
<thead>
<tr>
<th>Test</th>
<th>Browser</th>
<th>OS</th>
<th>Connection</th>
<th>Printer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NetScape</td>
<td>Windows</td>
<td>LAN</td>
<td>Local</td>
</tr>
<tr>
<td>2</td>
<td>NetScape</td>
<td>Linux</td>
<td>ISDN</td>
<td>Networked</td>
</tr>
<tr>
<td>3</td>
<td>NetScape</td>
<td>Macintosh</td>
<td>PPP</td>
<td>Screen</td>
</tr>
<tr>
<td>4</td>
<td>IE</td>
<td>Windows</td>
<td>ISDN</td>
<td>Screen</td>
</tr>
<tr>
<td>5</td>
<td>IE</td>
<td>Macintosh</td>
<td>LAN</td>
<td>Networked</td>
</tr>
<tr>
<td>6</td>
<td>IE</td>
<td>Linux</td>
<td>PPP</td>
<td>Local</td>
</tr>
<tr>
<td>7</td>
<td>Mozilla</td>
<td>Windows</td>
<td>PPP</td>
<td>Networked</td>
</tr>
<tr>
<td>8</td>
<td>Mozilla</td>
<td>Linux</td>
<td>LAN</td>
<td>Screen</td>
</tr>
<tr>
<td>9</td>
<td>Mozilla</td>
<td>Macintosh</td>
<td>ISDN</td>
<td>Local</td>
</tr>
</tbody>
</table>

**TABLE II**

**Test Suite to Cover all Pairs from Table I**
Why Pairwise?

- Many faults are caused by the interactions between two parameters
  - 92% block coverage, 85% decision coverage, 49% p-uses and 72% c-uses

- Not practical to cover all the parameter interactions
  - Consider a system with \( n \) parameter, each with \( m \) values. How many interactions to be covered?

- A “good” trade-off between test effort and test coverage
  - For a system with 20 parameters each with 15 values, pairwise testing only requires less than 412 tests, whereas exhaustive testing requires \( 15^{20} \) tests.
Outline

- Introduction
- The IPO Strategy
- Related Work
- 3-Way Testing and Beyond
- Conclusion
NP-Completeness

The problem of generating a **minimum** pairwise test set is NP-complete.

- Can be reduced to the **vertex cover** problem

Unlikely to find a **polynomial** time algorithm to solve the problem.

- **Greedy algorithms** are the first thing coming into the mind of a computer scientist
The Framework

Strategy In-Parameter-Order

begin
  /* for the first two parameters p₁ and p₂ */
  T := {(v₁, v₂) | v₁ and v₂ are values of p₁ and p₂, respectively}
  if n = 2 then stop;
  /* for the remaining parameters */
  for parameter pᵢ, i = 3, 4, ..., n do
    begin
      /* horizontal growth */
      for each test (v₁, v₂, ..., vᵢ₋₁) in T do
        replace it with (v₁, v₂, ..., vᵢ₋₁, vᵢ), where vᵢ is a value of pᵢ
      /* vertical growth */
      while T does not cover all pairs between pᵢ and
        each of p₁, p₂, ..., pᵢ₋₁ do
          add a new test for p₁, p₂, ..., pᵢ to T;
    end
  end
end
Horizontal Growth

Algorithm IPO_H(T, p_i)
// T is a test set. But T is also treated as a list with elements in arbitrary order.
{ assume that the domain of p_i contains values v_1, v_2, ..., and v_q;
  \( \pi = \{ \text{pairs between values of } p_i \text{ and values of } p_1, p_2, \ldots, \text{ and } p_{i-1} \}; \)
  if (|T| ≤ q)
    { for \(1 ≤ j ≤ |T|\), extend the jth test in T by adding value \(v_j\) and
      remove from \(\pi\) pairs covered by the extended test;
    }
  else
    { for \(1 ≤ j ≤ q\), extend the jth test in T by adding value \(v_j\) and
      remove from \(\pi\) pairs covered by the extended test;
      for \(q < j ≤ |T|\), extend the jth test in T by adding one value of \(p_i\)
      such that the resulting test covers the most number of pairs in \(\pi\), and
      remove from \(\pi\) pairs covered by the extended test;
    }
}
**Vertical Growth**

**Algorithm** $IPO_V(T, \pi)$

1. let $T'$ be an empty set;
2. for each pair in $\pi$
   1. assume that the pair contains value $w$ of $p_k$, $1 \leq k < i$, and value $u$ of $p_i$;
   2. if ($T'$ contains a test with “−” as the value of $p_k$ and $u$ as the value of $p_i$)
      modify this test by replacing the “−” with $w$;
   3. else
      add a new test to $T'$ that has $w$ as the value of $p_k$, $u$ as the value of $p_i$, and “−” as the value of every other parameter;
3. $T = T \cup T'$;
**Example (1)**

Consider a system with the following parameters and values:

- parameter $A$ has values $A_1$ and $A_2$
- parameter $B$ has values $B_1$ and $B_2$, and
- parameter $C$ has values $C_1$, $C_2$, and $C_3$
Example (2)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>B1</td>
<td>C1</td>
</tr>
<tr>
<td>A1</td>
<td>B2</td>
<td>C2</td>
</tr>
<tr>
<td>A2</td>
<td>B1</td>
<td>C3</td>
</tr>
<tr>
<td>A2</td>
<td>B2</td>
<td>C1</td>
</tr>
</tbody>
</table>

Horizontal Growth

Vertical Growth
**PairTest**

- A Java tool that implements the IPO strategy
- Supports the following types of test generation
  - Account for relations and constraints
  - Extend from an existing test set
  - Modify/extend an existing test set after changes of parameters, values, relations and constraints
- Has been used in IBM and software engineering classes at NCSU
Empirical Results (1)

Let \( n \) be the number of parameters, and \( d \) the domain size of each parameter. The size of a pairwise test set is in the order of \( O(\log n) \) and \( O(d^2) \).

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|c|}
\hline
n (\# of parameters) & 10 & 20 & 30 & 40 & 50 & 60 & 70 & 80 & 90 & 100 \\
\hline
s (\# of tests) & 31 & 34 & 41 & 42 & 48 & 48 & 51 & 51 & 51 & 53 \\
\hline
t (time in seconds) & 0.11 & 0.16 & 0.22 & 0.44 & 0.77 & 0.99 & 1.37 & 1.81 & 2.23 & 2.96 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
d (\# of values) & 5 & 10 & 15 & 20 & 25 & 30 \\
\hline
s (\# of Tests) & 47 & 169 & 361 & 618 & 956 & 1355 \\
\hline
t (time in seconds) & 0.05 & 0.28 & 0.72 & 1.54 & 2.96 & 5.16 \\
\hline
\end{array}
\]
Empirical Results (2)

Sizes of Pairwise Test Sets Generated by AETG and PairTest

<table>
<thead>
<tr>
<th>System</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
<th>S5</th>
<th>S6</th>
</tr>
</thead>
<tbody>
<tr>
<td>AETG</td>
<td>11</td>
<td>17</td>
<td>35</td>
<td>25</td>
<td>12</td>
<td>193</td>
</tr>
<tr>
<td>PairTest</td>
<td>9</td>
<td>17</td>
<td>34</td>
<td>26</td>
<td>15</td>
<td>212</td>
</tr>
</tbody>
</table>

S1: 4 3-value parameters
S2: 13 3-value parameters
S3: 61 parameters (15 4-value parameters, 17 3-value parameters, 29 2-value parameters)
S4: 75 parameters (1 4-value parameter, 39 3-value parameters, 35 2-value parameters)
S5: 100 2-value parameters
S6: 20 10-value parameters
Outline

- Introduction
- The IPO Strategy
- Related Work
- 3-Way Testing and Beyond
- Conclusion
**Classification**

- **Computational** methods that are mainly developed by computer scientists
  - AETG (from Telcordia), TCG (from JPL/NASA), DDA (from ASU), PairTest

- **Algebraic** methods that are mainly developed by mathematicians
  - Orthogonal Arrays
  - Recursive Construction
**AETG (1)**

- Starts with an empty set and adds one (complete) test at a time

- Each test is *locally optimized* to cover the most number of missing pairs:
  - Generate a random order of the parameters
  - Use a greedy algorithm to construct a test that covers the most uncovered pairs
  - Repeat the above two steps for a given number of times (suggested 50), and select the best one
AETG (2)

Add the 1st test
Add the 2nd test
Add the last test
**AETG vs IPO**

- AETG is fundamentally **non-deterministic**, whereas IPO is **deterministic**
- AETG has a higher order of complexity, both in terms of time and space, than IPO
- AETG is a commercial tool, and its license is very expensive, whereas IPO is open to the public.
Orthogonal Arrays (1)

- An orthogonal array $OA_\lambda(N; k, v, t)$ is an $N \times k$ array on $v$ symbols such that every $N \times t$ sub-array contains all tuples of size $t$ from $v$ symbols exactly $\lambda$ times.

- $N$ - Number of test cases
- $k$ - Number of parameters
- $v$ - Number of values of each parameter
- $t$ - Degree of interaction
- $\lambda$ - 1 for software testing and is often omitted

- For example, Table 2 is an orthogonal array $OA(9; 4, 3, 2)$
Orthogonal Arrays (2)

**OA (9; 4, 3, 2)**

<table>
<thead>
<tr>
<th>(b0, b1)</th>
<th>A = b1</th>
<th>B = b0 + b1</th>
<th>C = b0 + 2 * b1</th>
<th>D = b0</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, 0)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(0, 1)</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>(0, 2)</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(1, 0)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(1, 1)</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(1, 2)</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>(2, 0)</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>(2, 1)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>(2, 2)</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>
Orthogonal Arrays (3)

- Orthogonal arrays can be constructed very fast and are always optimal
  - Any extra test will cause a pair to be covered for more than once

However, there are several limitations:
- Orthogonal arrays do not always exist
- Existing methods often require $|v|$ be a prime power and $k$ be less than $|v| + 1$.
- Every parameter must have the same number of values
- Every $t$-way interaction must be covered at the same number of times
Recursive Construction (1)

- Covering arrays are a more general structure, which requires every $t$-way interaction be covered at least once.

- Constructing a covering array from one or more covering arrays with smaller parameter sets.

- Recursive construction can be fast, but it also has restrictions on the number of parameters and the domain sizes.
Recursive Construction (2)

Use $OA(27; 4, 3, 3)$ and $OA(9; 4, 3, 2)$ to construct $CA(27; 8, 3, 3) = 27 + 9 + 9 = 45$

| 0000 | 01 01 01 01 |
| 0012 | 01 12 12 12 |
| 0021 | 02 20 20 20 |
| ... | 10 20 10 12 |
| 2212 | 12 20 20 01 |
| 2220 | 20 20 01 12 |

Double each column

0 -> 01
1 -> 12
2 -> 20

0 -> 02
1 -> 10
2 -> 21
Outline

- Introduction
- The IPO Strategy
- Related Work
- 3-Way Testing and Beyond
- Conclusion
Why beyond 2-way?

- Software failures may be caused by more than two parameters
  - A recent NIST study by Rick Kuhn indicates that failures can be triggered by interactions up to 6 parameters

- Increased coverage leads to a higher level of confidence
  - Safety-critical applications have very strict requirements on test coverage
The Challenges

- The number of tests may increase rapidly as the degree of interactions increases
  - Assume that each parameter has 10 values. Then, pairwise testing requires at least 100 tests, 3-way testing at least $10^3$ tests, 4-way testing at least $10^4$ tests.

- Test generation algorithms must be more sensitive in terms of both time and space requirements

- The need for test automation becomes even more serious
  - Impractical to manually execute and inspect the results of a large number of test runs
State-of-the-Art

- Both algebraic and computational methods can be extended to 3-way testing and beyond.
- However, algebraic methods have fundamental restrictions on the systems they can apply.
- Computational methods are more flexible, but none of them are optimized for n-way testing with $n > 2$. 
Opportunities (1)

Possible ideas to reduce the number of tests

- Domain partitioning - identify equivalence values of each parameter
- Parameter constraints - exclude combinations that are not meaningful from the domain semantics
- Fault-oriented test generation - only include combinations that may contribute to one or more specific classes of faults
- Test budget - maximize the coverage of n-way interactions within a given number of tests
Opportunities (2)

Possible ways to improve the test generation algorithms

- Combination of algebraic and computational methods,
  - e.g., computational methods can be used to compute a starter covering array and then recursive construction can be used to expand the array
Opportunities (3)

Possible ideas for test automation

- **Test harness** that can automate test setup, test execution, and test results evaluation
- Automatically generate **test oracles** from a high level specification or by integration with tools based on formal methods, e.g., model checkers
Outline

- Introduction
- The IPO Strategy
- Related Work
- 3-Way Testing and Beyond
- Conclusion
Conclusion

- The problem of combinatorial testing is well-defined and has been used widely in practice.
- The IPO strategy is deterministic, has a lower order of complexity, and still produces competitive results.
- Algebraic methods, if applicable, are fast and can be optimal, whereas computational methods are heuristic but very flexible.
- Going beyond 2-way testing presents challenges and opportunities to the area of combinatorial testing.