Test Generation from Combinatorial Designs

Need Balance Requirement?

- If once failed with an interaction, will fail again
  - No need to test “a pair” multiple times
  - Exceptional cases?
- If deterministic failures, relax balance requirements, and produce fewer test cases
- Covering arrays
- Mixed level covering arrays
- Variable strength covering array

Covering Array

Fixed-Level Covering Array

- Given a finite set S of s symbols each,
  - N x k matrix, denoted by \( CA(N, k, s, t) \)
  - Any N x t subarray contains each t-tuple \( \text{at least} \ \lambda \) times
  - Not a balanced design

Examples

\( OA(8,5,2,2) \) vs \( CA(6,5,2,2) \)

<table>
<thead>
<tr>
<th>Run</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\( CA(6,5,2,2) = \)

<table>
<thead>
<tr>
<th>Run</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Mixed-Level Covering Array

- Given a finite set of $S$ of $s_1, s_2, \ldots, s_p$ symbols,
- $N \times Q$ matrix, denoted by $MCA(N, s_1^{k_1}, s_2^{k_2}, \ldots, s_p^{k_p}, t)$
- Any $N \times t$ subarray contains each $t$-tuple at least once
- Total num. of factors $Q = \sum_{i=1}^{p} k_i$

Example
- Compare to $MA(12, 2^{3 \times 2}, 2)$

<table>
<thead>
<tr>
<th>Run</th>
<th>Size</th>
<th>Toppings</th>
<th>Address</th>
<th>Phone</th>
<th>MCA(12, 2^{3 \times 2})</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Variable Strength Covering Array

- Given a finite set of $S$ of $s_1, s_2, \ldots, s_p$ symbols,
- $N \times Q$ matrix, denoted by
- Any $N \times t$ subarray contains each $t$-tuple at least $\lambda$ times
- Total num. of factors $Q = \sum_{i=1}^{p} k_i$
- Covers higher interactions for a subset $T$ of symbols

<table>
<thead>
<tr>
<th>Config. No</th>
<th>Option A</th>
<th>Option B</th>
<th>Option C</th>
<th>Option D</th>
<th>Option E</th>
<th>Option F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Generating Mixed-Level Covering Array

- In-Parameter-Order (IPO) method by Lei and Tai (2002)
  - Greedy method
  - Input: $2^r$ factors, (different) levels for each factors
  - Output: MCA

Steps
- Initialization: generate all pairs from 2 factors
- Horizontal growth for a remaining factor
- Vertical growth for a remaining factor
- Repeat 2nd and 3rd step

Step 1 - Initialization

- An example
  - Three factors A, B, and C.
  - Levels for A: $\{a_1, a_2, a_3\}$. B has 2 levels and C has 3 levels

- Generate all runs (pairs) from two factors
  - Denote each run by $t_1 \sim t_6$
    $$T = \{(a_1, b_1), (a_1, b_2), (a_2, b_1), (a_2, b_2), (a_3, b_1), (a_3, b_2)\}$$

- The IPO procedure would terminate at this point if the number of parameters $n=2$. 
**Step 2 - Horizontal Growth**

- **Step 2.1 - Compute all pairs between factors in runs and consider a remaining factor (call it F)**
  - Pairs between A and C, and B and C
  - All pairs in the set AP need to be covered by runs
  $AP = \{(a_1, c_3), (a_2, c_3), (a_2, c_1), (a_2, c_2), (a_3, c_1), (a_3, c_2), (a_3, c_3)\}$
  $T = \{(a_1, b_1), (a_1, b_2), (a_2, b_1), (a_2, b_2), (a_3, b_1), (a_3, b_2)\}$

- **Step 2.2 - Extend |F| runs in T by adding each level of F**
  - $t_1 = (a_2, b_1, c_1)$, $t_2 = (a_1, b_2, c_3)$, and $t_3 = (a_2, b_2, c_3)$
  - $AP = \{(a_1, c_3), (a_2, c_3), (a_2, c_1), (a_2, c_2), (a_3, c_1), (a_3, c_2), (a_3, c_3), (b_1, c_2), (b_2, c_1), (b_2, c_3)\}$

**Step 3 - Vertical Growth**

- For each not-yet-covered pairs in AP, add a new run to T
  - "don’t-care" value is used for factors not in the tuples
  - $AP = \{(a_1, c_3), (a_2, c_2), (a_3, c_1)\}$
  - $T = \{(a_1, b_1, c_1), (a_1, b_2, c_2), (a_2, b_2, c_3), (a_2, b_1, c_3), (a_3, b_2, c_3), (a_3, b_1, c_2)\}$
  - $t_7 \leftarrow (a_1, b_2, c_3)$
  - $t_8 \leftarrow (a_2, b_1, c_3)$
  - $t_9 \leftarrow (a_3, b_1, c_3)$

**AETG (Automatic Efficient Test Generator)**

- **Step 2.3 – Extend each remaining run in T by adding the best level of F**
  - **Heuristic: choose the level that covers the most**
  - $AP = \{(a_1, c_3), (a_2, c_1), (a_2, c_2), (a_3, c_1), (a_3, c_2), (a_3, c_3)\}$
    - t4 = (a_2, b_2), t5 = (a_2, b_1), t6 = (a_3, b_2)
    - t4 ← (a_2, b_2, c_1)
    - t5 ← (a_2, b_1, c_3)
    - t6 ← (a_3, b_2, c_3)

- **Step 3 - Generate r+1th test case**
  - Generate M different candidate test cases and choose the best one that covers the max. uncovered pairs
    - Choose the parameter $f_i$ with the max. benefit
    - Generate a random order for $f_2 \ldots f_i$
    - To choose $f_i$ value, count new pairs between $f_1 \ldots f_i$
    - Best M?
Applications of Interaction Test Design

- Applicable in a variety of domains, as long as
  - Input space can be modeled in factors and levels
  - Output is represented as performance metrics
    - # of faults detected, lines covered, ...
- We will look at
  - Configuration-aware regression testing
  - Product line testing
  - Fault characterization
  - Discovering high-coverage configurations

Configuration-Aware Regression Testing

- Context
  - Software fault detection depending on configurations
  - Only one configuration for regression testing?
- Problem
  - How to improve early fault detection capability in regression testing of configurable software
- Challenges
  - Combinatorial explosion

Configuration-Aware Regression Testing

- Approach
  - Generate MCA from options and their values
  - Run test suites and measure block coverage (BC) and fault detection (FD) per configuration
  - Prioritize configurations using interaction benefit
    - **Iterative greedy process**
    - Classify configurations with added FD or BC
    - Compute normalized benefit of "levels" of factors in each class
    - Compute benefit of "pair s" in each configuration
    - Compute **added pair**-benefit by each configuration

<table>
<thead>
<tr>
<th>Block Matrix: $B_s$</th>
<th>Fault Matrix: $F_s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_1$</td>
<td>$f_1$</td>
</tr>
<tr>
<td>$b_2$</td>
<td>$f_2$</td>
</tr>
<tr>
<td>$b_3$</td>
<td>$f_3$</td>
</tr>
<tr>
<td>$b_4$</td>
<td>$f_4$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration Matrix: $CM_s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_1$</td>
</tr>
<tr>
<td>$c_1$</td>
</tr>
<tr>
<td>$c_2$</td>
</tr>
<tr>
<td>$c_3$</td>
</tr>
<tr>
<td>$c_4$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Block Matrix: $B_s$</th>
<th>Fault Matrix: $F_s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_1$</td>
<td>$f_1$</td>
</tr>
<tr>
<td>$b_2$</td>
<td>$f_2$</td>
</tr>
<tr>
<td>$b_3$</td>
<td>$f_3$</td>
</tr>
<tr>
<td>$b_4$</td>
<td>$f_4$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration Matrix: $CM_s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_1$</td>
</tr>
<tr>
<td>$c_1$</td>
</tr>
<tr>
<td>$c_2$</td>
</tr>
<tr>
<td>$c_3$</td>
</tr>
<tr>
<td>$c_4$</td>
</tr>
</tbody>
</table>
Configuration-Aware Regression Testing

- Experiment with `vim`
  - 146~187 user-configurable options for 8 versions
  - Generate MCA of strength 2
    - e.g., MCA(60, 2, 2^3\times7^4\times6^2\times10^1) for version 2, mapped to `vimrc`
    - Compared to 60 random configurations
    - Compared to the default configuration

- Findings
  - MCA configurations detect more (seeded) faults
  - Use of suitable configurations for regression testing affects fault detection significantly
  - Prioritization enables early fault detection

Fault density between configurations

Findings

MCA configurations detect more (seeded) faults
Use of suitable configurations for regression testing affects fault detection significantly
Prioritization enables early fault detection

Covering Array in Product Line Testing

- Context
  - Software Product Line (SPL): commonality and variability
  - Variability space

- Problem
  - How to validate software families instantiated from diverse selections from variability space?

- Challenges
  - Combinatorial explosion
  - Constraints in SPL
Covering Array in Product Line Testing

- **Approach**
  - Orthogonal Variability Model (OVM) for product line modeling
  - Variation points
  - Variants
  - Variability dependencies – mandatory, optional, alternative, ...
  - Correlated binding – requires, excludes, ...

  ![Orthogonal Variability Model](Image)

**Approach (Cont.)**

- Mapping the OVM onto a relational model
  - Define the base “unconstrained” relation for all product line instances
    - a VP as a set of factors
    - variants as values
    - different mapping rules for dependency types
  - Restrict the relation
    - build sub-relations for each constraint
    - intersect with the current base relation

- Relational model as a Covering array
  - Greedy tuple selection from the restricted relation may work
  - Cumulative test coverage can also be used for targeted testing

Fault Characterization in Configuration Space

- **Context**
  - Software systems are designed to be option-configurable
  - Faults appear only on some option configurations

- **Problem**
  - How to pinpoint options and their values by searching a sampled configuration space

- **Challenges**
  - Combinatorial explosion
Fault Characterization in Configuration Space

- **Approach**
  - Modeling compile-/run-time option configuration space
  - Generating MCA (or VSCA) from the space
  - Run test suites with configurations in a distributed way
  - Classify failures
    - Partition test results per option setting
    - Find best partitioning option
    - Add edges to each option setting
    - Repeat from step 2

<table>
<thead>
<tr>
<th>Config</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>PASS</td>
</tr>
<tr>
<td>0 0 1</td>
<td>PASS</td>
</tr>
<tr>
<td>0 0 2</td>
<td>ERR #1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>PASS</td>
</tr>
<tr>
<td>0 1 1</td>
<td>PASS</td>
</tr>
<tr>
<td>0 1 2</td>
<td>PASS</td>
</tr>
<tr>
<td>1 0 0</td>
<td>ERR #1</td>
</tr>
<tr>
<td>1 0 1</td>
<td>ERR #1</td>
</tr>
<tr>
<td>1 0 2</td>
<td>ERR #1</td>
</tr>
<tr>
<td>1 1 0</td>
<td>ERR #1</td>
</tr>
<tr>
<td>1 1 1</td>
<td>ERR #1</td>
</tr>
</tbody>
</table>

Findings
- Low-strength CA performance showed good performance compared to exhaustive testing
- VSCA can be a good cost-performance tradeoff (hotspot)
- Fault characterization enables developers to pinpoint failure causes quickly

Discovering High-Coverage Configuration

- **Context**
  - Software systems are often highly-configurable
  - Configurations cover regions in software differently
    - E.g., PHP script enable option in a Web server

- **Problem**
  - 2-way CA? 3-way CA? Otherwise, 7-way CA?
  - How to discover configurations that capture key (high-strength) interactions in a scalable and effective way?

- **Challenges**
  - In the worst case, must test against all configurations
Discovering High-Coverage Configuration

- Code snippet from vsftpd
  - 6 binary configuration options
  - 8 regions of interest
  - 639 option interactions

- Look at details
  - 8 real interactions
  - Multiple interactions with one configuration
  - Wide code coverage with low-strength interactions
  - Must choose configurations carefully for some regions

Approach
- Find best node in i-Tree (i.e., partial option settings)
- Adaptively generate 2-/3-way CAs with remaining options
- Run test suite on all configurations and record coverage
- Discover new interactions and append to i-Tree
  - configuration clustering and classification of option settings

Experiment with vsftpd, ngIRCd, and MySQL

- Findings
  - Effective configuration space is not a cross product of options, but more like a union of smaller configurations
  - Higher coverage was achievable at low cost
    - ~ ½ configurations needed for vsftpd, compared to random selection
    - Can cover code regions associated with many options

Wrap-Up
- Interaction test design is practical
  - Latin square and orthogonal array can be used for balanced test design
  - CA reduces test effort by relaxing the balance requirement
  - Low-strength CAs were empirically proven to be useful
  - A useful plug-in for new ideas in various SE topics