17-708 SOFTWARE PRODUCT LINES: CONCEPTS AND IMPLEMENTATION

QUALITY ASSURANCE: TESTING

CHRISTIAN KAESTNER
CARNEGIE MELLON UNIVERSITY
INSTITUTE FOR SOFTWARE RESEARCH
READING ASSIGNMENT NOV 16

Textbook, Chapter 10
Thaker et al. Safe Composition. GPCE 2007
LEARNING GOALS

Understand what can be tested and how tests can be reused in a product line

Develop conditional tests in domain engineering and perform testing in domain engineering, understanding the limitations

Select a suitable test strategy for a given project
Requirements-based SPL testing

Source: Klaus Pohl and Andreas Metzger. 2006. Software product line testing. *Commun. ACM* 49, 12 (December 2006), 78-81
PRINCIPLES FOR SPL SYSTEM TESTING

P-1: Preserve Variability in Domain Test Artifacts

P-2: Test Commonalities in Domain Engineering

P-3: Use Reference Applications to Determine Defects in Frequently Used Variants

P-4: Test Commonalities based on a Reference Application

P-5: Test Correct Variability Bindings

P-6: Reuse Application Test Artifacts across Different Applications

Source: Klaus Pohl and Andreas Metzger. 2006. Software product line testing. Commun. ACM 49, 12 (December 2006), 78-81
DOMAIN TESTING

Domain Unit Testing
Domain Integration Testing
Domain System Test

Variability in Test Artifacts
Domain Test Artefact Reuse

Application Test Coverage
Application-specific Tests

TEST STRATEGIES

Brute Force Strategy

Pure Application Strategy

Sample Application Strategy

Commonality and Reuse Strategy

- Domain testing aims at testing common parts and preparing test artefacts for variable parts. Application testing aims at reusing the test artefacts for common parts and reusing the predefined, variable domain test artefacts to test specific applications.

<table>
<thead>
<tr>
<th></th>
<th>Time to create</th>
<th>Absent variants</th>
<th>Early validation</th>
<th>Learning effort</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>(BFS)</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>(PAS)</td>
<td>0</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>SAS</td>
<td>0</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>CRS</td>
<td>+</td>
<td>+</td>
<td>0</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Combined SAS/CRS</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 1. Test automation for each test practice

yes, this is done by the user community.
yes, we address that very thoroughly.
yes, but in an ad-hoc manner.
yes, we have AUTOMATED tests for this.

**Figure 2. Cross-Product Integration Testing**

OPEN VS CLOSED WORLD
What's the Specification?

Typically **global** property $x$ for every program

Syntactically correct, well-typed

Absence of double-free vulnerabilities

Returns positive number for parameter 3

Terminates within 10 seconds

Challenge is checking all configurations
e.g., $\forall p \in PL: p \models x$
Brute-Force
Feature-Based Specifications

Property $x$ for every program with feature $f$

No access to the file system

Renders “[:weather:]” as

Challenge is checking many configurations

e.g., $\forall p \in PL: (f \in p) \Rightarrow (p \models x)$
Feature-Based Specification

\[ \text{AG} \ (\text{incoming(email e)} \ & \ e.\text{isEncrypted} \Rightarrow \ ((\text{outgoing(email e)} \Rightarrow e.\text{isEncrypted}) \ R \ \text{outgoing(email e)}) \]
Brute-Force

One Configuration

Sampling

Complete Analysis

Check random configurations
Check representative configurations
Check maximum configuration
Check config. used by customers
Combinatorial interaction testing (pairwise, etc)
Code coverage heuristics