Generic Feature-Based Composition

Tijs van der Storm
Centrum voor Wiskunde en Informatica
storm@cwi.nl
Introduction

- Me: Tijs van der Storm
- Phd Student, project Deliver:
  - Intelligent Knowledge Management for Software Delivery
- My focus: software configuration management
- This talk:
  - Relating “variability” to “variation”
  - Steps towards bridging
    - Problem domain (features)
    - Solution domain (artifacts)
Overview

- Example product line
- Goal of this work: automation of configuration
- Description of
  - Variability model
  - Composition model
  - Link between them
- Conclusions
Example product line

Diagram showing a UML design for a product line with elements such as TreeFactory, list, array, and Visitor, with interactions and relationships indicated.
Some Questions

- What are valid configurations?
- What are the variation points?
- How are they bound?
- How many variants are there?
Goal: automatic instantiation

- Why?
  - Configuration space can be large
  - Binding is manual/error-prone

- Automation requires:
  - Formal description of configuration space
  - Formal link to solution space artifacts
Configuration impedance mismatch

- Configuration based on features
  - Abstract
  - Declarative
  - Logical

- Composition based on artifacts
  - Concrete
  - Structural

- Never the twain will meet?
Plan

- Configuration interfaces
  - Feature diagrams/descriptions
- Composition model
  - Dependency graphs
- Relate both through common logical formalism (propositional formulae)
- Derive consistency and product instantiation
Variability model

- Feature models

```
Tree
  +-------+------+
  |   ↘   |
  |       |
  | Factory |
  +-------+------+
    ↗     +-------+------+
    |     |   ↘   |
    |     |       |
  list array Strategy
    ↗     +-------+------+
    |     |   ↘   |
    |     |       |
  top-down bottom-up
```
Translation to logic

- **Textual description of diagram (FDL)**
  - Tree: all(Factory, Visitors?)
    - Factory: one-of(list, array)
    - Visitors: all(Strategy, logging?)
    - Strategy: one-of(top-down, bottom-up)

- **Translation to boolean sentence**
  - (Tree -> Factory) & (Factory -> ((list & not(array))) | (not(list) & array))) &
    - (Visitors -> Strategy) & (Strategy ->
      ((top-down & not(bottom-up))) | (not(top-down)) & bottom-up))))
Composition model

- Artifact dependency graphs

```
list.TreeFactory
array.TreeFactory
list.Tree
array.Tree
Tree
list.Visitability
array.Visitability
Visitability
Visitor
Strategy
TopDown
BottomUp
Logging
```

“depends on”
Genericity

- Variation can be implemented in many ways
  - design patterns
  - aspect oriented programming (AOP)
  - build files
  - ...

- Dependency relation sufficiently generic
  - independent of programming language
  - works on files or directories
Translation of dependencies

- Nodes are atoms, edges implication
  - (list.TreeFactory -> (TreeFactory & list.Tree)) &
    (array.TreeFactory -> (TreeFactory & array.TreeTree)) &
    (TreeFactory -> Tree) & (list.Tree -> Tree) &
    (array.Tree -> Tree) & (Visitability -> Tree) & ...

- Follows from artifacts themselves...
  - Future work: derive this automatically
Merging both domains

- Two boolean propositions:
  - Configuration interface
  - Dependency relations
- Bridged using implications:
  - (List & Visitors -> list.Visitability) &
    (array & Visitors -> array.Visitability) &
    (list -> list.TreeFactory) &
    (array -> array.TreeFactory) &
    (top-down -> TopDown) &
    (bottom-up -> BottomUp) &
    (logging -> Logging)
Configuration = valuation

<table>
<thead>
<tr>
<th>Features</th>
<th>Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature</td>
<td>Boolean formula</td>
</tr>
<tr>
<td>Dependency</td>
<td>Implication</td>
</tr>
<tr>
<td>Featurename</td>
<td>Atom</td>
</tr>
<tr>
<td>Artifact</td>
<td>Atom</td>
</tr>
<tr>
<td>Configurability</td>
<td>Satisfiability</td>
</tr>
<tr>
<td>Configuration</td>
<td>Valuation</td>
</tr>
<tr>
<td>Validity</td>
<td>Satisfaction</td>
</tr>
</tbody>
</table>

Set of all valuations = set of all valid instantiations
Valid configurations for *Tree*

<table>
<thead>
<tr>
<th><strong>Atomic Features</strong></th>
<th><strong>Artifacts</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>Tree, TreeFactory, array.TreeFactory, array.Tree</td>
</tr>
<tr>
<td>array, top-down</td>
<td>..., array.Visitability, Visitability, Strategy, TopDown</td>
</tr>
<tr>
<td>array, bottom-up</td>
<td>..., array.Visitability, Visitability, Strategy, BottomUp</td>
</tr>
<tr>
<td>array, top-down, logging</td>
<td>..., array.Visitability, Visitability, Strategy, TopDown, Logging</td>
</tr>
<tr>
<td>array, bottom-up, logging</td>
<td>..., array.Visitability, Visitability, Strategy, BottomUp, Logging</td>
</tr>
<tr>
<td>list</td>
<td>Tree, TreeFactory, list.TreeFactory, list.Tree</td>
</tr>
<tr>
<td>list, top-down</td>
<td>..., list.Visitability, Visitability, Strategy, TopDown</td>
</tr>
<tr>
<td>list, bottom-up</td>
<td>..., list.Visitability, Visitability, Strategy, BottomUp</td>
</tr>
<tr>
<td>list, top-down, logging</td>
<td>..., list.Visitability, Visitability, Strategy, TopDown, Logging</td>
</tr>
<tr>
<td>list, bottom-up, logging</td>
<td>..., list.Visitability, Visitability, Strategy, BottomUp, Logging</td>
</tr>
</tbody>
</table>
Maintaining the mapping

- Mapping co-evolves with code base
- But, weak coupling
  - Map features to *essential* artifacts
  - Many artifacts induced by transitivity
  - Static checking to spot errors
Conclusions

- Bridge from problem domain to solution domain for automating configuration
- Combination of two models
  - Features
  - Dependencies
- Future work:
  - Tool support
  - Dealing with evolution
  - Extension with binding actions
Thank you

Questions?