Product Line Engineering:
Product Derivation in an MDA framework

Tewfik Ziadi
Jean-Marc Jézéquel
Frédéric Fondement
Aspect weaving is The unifying paradigm!

MDA is mainly useful in a PLE context.
Executive Summary

- The analysis model has identified the variants between products (including Platform specificities)
  - Reified as language-level classes (inheritance, …)
  - Decorated with OCL meta-level constraints
- Systematic use of the Abstract Factory pattern
  - To specify a product among the family
- Model Transformations (at the meta-model level) to automatically derive a product
  - Using OCL2
The 3 Dimensions of Software Configuration Management: [Estublier et al. 95]

The Variant dimension
- Handle environmental differences

The revision dimension
- Evolution over time

The concurrent activities dimension
- Many developers are authorized to modify the same configuration item

Even with the help of sophisticated tools, the complexity might be daunting

Try to simplify it by reifying the variants of an OO system
Traditional Approaches

Patch the executable

- Device drivers
  - Source level, link time, boot time, on demand at runtime
- Static configuration table
- Conditional Compilation / Runtime Tests

```c
If (language == french) {
    #ifdef MSW
        io_puts(0, "Bonjour", 7);
    #elseif TEXT
        printf("Bonjour\n");
    #endif
} else {
    #ifdef MSW
        io_puts(0, "Hello", 5);
    #elseif TEXT
        printf("Hello\n");
    #endif
```

- Static and Dynamic configuration information intermingled
- Hard to change your mind on what should be static or dynamic…
Basic Idea

Abstract the Intent

- `io.write_line(language.hello)`

Rely on Dynamic Binding for the Details

Uncouple the variations from the selection process

- **Automatically derive a product using OCL2 meta-model transformation**
Case Study: The Mercure Product Line

43,980,465,111,040 possible variants

![Diagram]

- Mercure
- Engine
- Net Driver
- Manager
- GUI
- Language
- LanguageCat
- Message

1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
Exemples of Constraints at the PLA Level

Inheritance constraint

- Optional classes in Product Line Architecture can be omitted in certain products so a non-optional class cannot inherit from an optional one.

- OCL expression (at the M2 level):
  
  ```
  context Generalization
  inv self.parent.isStereotyped("optional") implies self.child.isStereotyped("optional")
  ```

Dependency constraint

- Idem

  ```
  context Dependency
  inv self.supplier->exists( S:ModelElement | .isStereotyped("optional") ) implies self.client->forall( C:ModelElement | C.isStereotyped("optional") )
  ```
Presence constraint.

- To express in a specific SPLA that the presence of the optional class C1 requires the presence of C2, we add the following OCL meta-model constraint.

  ```
  context Model
  inv presenceClass('C1') implies presenceClass('C2')
  ```

Mutual Exclusion constraint.

- To express in a specific SPLA that two optional classes cannot be present in the same Product, we add the following OCL meta-model constraint.

  ```
  context Model
  inv (presenceClass('C1') implies not presenceClass('C2'))
  and (presenceClass('C2') implies not presenceClass('C1'))
  ```
Reifying the Variants

Mercure Factory

Mercure

FullMercure
MinMercure
CustomMercure

<<Abstract>>
call_back
makes
1
1
1
1

<<GUI1>> <<GUI2>> new_gui() : GUI
<<Language2-1>> new_language() : Language
<<Manager1>> new_network_manager() : Manager
<<NetDriver1>> <<NetDriver2>> new_netdriver() : Net Driver
<<Engine1>> new_engine() : Engine
Dynamic Configuration

- aMercureConcreteFactory
  - make(Current)
    - Mercure
      - read()
        - aDynamicConfiguration
          - get_gui_type()
          - get_language_types()
          - get_manager_type()
          - get_netdriver_types()
          - get_engine_type()
          - run()

- GuiX
- LanguageX
- LanguageY
- ManagerX
- NetDriverX
- NetDriverY
- EngineX

Mercure

1112/09/2003 – 14h35
WP3 – Task 3.3
By limiting the range of variants available from a given Concrete Factory:

- Generate specialized code for the product
  - When only one *living* class for an abstract varying part:
    - Direct use of the relevant Concrete Class =>
    - Dynamic binding replaced with direct call (and even in lining)
  - When more than one *living* class
    - Dynamic binding (or replaced by *if then ... else*)

Implemented in SCM context using compilation – GNU SmallEiffel


Using OCL2 & UMLAUT framework
forall op in Operation.allInstances() {

-- The returned type of the operation determines the used variants
Class opReturnType :=
    (op.parameter->select(p:Parameter|p.kind = #return)).type

if opReturnType.isAbstract
then
    -- For multiple choice, we use stereotypes to specify the
    -- selected variants
    forall st in op.stereotype {
        selectVariant(st.name)
    }
else
    -- Here, we directly get the selected variant
    selectVariant(opReturnType.name)
endif
}
Class Diagram Handling

CustomMercure

<<GUI1>> <<GUI2>> new_gui() : GUI
<<Language2-1>> new_language() : Language
<<Manager1>> new_network_manager() : Manager
<<NetDriver1>> <<NetDriver2>> new_netdriver() : Net Driver
<<Engine1>> new_engine() : Engine

Mercure

Engine

Net Driver

Manager

GUI

Language

Message

1..* buffers

1 observe

1..2 +available

+Use

LanguageCat 1

Language 1-1

<<optional>> Language 1-N

<<optional>> Language 1-N

<<optional>> Language Cat 2

<<optional>> Language 2-1

<<optional>> Language 2-N

<<optional>> Language 2-N
Class Diagram Handling

Mercure

Net Driver

Manager 1

GUI

Language

Net Driver 1

Net Driver 2

Message

Language 1-1

Language 2-1

GUI 1

GUI 2

Manager 1

CustomMercure

<<GUI1>> <<GUI2>> new_gui() : GUI

<<Language2-1>> new_language() : Language

<<Manager1>> new_network_manager() : Manager

<<NetDriver1>> <<NetDriver2>> new_netdriver() : Net Driver

<<Engine1>> new_engine() : Engine
Model of PIM and Model of Transformation side by side on the CASE tool

Use a meta-level OCL2 interpreter/compiler
Our UMLAUT New Generation

GUI

XMI Parser

UML Model

UML/OCL2 Meta-Programs

Interpreter

UML Model

Java/Eiffel/C# AST

Compilers

XMI Generator

Compiler

XML

XSLT

XML

XMI Parser

Java/C#/etc. executable

Java/Eiffel/C# Source Code
Conclusion

- Method to uncouple the variations (reified as language-level objects) from the selection process
  - Based on the use of Creational Design Patterns
    - Abstract Factory
- All static configuration issues kept encapsulated in the Concrete Factory
- Model transformations with OCL2 makes it attractive
  - Experimented with GNU SmallEiffel compiler
  - Generalize the idea to UML using OCL2
  - “Constraint preserving” transformations
Derivation algorithm

- Pseudo-code
Derivation algorithm

Input: \( \text{PL}_\text{model}: \text{Model} \)
\( \text{aConcreteFactory}: \text{Class} \)

Output: \( \text{Product}_\text{model}: \text{Model} \)

-- Optional elements selection
Initiate selectedVariantsList to empty;

\[
\text{for each factory method in } \text{aConcreteFactory} \text{ do }
\]
\[
\text{initiate definedVariantsList to significant stereotypes of the factory; }
\]
\[
\text{if definedVariantsList is empty }
\]
\[
\text{then selectedVariantsList.add( all sub classes of the returned type); }
\]
\[
\text{else }
\]
\[
\text{selectedVariantsList.add(definedVariantsList) ; }
\]
\[
\text{endif}
\]
\[
\text{done}
\]

-- Model specialization
\[
\text{for each optional class C in } \text{PL}_\text{model} \text{ do }
\]
\[
\text{if (the class name of C not in selectedVariantsList) and ( names of all sub classes of C not in selectedVariantsList) }
\]
\[
\text{then }
\]
\[
\text{delete the class C from the } \text{PL}_\text{model;}
\]
\[
\text{endif}
\]
\[
\text{done}
\]

-- Model optimization
\[
\text{delete all other factories;}
\]
\[
\text{optimize inheritance;}
\]
\[
\text{Product}_\text{model} := \text{PL}_\text{model;}
\]