Query/Views/Transformations

An introduction to the MOF 2.0 QVT standard with focus on the Operational Mappings

Ivan Kurtev

ATLAS group, INRIA & University of Nantes, France
http://www.sciences.univ-nantes.fr/lina/atl/

Context of this work

- The present courseware has been elaborated in the context of the MODELWARE European IST FP6 project (http://www.modelware-ist.org/).
- Co-funded by the European Commission, the MODELWARE project involves 19 partners from 8 European countries. MODELWARE aims to improve software productivity by capitalizing on techniques known as Model-Driven Development (MDD).
- To achieve the goal of large-scale adoption of these MDD techniques, MODELWARE promotes the idea of a collaborative development of courseware dedicated to this domain.
- The MDD courseware provided here with the status of open source software is produced under the EPL 1.0 license.

Prerequisites

To be able to understand this lecture, a reader should be familiar with the following concepts, languages, and standards:

- Model Driven Engineering (MDE)
- The role of model transformations in MDE
- UML
- OCL
- MOF
- Basic programming concepts

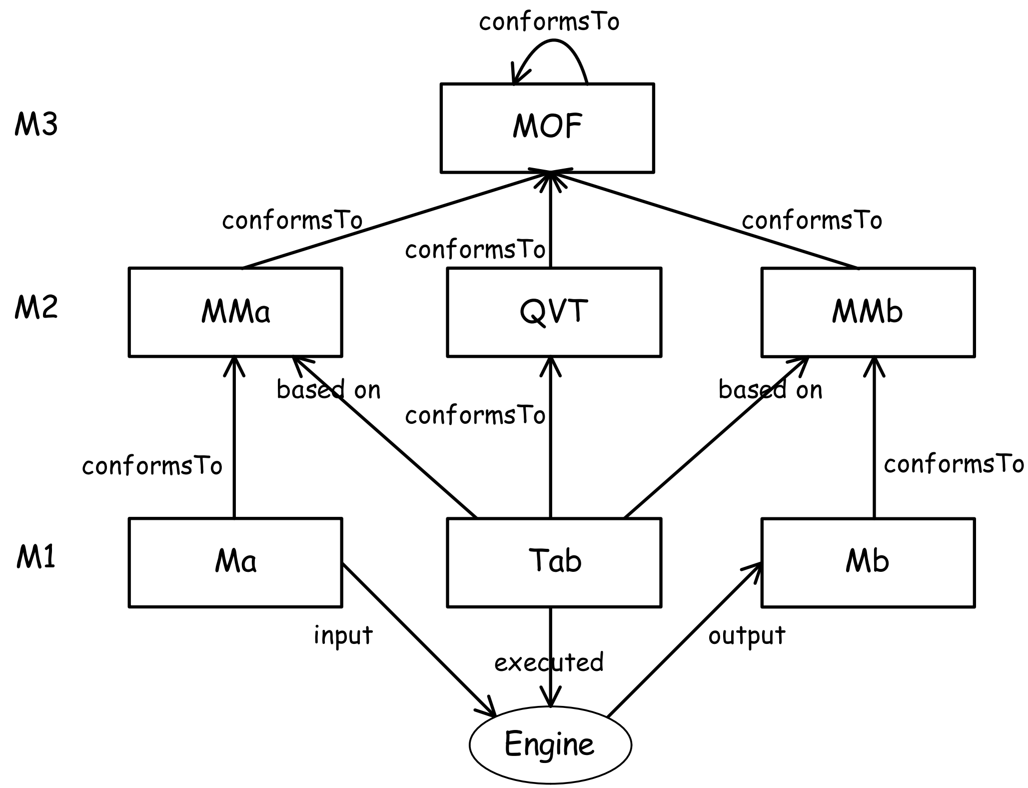
Outline

- Overview of QVT
- QVT Requirements
- QVT Languages:
  - Relations
  - Core
- Presentation of Operational Mappings
  - Case Study: Flattening UML Inheritance Hierarchies
  - Basic Language Constructs
- Conclusions

Overview

- **QVT** stands for **Q**uery/**V**iews/**T**ransformations
- OMG standard language for expressing queries, views, and transformations on MOF models
- OMG QVT Request for Proposals (QVT RFP, ad/02-04-10) issued in 2002
- Seven initial submissions that converged to a common proposal
- Current status (June, 2006): final adopted specification, OMG document ptc/05-11-01

QVT Operational Context



- Abstract syntax of the language is defined as MOF 2.0 metamodel
- Transformations (Tab) are defined on the base of MOF 2.0 metamodels (MMa, MMb)
- Transformations are executed on instances of MOF 2.0 metamodels (Ma)

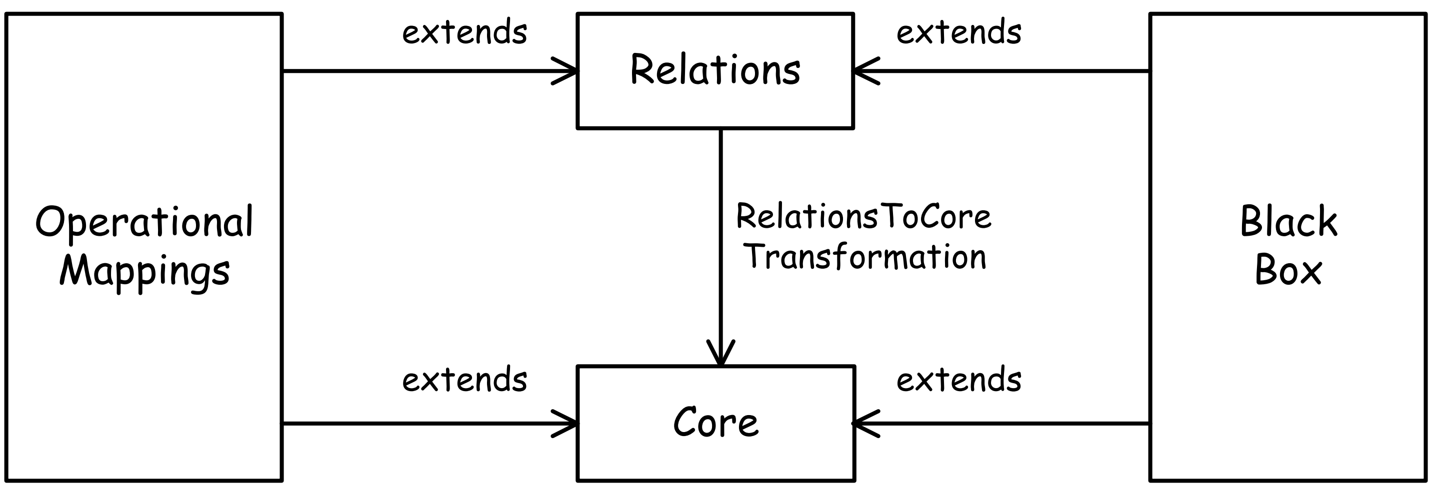
Requirements for QVT Language

Some requirements formulated in the QVT RFP

Mandatory requirements	
Query language	Proposals shall define a language for querying models
Transformation language	Proposals shall define a language for transformation definitions
Abstract syntax	The abstract syntax of the QVT languages shall be described as MOF 2.0 metamodel
Paradigm	The transformation definition language shall be declarative
Input and output	All the mechanisms defined by proposals shall operate on models instances of MOF 2.0 metamodels
Optional requirements	
Directionality	Proposals may support transformation definitions that can be executed in two directions
Traceability	Proposals may support traceability between source and target model elements
Reusability	Proposals may support mechanisms for reuse of transformation definitions
Model update	Proposals may support execution of transformations that update an

QVT Architecture

- Layered architecture with three transformation languages:
  - Relations
  - Core
  - Operational Mappings
- Black Box is a mechanism for calling external programs during transformation execution



Conformance Points for QVT Tools

	Interoperability Dimension				
Language Dimension		Syntax Executable	XMI Executable	Syntax Exportable	XMI Exportable
	Core				
	Relations				
	Operational Mappings				

- Language dimension indicates the language a tool may execute
- Interoperability dimension indicates the syntax that a tool can read
- 12 possible conformance points

! Note: Conformance to QVT is defined for tools and not for languages. The term "QVT compliant language" is not defined in the specification.

QVT Languages

- Relations
  - Declarative transformation language
  - Specification of relations over model elements
- Core
  - Declarative transformation language
  - Simplification of Relations language
- Operational Mappings
  - Imperative transformation language
  - Extends Relations language with imperative constructs

QVT is a set of three languages that collectively provide a hybrid "language".

Overview of Relations Language

- Declarative language based on relations defined on model elements in metamodels
- Object patterns that may be matched and instantiated
- Automatic handling of traceability links
- Transformations are potentially multidirectional
- Supported execution scenarios:
  - Check-only: verifies if given models are related in a certain way
  - Unidirectional transformations
  - Multidirectional transformations
  - Incremental update of existing models

Overview of Core Language

- Declarative language based on relations defined on model elements in metamodels
- Simpler object patterns
- Manual handling of traceability links
- Equal expressivity compared to the Relations language
- More verbose than the Relations language
- Core and Relations support the same set of execution scenarios
- Usage options:
  - Simple transformation language
  - Reference point for defining the semantics of the Relations language

Outline

- Overview of QVT
- QVT Requirements
- QVT Languages:
  - Relations
  - Core
- Presentation of Operational Mappings
  - Case Study: Flattening UML Inheritance Hierarchies
  - Basic Language Constructs
- Conclusions

Operational Mappings Language

This lecture presents Operational Mappings in details based on an example case study

- Case study: Flattening UML class hierarchies
- Overall transformation structure
- Mapping rules
- Querying source models
- Object resolution operations
- Creating target objects
- Flow of control
- Other features

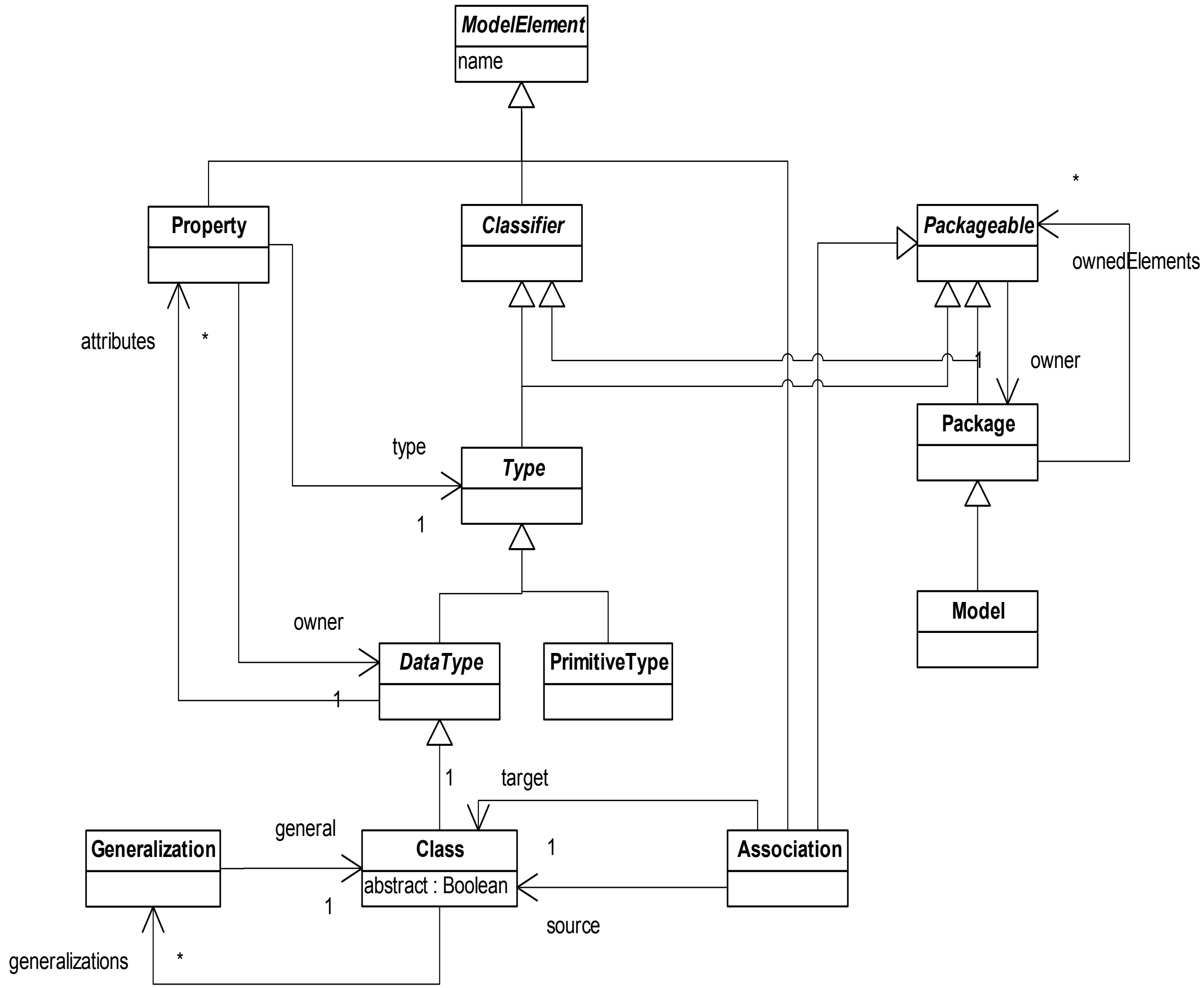
Case Study

Flattening UML class hierarchies: given a source UML model transform it to another UML model in which only the leaf classes (classes not extended by other classes) in inheritance hierarchies are kept.

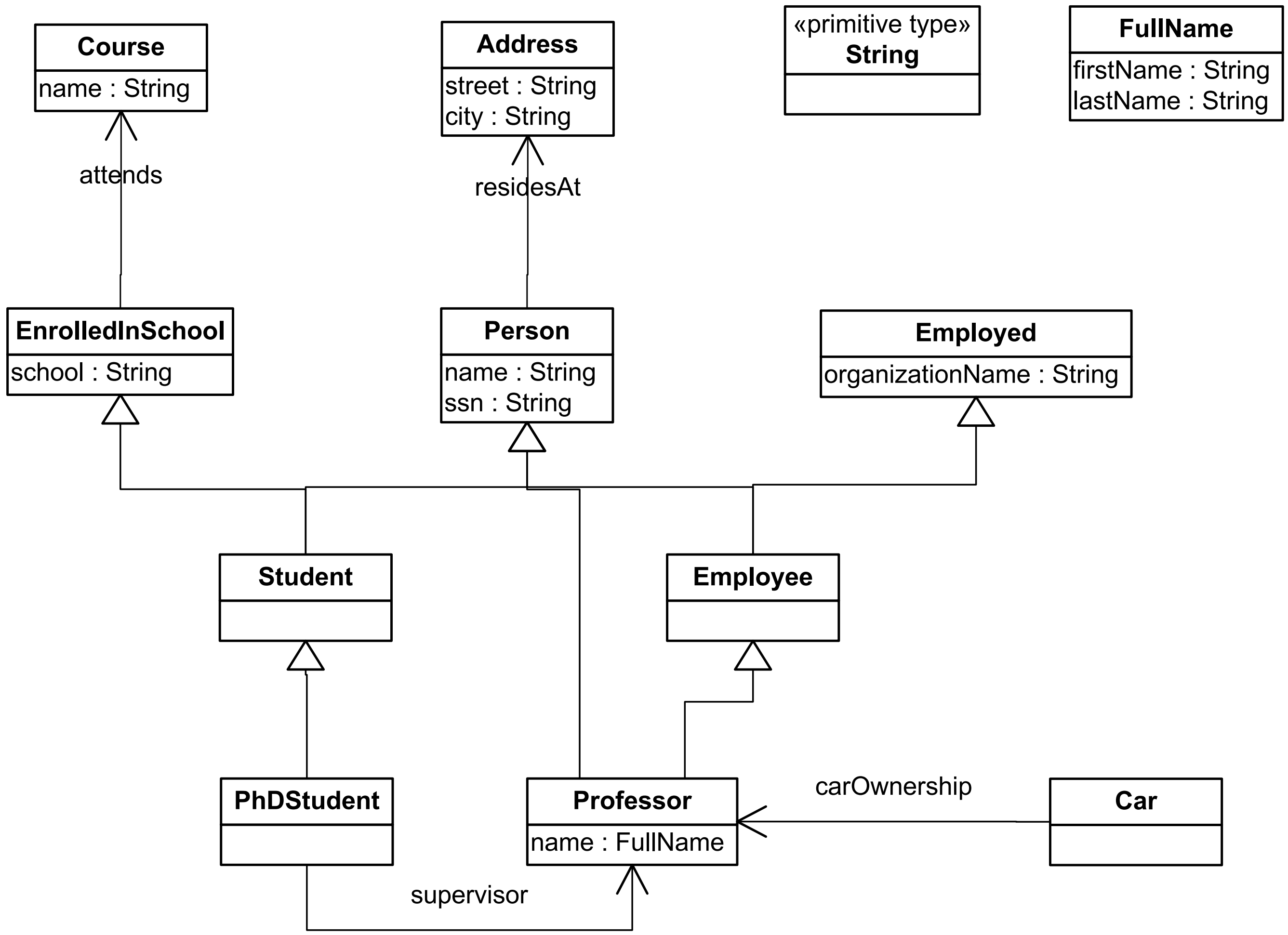
Rules:

- Transform only the leaf classes in the source model
- Include the inherited attributes and associations
- Attributes with the same name override the inherited attributes
- Copy the primitive types

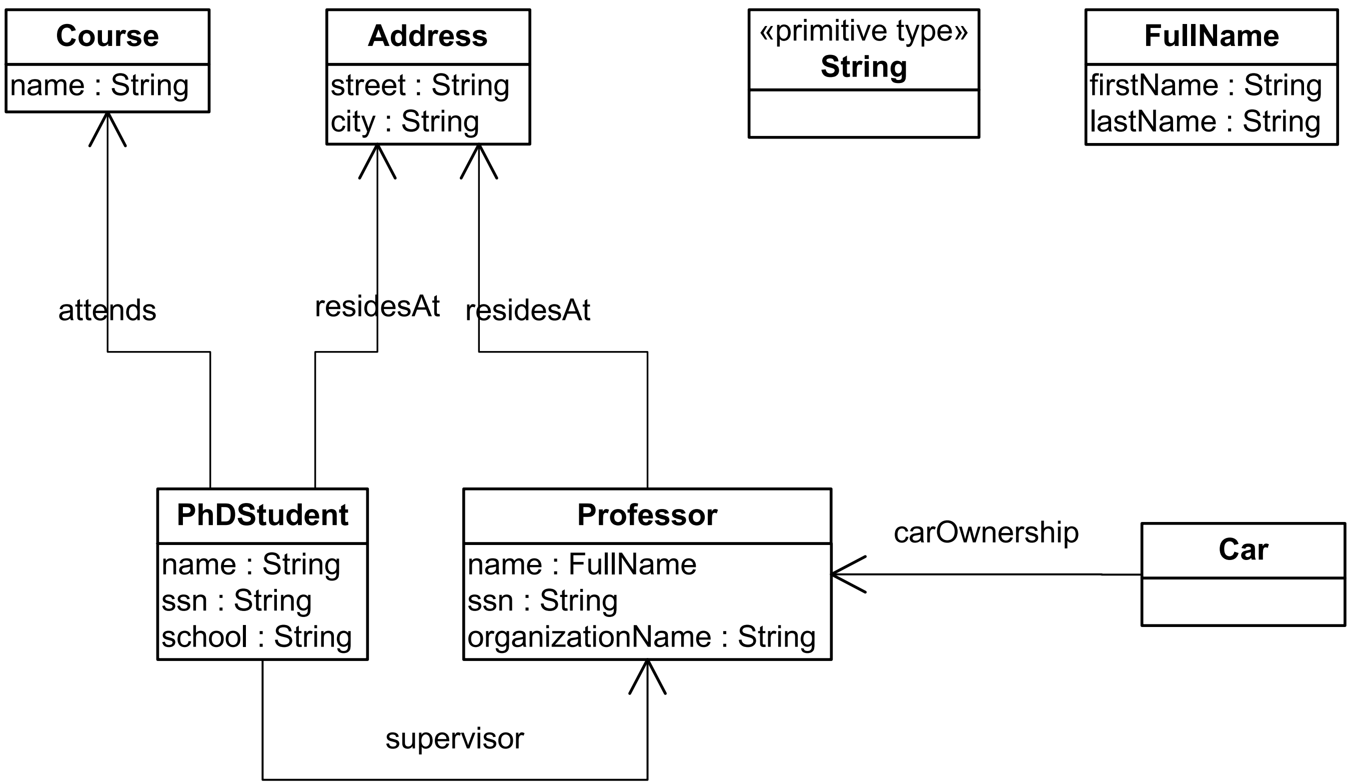
Source and Target Metamodel: SimpleUML



Example Input Model



Example Output Model



Model Transformation expressed in Operational Mappings Language

Overall structure of a transformation program:

```
transformation SimpleUML2FlattenSimpleUML(in source : SimpleUML
                                          out target : SimpleUML);
…………………………………………………………………

main() {}

…………………………………………………………………
…helpers………………………………………………
…mapping operations……………
```

Signature:
Declares the transformation name and the source and target metamodels.

in and **out** keywords indicate source and target model variables.

Entry point:
The execution of the transformation starts here by executing the operations in the body of **main**

Transformation elements:
Transformation consists of mapping operations and helpers. They form the transformation logic.

Mapping Operations

- A mapping operation maps one or more source elements into one or more target elements
- Always unidirectional
- Selects source elements on the base of a type and a Boolean condition (guard)
- Executes operations in its body to create target elements
- May invoke other mapping operations and may be invoked
- Mapping operations may be related by inheritance

Mapping Operations: Example (1)

- Consider the rule that transforms only leaf classes
  - Selects only classes without subclasses
  - Collects all the inherited properties
  - Creates new class in the target model

Signature and guard

```
mapping Class::leafClass2Class(in model : Model) : Class
   when {not model.allInstances(Generalization)->exists(g |
         g.general = self)}
{
  name:= self.name;
  abstract:= self.abstract;
  attributes:=
        self.derivedAttributes()->map property2property(self)->asOrderedSet();
}
```

Operation body

Mapping Operations: Example (2)

Operation Signature and Guard

An optional sequence of input/output parameters

Source element type

Operation name

Target element type

```
mapping Class::leafClass2Class(in model : Model) : Class
   when {not model.allInstances(Generalization)->exists(g |
         g.general = self)}
```

Guard

The Guard is an OCL expression used to filter source elements of a given type. The mapping operation is executed only on elements for which the guard expression is evaluated to true.

Mapping Operations: Example (3)

Operation Body

The predefined variable **self** refers to the source element on which the operation is executed

The keyword **map** is used to invoke another mapping operation named property2property over the elements returned by the helper derivedAttributes

```
name:= self.name;
abstract:= self.abstract;
attributes:=
    self.derivedAttributes()->map property2property(self)->asOrderedSet();
```

The left-hand side of the assignments denotes properties of the target element

Invocation of helper derivedAttributes

The mapping operation body contains initialization expressions for the properties of the target element. When an operation is executed over a source element the self variable is bound to it and an instance of the target type is created. Then the operation body is executed.

Helpers

- Helpers are operations associated to a type that return a result
- Both primitive and model types can be used
- Helpers may be used to perform complex navigations over source models
- Helpers have:
  - List of input parameters
  - An executable body
- Helper types:
  - Side-effect free: Query helper;
  - With side effect over input parameters: Helper

Helpers: Example

The query derivedAttributes is a side-effect free helper defined on classes. It returns an ordered set of properties that contains the attributes defined in a given class and the attributes derived from the its super classes. Derived attributes are overridden by the defined attributes with the same name. Note that this is a recursive helper. The variable self refers to the class on which the helper is executed

The context type of the helper derivedAttributes

Result type

```
query Class::derivedAttributes() : OrderedSet(Property){
  if self.generalizations->isEmpty() then  self.attributes
  else
   self.attributes->union(
        self.generalizations->collect(g |
          g.general.derivedAttributes()->select(attr |
            not self.attributes->exists(att | att.name = attr.name)
          )
        )->flatten()
   )->asOrderedSet()
  endif
}
```

Invoking Mapping Operations

Assume we have a mapping operation property2property that simply copies a property of a source class to a property of the target class. The target class is already created by the previously shown rule leafClass2Class.

In order to invoke property2property on every attribute of the source class we use the notation "->**map**". It implies an iteration over a list of source elements.

The notation "object.**map**" invokes a mapping operation on object as a source element.

Invocation of property2property on every member of the set returned by derivedAttributes query

```
……………………………………………………
attributes:=
    self.derivedAttributes()->map property2property(self)->asOrderedSet();
……………………………………………………
```

```
mapping Property::property2property(in ownerClass : Class) : Property{
  name:= self.name;
  type:= self.type;
  owner:= ownerClass;
}
```

Resolving Object References

Assume that we write a mapping operation that transforms associations in the source model to associations in the target model. In the target model an association relates two classes derived from other two classes in the source model. To identify these two classes in the target model the transformation engine maintains links among source and target model elements. These links are used for resolving object references from source to target model elements and back.

An example of a mapping operation that transforms associations and uses resolution of object references:

```
mapping Association::copyAssociation(sourceClass : Class) : Association {
   name:=self.name;
   source:=sourceClass.resolveByRule('leafClass2Class', Class)->first();
   target:= self.target.resolveByRule('leafClass2Class', Class)->first();
}
```

resolveByRule is an operation that looks for model elements of a given type (Class) in the target model derived from a source element by applying a given rule (leafClass2Class).

General Structure of Mapping Operations

```
mapping Type::operationName(((in|out|inout) pName : pType)* ) : (rName : rType)+
  when {guardExpression}
{
    init {}

    population {}

    end {}
}
```

init section contains code executed before the instantiation of the declared result elements

There exists an implicit instantiation section that creates all the output parameters not created in the init section. The trace links are created in the instantiation section.

population section contains code that sets the values or the result and the parameters declared as **out** or **inout**. The **population** keyword may be skipped. Population section is the default section in the operation body.

end section contains code executed before exiting the operation.

Object Creation and Population

Apart from the implicit creation of objects in the instantiation section there is an operation for creating and populating objects in mapping operations

Operation **object**:

Variable name and result type

```
object p : Property {
  name:= self.name;
  type:= self.type;
  owner:= ownerClass;
}
```

Population or property values

Imperative Constructs for Managing the Flow of Control

Operational Mappings is an imperative language. While many algorithms may be implemented just by a set of mapping operations that invoke each other and are supported by OCL expressions for navigation and iteration, there are cases where more sophisticated control flow is needed. The following imperative constructs are available:

- Compute
- While
- forEach
- Break
- Continue
- If-then-else

Features not Covered in the Lecture

- Packaging facilities:
  - transformation libraries
  - reuse of libraries
- Reuse facilities:
  - rule inheritance and merging
  - disjunctions of mapping operations
- Constructor operations
- Intermediate data
- Reusing and extending transformations
- Operation post condition: where clause

For more details consult the QVT specification: OMG document ptc/05-11-01

Conclusions (1)

- QVT: Query/Views/Transformations – the OMG standard language for model transformations in MDA/MDE;
- The issue of Views over models is not addressed;
- Query language based on OCL;
- A family of three transformation languages:
  - Relations: declarative language
  - Core: declarative language, simplification of Relations;
  - Operational Mappings: imperative transformation language that extends relations;
- Collectively QVT languages form a hybrid language;

Conclusions (2)

- Tool support is still insufficient (at the time of preparing of this lecture – June 2006);
- QVT is not proved yet in non-trivial industrial like scenarios;
- Many issues need further exploration:
  - Performance;
  - Testing;
  - Scalability of transformations;
  - Ease of use;
  - Handling change propagation;
  - Incremental transformations;
  - Adequacy of the reuse mechanisms;

Additional Materials

This lecture is accompanied by an implementation of the presented case study. The code, the documentation, and the example models are available from the MODELWARE web site.