

Großer Beleg  
XML Query Languages for Repositories Based  
on XML Documents

Ralf Wiebicke

April 2000

## Copyright

Copyright © 2000 Ralf Wiebicke.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is available at <http://www.gnu.org/copyleft/fdl.html>.

The OCL type information component developed together with this paper is Copyright © 2000 Ralf Wiebicke and published under the GNU Lesser General Public License.

## Availability

This document is available at <http://rw7.de/ralf/gbeleg99/intro.html> in several electronic forms including L<sup>A</sup>T<sub>E</sub>X, PostScript, PDF, Html and the original kLyx version.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Foundations and State-of-the-Art</b>	<b>4</b>
2.1	XML Query Languages	4
2.1.1	Overview of proposals	4
2.1.2	Requirements	5
2.2	XML Metadata Interchange (XMI)	5
<b>3</b>	<b>Type Information Component for OCL</b>	<b>7</b>
3.1	Analysis	7
3.1.1	Interface	7
3.1.2	Example	9
3.1.3	Dissolving Associations	11
3.1.4	Attribute Ambiguities	12
3.1.5	Operation Matching Ambiguities	13
3.2	Design	14
3.2.1	Type Information Component	14
3.2.2	XMI Parser	16
3.3	Implementation	18
3.3.1	Requirements	18
3.3.2	Operations with Side Effects	18
3.4	Tests	18
3.4.1	Tests Using the Royloy Example Model	18
3.4.2	Stress Test for the Type Information Component.	19
<b>4</b>	<b>Resulting Requirements</b>	<b>22</b>
4.1	Order of Instance Nodes	22
4.2	Transitive Closure	24
4.3	Summary	25
<b>5</b>	<b>Evaluation of XSLT</b>	<b>26</b>
5.1	Adapting XSLT to Java	26
5.2	Order of Instance Nodes	27
5.2.1	Parameter Polymorphism.	28

5.3	Transitive Closure . . . . .	28
5.3.1	Algorithm . . . . .	28
5.3.2	Implementation . . . . .	29
5.3.3	Test . . . . .	30
<b>6</b>	<b>Summary</b>	<b>32</b>
<b>A</b>	<b>Selected XSLT Scripts</b>	<b>34</b>
A.1	navigateParameterized.xsl . . . . .	34
A.2	navigateParameterized.xsl.result . . . . .	36
A.3	conformsTo.xsl . . . . .	37
A.4	conformsTo.xsl.result . . . . .	40

# Chapter 1

## Introduction

Looking at software engineering today, there is a strong emphasis on standards. When advertising for CASE tools, the best thing is to mention as often as possible those magic three letter words: XML, UML, MOF, XMI, XSL, RDF, DOM, SAX etc. So for the moment I will reformulate the title of the paper a bit:

XML Query Languages Like XSL for UML Models Based on  
XMI.

Four magic words, not bad.

This consciousness for standards is not by accident. Standards make CASE components work together. And there are many different components, that should work together: Design tools, browsers, report generators, reengineering tools and code generators, and all these running on top of a common repository.

Query languages are just another natural extension of this trend. Not only the repository is standardized by XML and XMI, even the method of accessing the repository is standardized on a level much higher than DOM. Since all CASE components rely on the repository, all components can benefit from a powerful, flexible access through a 4th generation language.

This paper tries to employ a XML query language for a typical task in software engineering: code generation. An OCL compiler [FF] generates java code for dynamically verifying invariants and pre/postconditions, and the query language provides access to the model information needed by the compiler.

## Chapter 2

# Foundations and State-of-the-Art

This chapter gives an overview of the technologies used in this report, as far as necessary for understanding.

### 2.1 XML Query Languages

XML is becoming the method of choice for representing structured information in a formalized way. However, it needs a convenient way to query and manipulate XML structures. It needs an equivalent to SQL.

#### 2.1.1 Overview of proposals

Several XML query languages have been proposed. I will give a short overview of some proposals and their relation to this paper.

**XML-QL.** XML-QL was designed at AT&T Labs. It's an extension of SQL with an additional CONSTRUCT clause for generating XML fragments. The current specification is available at [XMLQL]. XML-QL was the first language I examined. I found some serious lack of expressive power, which is described in chapter 4.

**XSLT.** XSLT is a proposal of the W3C available at [XSLT]. It's intended to translate arbitrary XML documents into Formatting Objects, a special XML instance designed to be rendered on screen. With XSLT I was able to solve the problems encountered on XML-QL. See chapter 5 for details.

**XQL.** According to [BC99], XQL is an extension to XSL pattern syntax, with a reduced expressive power. Therefore I didn't evaluate XQL any further. A draft is available at [XQL].

### 2.1.2 Requirements

[BC99] lists a comprehensive set of requirements for query languages. This section outlines the requirements I found being related to this paper.

**Joins.** A join compares two or more XML attributes or data. A typical comparison is the equality. [BC99] points out, that XML-QL supports joins, while XSLT does not. In this paper the join of XML-QL is solely used for the quite unsatisfying implementation of the transitive closure (see section 4.2). XSLT solves this problem much better with recursion.

**Matching of partially specified expressions with cyclic data.** With cyclic data it is possible to run the query processor into an infinite loop. This criterion examines, whether the language processor must detect such a situation and abort gracefully. [BC99] correctly points out, that XSLT does not support such a detection. This is concerns the script in A.3. If the generalization relation contains a cycle, this causes the script to loop infinitely. The XML-QL specification leaves this property unspecified.

**Querying the instance order.** Querying for XML elements appearing in a given order in the document. [BC99] claims, that XML-QL cannot do that. However, I found a construction suitable for this, as explained in 4.1, although this is not implemented in the prototype. XSLT and its prototype LotusXSL support instance order.

## 2.2 XML Metadata Interchange (XMI)

Roughly speaking, XMI is a XML representation for UML. But the details are a bit more complex:

**Meta Object Facility (MOF).** MOF is a standard for describing meta information. In contrast to UML it is much simpler. However, UML is described in MOF, so UML is a special MOF instance. This makes it possible to describe UML with MOF formally, which is quite smart.

**XML Metadata Interchange (XMI).** XMI is a method to represent a MOF instance in XML. Since UML is a (the) instance of MOF, XMI is a (the) method to represent UML in XML.

However, the concept has some drawbacks. The definition of XMI designed for more than just UML, it's designed for any meta model, which can be described in MOF. This makes XMI more complex. A XML representation especially for UML could be simpler, easier to read for humans and less tedious to parse.

Let me explain this on an example. Given the task to represent an operation with its parameters in XML, most results would probably look like this:

```

<Operation>
  <Parameter>...</Parameter>
  <Parameter>...</Parameter>
</Operation>

```

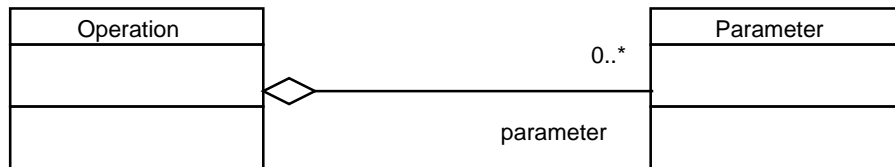
Reality in XMI is a bit more gossipy: (additions put in italics)

```

<Operation>
  <parameter>
    <Parameter>...</Parameter>
  </parameter>
  <parameter>
    <Parameter>...</Parameter>
  </parameter>
</Operation>

```

The definition of operations in MOF looks something like this:



The designer of UML knows, that “parameter” is the only way, to reach “Parameter” from “Operation”. But the designer of MOF and XMI cannot rely on this. There could be another association between “Operation” and “Parameter”. Therefore all `<Parameter>` elements must be enclosed by `<parameter>` elements, no matter how unneccessary it looks.

## Chapter 3

# Type Information Component for OCL

This chapter describes the problem I solved for verification of XML-QL. It describes the problem in detail, so the reader can understand the difficulties I found during implementation.

### 3.1 Analysis

The type information component is a part of the OCL compiler developed by Frank Finger. The compiler does some type checking on OCL constraints. Therefore it needs information from the model. The compiler supplies a java interface, which the type information component has to implement. Up to this point, there was only an implementation featuring a hard-wired example model. The component developed here enables the OCL compiler to process OCL constraints on arbitrary models given in XMI.

#### 3.1.1 Interface

This is the interface, as given by the OCL compiler. For further details see [FF] package `tudresden.oc1.check.types`.

**ModelFacade** represents a whole model, with all its classes, datatypes and relationships. The information is extracted from a XMI file.

```
interface ModelFacade
{
    Any getClassifier(String name);
}
```

**getClassifier** retrieves an object from the model, representing a class/datatype with the given name. This object is subject to further queries, cover-

ing the details of this class. By implementing the interface `Any`, the object makes these queries available.

**Any** specializes `Type`, that it is a non-collection type. All methods of this interface are inherited from `Type`.

```
interface Any extends Type
{
}
```

Collections use `Any` as their element type, since this must be a non-collection type. (Nested collections are automatically flattened in OCL.)

**Type** represents a arbitrary type of the type information model. It may be a class, a basic datatype or a collection of those.

```
interface Type
{
    boolean conformsTo(Type type);
    Type navigateQualified(String name, Type[] qualifiers);
    Type navigateParameterized(String name, Type[] params);
}
```

**conformsTo** checks, whether the type can be an instance of a formal parameter of the given type. For classes this is the transitive closure of the generalization relation being extended to be reflexive.

**navigateQualified** queries the class for a structural feature (an attribute or an association partner by its rolename). It returns the object representing the type of this feature. According to [OCL] section 5.4.1. missing rolenames are generated automatically (the class name starting with a lowercase letter). For association partners with multiplicities greater the one, a collection of the partners type is returned. The parameter `qualifiers` is used for qualified associations. This method cannot be used for operations, even operations without parameters. See section 3.1.4 about potential ambiguities .

**navigateParameterized** queries the class for a behavioral feature (an operation). The second parameter specifies the parameter sequence of the requested method. This request is valid only for methods having the `isQuery` flag set. Methods with return type `void` are unavailable for OCL. See section 3.1.5 about potential ambiguities.

All queries have to care about inherited features and polymorphism of parameters. This is explained below by example.

Methods `navigateQualified`, `navigateParameterized` and `getClassifier` never return null. Instead they throw an exception, if the queried feature/classifier is not available.

### 3.1.2 Example

This section explains the interface introduced above in detail using an example model. For the example see figure 3.1.

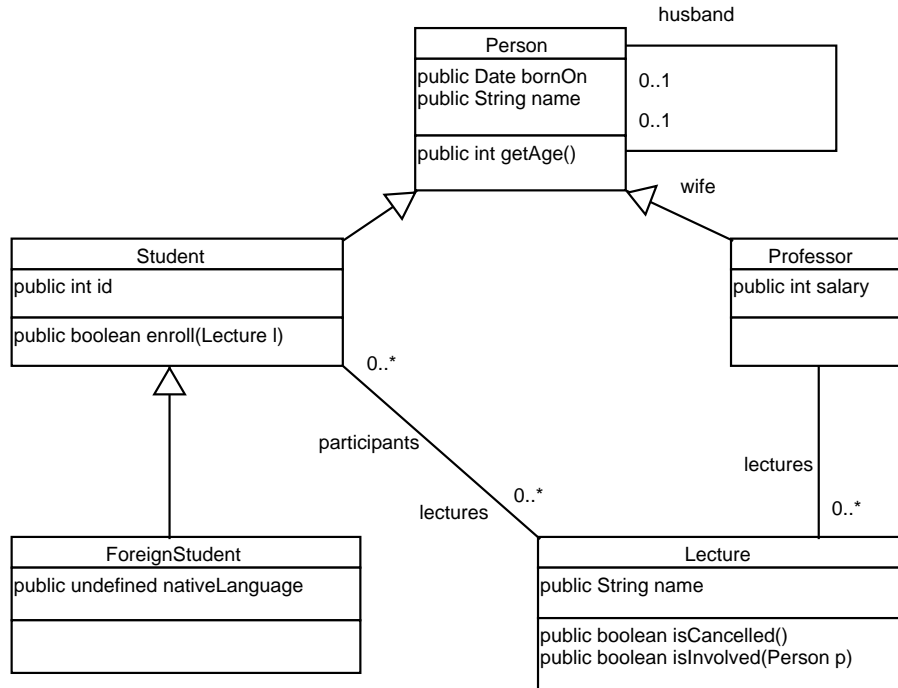


Figure 3.1: Example Model for the Type Interface

**Creating the model.** At first the model has to be created. Using the XMI parser this is achieved by

```
ModelFacade model=XmiParser.getModel("example.xmi");
```

**Querying classifiers.** The model is used to get objects representing classes of the model. These objects are subject to further queries.

```
Type person=model.getClassifier("Person");
Type lecture=model.getClassifier("Lecture");
```

I suppose similar procedure for all classes of the model to be used below.

**Checking generalizations.** Generalizations between classes of the given model are checked with `conformsTo`.

```

student.conformsTo(person);
!person.conformsTo(student);
foreignstudent.conformsTo(person); // yes its transitive
student.conformsTo(student);      // and reflexive too!

```

**Querying features.** At first some obvious examples.

```

lecture.navigateQualified("name", null) ==1
  Basic.STRING2;
lecture.navigateQualified("professor", null) ==
  professor;
lecture.navigateParameterized(
  "isInvolved", { person } ) == Basic.BOOLEAN;

```

Note that “professor” is an implicit rolename as described in [OCL] section 5.4.1. Implicit role names may cause ambiguities. For details see section 3.1.4.

**Multiplicities.** Association ends featuring multiplicities of more than one result in collections.

```

professor.navigate("lectures")==
  new Collection(SET, lecture);

```

**Inherited features.** The model must find all features of a class. This includes features inherited from super classes.

```

student.navigate("bornOn")==date;
student.navigate("husband")==person;
student.navigate("getAge", {} )==
  Basic.INTEGER;
lecture.navigate("isInvolved", { person } )==
  Basic.BOOLEAN;

```

**Polymorphism.** The model must find operations, even if parameter types are not exactly the same as in the operation definition. For demonstration, the last statement of the paragraph above is slightly modified.

```

lecture.navigate("isInvolved", { student } )==
  Basic.BOOLEAN;
lecture.navigate("isInvolved", { professor } )==
  Basic.BOOLEAN;

```

Note that this may result in ambiguities in method matching. For details see section 3.1.5.

---

<sup>1</sup> Actually, the method equals has to be called instead of the equality operator. The operator is used here for brevity only. However, most objects are implemented using flyweight pattern, so the equality operator would work as well.

<sup>2</sup> Class Basic contains predefined objects representing built-in OCL types.

### 3.1.3 Dissolving Associations

Associations in UML are entities of their own. The type information component dissolves associations into a set of structural features for each class of the association. This section describes the algorithm used to dissolve an association formally. The algorithm closely follows [OCL].

**Association.** An association is a tuple of a set of association ends and an association class. The association class is optional.<sup>3</sup> The set of association ends must contain at least two members. The associations name is not of interest.

**Association End.** An association end is a tuple of the following components: the *rolename*, the *class* connected to the association end, the flag *isMultiple* describing whether multiplicities above one are allowed and the flag *isOrdered* provided by the model.

**Name of an Association End.** The association ends name is the rolename, if provided by the model. Otherwise it is the name of the class, starting with a lowercase letter.

#### Navigation between Association Ends.

In a first step, we don't care about association classes.

**Type of an Association End.** The association ends type is determined by the following table.

isMultiple	isOrdered	Type
false	-	class
true	false	set of class
true	true	sequence of class

Bags are never generated by the type information model.

**Structural Feature of an Association End.** Each association end is represented by a structural feature. The name of the feature is the name of the association end. The type of the feature is the type of the association end.

**Creating features.** For each pair (A,B) of association ends with  $A \neq B$  the structural feature representing B is added to the class connected to association end A.

---

<sup>3</sup>This is slightly different from the UML, where an association class inherits from both association and class.

### Navigation to Association Classes.

The second step makes the association class reachable from the classes connected to association ends.

**othersAreMultiple.** For each association end  $E$  of an association  $A$ , there is a flag *othersAreMultiple*, which is true, if at least one of the other association ends has the *isMultiple* flag set:

$$othersAreMultiple(E) = \exists e \in A : e \neq E \wedge isMultiple(e)$$

**Creating features.** To each class connected to an association end a structural feature is added. The name of the feature is the name of the association class (which is the name of the association) starting with a lower-case letter. The type of the feature is determined by the following table.

othersAreMultiple	Type
false	association class
true	set of association class

### Navigation from Association Classes.

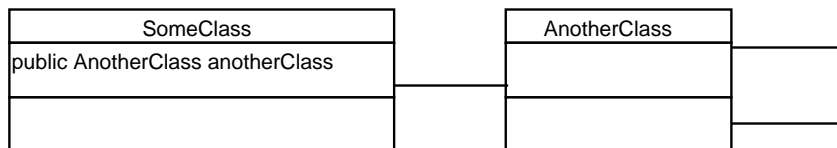
The third step makes the association ends reachable from the association class.

**Creating features.** For each association end a structural feature is added to the association class. The name of the feature is the *name of the association end*. The type of the feature is the class connected to the association end.

Note that when navigating from the association class to association ends the model never returns collections, regardless of the association ends multiplicity.

#### 3.1.4 Attribute Ambiguities

An attribute ambiguity occurs, if a classifier has more than one feature of the same name. Consider the following example.



SomeClass has two features called “anotherClass”:

1. The attribute “anotherClass”
2. The association partner “AnotherClass”, which is accessible by an implicit rolename “anotherClass”, as described in section 3.1.3.

According to [OCL], this ambiguity causes both features “anotherClass” to be unavailable for OCL.

The same applies to “AnotherClass”. It has two features “anotherClass”. Both are implicit rolenames of the reflexive association. Again, this causes both features “anotherClass” to be unavailable for OCL.

[OCL] does not specify, whether a subclass may override an ambiguous feature, thus making the feature available again. For example, suppose a subclass “SubSomeClass” of “SomeClass” having an attribute “anotherClass”. One could argue about, whether this feature is available for OCL or not. Current implementation of the type information component makes the feature available again, however this could be changed easily.

### 3.1.5 Operation Matching Ambiguities

Operation matching is a problem to be solved by any compiler. There is a operation call and given set of operations with the same name. The question is, which operation actually gets called.

Suppose a generalizationship of two classes

```
class ClassA {};  
class ClassB extends ClassA {};
```

and the following model.

```
class SomeClass  
{  
    operation(ClassA a);  
    operation(ClassB b);  
}
```

Now lets query for `operation(ClassB)`.

```
someClass.navigateParameterized("operation", { ClassB } );
```

According to the interface definition, navigate must care about parameter polymorphism. Thus, there are two operations matching the query: `operation(ClassA)` and `operation(ClassB)`. Of course the second one matches “better”.

But its not as easy, as it looks. To explain, the model above is extended.

```
class SomeClass  
{  
    operation(ClassA a, ClassB b);  
    operation(ClassB b, ClassA a);  
}
```

The new query

```
someClass.navigateParameterized("operation",
    { ClassA, ClassA } );
```

will get the type checker into serious problems. Both operations match to the query, and both match equally well. There is no way to decide, which operation is meant.

The current implementation of the type information component cannot handle either of the queries. Instead, it will generate an exception due to ambiguous operations. More formally:

**Operation matching.** An operation definition  $(dn, dp_1, dp_2, \dots, dp_{dl})$  matches an operation query  $(qn, qp_1, qp_2, \dots, qp_{ql})$  if and only if all subsequent conditions hold:

1. The operations name matches:  $dn = qn$ .
2. The parameter sequence length matches:  $dl = ql$ .
3. The parameters match:  $\forall i \in [1..dl] : qp_i.conformsTo(dp_i)$

**Operation ambiguity.** An operation query is ambiguous, if and only if there is more than one operation definition matching the operation query as defined above.

A more sophisticated algorithm could handle at least some of the queries, which are rejected by the current implementation. For an idea see [Java] section 15.11.2.2: CHOOSING THE MOST SPECIFIC METHOD. This has not been implemented for simplicity. Up to now the OCL specification does not specify operation matching in detail.

## 3.2 Design

This section explains the design of the component I developed. It consists of two parts:

**Type Information Component.** The model itself, storing all information needed to handle the queries from the OCL compiler.

**XMI Parser.** Creates the model from the information provided by the XMI file.

See figure 3.2 how both parts fit into the architecture of the whole system.

### 3.2.1 Type Information Component

See the figure 3.3 for an static UML structure of the type information component.

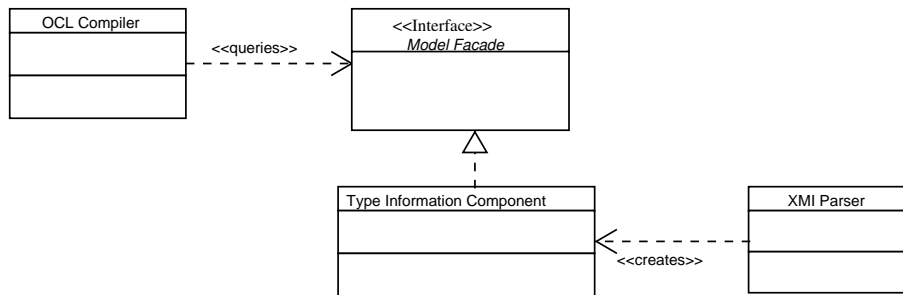


Figure 3.2: Architecture of the Type Checking Subsystem

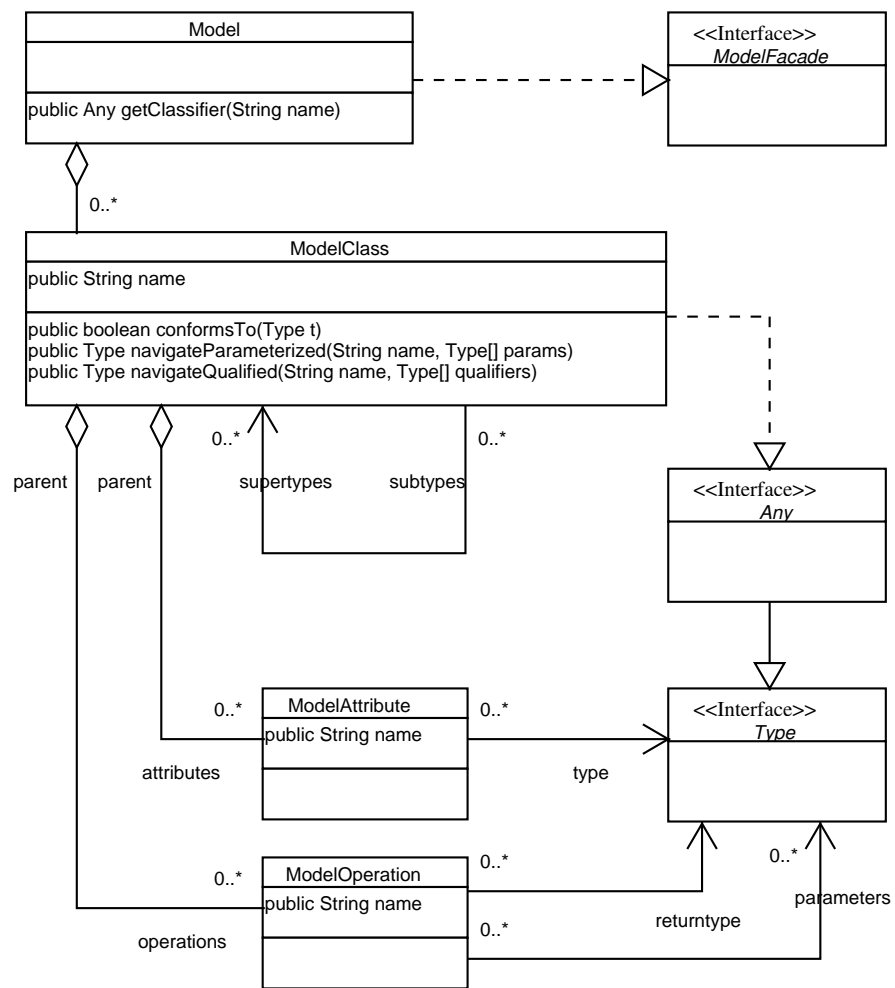


Figure 3.3: Design of the Type Information Component

**Model Facade.** At the right there is the interface provided by the OCL compiler. This has been discussed in detail in section 3.1.1.

**Type Information Component.** The classes at the left store the type information and implement the interface `ModelFacade` to answer the queries from the compiler.

**Model** represents a whole Model. It aggregates all classes of the model. It implements the `getClassifier()` method inherited from `ModelFacade` by looking up the class with the given name.

**ModelClass** represents a class of the model. It knows about its supertypes and aggregates all attributes and operations. `ModelClass` implements `Any`, so it is directly responsible for all queries about the class.

**ModelAttribute** represents an attribute of a class. It stores the attributes name and its type.

**ModelOperation** represents an operation of a class. It stores the operations name, its parameter sequence and its return type. Note that parameters is an ordered association.

**Associations.** It is not possible to store associations explicitly. Instead, associations are dissolved into a set of structural features (attributes) according to section 3.1.3. Dissolving associations is implemented by the method `ModelAssociation.dissolve(Model)`.

**Flattening.** After creating the model structure, there is a final step of flattening the model. This means, that for each class

1. The set of supertypes is extended to include all indirect supertypes.
2. The set of attributes is extended to include all inherited attributes, which are not overridden in this class. However, overriding an attribute is considered to be suspect and causes a warning to standard out.
3. The set of operations is extended to include all inherited operations, which are not overridden in this class. Overriding an operation with a different return type is considered to be suspect and causes a warning to standard out.

After flattening the model, it cannot be modified any longer. The Type Information Component is designed to be created at once, and then used without any further modifications. This simplifies the design significantly and makes better performance possible.

### 3.2.2 XMI Parser

The parser is essentially a collection of methods, creating the Type Information Component. So there is not much to be designed.

**The XML Parser used.** The XMI parser uses XML4J to parse the XML file. XML4J is provided by IBM for free. XML4J implements the DOM model provided by the W3C. This standardizes the way, a java application accesses a XML file. Thus, the XMI parser does not depend on an interface proprietary to IBM.

**XMI Adapters.** A XMI adapter adapts the parser to different versions of XMI. For the XMI Parser there are two different versions to be distinguished. Both differ in naming of tags and attributes. See the table for details.

Version	OMG	IBM
	Rational Rose XMI Plugin	Argo/UML 0.7
attribute name	xmi.id	XMI.id
attribute name	xmi.value	XMI.value
attribute name	xmi.idref	target
element name	Foundation.Core.Class	Class
element name	Foundation.Core.ModelElement.name	name
element name	Foundation.Core.Generalization	Generalization
element name	Foundation.Core.Generalization.subtype	subtype
	... and so on.	

Originally, XMI has been developed by IBM. The OMG adopted XMI as a standard representation of UML. Unfortunately, the OMG has prefixed all element names with their packages of the UML specification. This makes XMI documents less readable by humans, more tedious to parse and noticeably larger in size.

**Element Nesting.** There are further differences of the XMI generated by the tools mentioned above. One example is illustrated here.

This is an extract of the XMI structure generated by Argo/UML:

```
<Model>
  <ownedElement>
    <Class> </Class>
  </ownedElement>
  <ownedElement>
    <Class> </Class>
  </ownedElement>
</Model>
```

The XMI generated by Rational Rose is different: The two lines put in italics are missing.<sup>4</sup>

In other words: Argo puts each `<Class>` element into its own `<ownedElement>` element, while Rose put all `<Class>` elements into a single `<ownedElement>` element.

This problem is solved by defensive programming. The parser simply accepts multiple `<ownedElement>` elements, each having multiple `<Class>` elements nested inside.

The table below list all similar cases of different nesting, the author found during implementation. Probably, this list is not complete.

Class	Rolename	Associated Classes
Model	ownedElement	Class, Association, Generalization
Association	connection	AssociationEnd
Class	feature	Attribute, Operation
Operation	parameter	Parameter

### 3.3 Implementation

This section discusses selected issues of the implementation.

#### 3.3.1 Requirements

The type information component with the XMI parser requires the Java Collections API. Since the OCL compiler does depend on Java Collections too, this is no problem.

Additionally XML4J is needed. It is available at the IBM website. Development and testing was done using version 2.0.15.

The parser was tested with Argo/UML 0.7 and Rational Rose 98sp1 with the Unisys XMI Plugin.

#### 3.3.2 Operations with Side Effects

[OCL] specifies, that OCL expressions may not use operations with side effects. XMI makes this information available by the `isQuery` element. However, neither Rose or Argo do support setting the `isQuery` information by the user. Instead, it is set to false for all operations. Therefore the XMI parser ignores it.

### 3.4 Tests

This section describes the test cases performed on the type information subsystem. There are several tests, some replacing some of the components with

---

<sup>4</sup>Additionally, tag names are different as explained above.

test drivers. For understanding, please refer to the type checking subsystem architecture figure 3.2 on page 15.

### 3.4.1 Tests Using the Royloy Example Model

The following tests use a model featuring the “Royalties and Loyalties” example model. The OCL compiler processes some OCL expressions written by Frank Finger for the example model.

**Testing the Type Information Component.** In this test case the XMI Parser component replaced by a test driver. The test driver directly creates the example model.

**Testing the XMI Parser.** This test case runs the entire system. The XMI Parser creates the example model twice from two XMI documents. The first one was created using Argo/UML 0.7. The second document has been exported from Rational Rose using the Unisys Plugin.

### 3.4.2 Stress Test for the Type Information Component.

The example model of the tests above does not cover the more subtle problems of the type information component. The model lacks

- overriding features
- indirect generalizations
- attribute ambiguities (see section 3.1.4)
- method ambiguities (see section 3.1.5)
- different package names.

The stress test features a model covering all these issues. See figure 3.4.

The test case replaces the OCL compiler (see figure 3.2 again) by a test driver querying the Model Facade. This makes more precise testing possible. For failing queries it is additionally checked, whether they fail for the intended reason. Possible reasons for failure are attribute ambiguity, method ambiguity and the simple absence of the feature queried.

**Attribute Ambiguities.** The following queries throw an `OclTypeException` due to attribute ambiguities:

```
gamma.navigateQualified("three", null)
```

fails because the implicit rolename “three” clashes with the attribute of the same name.

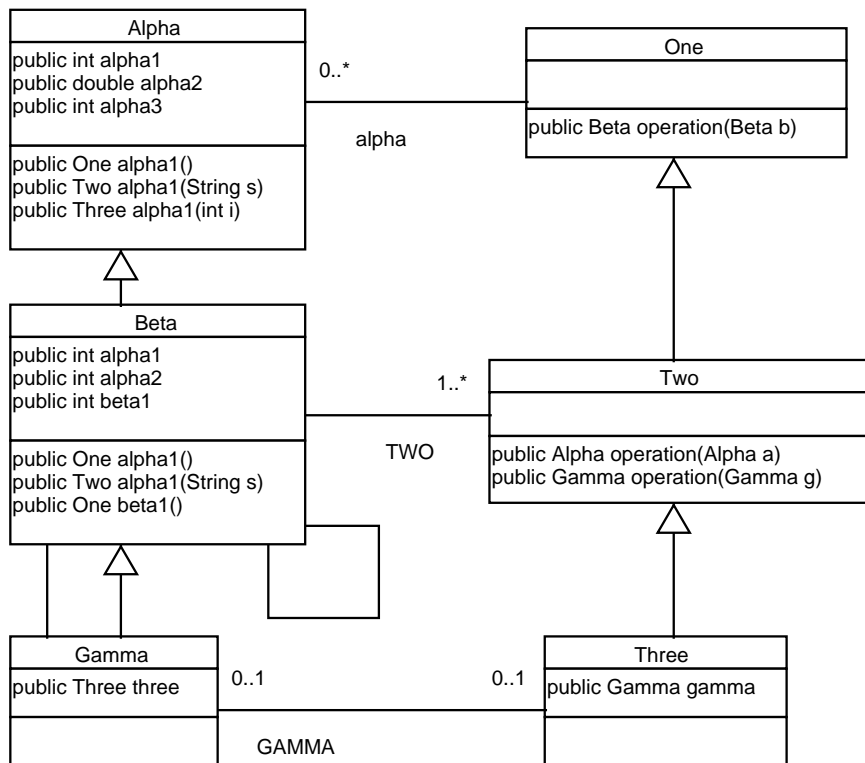


Figure 3.4: Stress Test Model

```
beta.navigateQualified("beta", null)
```

fails because the recursive association of beta generates two implicit rolenames "beta".

However,

```
gamma.navigateQualified("beta", null)
```

returns Beta, since the ambiguous feature inherited from Beta is overridden in Gamma.

**Method Ambiguities.** This query throws an OclTypeException due to method ambiguity.

```
two.navigateParameterized("operation", { beta })
```

This query can easily be changed, so that the ambiguity is eliminated.

```
one.navigateParameterized("operation", { beta }) == beta  
two.navigateParameterized("operation", { gamma }) == gamma
```

## Chapter 4

# Resulting Requirements

This chapter describes requirements a query language must fulfill to be able to implement the OCL compilers type information component.

### 4.1 Order of Instance Nodes

This section shows by example, that a query language must be able to evaluate the order of nodes in the document instance.

**Example.** Given the following example, written down in Java.

```
class someClass
{
    Return1 operation(int x, String y);
    Return2 operation(String y, int x);
}
```

When the type component is queried for the first of both operations , it looks like this.

```
someclass.navigateParameterized("operation",
    { Basic.INTEGER, Basic.STRING } );
```

Obviously, this expression should evaluate to **Return1**.

Lets see, how this query can be implemented using XML-QL. At first the (simplified) XMI representation of the example above.

```
<Class name="someClass">
  <Operation name="operation">
    <Parameter kind="in" name="x" type="int" />
    <Parameter kind="in" name="y" type="String" />
    <Parameter kind="return" type="Return1" />
  </Operation>
</Class>
```

```

    </Operation>
  <Operation name="operation">
    <Parameter kind="in" name="y" type="String" />
    <Parameter kind="in" name="x" type="int" />
    <Parameter kind="return" type="Return2" />
  </Operation>
</Class>

```

Note that both operations differ in the order of the `Parameter` elements only. Now lets have a first try of the XML-QL query.

```

WHERE
<Class name="someClass">
  <Operation name="operation">
    <Parameter kind="in" type="int" />
    <Parameter kind="in" type="String" />
    <Parameter kind="return" type=$r />
  </Operation>
</Class>
IN "example.xmi",
CONSTRUCT <result>$r</result>

```

This query is not yet correct. By default, XML-QL matches without checking order of nodes. Thus the query matches to both operations and returns

```

<result>Return1</result>
<result>Return2</result>

```

To achieve the correct behavior, the query is slightly extended. Enclosed in square brackets are element-order variables. These variables are bound to the index in the local order of document nodes. An additional precondition rejects matches with the undesired order. Additions are put in *italic*.

```

WHERE
<Class name="someClass">
  <Operation name="operation">
    <Parameter kind="in" type="int" /> [$i]
    <Parameter kind="in" type="String" /> [$j]
    <Parameter kind="return" type=$r />
  </Operation>
</Class>
IN "example.xmi",
$i < $j
CONSTRUCT <result>$r</result>

```

This query matches only the first operation. Correctly it produces

```

<result>Return1</result>

```

Note that the XML-QL prototype currently available from AT&T Labs does not support queries for instance order.

**Evaluation of impact.** It has to be admitted, that the problem described here is somewhat exotic. For a collision to occur, it needs two operations in a class, differing in the order of their parameters only.

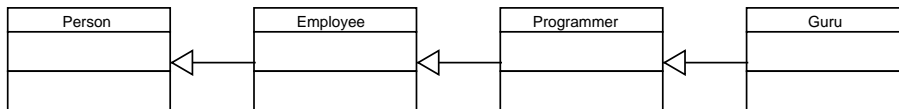
The problem of transitive closures described below is much more important.

## 4.2 Transitive Closure

This section shows by example, that a query language must be able to process queries including the transitive closure of a given relation.

The type component must implement a query for generalizations. Given two classes the query must determine whether one class is a (possibly indirect) subtype of the other.

Our example features four classes in a linear generalization.



XMI represents this model as a relationship table. Somewhat simplified, it looks like this.

```

<generalization supertype="Person"
  subtype="Employee" />
<generalization supertype="Employee"
  subtype="Programmer" />
<generalization supertype="Programmer"
  subtype="Guru" />
  
```

For an easy start, we retrieve the generalization from Employee to Person.

```

WHERE
<generalization supertype="Person"
  subtype="Employee">
IN "example.xmi"
CONSTRUCT yes
  
```

When querying for the indirect generalization from Programmer to Person, we use a self join.

```

WHERE
<generalization supertype="Person" subtype=$a>
IN "example.xmi",
<generalization supertype=$a subtype="Programmer">
IN "example.xmi"
CONSTRUCT yes
  
```

This works for generalizations of any order. From Guru to Person it takes three steps.

```
WHERE
  <generalization supertype="Person" subtype=$a>,
  IN "example.xmi"
  <generalization supertype=$a subtype=$b>,
  IN "example.xmi"
  <generalization supertype=$b subtype="Guru">
  IN "example.xmi"
CONSTRUCT yes
```

However, there is a serious drawback. These queries need to know the “generalization distance” in advance.

To address this issue, the query language must feature recursion. XML-QL does not provide recursion. Chapter 5 shows, that XSLT solves this problem.

**Evaluation of impact.** This requirement is essential for a query language to be suitable for implementing the type component. Note that the query for generalizations is only the simplest case. The type checker must also care about inherited features and polymorphism of operation parameters. This cannot be implemented without traversing arbitrary generalization trees.

### 4.3 Summary

This chapter introduced two requirements for a query language. The need for evaluating node order will be rare, but certainly the inability to handle these situations well leaves the author with strong discomfort. However, the need for querying over transitive closures is essential. For a query language to be suitable for our task, there is no way around it.

## Chapter 5

# Evaluation of XSLT

This chapter shows, that the type information component as described in the chapters before can be implemented using XSLT. It will be shown, that XSLT fulfills the requirements described in the previous chapter.

This section will not produce an implementation working together with the OCL compiler. Instead it will discuss the critical points and possible solutions. I will develop code fragments in XSLT, without integrating them into the compiler.

### 5.1 Adapting XSLT to Java

This section designs the adaption mechanism between Java interface introduced in section 3.1.1 and XSLT. In fact, this adaption mechanism will work with any query language. This adapter will not be implemented, but we have to think about it.

At first, XSLT like any other query language cannot handle anything else than strings. Thus we have to transform all types and classes to strings.

**ModelFacade.** This class represents the whole model. There is only one instance at a time, so we don't have to care about it.

**Type.** This class represents a basic datatype, a class or a association class. All these entities are represented by corresponding elements in XMI, and each of these elements carries a unique and mandatory `xmi.id` attribute. The value of this attribute can be used to identify each instance of Type.

**Any.** Any is derived from Type without adding any further features, so the same arguments apply.

**Boolean.** This is needed as return type of `Type.conformsTo`. It is represented by a string being either empty for false or having the value "TRUE" for true.

Applying these transformations to the interface, the operations of the model facade are adapted as follows:

**ModelFacade.getClassifier** has the signature `Any getClassifier(String name)`. The input parameter `name` is already a string, and the implicit parameter `this` is the `ModelFacade` itself, and can be ignored. The corresponding XSLT script results in the `xmi.id` attribute value of the XMI element representing the queried instance of `Any`.

**Type.conformsTo** has the signature `boolean conformsTo(Type type)`. The input parameter `type` is represented by its `xmi.id` value, the same goes for `this`. The XSLT script returns a string representing the boolean value as describes above.

**Type.navigateQualified** has the signature `Type navigateQualified(String name, Type[] qualifiers)`. Parameter `name` is a string already, `qualifiers` and `this` are given as their `xmi.id` values. The script results in the `xmi.id` value of the queried type.

**Type.navigateParameterized** is adapted exactly like `navigateQualified`.

Given this adaption mechanism we will do some scripting in XSLT.

## 5.2 Order of Instance Nodes

This section shows, that XSLT is able to query the order of nodes in the document instance. Therefore the operation `Type.navigateParameterized` is partially implemented in XSLT.

For the test case I slightly extended the example model from section 4.1. The script queries for `operation(String y, int x)` and correctly produces `Return2`.

```
class someClass
{
    Return1 operation(int x, String y);
    Return2 operation(String y, int x);
    Return3 operation(int x);
    Return4 otherOperation(String y, int x);
}
```

The problem of node order is easily solved in XSLT using node number qualifiers. As shown below, the first parameter of an operation is accessed by writing `parameter[1]`.

```
<xsl:template match="Operation[name='operation']">
  <xsl:apply-templates select="parameter[1]"/>
</xsl:template>
```

The entire script is included in appendix A.1. For the resulting output see A.2.

### 5.2.1 Parameter Polymorphism.

The script does not care about parameter polymorphism, since this chapter does not intend to develop a fully functional XSLT version of `navigateParameterized`. The script simply tests the parameter types on equality.

Note, that this cannot be solved by simply invoking the XSLT script for `conformsTo`, which is explained in the next section. Instead, the java implementation must be called, since basic types and collections are handled there. At the very end of my work, I discovered a new requirement for XML query languages: the possibility to call java methods from the script.

## 5.3 Transitive Closure

This section shows, that XSLT is able to query the transitive closure of a given relation. Therefore the operation `Type.conformsTo` is implemented in XSLT.

### 5.3.1 Algorithm

At first some definitions. The function `conformsTo` is represented by the circle operator:

$$A.conformsTo(B) = A \circ B$$

Function `sup` and `sub` retrieve all supertypes/subtypes of a given type:

$$sup : Type \rightarrow P(Type)$$

$$sub : Type \rightarrow P(Type)$$

Of course we use recursion. The first try is

$$A \circ B = (A = B) \vee \exists t \in sup(A) : t \circ B.$$

This traverses from subtypes to supertypes. We could go the other direction:

$$A \circ B = (A = B) \vee \exists t \in sub(B) : A \circ t.$$

Usually a generalization relation is more often branched in top-down direction, so traversing down-top should be faster.

During implementation I found, that a variation of the algorithm is easier to write in XSLT. I split it up into two parts:

$$A \circ B = (A = B) \vee A \bullet B$$

$$A \bullet B = \exists t \in sup(A) : (t = B) \vee t \bullet B$$

The first part is implemented very efficiently in the java wrapper. The second part featuring the recursion is implemented in XSLT. This is explained in the next section.

### 5.3.2 Implementation

This chapter explains the implementation. For the complete source see appendix A.3.

The implementation is a cascade of “procedure calls”, descending into the depth of the DOM tree. For example, the following rule descends into all `Model` elements nested into the `XMI.content` element. The currently queried subtype and supertype are stored in parameters “sub” and “super”.

```
<xsl:template match="XMI.content">
  <xsl:param name="sub"  />
  <xsl:param name="super" />
  <xsl:apply-templates select="Model">
    <xsl:with-param name="sub">
      <xsl:value-of select="$sub"/>
    </xsl:with-param>
    <xsl:with-param name="super">
      <xsl:value-of select="$super"/>
    </xsl:with-param>
  </xsl:apply-templates>
</xsl:template>
```

Eventually there is a generalizationship found with the subtype in question. This rule branches to the rule for supertypes.

```
<xsl:template
  match="Generalization[subtype[
    XMI.reference[@target=$sub]]]"
  priority="2">
  <xsl:param name="sub"  />
  <xsl:param name="super" />
  <xsl:apply-templates select="supertype"
    mode="supertype">
    <xsl:with-param name="super">
      <xsl:value-of select="$super" />
    </xsl:with-param>
  </xsl:apply-templates>
</xsl:template>
```

Eventually the supertype of the generalizationship is the supertype in question. The following rule matches this and generates a “TRUE”, which is evaluated by the java wrapper.

```
<xsl:template
  match="supertype[XMI.reference[@target=$super]]"
  priority="2"
  mode="supertype">
```

```

    <supertype super="{ $super }">
      <xsl:text>[TRUE]</xsl:text>
    </supertype>
  </xsl:template>

```

Note that the rule above has an increased priority. If the supertype does not match, there are another two rules for the recursion.

```

<xsl:template match="supertype" mode="supertype">
  <xsl:param name="super" />
  <xsl:apply-templates select="XMI.reference"
    mode="supertype">
    <xsl:with-param name="super">
      <xsl:value-of select="$super"/>
    </xsl:with-param>
  </xsl:apply-templates>
</xsl:template>

<xsl:template match="XMI.reference" mode="supertype">
  <xsl:param name="super" />
  <xsl:variable name="nextsub" select="@target" />
  <xsl:apply-templates select="/XMI/XMI.content">
    <xsl:with-param name="sub">
      <xsl:value-of select="$nextsub"/>
    </xsl:with-param>
    <xsl:with-param name="super">
      <xsl:value-of select="$super"/>
    </xsl:with-param>
  </xsl:apply-templates>
</xsl:template>

```

Actually, the recursion is performed in the second rule above. The `apply-templates` element branches (nearly) to the top of the DOM tree. The subtype is not the same of the current query. Instead the subtype of the current generalizationship is used. This shortens the generalization distance by 1.

### 5.3.3 Test

This section tests the XSLT script performing the `conformsTo` operation. Therefore I developed a test case featuring some kind of worst-case generalizationship. See figure 5.1.

For the test I queried the generalizationship from class C1 to class A3. Note, that there are two paths from C1 to A3, one via B2 and another via B3. Accordingly, the XSLT script produces the `[TRUE]` twice, once for each path.

Additionally the XSLT script produces debugging output. The output of the test query is available in appendix A.4, which can be used to trace the recursion path through the DOM tree.

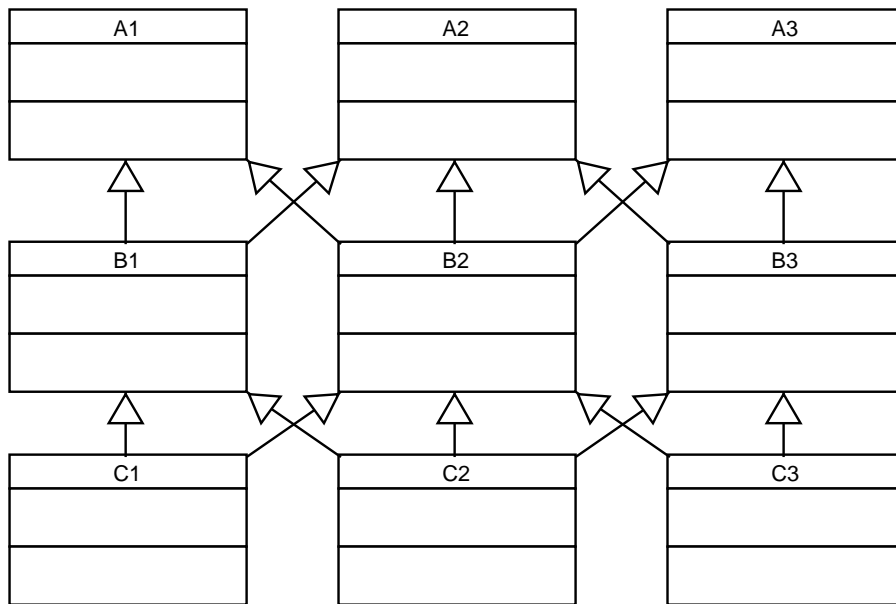


Figure 5.1: Test Case for `conformsTo` in XSLT

For the test I used the Lotus XSLT processor version 0.18.5.

## Chapter 6

# Summary

The task was to evaluate XML query languages for querying XML based repositories with an emphasis on XMI documents. For the evaluation I had to implement an experimental application in the context of software engineering, especially code generation.

The application I chose is a component of the OCL compiler developed by Frank Finger in [FF]. It provides the subset of the information of a UML model to the OCL compiler. This component has been described extensively in chapter 3.

For the implementation I gave XML-QL the first chance. After some experimenting I found out, that XML-QL isn't powerful enough to implement this component. This hit me by surprise. When I started with XML-QL I had the naive assumption, that all of the more complex query languages could handle this task. The exact reasons has been analyzed in chapter 4.

When XML-QL went out of the arena, I decided together with my tutor not to try another query language. Instead I did the implementation in pure Java. There were two reasons for this:

1. The prototypes of several query languages proofed to be instable and incomplete. It would be very hard to predict, how much time an implementation would take, if possible at all.
2. The OCL compiler became more and more important for the chair. The component to be developed by me was the last part missing. The java implementation was a safe bet, that the work could be finished in a deterministic amount of time.

After the java implementation I got interested in XSLT. This is a quite powerful query language, and so I tried, whether it could implement the component for the OCL compiler. Since the java implementation was already up and running, I didn't work on a fully functional version to be integrated with the OCL compiler. Instead, I concentrated on the problems I found with XML-QL. In chapter 5 I

describe two XSLT scripts, solving these two problems. This suggests, that it is at least possible to implement the component in XSLT.

**What finally came out.**

- The realization, that XML-QL lacks some functionality needed in real life.
- Two requirements for XML query languages, probably needed for most applications querying UML models represented in XMI.
- A fully functional and well integrated type information component for the OCL compiler.
- Two experimental and incomplete XSLT scripts, proofing that XSLT provides some functionality, that XML-QL lacks.
- Lots of new knowledge and experiences for me.

**Usability of XML query languages.** Now a word on the usability of the query languages (XML-QL and XSLT) in real life. To be honest I cannot say something really significant about this. The main problem for me was the quality of the prototypes I used. They are instable and incomplete. Sometimes something failed and I didn't know why. Sometimes something worked and I didn't know why. Error messages are poorly expressive, and often there is simply a `NullPointerException` thrown. I cannot imagine, how to use these prototypes for something more demanding than an evaluation exercise. So I didn't get the chance to use the query languages on a *real* task, and I cannot say, how they suit this.

For a practical application of query languages the implementations will have to become much more better. Until then, XML query languages will be of academic interest only.

# Appendix A

## Selected XSLT Scripts

### A.1 navigateParameterized.xsl

To understand the script and the following output, you need the `xmi.id`'s of the types involved:

Type	Id
int	S.100006
String	S.100003
Return2	S.100082

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/XSL/Transform/1.0"
                xmlns:xt="http://www.jclark.com/xt"
                extension-element-prefixes="xt">

<xsl:template match="XMI">
  <xsl:text>
</xsl:text>
  <XMI>
    <xsl:apply-templates select="XMI.content">
    </xsl:apply-templates>
    <xsl:text>
</xsl:text>
  </XMI>
</xsl:template>

<xsl:template match="XMI.content">
  <xsl:text>
</xsl:text>

```

```

    <XMI.content>
      <xsl:apply-templates select="Model">
        </xsl:apply-templates>
      <xsl:text>
    </xsl:text>
    </XMI.content>
  </xsl:template>

  <xsl:template match="Model">
    <xsl:text>
      </xsl:text>
    <Model name="{name}">
      <xsl:apply-templates select="ownedElement">
        </xsl:apply-templates>
      <xsl:text>
        </xsl:text>
    </Model>
  </xsl:template>

  <xsl:template match="ownedElement">
    <xsl:apply-templates select="Model|Class" />
  </xsl:template>

  <xsl:template match="Class">
  </xsl:template>

  <xsl:template match="Class[@XMI.id='S.100027']" priority="2">
    <xsl:text>
      </xsl:text>
    <Class name="{name}">
      <xsl:apply-templates select="feature" />
      <xsl:text>
        </xsl:text>
    </Class>
  </xsl:template>

  <xsl:template match="feature">
    <xsl:apply-templates select="Operation" />
  </xsl:template>

  <xsl:template match="Operation">
  </xsl:template>

  <xsl:template match="Operation[name='operation']" priority="2">
    <xsl:text>
      </xsl:text>
  </xsl:template>

```

```

<Operation name="{name}">
  <xsl:apply-templates
    select="parameter[1]/Parameter[1]/type[1]/XMI.reference[1]"
    mode="P1"/>
  <xsl:text>
    </xsl:text>
</Operation>
</xsl:template>

<xsl:template match="XMI.reference">
</xsl:template>

<xsl:template match="XMI.reference[@target='S.100003']" mode="P1" priority="2">
  <xsl:text>
    IS_OK_1</xsl:text>
  <xsl:apply-templates
    select="../../../parameter[2]/Parameter[1]/type[1]/XMI.reference[1]"
    mode="P2"/>
</xsl:template>

<xsl:template match="XMI.reference[@target='S.100006']" mode="P2" priority="2">
  <xsl:text>
    IS_OK_2</xsl:text>
  <xsl:apply-templates
    select="../../../parameter[3]/Parameter[1]/type[1]/XMI.reference[1]"
    mode="PR"/>
</xsl:template>

<xsl:template match="XMI.reference" mode="PR" priority="2">
  <xsl:text>
    [</xsl:text>
    <xsl:value-of select="@target"/>
    <xsl:text>]</xsl:text>
</xsl:template>

</xsl:stylesheet>

```

## A.2 navigateParameterized.xsl.result

Apart from the query result, the script produces debugging output, which may be used to trace the path of the query processor through the document.

```

<XMI>
  <XMI.content>

```

```

<Model name="untitledpackage">
  <Class name="someClass">
    <Operation name="operation">
    </Operation>
    <Operation name="operation">
      IS_OK_1
      IS_OK_2
      [S.100082]
    </Operation>
    <Operation name="operation">
    </Operation>
  </Class>
</Model>
</XMI.content>
</XMI>

```

### A.3 conformsTo.xsl

Again, the id's for types involved:

Type	Id
A1	S.100027
A2	S.100073
A3	S.100074
B1	S.100075
B2	S.100076
B3	S.100077
C1	S.100078
C2	S.100079
C3	S.100080

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/XSL/Transform/1.0"
  xmlns:xt="http://www.jclark.com/xt"
  extension-element-prefixes="xt">

<xsl:template match="XMI">
  <xsl:param name="sub" >S.100079</xsl:param>
  <xsl:param name="super">S.100074</xsl:param>
  <xsl:text>
</xsl:text>
  <XMI sub="{ $sub}" super="{ $super}">
    <xsl:apply-templates select="XMI.content">
      <xsl:with-param name="sub" ><xsl:value-of select="{ $sub}" /></xsl:with-param>

```

```

        <xsl:with-param name="super"><xsl:value-of select="$super" /></xsl:with-param>
    </xsl:apply-templates>
    <xsl:text>
</xsl:text>
</XMI>
</xsl:template>

<xsl:template match="XMI.content">
    <xsl:param name="sub" >nixxsub</xsl:param>
    <xsl:param name="super">nixxsuper</xsl:param>
    <xsl:text>
</xsl:text>
    <XMI.content sub="{ $sub}" super="{ $super}">
        <xsl:apply-templates select="Model">
            <xsl:with-param name="sub" ><xsl:value-of select="$sub" /></xsl:with-param>
            <xsl:with-param name="super"><xsl:value-of select="$super" /></xsl:with-param>
        </xsl:apply-templates>
        <xsl:text>
</xsl:text>
    </XMI.content>
</xsl:template>

<xsl:template match="Model">
    <xsl:param name="sub" >nixxsub</xsl:param>
    <xsl:param name="super">nixxsuper</xsl:param>
    <xsl:text>
</xsl:text>
    <Model name="{name}" sub="{ $sub}" super="{ $super}">
        <xsl:apply-templates select="ownedElement">
            <xsl:with-param name="sub" ><xsl:value-of select="$sub" /></xsl:with-param>
            <xsl:with-param name="super"><xsl:value-of select="$super" /></xsl:with-param>
        </xsl:apply-templates>
        <xsl:text>
</xsl:text>
    </Model>
</xsl:template>

<xsl:template match="ownedElement">
    <xsl:param name="sub" >nixxsub</xsl:param>
    <xsl:param name="super">nixxsuper</xsl:param>
    <xsl:apply-templates select="Model|Generalization">
        <xsl:with-param name="sub" ><xsl:value-of select="$sub" /></xsl:with-param>
        <xsl:with-param name="super"><xsl:value-of select="$super" /></xsl:with-param>
    </xsl:apply-templates>
</xsl:template>

```

```

<xsl:template match="Generalization">
</xsl:template>

<xsl:template match="Generalization[subtype[XMI.reference[@target=$sub]]]"
  priority="2">
  <xsl:param name="sub" >nixxsub</xsl:param>
  <xsl:param name="super">nixxsuper</xsl:param>
  <xsl:text>
    </xsl:text>
  <Generalization name="{@XMI.id}" sub="{@sub}" super="{@super}">
    <xsl:apply-templates select="supertype" mode="supertype">
      <xsl:with-param name="super"><xsl:value-of select="$super" /></xsl:with-param>
    </xsl:apply-templates>
    <xsl:text>
      </xsl:text>
    </Generalization>
  </xsl:template>

<xsl:template match="supertype[XMI.reference[@target=$super]]"
  priority="2"
  mode="supertype">
  <xsl:text>
    </xsl:text>
  <supertype super="{@super}">
    <xsl:text>
      [TRUE]</xsl:text>
    <xsl:text>
      </xsl:text>
    </supertype>
  </xsl:template>

<xsl:template match="supertype" mode="supertype">
  <xsl:param name="super">nixxsuper</xsl:param>
  <xsl:text>
    </xsl:text>
  <supertype super="{@super}">
    <xsl:apply-templates select="XMI.reference" mode="supertype">
      <xsl:with-param name="super"><xsl:value-of select="$super" /></xsl:with-param>
    </xsl:apply-templates>
    <xsl:text>
      </xsl:text>
    </supertype>
  </xsl:template>

<xsl:template match="XMI.reference" mode="supertype">
  <xsl:param name="super">nixxsuper</xsl:param>

```

```

<xsl:variable name="nextsub" select="@target" />
<xsl:text>
  </xsl:text>
<XMI.reference super="{super}" nextsub="{nextsub}">
  <xsl:apply-templates select="/XMI/XMI.content">
    <xsl:with-param name="sub" ><xsl:value-of select="$nextsub" /></xsl:with-param>
    <xsl:with-param name="super"><xsl:value-of select="$super" /></xsl:with-param>
  </xsl:apply-templates>
  <xsl:text>
    </xsl:text>
  </XMI.reference>
</xsl:template>

</xsl:stylesheet>

```

## A.4 conformsTo.xsl.result

```

<XMI super="S.100074" sub="S.100079">
  <XMI.content super="S.100074" sub="S.100079">
    <Model super="S.100074" sub="S.100079" name="untitledpackage">
      <Generalization super="S.100074" sub="S.100079" name="S.100086">
        <supertype super="S.100074">
          <XMI.reference nextsub="S.100075" super="S.100074">
            <XMI.content super="S.100074" sub="S.100075">
              <Model super="S.100074" sub="S.100075" name="untitledpackage">
                <Generalization super="S.100074" sub="S.100075" name="S.100081">
                  <supertype super="S.100074">
                    <XMI.reference nextsub="S.100073" super="S.100074">
                      <XMI.content super="S.100074" sub="S.100073">
                        <Model super="S.100074" sub="S.100073" name="untitledpackage">
                          </Model>
                        </XMI.content>
                      </XMI.reference>
                    </supertype>
                  </Generalization>
                <Generalization super="S.100074" sub="S.100075" name="S.100135">
                  <supertype super="S.100074">
                    <XMI.reference nextsub="S.100027" super="S.100074">
                      <XMI.content super="S.100074" sub="S.100027">
                        <Model super="S.100074" sub="S.100027" name="untitledpackage">
                          </Model>
                        </XMI.content>
                      </XMI.reference>
                    </supertype>
                  </Generalization>
                </Model>
              </XMI.content>
            </XMI.reference>
          </supertype>
        </Generalization>
      </Model>
    </XMI.content>
  </XMI.reference>
</XMI>

```

```

        </Generalization>
    </Model>
</XMI.content>
    </XMI.reference>
    </supertype>
</Generalization>
    <Generalization super="S.100074" sub="S.100079" name="S.100087">
        <supertype super="S.100074">
            <XMI.reference nextsub="S.100077" super="S.100074">
<XMI.content super="S.100074" sub="S.100077">
    <Model super="S.100074" sub="S.100077" name="untitledpackage">
        <Generalization super="S.100074" sub="S.100077" name="S.100084">
            <supertype super="S.100074">
                <XMI.reference nextsub="S.100073" super="S.100074">
<XMI.content super="S.100074" sub="S.100073">
    <Model super="S.100074" sub="S.100073" name="untitledpackage">
        </Model>
</XMI.content>
    </XMI.reference>
    </supertype>
</Generalization>
    <Generalization super="S.100074" sub="S.100077" name="S.100137">
        <supertype super="S.100074">
            [TRUE]
        </supertype>
    </Generalization>
</Model>
</XMI.content>
    </XMI.reference>
    </supertype>
</Generalization>
    <Generalization super="S.100074" sub="S.100079" name="S.100139">
        <supertype super="S.100074">
            <XMI.reference nextsub="S.100076" super="S.100074">
<XMI.content super="S.100074" sub="S.100076">
    <Model super="S.100074" sub="S.100076" name="untitledpackage">
        <Generalization super="S.100074" sub="S.100076" name="S.100082">
            <supertype super="S.100074">
                <XMI.reference nextsub="S.100027" super="S.100074">
<XMI.content super="S.100074" sub="S.100027">
    <Model super="S.100074" sub="S.100027" name="untitledpackage">
        </Model>
</XMI.content>
    </XMI.reference>
    </supertype>
</Generalization>

```

```

    <Generalization super="S.100074" sub="S.100076" name="S.100083">
      <supertype super="S.100074">
        [TRUE]
      </supertype>
    </Generalization>
    <Generalization super="S.100074" sub="S.100076" name="S.100136">
      <supertype super="S.100074">
        <XMI.reference nextsub="S.100073" super="S.100074">
<XMI.content super="S.100074" sub="S.100073">
  <Model super="S.100074" sub="S.100073" name="untitledpackage">
    </Model>
  </XMI.content>
    </XMI.reference>
      </supertype>
    </Generalization>
  </Model>
</XMI.content>
    </XMI.reference>
      </supertype>
    </Generalization>
  </Model>
</XMI.content>
</XMI>

```

# List of Figures

3.1	Example Model for the Type Interface . . . . .	9
3.2	Architecture of the Type Checking Subsystem . . . . .	15
3.3	Design of the Type Information Component . . . . .	15
3.4	Stress Test Model . . . . .	20
5.1	Test Case for conformsTo in XSLT . . . . .	31

# Bibliography

- [BC99] Angela Bonifati, Stefano Ceri. Comparative Analysis of Five XML Query Languages. Sep. 99,
- [DFF\*99] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy. David Maier, Dan Suciu. Querying XML Data. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering.
- [XMLQL] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, Dan Suciu. A Query Language for XML. <http://www.research.att.com/~mff/files/final.html>.
- [XSLT] XSL Transformations (XSLT), version 1.0. W3C Working Draft 13 August 1999. <http://www.w3.org/TR/xslt>
- [XQL] XML Query Language. <http://www.w3.org/TandS/QL/QL98/pp/xql.html>
- [FF] Frank Finger. Design and Implementation of a Modular OCL Compiler. Diplomarbeit. TU-Dresden 2000. <http://dresden-ocl.sourceforge.net/>
- [UML] OMG Unified Modeling Language Specification, Version 1.3, June 1999.
- [OCL] Object Constraint Language Specification. Chapter 7 in [UML].
- [Java] James Gosling, Bill Joy, Guy Steele. The Java Language Specification. Edition 1.0. August 1996.