Date: January 2008

MOF Model to Text Transformation Language, v1.0

OMG Available Specification

OMG Document Number: formal/2008-01-16
Standard document URL: http://www.omg.org/spec/MOFM2T/1.0/PDF
Associated Schema File*: http://www.omg.org/spec/MOFM2T/20071001
http://www.omg.org/spec/MOFM2T/20071001/.xml

---

* ptc/07-10-02

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE.
IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 140 Kendrick Street, Needham, MA 02494, U.S.A.

TRADEMARKS

MDA®, Model Driven Architecture®, UML®, UML Cube logo®, OMG Logo®, CORBA® and XMI® are registered trademarks of the Object Management Group, Inc., and Object Management Group™, OMG™ , Unified Modeling Language™, Model Driven Architecture Logo™, Model Driven Architecture Diagram™, CORBA logos™, XMI Logo™, CWM™, CWM Logo™, IIOP™ , MOF™ , OMG Interface Definition Language (IDL)™, and OMG Systems Modeling Language (SysML)™ are trademarks of the Object Management Group. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

OMG's Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page *http://www.omg.org*, under Documents, Report a Bug/Issue (http://www.omg.org/technology/agreement.htm).

Table of Contents

Preface

About the Object Management Group

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at *http://www.omg.org/*.

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. A catalog of all OMG Specifications is available from the OMG website at:

http://www.omg.org/technology/documents/spec_catalog.htm

Specifications within the Catalog are organized by the following categories:

OMG Modeling Specifications

- UML
- MOF
- XMI
- CWM
- Profile specifications.

OMG Middleware Specifications

- CORBA/IIOP
- IDL/Language Mappings
- Specialized CORBA specifications
- CORBA Component Model (CCM).

Platform Specific Model and Interface Specifications

- CORBAservices

- CORBAfacilities
- OMG Domain specifications
- OMG Embedded Intelligence specifications
- OMG Security specifications.

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. (as of January 16, 2006) at:

OMG Headquarters
140 Kendrick Street
Building A, Suite 300
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*

Certain OMG specifications are also available as ISO standards. Please consult *http://www.iso.org*

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text

Helvetica/Arial - 10 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier - 10 pt. Bold:` Programming language elements.

Helvetica/Arial - 10 pt: Exceptions

Note – Terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to *http://www.omg.org/technology/agreement.htm*.

1 Scope

This specification defines the MOF to Text Template Language (Mof2Text), version 1.0. Mof2Text is aligned with UML 2.0, MOF 2.0, and OCL 2.0.

2 Conformance

There are four levels of compliance as shown below. A compliance level is defined in terms of the supported syntax and features.

Syntax Features	Core	Advanced
Abstract	Minimal	Intermediate
Concrete	Basic	Complete

- Abstract syntax compliance: The tool can read and interpret Mof2Text specifications in model form.
- Concrete syntax compliance: The tool can read and execute Mof2Text specifications in concrete syntax form. A tool supporting the concrete syntax also supports the abstract syntax.
- Core feature compliance: The tool supports core language features namely *Template*, *Query*, and *Module*.
- Advanced feature compliance: In addition to the core features, the tool also supports advanced features namely *Module extension, Template Overriding*, *Text mode switching*, and *Macros*.

3 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply.

- MOF 2.0 Core Specification (formal/2006-01-01)
- OCL 2.0 Specification (formal/2006-05-01)
- QVT Specification (ptc/05-11-01)
- UML 2.0 Superstructure Specification (formal/05-07-04)

4 Definitions and Terms

Block

A block groups text producing expressions of a template.

Macro

A Macro provides a way to extend the template language.

Module

A module is a mechanism for structuring transformation specifications.

ProtectedAreaBlock

A protectedAreaBlock identifies the text part that needs to be preserved across model-to-text transformations.

Template

A template specifies a text template with placeholders for data to be extracted from models.

TraceBlock

A trace block associates model elements, for traceability purpose, with a block of text to be generated.

5 Symbols

There are no symbols defined in this specification.

6 Additional Information

6.1 Acknowledgements

The following companies submitted and/or supported parts of this specification:

- Compuware Corporation
- France Telecom
- Interactive Objects Software GmbH
- Mentor Graphics Corporation
- Pathfinder Solutions
- SINTEF
- Softeam
- Tata Consultancy Services

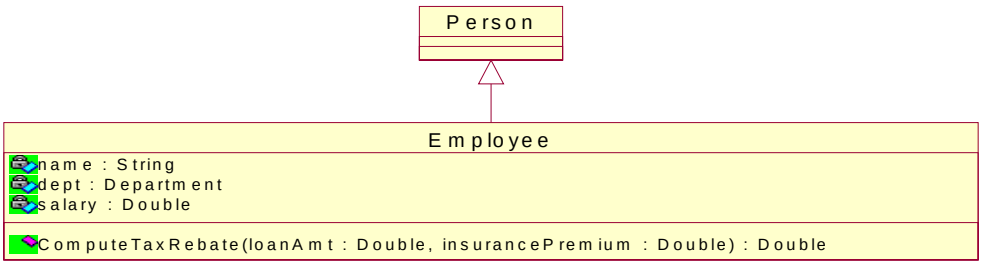
7 Overview

MDA places modeling at the heart of the software development process. Various models are used to capture various aspects of the system in a platform independent manner. Sets of transformations are then applied to these platform independent models (PIM) to derive platform specific models (PSM). These PSMs need to be eventually transformed into software artifacts such as code, deployment specifications, reports, documents, etc. QVT standard addresses the needs of model – to – model transformation (e.g., PIM – to – PIM, PIM – to – PSM and PSM – to – PSM). The MOF Model to Text (mof2text) standard addresses how to translate a model to various text artifacts such as code, deployment specifications, reports, documents, etc. Essentially, the mof2text standard needs to address how to transform a model into a linearized text representation. An intuitive way to address this requirement is a template based approach wherein the text to be generated from models is specified as a set of text templates that are parameterized with model elements.

A template-based approach is used wherein a *Template* specifies a text template with placeholders for data to be extracted from models. These placeholders are essentially expressions specified over metamodel entities with queries being the primary mechanisms for selecting and extracting the values from models. These values are then converted into text fragments using an expression language augmented with a string manipulation library. *Template* can be composed to address complex transformation requirements. Large transformations can be structured into *modules* having public and private parts.

For example, the following *Template* specification generates a Java definition for a UML class.

```
[template public classToJava(c : Class)]
class [c.name/]
{
   // Constructor
   [c.name/]()
   {
   }
}
[/template]
```



For a class ‘Employee’ (shown in the figure above), the following text will be generated:

```
class Employee
{
      // Constructor
      Employee()
      {
      }
}
```

As shown above, the specification has a WYSIWYG character with the output preserving indentation, white spaces, etc. from the specification.

A *Template* can invoke other *Templates*. Invocation of a *Template* is equivalent to *in situ* placement of the text produced by the *Template* being invoked.

```
[template public classToJava(c : Class)]
class [c.name/]
{
   // Attribute declarations
   [attributeToJava(c.attribute)/]

   // Constructor
   [c.name/]()
   {
   }
}
[/template]

[template public attributeToJava(a : Attribute)]
[a.type.name/] [a.name/];
[/template]
```

Template *classToJava* invokes template *attributeToJava* for each attribute of the class and puts ';' as a separator between the text fragments produced by each invocation of *attributeToJava* template.

```
class Employee
{
      // Attribute declarations
      String name;
      Department dept;
      Double salary;

      // Constructor
      Employee()
      {
      }
}
```

Instead of defining two templates separately, a template can iterate over a collection by using the *for* block. Using the *for* block preserves WYSIWYG-ness and improves readability.

For example the *classToJava* template above can use the *for* block as shown below:

```
[template public classToJava(c : Class)]
class [c.name/]
{
   // Attribute declarations
   [for(a : Attribute | c.attribute)]
   [a.type.name/] [a.name/];
   [/for]
```

```
   // Constructor
   [c.name/]()
   {
   }
}
[/template]
```

The for block declares a loop variable '**a**' of type Attribute and produces for each Attribute in the collection **c.attribute** the text between the *[for]* and *[/for]*.

A template can have a guard that decides whether the template can be invoked. For example, the following *classToJava* template is invoked only if the class is concrete.

```
[template public classToJava(c : Class) ? (c.isAbstract = false)]
class [c.name/]
{
   // Attribute declarations
   [attributeToJava(c.attribute)/]

   // Constructor
   [c.name/]()
   {
   }
}
[/template]
```

Complex model navigations can be specified using *queries*. The following example shows use of a query *allOperations* to collect operations of all abstract parent classes of a class in a class hierarchy.

```
[query public allOperations(c: Class) : Set ( Operation ) =
c.operation->union( c.superClass->select(sc|sc.isAbstract=true)-
>iterate(ac : Class;
         os:Set(Operation) = Set{}| os->union(allOperations(ac)))) /]

[template public classToJava(c : Class) ? (c.isAbstract = false)]
class [c.name/]
{
// Attribute declarations
   [attributeToJava(c.attribute)/]
   // Constructor
   [c.name/]()
   {
   }
   [operationToJava(allOperations(c))/]
}

[/template]

[template public operationToJava(o : Operation)]
[o.type.name/] [o.name/] ([for(p:Parameter | o.parameter)
separator(',')] [p.type/] [p.name/] [/for]);
[/template]
```

Testing if a modelelement is of a certain type, and if it is, declaring a variable of that type that is used later in the transformation is supported directly by specifying a let block statement.

```
[template public classToJava(c : Class)]
[let ac : AssociationClass = c ]
class [c.name/]
{
   // Attribute declarations
   [attributeToJava(c.attribute)/]
   // Constructor
   [c.name/]()
   {
   }

   // Association class methods
   [for (t:Type | ac.endType)]
   Attach_[t.name/]([t.name/] p[t.name/])
   {
   // Code for the method here
   }
   [/for]
}
[/let]
[/template]
```

The let block tests whether the actual argument c is of type AssocationClass, and if it is, it declares a variable ac of type AssociationClass that can be used inside the let block.

If the test fails, the let block does not produce any text.

Large transformation specifications can be structured into *Module*s. A *Module* consists of a set of *Templates* and *Queries* and has a public and a private part. Public part exposes *Templates* and *Queries* that can be invoked from other modules. A *Module* can have import dependency on other *Module*s. This allows the importing *Module* to invoke the *Templates* and *Queries* exported by the imported *Module*s.

A transformation (template) can be started by directly invoking a public template with the correct parameters. There is no explicit notion of a main template.

A *Template* can override one or more other *Templates*. A *Module* can extend another *Module* by overriding some of its *Templates*. A Module can extend another module (inheritance) in a sub-super relationship. Only single inheritance is supported. The specializing module inherits all templates from its super and can access or override all public and protected templates.

Overriding is a mechanism to selectively modify the behavior of a *Module*. An overriding template should have the same number of parameters as the overridden template with compatible types. The overriding template is invoked in place of the overridden template when the parameter types match and the guard condition of the former evaluates to true.

7.1 Escape Direction

A template has WYSIWYG nature with the text to be output being specified in exactly the way it should look in the output. In most of the cases, this style of template specification is intuitive where the text producing logic is specified in quoted form i.e., delimited by '[' and ']'. However, there may be cases where the quantity of the text producing logic far outweighs the text being produced. In this case, it is more intuitive to specify the text producing logic without use of special delimiters. This is achieved by setting the escape direction to the required mode i.e., *text-explicit* or *code-explicit* with text-explicit being the default mode.

Syntax for escape direction is similar to java annotations, with possible parameters to control the escape character(s) used:

```
@text-explicit
```

Or

```
@code-explicit
```

Example

```
@text-explicit
[template public classToJava(c : Class)]
class [c.name/]
{
   // Constructor
   [c.name/]()
   {
   }
}
[/template]

@code-explicit
template public classToJava(c : Class)
'class 'c.name '
{
   // Constructor
   'c.name'()
   {
   }
}'
/template
```

In code explicit form, the output text is escaped instead of the transformation (template) code. The characters used to provide escaping in the two different modes has a default value, but may be modified as parameter to the @code/text-explicit annotation.

In code explicit mode, blocks are ended using a slash followed by the block keyword (e.g., for … /for).

```
for(a : Attribute | c.attribute) separator( ',' )
  a.name ' ' a.type.name
/for
```

Some additional examples on code-explicit mode of the previous code snippets are given below:

```
query public allOperations(c: Class) : Set ( Operation ) =
c.operation->union( c.superClass->select(sc|sc.isAbstract=true)->iterate(ac : Class;
        os:Set(Operation) = Set{}| os->union(allOperations(ac))))
/query

template public classToJava(c : Class) ? (c.isAbstract = false)]
‘class [c.name/] ‘
{
   // Attribute declarations
   ‘ attributeToJava(c.attribute)’

   // Constructor
   ‘ c.name ‘()
   {
   }
      ‘operationToJava(allOperations(c))’
}’
/template

template public operationToJava(o : Operation)
o.type.name ‘ ‘ o.name ‘ (‘ for(p:Parameter | o.parameter)
separator(',') p.type ‘ ‘ p.name /for’);’
/template
```

The alternative with a type checking let clause.

```
template public classToJava(c : Class)
let ac : AssociationClass = c
‘class ‘c.name’
{
   // Attribute declarations
   ‘attributeToJava(c.attribute)’

   // Constructor
   ‘c.name’()
   {
   }

   // Association class methods
   ‘for (t:Type | ac.endType)’
   Attach_’t.name’(‘t.name’ p’t.name’)
   {
   // Code for the method here
   }
   ‘/for’
}
‘/let
/template
```

The escape characters

The escape characters for text-explicit and code-explicit has a default representation.

For text-explicit mode, [and] delimiters are used: `TABLE [c.name/] (`

For code-explicit mode, the single quote character is used: ‘TABLE ‘ c.name ‘(

The escape character(s) used can be changed by parameters given to the escape direction annotation property: e.g., @code-explicit (#) – Which defines ‘#’ as the escape character for escaped text. Normally, however, the default escape should be used.

7.2 Traceability

A Trace block relates text that is produced in a block to a set of model elements that are provided as parameters. A Trace block will typically lead to comments in the produced text. Model based code generation is one of the principal applications of model to text transformation. Mof2Text provides support for tracing model elements to text parts. Text parts to be traced must be delimited with special keywords. Additionally, text parts may be marked as protected. Such text parts are preserved and not overwritten by subsequent model-to-text transformations. Information concerning the originating template for text output should also be part of the trace information.

Text parts must be able to relate unambiguously to a set of model elements and must have a unique identification.

```
[template public classToJava(c : Class)]
[trace(c.id()+ '_definition') ]
class [c.name/]
{
   // Constructor
   [c.name/]()
   {
   [protected('user_code')]
   ; user code
   [/protected]

   }
}
[/trace]
[/template]
```

In the example above, the trace block identifies the text to be traced by relating the generated text to model element ‘c’ of type Class. The protected block identifies the text part that needs to be preserved between subsequent model-to-text transformations. It produces delimiters in the output text to clearly identify the protected part. Since such delimiters are specific to a target language, they are not defined in this standard. An implementation tool is responsible for producing the correct delimiters for the desired target language.

7.3 Directing Output to Files

The file block specifies the file to which the generated text should be sent. The file block has three parameters; a uri which denotes the name of the file, a Boolean flag indicating whether the file is to be opened in append mode or not, and an optional unique id, typically derived from modelelement identifiers as in the traceability specification. For instance, a transformation tool can use this id to find a file that was generated in a previous session even when a modelelement was renamed (and the modelelement name was used in the uri of the file) or when the file name has been changed by the

template writer. This will enable protected text parts, if any, to be preserved across transformations. File blocks can be nested with the file associated with the current file block receiving the output. Files may be opened in ‘append’ mode. The default mode of opening files is ‘overwrite.’ This is useful for writing debug information in a log file, for example.

Example

```
[template public classToJava(c : Class)]
[file (‘file:\\’+c.name+’.java’, false, c.id + ‘impl’)]
[file('log.log', true)]processing [class.name/][/file]
class [c.name/]
{
   // Constructor
   [c.name/]()
   {
   }
}
[/file]
[/template]
```

Suppose the above specification was run on a class named ‘cust,’ it would produce Java code in cust.java file and a log entry ‘processing cust’ in log.log file. Suppose after generation, the storage specification was added in the protected area of file cust.sql. Even if the classname is changed later, say to ‘customer,’ a tool will be able to retain the storage specification in the new file customer.sql as the file block takes unique id as a parameter that hasn’t changed (for ‘cust’ object). The uri ‘stdout’ denotes the stdout output stream.

7.4 Macros

Macros provide a way to extend the language. A macro can be used in template specifications.

An example of a macro definition:

```
[macro javaMethod(Type type, String methodName, String resultName, Body body)]
public [typeName(type)/] [methodName/] () {
   [typeName(type)/] [resultName/] = null;
   [body/]
   return [resultName];
}
[/macro]
```

The macro can be invoked as follows:

```
[javaMethod([query.oclExp.type/], query.name, "result")]
    result = [javaocl(query.oclExp)/];
[/javaMethod]
```

A macro must have next to a number of ordinary formal parameters, one parameter of predefined type Body. In the example, the body of the macro call javaMethod is passed to the Body parameter ‘body,’ and the invoked call inserts the body where body is referred. The macro is expanded in the context of the macro call.

Macros can be used to implement comments. The following macro specifies a comment block implementation:

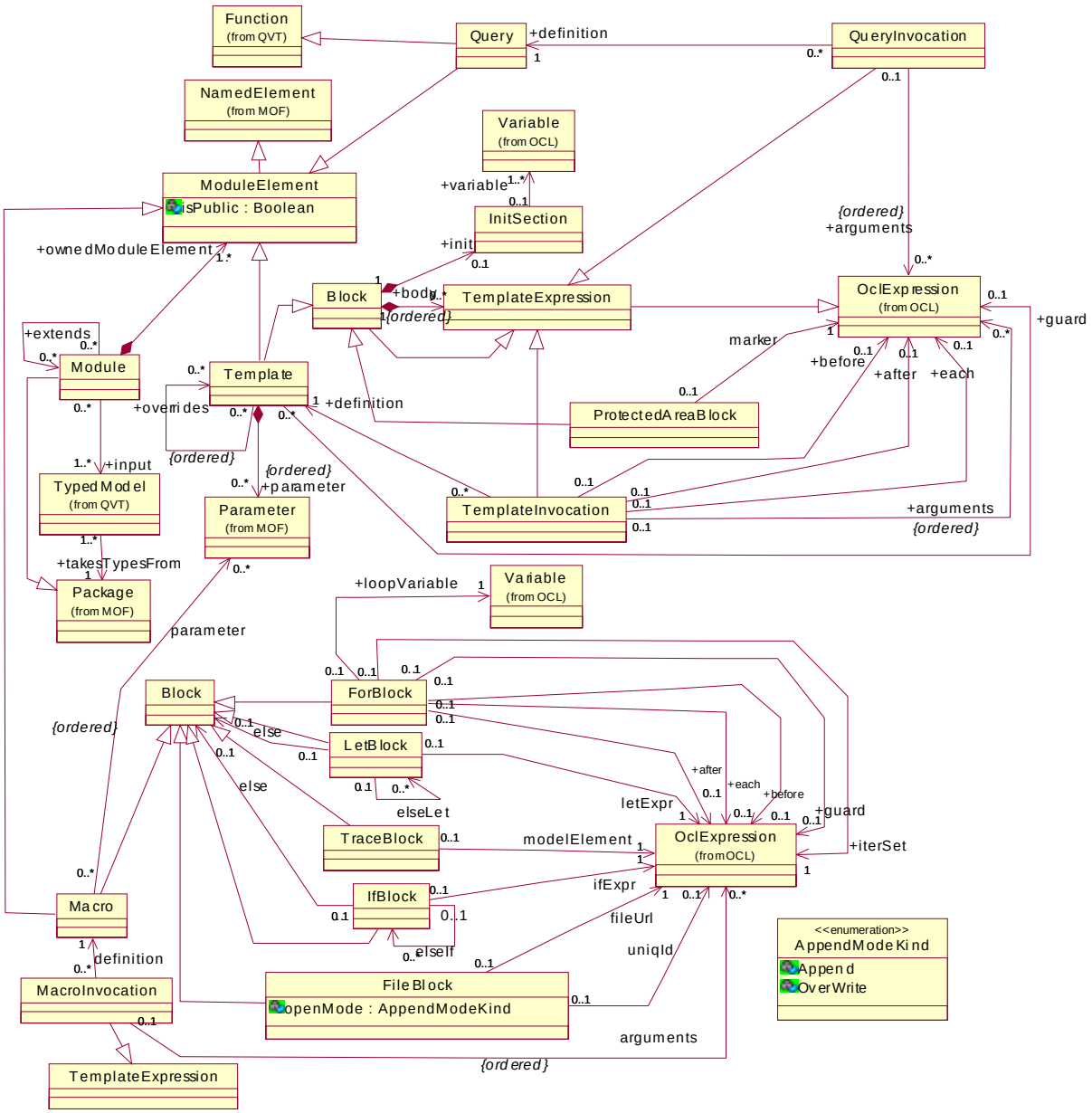
```
[macro comment (Body b)][/macro]
```

The macro can be used as follows:

```
[comment()]
... some code here
[/comment]
```

8 Template Language Specification

8.1 Metamodel



8.1.1 Module

A *module* is a mechanism for structuring transformation specifications. It defines a namespace for the *module elements* it contains. A *module* has a public and a private part. Public part exposes *module elements* that can be used by other modules. A *module* can have import dependency on other modules whose public *module elements* it can use. A *module* can extend one or more other modules by overriding some of their *templates*.

Supertypes

- Package
  A module behaves like a package in providing a namespace for its contents.

Associations

- *input*
  TypedModel containing the model elements to be transformed to text
- *ownedModuleElement*
  A set of contained *module elements*
- *extends*
  A set of modules being extended by this module

8.1.2 ModuleElement

Module element is an abstract class that is specialized by *template* and *query.* A *module element* has a unique name within the containing *module*, and must indicate whether it's public i.e., whether it can be used outside the containing *module*. Input parameters of a *module element* are *Parameters* whose types must be MOF types. These types must be defined in the meta model package specified by the typed model associated with the containing *module*.

Supertypes

- NamedElement

Attributes

- *isPublic : Boolean*
  Specifies whether the *module element* is visible outside the contained *module*.

Associations

- *parameter*
  Input parameters of the *module element*

8.1.3 Template

A *template* specifies a text template with placeholders for data to be extracted from models. These placeholders are expressions specified in terms of metamodel entities and are evaluated over instances of these metamodel entities. *Template* is a specialization of *block*. A *template* can have a *guard* (inherited from *block*) that specifies when it can be invoked. A *template* can override one or more other *templates*. An overriding *template* should have the same number of parameters as the overridden *template* with compatible types. The overriding *template* is invoked in place of the overridden *template* when the parameter types match and the guard condition of the overriding *template* evaluates to true.

In case of a *template* being overridden by multiple *templates*, the *guard* should be specified such that only one of the overriding *templates* is selected. In case of *guards* of more than one overriding *templates* evaluating to true, one of them will get selected arbitrarily.

Consider a *template* T1 with parameter type PT1 being overridden by a *template* T2 with parameter type PT2. If PT2 is the same as PT1, then invocation of T1 will result in invocation of T2. If PT2 is a subtype of PT1, then invocation of T1 with instance of PT2 as an argument will result in invocation of T2 and invocation of T1 with an argument that is an instance of PT1 but not an instance of PT2 will result in invocation of T1.

If PT2 is a supertype of PT1, then invocation of T1 with instance of PT1 or PT2 as an argument will result in invocation of T2. If T1 and T2 have more than one parameter, then if for any of the parameters, the decision is in favor of T1, then T1 is invoked.

An overriding *template* can invoke the overridden *template* from its body using [super/] expression. A template can override one or more templates.

Let's say that T2 overrides T1 and T3. Then T2 may be invoked in three different scenarios:

1. Invocation of T1 resulted in invocation of T2. In this case super points to T1.
2. Invocation of T3 resulted in invocation of T2. In this case super points to T3.
3. T2 is directly invoked. In this case we have ambiguity. This is solved by a default resolution strategy: we choose the first listed template (in this case T1).

It is possible to specify protected sections in the generated text for manual additions. Subsequent applications of the *template* preserve this text.

Supertypes

- Block

Associations

- *guard*
  Guard condition of the *Template*
- *parameter*
  Parameters of the *template*
- *overrides*
  A set of *templates* being overridden by this *template*

8.1.4 Block

A *block* groups text producing expressions of a *template*. A *block* can have *init section* that initializes a set of variables that can be used in its *body.* A *block* can contain other *blocks*.

Supertypes

- TemplateExpression

Associations

- *body*
  An ordered set of *template expressions* specifying the text of the *block*
- *init*
  A set of variable initializations to be used in the *body* of the *block*

8.1.5 InitSection

An *InitSection* contains a set of variable initializations to be used in the *body* of its owning *block*.

Associations

- *body*
  An ordered set of *template expressions* specifying the text of the *block*

8.1.6 TemplateExpression

A *template expression* specializes *ocl expression* for the purpose of text generation. The type of a *template expression* specifying *body* of a *template* must be a *String*.

Supertypes

- OCLExpression

Subtypes

- ProtectedAreaExpression
- TemplateInvocation
- QueryInvocation

8.1.7 ProtectedAreaBlock

A *protected area block* identifies the text part that needs to be preserved across model-to-text transformations. It produces delimiters in the output text to clearly identify the protected part. Changes introduced in the protected part are preserved in subsequent transformations.

Supertypes

- Block

Associations

- *marker*
  Expression for producing the begin and end markers of the delimited section.

8.1.8 ForBlock

A *for block* specifies a text segment that needs to be processed repeatedly over a set of *model elements*. A *for block* can have a guard that specifies when the body of the block can be executed.

Supertype

- Block

Associations

- *iterSet*
  The set over which the loop *body* is processed iteratively.
- *loopVariable*
  The variable that binds to an element of *iterSet*.
- *body*
  Specifies the loop *body* (inherited from *block*).
- *each*
  Expression being evaluated after every iteration except the last
- *before*
  Expression being evaluated before the first iteration.
- *after*
  Expression being evaluated after the last iteration.
- *guard*
  Guard condition of the *For.*

8.1.9 QueryInvocation

A *QueryInvocation* is an expression that specifies invocation of a *Query.*

Associations

- *arguments*

8.1.10 TemplateInvocation

A *template invocation* specifies one or more invocations of a *template*. An argument of a *template invocation* is specified by an expression that could evaluate either to a single value or a set of values. The rules for determining which *template* to invoke and how many times are as follows:

- The types of arguments should match the types of the corresponding parameters. An argument type matches the parameter type when the latter is either the same type or a super type. If the argument is a set and the parameter is a singleton, then the *template* is invoked for each member of the set.
- If the argument is a set and the parameter is a set, then the *template* is invoked once.
- If the argument is a singleton and the parameter is a set, then the *template* is invoked once with the singleton set.

- If the *template* has k singleton parameters and the corresponding arguments are sets, then the *template* is invoked for each member of the cross product of the k sets.
- If a *template* is overridden, then the overriding *template* is invoked as described earlier.

Associations

- *definition*
  The *template* being invoked
- *arguments*
  Expressions evaluating the arguments
- *each*
  Expression being evaluated after each invocation except the last when a *template* is invoked multiple times.
- *before*
  Expression being evaluated before the first invocation of the *template*.
- *after*
  Expression being evaluated after the last invocation of the *template*.

8.1.11 TypedModel

Reused from QVT. A *typed model* specifies candidate input model to be transformed to text. At runtime, a model that is passed to the transformation is constrained to contain only those model elements whose types are specified in the set of model packages associated with the typed model.

Associations

- *takesTypesFrom*
  Package containing the metamodel which provides the types for the model.

8.1.12 Package

Reused from MOF 2.0.

8.1.13 Parameter

Reused from MOF 2.0.

8.1.14 Function

Reused from QVT.

8.1.15 Query

As specified by its supertype *function*, a *query* is a side-effect-free operation. It is owned by a module. A *query* is required to produce the same result each time it is invoked with the same arguments. A *query* is specified by an OCL expression.

Supertypes

- Function, ModuleElement

8.1.16 LetBlock

A let block declares and initializes a variable of type subtype when the cast is successful. Essentially, it can be seen as a test that checks whether the object bound to the supertype variable is actually an instance of the specified subtype and execution of the associated block on success. Multiple such tests can be grouped together in a Let statement on the lines of if-elseif chain.

Supertypes

Block

Associations

- *letExpr*
  The subtype testing and assignment expression
- *elseLet*
  A chain of alternate Let blocks on the lines of if-elseif chain
- *else*
  A block to be executed when none of the let expressions match

8.1.17 FileBlock

A *file block* specifies the file to which the generated text should be sent. The file block has a uri that denotes the name of the file and an optional unique id, typically derived from modelelement identifiers. For instance, a transformation tool can use this id to find a file that was generated in a previous session even when a modelelement was renamed (and the modelelement name was used in the uri of the file) or when the file name has been changed by the template writer. This will enable protected text parts, if any, to be preserved across transformations. A file may be opened in append mode.

Supertypes

- Block

Attributes

- *appendMode : AppendModeKind*
  Mode in which the file is to be opened

Associations

- *fileUrl*
  File to be opened
- *uniqId*
  Unique id associated with the file

8.1.18 TraceBlock

A *trace block* associates model elements, for traceability purpose, with a block of text to be generated.

Supertypes

- Block

Associations

- *modelElement*
  modelelement to be associated with the text block

8.1.19 Macro

A Macro provides a way to extend the language. A macro can be invoked from a template body.

Supertypes

- Block, ModuleElement

Associations

- *parameter*
  Parameters of the macro. The last parameter must be of a special type Body.

8.1.20 MacroInvocation

A *MacroInvocation* is an expression that specifies invocation of a *Macro*.

Supertypes

- TemplateExpression

Associations

- *arguments*
  The last argument must be of special type *TemplateExpression*

8.1.21 IfBlock

An *If block* allows specification of conditional execution of the associated template block. Multiple such conditions can be grouped together in an if-elseif chain.

Supertypes

- Block

Associations

- *ifExpr*
  Conditional expression that needs to evaluate to true for the associated block to be executed.

- *elseIf*
  A chain of conditional blocks.
- *else*
  A block to be executed when none of the conditional expressions evaluate to true.

8.2 Concrete Syntax

The concrete syntax is defined using EBNF.

Text following '//' upto the end-of-line is treated as a comment. Grammar does not specify a rule for comments.

Keywords
module, import, extends, template, query, public, private, protected, guard, init, overrides, each, before, after, for, if, elseif, else, let, elselet, trace, macro, file, mode, text_explicit, code_explicit, super, stdout

Grammar

```
<module> ::= <module_decl> ( <import_decl> )*  [ <queries_section> ] [
<templates_section> ] [ <macros_section> ]

<module_decl> ::= '[module' <PathNameCS> '(' <PathNameCS> ')'
[extends_decl] '/]' |
         'module' <PathNameCS> [extends_decl]

<extends_decl> ::= 'extends' <PathNameCS> ( ',' <PathNameCS> )*

<import_decl> ::= '[import' <PathNameCS> '/]' |
'import' <PathNameCS>

<queries_section> ::= ( <query_defn> | <query_defn_code> )*

<query_defn> ::= '[query' <visibility> <PathNameCS> '(' <arglist> ')' ':' <typeCS>
'=' <OclExpressionCS> '/]'
<query_defn_code> ::= 'query' <visibility> <PathNameCS> '(' <arglist> ')' ':'
<typeCS> '=' <OclExpressionCS>


<visibility> ::= 'public' | 'private'

<arglist> ::= (arg_decl ( ',' arg_decl )* )?

<arg_decl> ::= <SimpleNameCS> ':' <typeCS>

<actualarglist> ::= ( <OclExpressionCS> ( ',' <OclExpressionCS> )* )?

<macros_section> ::= ( <macro_defn> | <macro_defn_code> )*

<templates_section> ::= ( <mode>? <template_defn> )*
```

```
<template_defn>   ::= <template_defn_text> | <template_defn_code>
<template_defn_text> ::= '[template' <signature> ‘]’ <production> '[/template]'
<template_defn_code> ::= template <signature> <production_code> ‘/template’

<signature>  ::= <visibility> <PathNameCS> '(' <arglist> ')' <overrides>? [ <guard> ]
[ <init> ]

<macro_defn> ::= '[macro <PathNameCS> '(' <arglist> ')' ']' <production> '[/macro ]'
<macro_defn_code> ::= 'macro <PathNameCS> '(' <arglist> ')'  <production_code> '/
macro'

<overrides> ::= 'overrides' PathNameCS ( ',' <PathNameCS> )*

<production>::=( <filecmd> | <literal> | <protected> |
<tracecmd> | <forcmd> | <ifcmd> | <letcmd> | '[' <OclExpressionCS> '/]' )*
<production_code>::= (<filecmd_code> | <literal_code> | <protected_code>|
<tracecmd_code> | <forcmd_code> | <ifcmd_code> | <letcmd_code> | <OclExpressionCS>
)*

<guard> ::= '?' '(' <OclExpressionCS> ')'


<init> ::= '{' (  VariableDeclarationCS ';' )+ '}'

<filecmd> ::= '[file' '(' (<OclExpressionCS> | 'stdout' ) [ ‘,’ <OclExpessionCS>] [
‘,’ <OclExpressionCS> ] ')' ']' <production> '[/file]'

<filecmd_code> ::= ‘file’ ‘(' (<OclExpressionCS> | ‘stdout’ ) [ ‘,’ <OclExpessionCS>]
[ ‘,’ <OclExpressionCS> ] ')' <production_code> '/file'


<protected> ::= '[protected' '(' <OclExpressionCS> ')' ']' <production> '[/pro-
tected]'
<protected_code> ::= 'protected' '(' <OclExpressionCS> ')'  <production_code> '/pro-
tected'


<tracecmd> ::= '[trace '(' <OclExpressionCS> ')' ']' <production> '[/trace]'
<tracecmd_code> ::= 'trace '(' <OclExpressionCS> ')'  <production_code> '/trace'


<OclExpressionCS> ::= ( <PropertyCallExpCS> | <VariableExpCS> | <LiteralExpCS> |
<LetExpCS> | <OclMessageExpCS> | <ifExp> |  <invocation> )

<ifExp> ::= <OclExpressionCS> '?' <OclExpressionCS> ':' <OclExpressionCS>

<invocation> ::= ( <PathNameCS> '(' <actualarglist> ')' | 'super' ) [ <before> ] [
<separator> ] [ <after> ]

<before> ::= 'before' '(' <OclExpressionCS> ')'
```

```
<separator> ::= 'separator' '(' <OclExpressionCS> ')'

<after> ::= 'after' '(' <OclExpressionCS> ')'

<forcmd> ::= '[for' '(' <arg_decl> '|' <OclExpressionCS> ')' [ <before> ] [ <separa-
tor> ] [ <after> ] [ <guard> ] [ <init> ] ']' <production> '[/for]'
<forcmd_code> ::= 'for' '(' <arg_decl> '|' <OclExpressionCS> ')' [ <before> ] [ <sep-
arator> ] [ <after> ] [ <guard> ] [ <init> ] <production_code> '/for'


<ifcmd> ::= '[if' '(' <OclExpressionCS> ')' ']' <production> (<elseif>  )* [ <else> ]
'[/if]'
<ifcmd_code> ::= 'if' '(' <OclExpressionCS> ')' <production_code> (<elseif_code>  )*
[ <else_code> ] '/if'


<elseif> ::= '[elseif' '(' <OclExpressionCS> ')' ']' <production> ('[/elseif]')?
<elseif_code> ::= 'elseif' '(' <OclExpressionCS> ')' <production_code> ('/elseif') ?

<else> ::= '[else]' <production> ('[/else]')?
<else_code> ::= 'else' <production_code> ('/else')


<letcmd> ::= '[let' <VariableDeclarationCS> ']' <production> (<elselet>  )* [ <else>
] '[/let]'
<letcmd_code> ::= 'let' <VariableDeclarationCS> <production_code> (<elselet_code>
)* [ <else_code> ] '/let'

<elselet> ::= '[elselet]' <VariableDeclarationCS> <production> '[/elselet]'
<elselet_code> ::= 'elselet' <VariableDeclarationCS> <production_code> '/elselet'

<mode> ::= '@' 'text-explicit' |
     '@' 'code-explicit'

<literal> is a text string not enclosed in quotes
<literal_code> is a text string enclosed in single quotes
```

8.3 Library

The OCL String library has been extended with the following functions. Since OCL understands MOF operations, these functions need to be wrapped as operations of some class not necessarily from the source model being transformed.

8.3.1 String

substitute(String r, String t) : String

Substitutes substring *r* in *self* by substring *t* and returns the resulting string. If there is no occurrence of the substring, it returns the original string.

index(String r) : Integer

Returns the index of substring *r* in *self*, or -1 if *r* is not in *self*.

first(Integer n) : String

Returns first *n* characters of *self*, or *self* if size of *self* is less than *n*.

last(Integer n) : String

Returns last *n* characters of *self*, or *self* if size of *self* is less than *n*.

strstr(String r) : Boolean

Searches for string *r* in *self*. Returns true if found, false otherwise.

toUpper() : String

Creates a copy of *self* with all characters converted to uppercase and returns it.

toLower() : String

Creates a copy of *self* with all characters converted to lowercase and returns it.

strtok(String s1, Integer flag) : String

Breaks the string *self* into a sequence of tokens each of which is delimited by any character in string *s1*. The parameter *flag* should be 0 when *strtok* is called for the first time, 1 subsequently.

strcmp(String s1) : Integer

Returns an integer less than zero, equal to zero, or greater than zero depending on whether *s1* is lexicographically less than, equal to, or greater than *self*.

isAlpha() : Boolean

Returns true if *self* consists only of alphabetical characters, false otherwise.

isAlphanum() : Boolean

Returns true if *self* consists only of alphanumeric characters, false otherwise.

toUpperFirst() : String

Creates a copy of *self* with first character converted to uppercase and returns it.

toLowerFirst() : String

Creates a copy of *self* with first character converted to lowercase and returns it.

8.3.2 Integer

toString(Integer i) : String

Converts the integer *i* to a string.

8.3.3 Real

toString(Real r) : String

Converts the real *r* to a string.

8.4 Whitespace Handling

Text production rules will be easier to understand by viewing the body of a template (or a block expression) as follows:

<block-body> ::= <body-element>*
<body-element> ::= <literalString> | <whitespace> | <expression> | <BOL-indicator>
<whitespace> ::= space | tab | newline
<expression> ::= <stand-alone-block-expression> | <embedded-block-expression> |
<stand-alone-template-invocation> | <embedded-template-invocation> |
<other-expression>
<BOL-indicator> ::= '^'

<stand-alone-block-expression> is a block expression (single or multi-line) that is not surrounded by other body elements on the lines where the block head and the tail occur.

<embedded-block-expression> is a block expression that is surrounded by other non-whitespace body elements on the lines where block head or tail occur.

<stand-alone-template-invocation> is a template invocation expression that stands on a line all by itself.

<embedded-template-invocation> is a template invocation that is surrounded by other body elements on the same line.

<other-expression> refers to all expressions other than the block and template invocation expressions.

Text production rules:

- All body elements produce text
- The text output of a block-body is the sequential concatenation of the text outputs of its body elements
- Rules for identifying starting and ending of block body:
  - template: body starts at the beginning of the next line after the template head, and ends on the last character (excluding the new line) of the line previous to the template tail.
  - multi-line-block: body starts at the beginning of the next line after the block head, and ends on the last character (excluding the new line) of the line previous to the block tail.
  - single-line-block: body starts after the closing bracket of the block head and ends before the starting bracket of the block tail.
- Text outputs of different body elements:
  - A literal string is output as is.
  - A whitespace character is output as is.
  - The text produced by the execution of <other-expression> is output as is.
  - The text produced by the execution of <embedded-block-expression> is output as is.

- The text produced by the execution of <embedded-template-invocation> is output as is.
- <stand-alone-block-expression> needs special handling:
  - Ignore the whitespace characters occurring before the beginning of the head.
  - In the case of a multi-line ‘for block,’ when a separator character is not specified, use ‘new line’ as the default separator between the outputs produced by successive iterations.
- <stand-alone-template-invocation> needs special handling:
  - Add the whitespace preceding the invocation expression before each line of the text produced by the invoked template.
  - In the case of an iterative template invocation (e.g., when the argument is a collection), if a separator character is not specified, use ‘new line’ as the default separator between the outputs produced by successive iterations.

<BOL-indicator> is a special character (‘^’) that marks the beginning of a line – i.e., on the line on which this character appears, the whitespace preceding the character should be excluded from the text output.

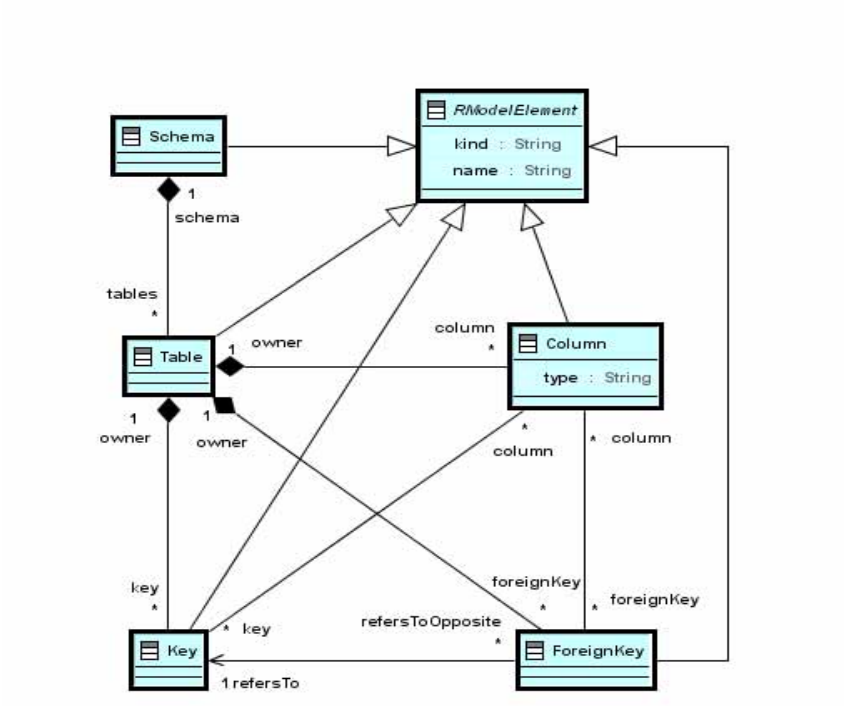
In code-explicit mode, all whitespace must be explicitly specified.

Annex A: Examples

(normative)

A.1 Example 1

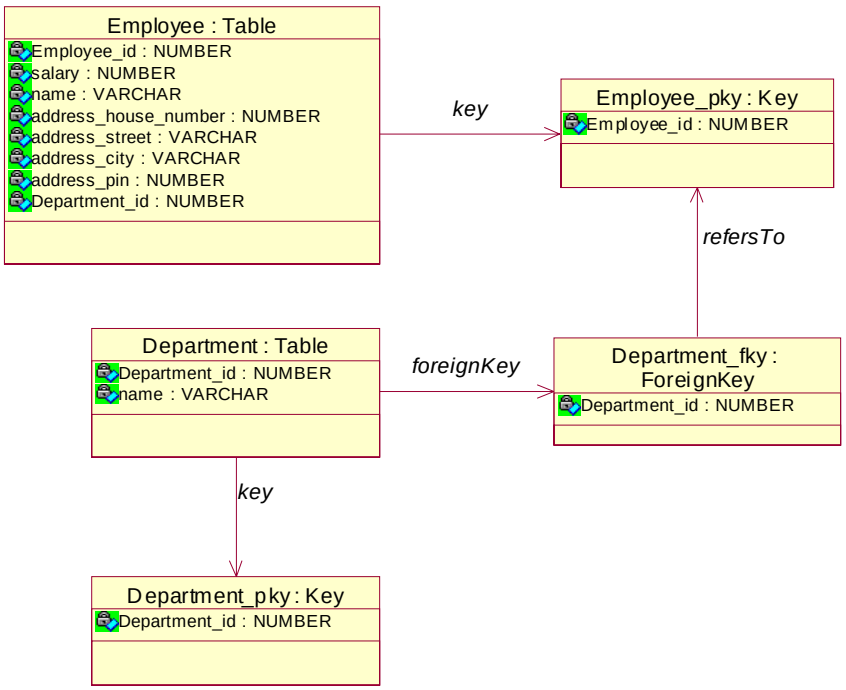
The example shows transformation specifications for transforming an RDBMS model to Oracle DDL. A simplified RDBMS metamodel is shown below.




```
[module DDLgen(RDBMS)/]

[template public SchemaToDDL (s: Schema)]
[for (t:Table | s.table)]
[TableToDDL(t)/]
[/for]
[/template]

[template public TableToDDL(t: Table)]
CREATE TABLE [t.name/] (
[for (c:Column|t.column) separator(',')]
[c.name/] [c.type/]
```

```
[/for]
);
[KeyToDDL(t.key)/]
[foreignKeyToDDL(t.foreignKey)/]
[/template]

[template private KeyToDDL(k:Key)]
ALTER TABLE [k.owner.name/] ADD (
CONSTRAINT [k.name/] PRIMARY KEY ([for (c:Column|k.column) separator(',')]c.name[/
for])
);
[/template]

[template private ForeignKeyToDDL(fk:ForeignKey)]
ALTER TABLE [fk.owner.name] ADD (
CONSTRAINT [fk.name/] FOREIGN KEY ([for (c:Column|fk.column) separator(',')]c.name[/
for])
REFERENCES [fk.refersTo.owner.name/] ([for (c:Column|fk.refersTo.column) separa-
tor(',')]c.name[/for])
ON DELETE CASCADE
);
[/template]

[/module]
```

A sample input model is shown below where Employee and Department are persistent classes.



Generated text is shown below.

```
CREATE TABLE Employee (
   Employee_id NUMBER,
```

```
   salary NUMBER,
   name VARCHAR,
   address_house_number NUMBER,
   address_street VARCHAR,
   address_city VARCHAR,
   address_pin NUMBER,
   Department_id NUMBER
);

ALTER TABLE Employee ADD (
   CONSTRAINT Employee_pky PRIMARY KEY (Employee_id)
);

ALTER TABLE Employee ADD (
   CONSTRAINT Department_fky FOREIGN KEY (Department_id)
      REFERENCES Department (Department_id)
      ON DELETE CASCADE
);

CREATE TABLE Department (
   Department_id NUMBER,
   name VARCHAR
);

ALTER TABLE Department ADD (
   CONSTRAINT Department_pky PRIMARY KEY (Department_id)
);
```

A.2 Example2

Above example in code explicit mode.

```
module DDLgen(RDBMS)

template public SchemaToDDL (s: Schema)
for (t:Table | s.table)
TableToDDL(t)
/for
/template

template public TableToDDL(t: Table)
'CREATE TABLE 't.name ' (
‘  for (c:Column|t.column) separator(',')’
‘c.name ' ' c.type’
‘/for
');'
KeyToDDL(t.key)
foreignKeyToDDL(t.foreignKey)
/template

template private KeyToDDL(k:Key)
'ALTER TABLE 'k.owner.name 'ADD (
CONSTRAINT ' k.name ' PRIMARY KEY (' for(c:Column|k.column) separator(',') c.name /
for ')
);'
/template

template private ForeignKeyToDDL(fk:ForeignKey)
'ALTER TABLE ' fk.owner.name ' ADD (
```

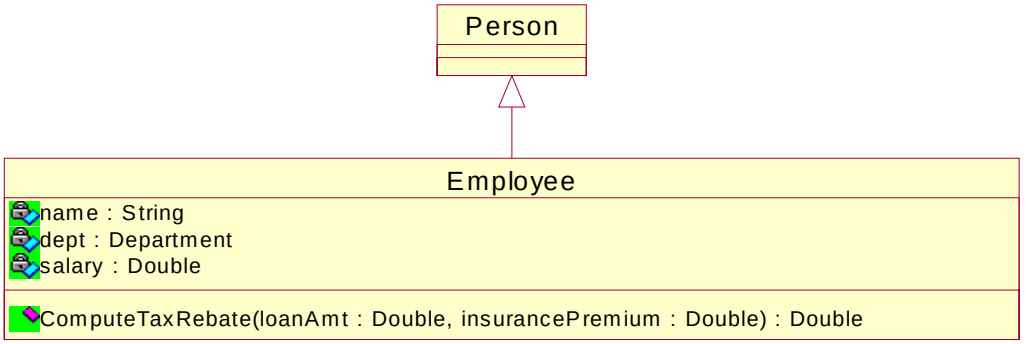
```
CONSTRAINT ' fk.name ' FOREIGN KEY (' for (c:Column|fk.column) separator(',') c.name
/for ')
REFERENCES ' fk.refersTo.owner.name ` ('for (c:Column|fk.refersTo.column) separa-
tor(',') c.name /for ')
ON DELETE CASCADE
);'
/template
/module
```

A.3 Example3

This example shows a template for generating C++ class header from a UML class model. It generates getters and setters for the attributes, a bitvector member to keep track of which attributes have values set and which do not, and a constructor for the class that initializes the bitvector member. User can further modify the constructor body between the generated delimiters.

```
[module class_header_gen /]

[template public class_header(c : Class) { int count = -1; } ]
[file (c.name +'.cpp', false)]
[trace(c.id() +'_header')]

  // Bit vector #definies
  [for(a : Attribute) | c.attribute) { count = count + 1; }]
       #define [a.name/]_BIT [count/]
  [/for]

  class [c.name/] [for(c:Class | c.super) before(':') separator(',')] [c.name/] [/for]
  {
    bool bitVector [‘[‘+c.attribute->size()+’]’/];

    // Attribute declarations
    [for (a : Attribute) | c.attribute)]
      [a.type.name/] [if(isComplexType(a.type)]*[/if] [a.name/];
    [/for]

    // Constructor
    [c.name/]()
    {
      // initialize bit vector
      for (int i = 0; i < [c.attribute->size()/]; i++)
      {
        bitVector[‘[i]’] = 0;
      }
      [protected ('user_code')]
           // your code here
      [/protected]
    }

    // Attribute set/get/isSpecified methods
    [for (a : Attribute) | c.attribute)]
    void Set[a.name/] ([a.type.name/] [if(isComplexType(a.type)] * [/if] p[a.name/])
    {
      bitVector[’[‘+a.name+’_BIT]’] = 1;
      [a.name/] = p[a.name/];
    }

    [a.type.name/] Get[a.name/] () {return [a.name/];}
```

```
    bool isSpecified[a.name/]() {return bitVector['['+a.name+'_BIT]'];}
    [/for]

    // Method declarations
    [for (o : Operation) | c.operation) ]
    [o.type.name/] [o.name/] ([for(p:Parameter | o.parameter) separator(',')] [p.type/] [p.name/] [/for]);
    [/for]
  }
[/trace]
[/file]
[/template]
```



Output of the above template for the model shown above is given below:

```
// modelelement$employee_id$_header

#define name_BIT 0
#define dept_BIT 1
#define salary_BIT 2

class Employee : public Person
{
    char bitVector[3];

    // Attribute declarations
    String name;
    Department *dept;
    double salary;

    // Constructor
    Employee()
    {
for (int i = 0; i < 3; i++)
      {
         bitVector[i] = 0;
      }
// protected$user_code$modelelement$employee_id$_header
     // your code here
        // 1$modelelement$employee_id$_header
    }

    // Attribute set/get/isSpecified methods
    void Setname(String pname)
    {
```

```
        bitVector[name_BIT] = 1;
        name = pname;
    }

    String Getname() {return name;}

    bool isSpecifiedname() {return bitVector[name_BIT];}

    void Setdept(Department *pdept)
    {
        bitVector[dept_BIT] = 1;
        dept = pdept;
    }

    Department* Getdept() {return dept;}

    bool isSpecifieddept() {return bitVector[dept_BIT];}

    void Setsalary(double psalary)
    {
        bitVector[salary_BIT] = 1;
        salary = psalary;
    }

    double Getsalary() {return salary;}

    bool isSpecifiedsalary() {return bitVector[salary_BIT];}

    // Method declarations
    double ComputeTaxRebate( double loanAmt, double insurancePremium );
}
```

A.4 Metamodel in XMI

```
<?xml version="1.0" encoding="UTF-8"?>
<emof:Package xmi:version="2.0"
   xmlns:xmi="http://www.omg.org/XMI" xmlns:emof="http://schema.omg.org/spec/mof/2.0/emof.xmi"
xmi:id="metamodel"
   name="metamodel" uri="http:///metamodel.ecore">
  <nestedPackage xmi:id="metamodel.mof" name="mof" uri="http:///metamodel/mof.ecore">
   <ownedType xmi:type="emof:Class" xmi:id="metamodel.mof.Package" name="Package"/>
   <ownedType xmi:type="emof:Class" xmi:id="metamodel.mof.Parameter" name="Parameter"/>
   <ownedType xmi:type="emof:Class" xmi:id="metamodel.mof.NamedElement" name="NamedElement"/>
   <xmi:Extension extender="http://www.eclipse.org/emf/2002/Ecore">
    <nsPrefix>metamodel.mof</nsPrefix>
   </xmi:Extension>
  </nestedPackage>
  <nestedPackage xmi:id="metamodel.mtt" name="mtt" uri="http:///metamodel/mtt.ecore">
   <ownedType xmi:type="emof:Class" xmi:id="metamodel.mtt.TemplateInvocation" name="TemplateInvo-
cation"
     superClass="metamodel.mtt.TemplateExpression">
    <ownedAttribute xmi:id="metamodel.mtt.TemplateInvocation.definition" name="definition" isOr-
dered="false"
      lower="1" type="metamodel.mtt.Template"/>
    <ownedAttribute xmi:id="metamodel.mtt.TemplateInvocation.arguments" name="arguments"
      upper="*" type="metamodel.ocl.OclExpression"/>
    <ownedAttribute xmi:id="metamodel.mtt.TemplateInvocation.before" name="before" isOrdered="false"
      type="metamodel.ocl.OclExpression"/>
    <ownedAttribute xmi:id="metamodel.mtt.TemplateInvocation.each" name="each" type="meta-
```

```
model.ocl.OclExpression" isOrdered="false"/>
    <ownedAttribute xmi:id="metamodel.mtt.TemplateInvocation.after" name="after" isOrdered="false"
      type="metamodel.ocl.OclExpression"/>
  </ownedType>
  <ownedType xmi:type="emof:Class" xmi:id="metamodel.mtt.Template" name="Template"
    superClass="metamodel.mtt.Block metamodel.mtt.ModuleElement">
    <ownedAttribute xmi:id="metamodel.mtt.Template.overrides" name="overrides" upper="*"
      type="metamodel.mtt.Template"/>
    <ownedAttribute xmi:id="metamodel.mtt.Template.parameter" name="parameter" upper="*"
      type="metamodel.mof.Parameter" isComposite="true"/>
    <ownedAttribute xmi:id="metamodel.mtt.Template.guard" name="guard" type="metamodel.ocl.OclEx-
pression" isOrdered="false"/>
  </ownedType>
  <ownedType xmi:type="emof:Class" xmi:id="metamodel.mtt.QueryInvocation" name="QueryInvoca-
tion"
    superClass="metamodel.mtt.TemplateExpression">
    <ownedAttribute xmi:id="metamodel.mtt.QueryInvocation.definition" name="definition" isOr-
dered="false"
      lower="1" type="metamodel.mtt.Query"/>
    <ownedAttribute xmi:id="metamodel.mtt.QueryInvocation.arguments" name="arguments"
      upper="*" type="metamodel.ocl.OclExpression"/>
  </ownedType>
  <ownedType xmi:type="emof:Class" xmi:id="metamodel.mtt.Module" name="Module" super-
Class="metamodel.mtt.Template metamodel.mof.Package">
    <ownedAttribute xmi:id="metamodel.mtt.Module.extends" name="extends" upper="*" isOr-
dered="false"
      type="metamodel.mtt.Module"/>
    <ownedAttribute xmi:id="metamodel.mtt.Module.ownedModuleElement" name="ownedModuleEle-
ment" isOrdered="false"
      lower="1" upper="*" type="metamodel.mtt.ModuleElement" isComposite="true"/>
    <ownedAttribute xmi:id="metamodel.mtt.Module.input" name="input" lower="1" upper="*" isOr-
dered="false"
      type="metamodel.qvt.TypedModel"/>
  </ownedType>
  <ownedType xmi:type="emof:Class" xmi:id="metamodel.mtt.ProtectedAreaBlock" name="ProtectedAr-
eaBlock"
    superClass="metamodel.mtt.Block">
    <ownedAttribute xmi:id="metamodel.mtt.ProtectedAreaBlock.marker" name="marker" isOr-
dered="false"
      lower="1" type="metamodel.ocl.OclExpression"/>
    <ownedAttribute xmi:id="metamodel.mtt.ProtectedAreaBlock.end" name="end" lower="1" isOr-
dered="false"
      type="metamodel.ocl.OclExpression"/>
  </ownedType>
  <ownedType xmi:type="emof:Class" xmi:id="metamodel.mtt.ModuleElement" name="ModuleElement"
    superClass="metamodel.mof.NamedElement">
    <ownedAttribute xmi:id="metamodel.mtt.ModuleElement.isPublic" name="isPublic" isOr-
dered="false">
      <type xmi:type="emof:PrimitiveType" href="http://www.eclipse.org/emf/2002/
Ecore.emof#ecore.EBooleanObject"/>
    </ownedAttribute>
  </ownedType>
  <ownedType xmi:type="emof:Class" xmi:id="metamodel.mtt.TemplateExpression" name="TemplateEx-
pression"
    superClass="metamodel.ocl.OclExpression"/>
  <ownedType xmi:type="emof:Class" xmi:id="metamodel.mtt.InitSection" name="InitSection">
    <ownedAttribute xmi:id="metamodel.mtt.InitSection.variable" name="variable" isOrdered="false"
      lower="1" upper="*" type="metamodel.ocl.Variable"/>
  </ownedType>
  <ownedType xmi:type="emof:Class" xmi:id="metamodel.mtt.Block" name="Block" superClass="meta-
model.mtt.TemplateExpression">
```

```
    <ownedAttribute xmi:id="metamodel.mtt.Block.body" name="body" upper="*" type="meta-
model.mtt.TemplateExpression"
        isComposite="true"/>
    <ownedAttribute xmi:id="metamodel.mtt.Block.init" name="init" type="metamodel.mtt.InitSection"
isOrdered="false"
        isComposite="true"/>
  </ownedType>
  <ownedType xmi:type="emof:Class" xmi:id="metamodel.mtt.ForBlock" name="ForBlock"
      superClass="metamodel.mtt.Block">
    <ownedAttribute xmi:id="metamodel.mtt.ForBlock.loopVariable" name="loopVariable" isOr-
dered="false"
        lower="1" type="metamodel.ocl.Variable"/>
    <ownedAttribute xmi:id="metamodel.mtt.ForBlock.iterSet" name="iterSet" lower="1" isOr-
dered="false"
        type="metamodel.ocl.OclExpression"/>
    <ownedAttribute xmi:id="metamodel.mtt.ForBlock.Variable" name="Variable" upper="*" isOr-
dered="false"
        type="metamodel.ocl.Variable"/>
    <ownedAttribute xmi:id="metamodel.mtt.ForBlock.before" name="before" type="metamodel.ocl.OclEx-
pression" isOrdered="false"/>
    <ownedAttribute xmi:id="metamodel.mtt.ForBlock.after" name="after" type="metamodel.ocl.OclEx-
pression" isOrdered="false"/>
    <ownedAttribute xmi:id="metamodel.mtt.ForBlock.each" name="each" type="metamodel.ocl.OclEx-
pression" isOrdered="false"/>
    <ownedAttribute xmi:id="metamodel.mtt.ForBlock.guard" name="guard" type="metamodel.ocl.OclEx-
pression" isOrdered="false"/>
  </ownedType>
  <ownedType xmi:type="emof:Class" xmi:id="metamodel.mtt.LetBlock" name="LetBlock"
      superClass="metamodel.mtt.ForBlock metamodel.mtt.Block">
    <ownedAttribute xmi:id="metamodel.mtt.LetBlock.elseLet" name="elseLet" upper="*" isOr-
dered="false"
        type="metamodel.mtt.LetBlock"/>
    <ownedAttribute xmi:id="metamodel.mtt.LetBlock.else" name="else" type="metamodel.mtt.Block"
isOrdered="false"/>
    <ownedAttribute xmi:id="metamodel.mtt.LetBlock.ForBlock" name="ForBlock" upper="*" isOr-
dered="false"
        type="metamodel.mtt.ForBlock"/>
    <ownedAttribute xmi:id="metamodel.mtt.LetBlock.letExpr" name="letExpr" lower="1" isOr-
dered="false"
        type="metamodel.ocl.OclExpression"/>
  </ownedType>
  <ownedType xmi:type="emof:Class" xmi:id="metamodel.mtt.NewClass2" name="NewClass2"/>
  <ownedType xmi:type="emof:Class" xmi:id="metamodel.mtt.FileBlock" name="FileBlock"
      superClass="metamodel.mtt.Block">
    <ownedAttribute xmi:id="metamodel.mtt.FileBlock.openMode" name="openMode" type="meta-
model.mtt.AppendModeKind"/>
    <ownedAttribute xmi:id="metamodel.mtt.FileBlock.fileUrl" name="fileUrl" lower="1" isOrdered="false"
        type="metamodel.ocl.OclExpression"/>
    <ownedAttribute xmi:id="metamodel.mtt.FileBlock.uniqId" name="uniqId" type="metamodel.ocl.OclEx-
pression" isOrdered="false"/>
  </ownedType>
  <ownedType xmi:type="emof:Class" xmi:id="metamodel.mtt.TraceBlock" name="TraceBlock"
      superClass="metamodel.mtt.Block">
    <ownedAttribute xmi:id="metamodel.mtt.TraceBlock.modelElement" name="modelElement" isOr-
dered="false"
        lower="1" type="metamodel.ocl.OclExpression"/>
  </ownedType>
  <ownedType xmi:type="emof:Class" xmi:id="metamodel.mtt.Macro" name="Macro" superClass="meta-
model.mtt.Block metamodel.mtt.ModuleElement">
    <ownedAttribute xmi:id="metamodel.mtt.Macro.parameter" name="parameter" upper="*"
        type="metamodel.mof.Parameter"/>
```

```
  </ownedType>
  <ownedType xmi:type="emof:Class" xmi:id="metamodel.mtt.MacroInvocation" name="MacroInvocation"
      superClass="metamodel.mtt.TemplateExpression">
    <ownedAttribute xmi:id="metamodel.mtt.MacroInvocation.Macro" name="Macro" lower="1" isOrdered="false"
       type="metamodel.mtt.Macro"/>
    <ownedAttribute xmi:id="metamodel.mtt.MacroInvocation.TemplateExpression" name="TemplateExpression" isOrdered="false"
       upper="*" type="metamodel.mtt.TemplateExpression"/>
    <ownedAttribute xmi:id="metamodel.mtt.MacroInvocation.arguments" name="arguments"
       upper="*" type="metamodel.ocl.OclExpression"/>
  </ownedType>
  <ownedType xmi:type="emof:Class" xmi:id="metamodel.mtt.IfBlock" name="IfBlock"
     superClass="metamodel.mtt.Block">
    <ownedAttribute xmi:id="metamodel.mtt.IfBlock.ifExpr" name="ifExpr" lower="1" isOrdered="false"
       type="metamodel.ocl.OclExpression"/>
    <ownedAttribute xmi:id="metamodel.mtt.IfBlock.else" name="else" type="metamodel.mtt.Block" isOrdered="false"/>
    <ownedAttribute xmi:id="metamodel.mtt.IfBlock.elseIf" name="elseIf" upper="*" isOrdered="false"
       type="metamodel.mtt.IfBlock"/>
  </ownedType>
  <ownedType xmi:type="emof:Enumeration" xmi:id="metamodel.mtt.AppendModeKind" name="AppendModeKind">
    <ownedLiteral xmi:id="metamodel.mtt.AppendModeKind.Append" name="Append"/>
    <ownedLiteral xmi:id="metamodel.mtt.AppendModeKind.OverWrite" name="OverWrite">
      <xmi:Extension extender="http://www.eclipse.org/emf/2002/Ecore">
        <value>1</value>
      </xmi:Extension>
    </ownedLiteral>
  </ownedType>
  <ownedType xmi:type="emof:Class" xmi:id="metamodel.mtt.Query" name="Query" superClass="metamodel.mtt.ModuleElement metamodel.qvt.Function"/>
  <nestedPackage xmi:id="metamodel.mtt.newpackage" name="newpackage" uri="http:///metamodel/mtt/newpackage.ecore">
    <nestedPackage xmi:id="metamodel.mtt.newpackage.newpackage2" name="newpackage2"
       uri="http:///metamodel/mtt/newpackage/newpackage2.ecore">
      <xmi:Extension extender="http://www.eclipse.org/emf/2002/Ecore">
        <nsPrefix>metamodel.mtt.newpackage.newpackage2</nsPrefix>
      </xmi:Extension>
    </nestedPackage>
    <nestedPackage xmi:id="metamodel.mtt.newpackage.qvt_1" name="qvt_1" uri="http:///metamodel/mtt/newpackage/qvt_1.ecore">
      <xmi:Extension extender="http://www.eclipse.org/emf/2002/Ecore">
        <nsPrefix>metamodel.mtt.newpackage.qvt_1</nsPrefix>
      </xmi:Extension>
    </nestedPackage>
    <nestedPackage xmi:id="metamodel.mtt.newpackage.mof_2" name="mof_2" uri="http:///metamodel/mtt/newpackage/mof_2.ecore">
      <xmi:Extension extender="http://www.eclipse.org/emf/2002/Ecore">
        <nsPrefix>metamodel.mtt.newpackage.mof_2</nsPrefix>
      </xmi:Extension>
    </nestedPackage>
    <nestedPackage xmi:id="metamodel.mtt.newpackage.ocl_1" name="ocl_1" uri="http:///metamodel/mtt/newpackage/ocl_1.ecore">
      <xmi:Extension extender="http://www.eclipse.org/emf/2002/Ecore">
        <nsPrefix>metamodel.mtt.newpackage.ocl_1</nsPrefix>
      </xmi:Extension>
    </nestedPackage>
    <xmi:Extension extender="http://www.eclipse.org/emf/2002/Ecore">
      <nsPrefix>metamodel.mtt.newpackage</nsPrefix>
```

```
    </xmi:Extension>
   </nestedPackage>
   <xmi:Extension extender="http://www.eclipse.org/emf/2002/Ecore">
    <nsPrefix>metamodel.mtt</nsPrefix>
   </xmi:Extension>
  </nestedPackage>
  <nestedPackage xmi:id="metamodel.ocl" name="ocl" uri="http:///metamodel/ocl.ecore">
   <ownedType xmi:type="emof:Class" xmi:id="metamodel.ocl.OclExpression" name="OclExpression"/>
   <ownedType xmi:type="emof:Class" xmi:id="metamodel.ocl.Variable" name="Variable"/>
   <xmi:Extension extender="http://www.eclipse.org/emf/2002/Ecore">
    <nsPrefix>metamodel.ocl</nsPrefix>
   </xmi:Extension>
  </nestedPackage>
  <nestedPackage xmi:id="metamodel.qvt" name="qvt" uri="http:///metamodel/qvt.ecore">
   <ownedType xmi:type="emof:Class" xmi:id="metamodel.qvt.TypedModel" name="TypedModel">
    <ownedAttribute xmi:id="metamodel.qvt.TypedModel.takesTypesFrom" name="takesTypesFrom" isOr-
dered="false"
      lower="1" type="metamodel.mof.Package"/>
   </ownedType>
   <ownedType xmi:type="emof:Class" xmi:id="metamodel.qvt.Function" name="Function"/>
   <xmi:Extension extender="http://www.eclipse.org/emf/2002/Ecore">
    <nsPrefix>metamodel.qvt</nsPrefix>
   </xmi:Extension>
  </nestedPackage>
  <xmi:Extension extender="http://www.eclipse.org/emf/2002/Ecore">
   <nsPrefix>metamodel</nsPrefix>
  </xmi:Extension>
</emof:Package>
```

INDEX