MOF QVT Final Adopted Specification

This OMG document replaces the submission document (ad/05-07-01) and the Draft Adopted specification (ptc/05-10-02). It is an OMG Final Adopted Specification and is currently in the finalization phase. Comments on the content of this document are welcomed, and should be directed to *issues@omg.org* by March 31, 2007.

You may view the pending issues for this specification from the OMG revision issues web page *http://www.omg.org/issues/*.

The FTF Report for this specification will be published on July 9, 2007. Please download the appropriate document from the OMG Specifications Catalog.

OMG Adopted Specification
ptc/05-11-01

Date: November 2005

Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification

Final Adopted Specification

ptc/05-11-01

Copyright ©2005 Codagen Technologies Corp
Copyright ©2005 Compuware
Copyright ©2005 DSTC
Copyright ©2005 France Telecom
Copyright ©2005 IBM
Copyright ©2005 INRIA
Copyright ©2005 Interactive Objects
Copyright ©2005 Kings College London
Copyright © 2005, Object Management Group
Copyright ©2005 Softeam
Copyright ©2005 Sun Microsystems
Copyright ©2005 Tata Consultancy Services
Copyright ©2005 Thales
Copyright ©2005 TNI-Valiosys
Copyright ©2005 University of Paris VI
Copyright ©2005 University of York

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, 250 First Avenue, Needham, MA 02494, U.S.A.

TRADEMARKS

The OMG Object Management Group Logo®, CORBA®, CORBA Academy®, The Information Brokerage®, XMI® and IIOP® are registered trademarks of the Object Management Group. OMG™, Object Management Group™, CORBA logos™, OMG Interface Definition Language (IDL)™, The Architecture of Choice for a Changing World™, CORBAservices™, CORBAfacilities™, CORBAmed™, CORBAnet™, Integrate 2002™, Middleware That's Everywhere™, UML™, Unified Modeling Language™, The UML Cube logo™, MOF™, CWM™, The CWM Logo™, Model Driven Architecture™, Model Driven Architecture Logos™, MDA™, OMG Model Driven Architecture™, OMG MDA™ and the XMI Logo™ 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

1 Scope 1

2 Conformance 1

2.1 Conformance Points 1

2.2 Language Dimension 2

2.3 Interoperability Dimension 2

2.4 Conformance of QVT definitions 2

3 Normative References 3

4 Definitions and Terms 4

4.1 Glossary 4

5 Additional Information 6

5.1 Changes To Adopted OMG Specifications 6

5.2 Structure of the Specification 6

5.3 Acknowledgements 7

6 QVT Overview 9

6.1 Two Level Declarative Architecture 9

6.1.1 Relations 9

6.1.2 Core 9

6.1.3 Virtual Machine Analogy 10

6.2 Imperative Implementations 10

6.2.1 Operational Mappings Language 10

6.2.2 Black Box Implementations 10

6.3 Execution Scenarios 11

6.4 MOF Metamodels 11

7 The Relations Language 13

7.1 Transformations and Model Types 13

7.1.1 Transformation Execution Direction 13

7.2 Relations and Domains 13

7.2.1 When and Where Clauses 14
7.2.2 Top-level Relations 14
7.2.3 Check and Enforce 15

7.3 Pattern Matching 15

7.4 Keys and Object Creation using Patterns 16

7.5 Restrictions on Expressions 17

7.6 Change Propagation 18

7.7 In-place Transformations 18

7.8 Integrating Black-box Operations with Relations 18

7.9 Executing a Transformation in Checkonly mode 19

7.10 Detailed Semantics 19
7.10.1 Checking Semantics 20
7.10.2 Enforcement Semantics 20
7.10.3 Pattern Matching Semantics 21

7.11 Abstract Syntax and Semantics 24
7.11.1 QVTBase Package 24
7.11.1.1 Transformation 25
7.11.1.2 TypedModel 26
7.11.1.3 Domain 27
7.11.1.4 Rule 27
7.11.1.5 Function 28
7.11.1.6 FunctionParameter 28
7.11.1.7 Predicate 29
7.11.1.8 Pattern 29
7.11.2 QVTTemplate Package 30
7.11.2.1 TemplateExp 30
7.11.2.2 ObjectTemplateExp 31
7.11.2.3 CollectionTemplateExp 31
7.11.2.4 PropertyTemplateItem 32
7.11.3 QVTRelation Package 33
7.11.3.1 Relation 33
7.11.3.2 RelationDomain 34
7.11.3.3 DomainPattern 34
7.11.3.4 Key 35
7.11.3.5 RelationImplementation 35

7.12 Standard Library 36

7.13 Concrete Syntax 36
7.13.1 Relations Textual Syntax Grammar 36
7.13.2 Expressions syntax (extensions to OCL) 37
7.13.3 Graphical Syntax 37
7.13.3.1 Introduction 37
7.13.3.2 Graphical Notation Elements 42
7.13.3.3 Variations in Graphical Notation 44

8 Operational Mappings 45

8.1 Overview 45

8.1.1 Operational transformations 45
8.1.2 Model types 46
8.1.3 Libraries 46
8.1.4 Mapping operations 47
8.1.5 Object creation and population in mapping operations 48
8.1.6 Inlining mapping operations........ 49
8.1.7 Using constructor operations........ 50
8.1.8 Helpers 50
8.1.9 Intermediate data 51
8.1.10 Updating objects and resolving object reference........ 52
8.1.11 Composing transformations 53
8.1.12 Reuse facilities for mapping operations 54
8.1.13 Disjunction of mapping operations 55
8.1.14 Type extensions 55
8.1.15 Imperative expressions 56
8.1.16 Pre-defined variables: this, self and result 57
8.1.17 Null 57
8.1.18 Advanced features: dynamic definition and parallelism 57

8.2 Abstract Syntax and Semantics 58

8.2.1 The QVTOperational Package. 59
8.2.1.1 OperationalTransformation 60
8.2.1.2 Library 63
8.2.1.3 Module 64
8.2.1.4 ModuleImport 65
8.2.1.5 ModelParameter 66
8.2.1.6 ModelType 66
8.2.1.7 VarParameter........ 68
8.2.1.8 DirectionKind........ 68
8.2.1.9 ImportKind 69
8.2.1.10 ImperativeOperation 70
8.2.1.11 EntryOperation 70
8.2.1.12 Helper 71
8.2.1.13 Constructor 72
8.2.1.14 ContextualProperty 72
8.2.1.15 MappingOperation 73
8.2.1.16 MappingParameter 75
8.2.1.17 OperationBody 76
8.2.1.18 ConstructorBody 76
8.2.1.19 MappingBody 77
8.2.1.20 ImperativeCallExp 79
8.2.1.21 MappingCallExp........ 79
8.2.1.22 ResolveExp........ 81
8.2.1.23 ResolveInExp........ 82
8.2.1.24 ObjectExp 82
8.2.2 The ImperativeOCL Package 83
8.2.2.1 ImperativeExpression........ 85

8.2.2.2 BlockExp 85
8.2.2.3 ComputeExp 86
8.2.2.4 WhileExp 86
8.2.2.5 ImperativeLoopExp 87
8.2.2.6 ForExp 87
8.2.2.7 ImperativeIterateExp 88
8.2.2.8 SwitchExp 90
8.2.2.9 AltExp 91
8.2.2.10 VariableInitExp 92
8.2.2.11 AssignExp 92
8.2.2.12 UnlinkExp 93
8.2.2.13 TryExp 94
8.2.2.14 RaiseExp 94
8.2.2.15 ReturnExp 95
8.2.2.16 BreakExp 95
8.2.2.17 ContinueExp 95
8.2.2.18 LogExp 95
8.2.2.19 AssertExp 96
8.2.2.20 SeverityKind 97
8.2.2.21 TupleExp 97
8.2.2.22 UnpackExp 97
8.2.2.23 InstantiationExp 97
8.2.2.24 Typedef 99
8.2.2.25 ListType 100
8.2.2.26 DictionaryType 100
8.2.2.27 AnonymousTupleType 100
8.2.2.28 TemplateParameterType 101
8.2.2.29 DictLiteralExp 101
8.2.2.30 DictLiteralPart 02
8.2.2.31 AnonymousTupleLiteralExp 102
8.2.2.32 AnonymousTupleLiteralPart 102

8.3 Standard Library 102

8.3.1 Predefined types 103
8.3.1.1 Transformatio 103
8.3.1.2 Model 103
8.3.1.3 Status 103
8.3.2 Synonym types and synonym operations 103
8.3.3 Operations on objects 103
8.3.3.1 repr 104
8.3.4 Operations on elements 104
8.3.4.1 _localId 104
8.3.4.2 _globalId 104
8.3.4.3 metaClassName 104
8.3.4.4 subobjects 104
8.3.4.5 allSubobjects 104
8.3.4.6 subobjectsOfType 104
8.3.4.7 allSubobjectsOfType 104
8.3.4.8 subobjectsOfKind 105
8.3.4.9 allSubobjectsOfKind 105
8.3.4.10 clone 105

8.3.4.11 deepclone ... 105
8.3.4.12 markedAs ... 105
8.3.4.13 markValue ... 105
8.3.4.14 stereotypedBy ... 105
8.3.5 Operations on models ... 105
8.3.5.1 objects ... 106
8.3.5.2 objectsOfType ... 106
8.3.5.3 rootObjects ... 106
8.3.5.4 removeElement ... 106
8.3.5.5 asTransformation ... 106
8.3.5.6 copy ... 106
8.3.5.7 createEmptyModel ... 106
8.3.6 Operations on Transformations ... 106
8.3.6.1 transform ... 107
8.3.6.2 parallelTransform ... 107
8.3.6.3 wait ... 107
8.3.6.4 raisedException ... 107
8.3.6.5 failed ... 107
8.3.6.6 succeeded ... 107
8.3.7 Operations on dictionaries ... 107
8.3.7.1 get ... 107
8.3.7.2 hasKey ... 108
8.3.7.3 defaultget ... 108
8.3.7.4 put ... 108
8.3.7.5 clear ... 108
8.3.7.6 size ... 108
8.3.7.7 values ... 108
8.3.7.8 keys ... 108
8.3.7.9 isEmpty ... 108
8.3.8 Operations on lists ... 108
8.3.8.1 add ... 109
8.3.8.2 prepend ... 109
8.3.8.3 insertAt ... 109
8.3.8.4 joinfields ... 109
8.3.9 Operations on strings ... 109
8.3.10 Operations on numeric types ... 111
8.3.11 Predefined tags ... 111

8.4 Concrete Syntax ... 112
8.4.1 Files ... 112
8.4.2 Comments ... 112
8.4.3 Shorthands used to invoke specific pre-defined operations ... 112
8.4.4 Other language shorthands ... 113
8.4.5 Notation for metamodels ... 113
8.4.6 EBNF ... 113
8.4.6.1 Syntax for module definitions ... 114
8.4.6.2 Syntax for the expressions ... 116

9 The Core Language ... 119

9.1 Comparison with the Relational Language ... 119

9.2 Transformations and Model Types ... 119
9.3 Mappings ... 119
9.4 Patterns ... 120
9.5 Bindings ... 121
9.6 Binding Dependencies ... 121
9.7 Guards ... 122
9.8 Bottom Patterns ... 122
9.9 Checking ... 122
9.9.1 Checking formally defined ... 123
9.10 Enforcement ... 124
9.10.1 Enforcement formally defined ... 125
9.11 Realized Variables ... 126
9.12 Assignments ... 126
9.13 Enforcement Operations ... 127
9.14 Mapping Refinement ... 127
9.15 Mapping Composition ... 128
9.16 Functions ... 128
9.17 Abstract Syntax and Semantics ... 128
9.17.1 CorePattern ... 129
9.17.2 Area ... 130
9.17.3 GuardPattern ... 130
9.17.4 BottomPattern ... 130
9.17.5 CoreDomain ... 131
9.17.6 Mapping ... 131
9.17.7 RealizedVariable ... 133
9.17.8 Assignment ... 134
9.17.9 EnforcementMode ... 134
9.17.10 EnforcementOperation ... 135
9.18 Concrete Syntax ... 135
10 Relations to Core Transformation ... 137
10.1 Mapping Approach ... 137
10.2 Mapping Rules ... 138
10.3 Relational Expression of Relations To Core Transformation ... 145
Annex A ... 163
Annex B - Semantics of Relations ... 185

1 Scope

2 Conformance

QVT language conformance is specified along two orthogonal dimensions: the *language dimension* and the *interoperability dimension*. Each dimension specifies a set of named levels. Each intersection of the levels of the two dimensions specifies a valid QVT conformance point. All conformance points are valid by themselves, which implies that *there is no general notion of "QVT conformance"*. Instead, a tool shall state which conformance points it implements, as described below in Section 3.1.

2.1 Conformance Points

Any combination of two named levels, one from each dimension, constructs a conformance point. Figure 1 specifies the 12 different possible conformance points. A tool can claim to be conformant according to one or more of these 12 conformance points.

Language \ Interoperability	Syntax Executable	XMI Executable	Syntax Exportable	XMI Exportable
Core				
Relations				
Operational				

Figure 2.1 - Conformance Table

By convention a conformance point is denoted using the abbreviation

QVT - <language level> - <interoperability level>

For example, a tool could be *QVT-Relations-SyntaxExecutable*, *QVT-Relations-XMIExportable*, and *QVT-Core-SyntaxExportable*. Another tool could be *QVT-Operational-SyntaxExecutable* and *QVT-Operational-XMIExportable*.

There is one implicit requirement: A tool which is *QVT-SyntaxExecutable* or *QVT-XMIExecutable* for a particular language level shall also be *QVT-SyntaxExportable* conformant or *QVT-XMIExportable* conformant, respectively, for the same language level.

2.2 Language Dimension

The language dimension consists of the three named language levels:

- Core: The Core language is described in Section 11. This includes the ability to insert black-box implementations via MOF operations as specified.
- Relations: The Relations language is described in Section 9. This includes the ability to insert black-box implementations via MOF operations as specified.
- Operational: The Operational Mappings language is described in Section 10.

2.3 Interoperability Dimension

The interoperability dimension has four named interoperability levels:

- SyntaxExecutable: An implementation shall provide a facility to import or read, and then execute the **concrete syntax** description of a transformation in the language given by the language dimension. The execution shall be according to the semantics of the chosen language as described in this document.
- XMIExecutable: An implementation shall provide a facility to import or read, and then execute an **XMI serialization** of a transformation description which conforms to the MOF meta-model of the language given by the language dimension. The execution shall be according to the semantics of the chosen language as described in this document.
- SyntaxExportable: An implementation shall provide a facility to export a model-to-model transformation in the **concrete syntax** of the language given by the language dimension.
- XMIExportable: An implementation shall provide a facility to export a model-to-model transformation into its **XMI serialization** which conforms to the MOF meta-model of the language given by the language dimension.

2.4 Conformance of QVT definitions

Figure 1 defines compliance points for tools. We address here conformance of transformations definitions written in QVT.

The authors of a QVT transformation definition shall indicate:

1. The language dimension being used,
2. Whether back-box operations are being used. If black-box operations are used then a suitable description of the operations should also be provided including a signature expressed in OCL syntax.

By convention the following terms should be used when claiming QVT compliance of a transformation definition:

QVT - <language-level> or

QVT - <language-level> *

The asterisk symbol means that black-boxes are used. This gives the following possible set of values: QVT-Core, QVT-Core*, QVT-Relations, QVT-Relations*, QVTOperational and QVT-Operational*.

3 Normative References

The QVT specification depends on the following two OMG specifications:

- MOF 2.0 Specification
- OCL 2.0 Specification

4 Definitions and Terms

4.1 Glossary

Area	In the context of a *core mapping* an area is a pair of ***patterns***, consisting of a *guard pattern* and a *bottom pattern*.
Bottom Pattern	A pattern which is checked or enforced for the bindings generated by the *guard pattern* of the same *area* of a *mapping*, and other patterns which this area is related to for execution in a particular direction.
Core Domain	A specialized kind of *domain* that forms part of a *mapping*. A core domain is also an *area* which defines a pair of *patterns*, consisting of a *guard pattern* and a *bottom pattern*.
Core Transformation	A transformation definition formalized by a list of *core mappings*.
Domain	A domain is a distinguished set of variables to be matched in a *typed model*. It is related to other domains by a transformation *rule*. Domain is an abstract type in the QVTBase package which has concrete types RelationDomain and CoreDomain. Domains have flags to indicate whether they are *checkonly* or *enforced*. When a transformation is executed with the typed model of this domain as its target model, and it is an enforced domain, values may be created or destroyed in the typed model in order to satisfy the rules of the relation to which it belongs.
Guard Pattern	A *pattern* which must hold as a precondition to the application of the *bottom pattern* related to it in an *area* of a core mapping.
Identifying Property	A property of class that is part of a **key** defined in a relational transformation.

Incremental Update Once a relationship (a set of *trace instances*) has been established between models by executing a transformation, small changes to a source model may be propagated to a target model by re-executing the transformation in the context of the trace, causing only the relevant target model elements to be changed, without modifying the rest of the model.

Key In the context of *relations* a key is a definition of which properties of a MOF class, in combination, can uniquely identify an instance of that class. These properties are called *identifying properties*, and are used when matching *template patterns* to determine how many instances of a class should exist in a relationship by creating or locating an instance for each unique key that can be derived from the values bound to the identifying properties.

Mapping (Core) A transformation *rule* in a Core transformation description. It is an *area*, consisting of a pair of *patterns* which are designed to locate or create instances of the *trace classes* which store the relationships between models. It also owns a set of **core** *domains* which identify the model elements in those models to be related to one another.

Mapping Operation An operation implementing a part of a transformation. It defines a signature and a structured and imperative body. It is associated with a relation for which it is a refinement.

Model Type In the context of an *operational transformation* a model type represents the type of the models involved in the transformation. A model type is defined by a metamodel, a conformance kind ñ strict or effective - and an optional set of constraint expressions. The metamodel defines the set of classes and property elements that are expected by the transformation, and is captured in a set of MOF Packages. Effective compliance allows flexible transformations to be defined that can be applied to similar metamodels.

Operational Transformation A transformation definition that is formalized by a list of mapping operations.

Relation A relation is a subset of an N-ary product of sets, A_1 x A_2 x ... x A_N, and may be represented as a set of N-tuples (a_1, a_2,..., a_N). In the context of MOF, each set A_K, called a *domain*, is a MOF type, and a relation will be populated by tuples referring to model elements of those types which exist in MOF extents.

A Relation in the QVT specification also defines the *rules* by which the exact subset of model elements to be related is determined. These *rules* comprise variables of additional MOF types, *template pattern* matches on the structure of the *relation domains* which bind values to the variables, OCL constraints over the *relation domains* and variables of the relation, and assertions that other relations hold.

Relations imply the existence of equivalent *trace classes* which have properties for each of its domains, and whose set of *trace instances* are equivalent to the relation's population of tuples.

Relational Transformation A transformation definition that is formalized by a list of relations.

Relation Domain A specialization of the concept of a *domain*. In a Relation a domain is a type which may be the root of a *template pattern*, which can match any model element navigable from that type.

A domain implies the existence of a property of the same type in a *trace class* derived from the relation to which it belongs.

Template Pattern A template pattern is a combination of a literal as defined in OCL which can match against instances of a class and values for any of its properties, including recursive matching of other class instances which are values of those properties. It also allows for the binding of variables to any value matched in that structure, including collections of values. Template Patterns are part of the definition of a *relation domain*. They provide a terse, user-friendly expression of what can be quite verbose expressions in ordinary OCL. Template Patterns rely on *identifying properties* of classes, defined by *keys* to further simplify the specification of relationships between relation domains.

Trace Class A MOF class with properties that refer to objects and values in models that are related by a transformation. Instances of these classes (*trace instances*) are created during the execution of a transformation so that relationships between models that are created by the execution can be stored. In the context of the Relations Language, a trace class is derived from each Relation, with a property to represent each domain of the relation.

Trace Instance An instance of a *trace class* which represents the linkage between models established by a transformation execution. These instances may be used to aid in propagating *incremental updates* to a source model into a target model without re-executing the entire transformation.

5 Additional Information

5.1 Changes To Adopted OMG Specifications

This specification does not make any changes to existing OMG specifications.

5.2 Structure of the Specification

This specification defines three related transformation languages: Relations, Operational Mappings, and Core.

Section 6 - QVT Overview, describes the relationships between the three language models, and gives an overview of their purposes and features.

Section 7 - The Relations Language, provides the details of this language, and its evaluation semantics. It shows the MOF metamodel and describes the elements of that model. It gives the concrete syntax for the language. It also describes how black-box operation implementations can be used.

Section 8 - Operational Mappings, provides the details of this language in terms of imperative mappings which extend the QVTRelation package introduced in the previous section, as well as side-effect extensions to OCL 2.0, and their evaluation semantics. It shows the MOF metamodel and describes the elements of that model. It gives the concrete syntax for the language.

Section 9 - The Core Language, describes the Core on which the semantics of the Relations Language is based. The core evaluation semantics is given in semi-formal set-theoretic notation. Then the MOF metamodel and descriptions of its elements are given.

Section 10 - Relations to Core Transformation, gives the transformation from an arbitrary relations specification for a particular execution direction, to an equivalent core specification and trace classes. This allows relational transformation descriptions to be understood in terms of the formal semantics of the Core.

Finally, Annex A - Additional Examples, provides some whole transformation examples to augment the excerpt examples shown inline in the rest of the specification.

5.3 Acknowledgements

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

- Adaptive
- Alcatel
- Artisan Software
- Borland
- CBOP
- CEA
- Codagen Technologies Corp.
- Colorado State University
- Compuware
- DSTC
- France Telecom
- Hewlett Packard
- INRIA
- International Business Machines
- Interactive Objects
- Kinetium
- King’s college London
- LIFL
- Softeam
- Sun Microsystems
- Tata Consultancy Services
- Thales
- TNI-Valiosys
- Unisys
- University of Paris VI
- University of York
- Xactium

The following people have been chiefly responsible for the work involved in this version of the specification:

Wim Bast, Michael Murphree (Compuware), Michael Lawley, Keith Duddy (DSTC), Mariano Belaunde (France Telecom R&D), Catherine Griffin, Shane Sendall (IBM), Didier Vojtisek, Jim Steel (INRIA), Simon Helsen (Interactive Objects), Laurence Tratt (King's College London), Sreedhar Reddy, R. Venkatesh (Tata Consultancy Services), Xavier Blanc (University of Paris VI)

Many other people have made contributions to the ideas in this specification. This is an incomplete list:

Steve Mellor, Michel Brassard, Eric Brière, Tracy Gardner, Alan Kennedy, Kerry Raymond, Anna Gerber, Laurent Rioux, Madeleine Faugère, Benoit Langlois, Jens Rommel, Philippe Desfray, Biju Appukuttan, Tony Clark, Andy Evans, Girish Maskeri, Paul Sammut, James Willans, Jim Rumbaugh, Jean Bézivin, Frédéric Jouault, Erwan Breton, Martin Matula, Pete Rivett, Roy Gronmo.

6 QVT Overview

The QVT specification has a hybrid declarative/imperative nature, with the declarative part being split into a two-level architecture. We start by explaining the two-level architecture of the declarative part, as it forms the framework for the execution semantics of the imperative part.

6.1 Two Level Declarative Architecture

The declarative parts of this specification are structured into a two-layer architecture.

The layers are:

- A user-friendly *Relations* metamodel and language which supports complex object pattern matching and object template creation. Traces between model elements involved in a transformation are created implicitly.
- A *Core* metamodel and language defined using minimal extensions to EMOF and OCL. All trace classes are explicitly defined as MOF models, and trace instance creation and deletion is defined in the same way as the creation and deletion of any other object.

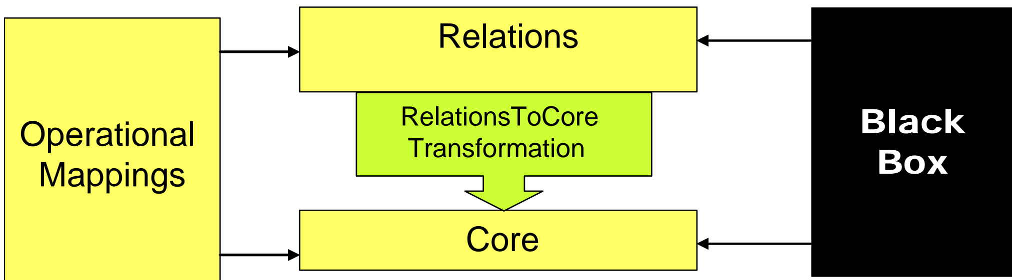


Figure 6.1 - Relationships between QVT metamodels

6.1.1 Relations

A declarative specification of the relationships between MOF models. The Relations language supports complex object pattern matching, and implicitly creates trace classes and their instances to record what occurred during a transformation execution. Relations can assert that other relations also hold between particular model elements matched by their patterns. The semantics of Relations are defined in a combination of English and first order predicate logic in Section 9.10, as well as by a standard transformation for any Relations model to trace models and a Core model with equivalent semantics. This transformation can be found in Chapter 10. It can be used purely as a formal semantics for Relations, or as a way of translating a Relations model to a Core model for execution on an engine implementing the Core semantics.

6.1.2 Core

This is a small model/language which only supports pattern matching over a flat set of variables by evaluating conditions over those variables against a set of models. It treats all of the model elements of source, target and trace models symmetrically. It is equally powerful to the Relations language, and because of its relative simplicity, its semantics can be defined more simply, although transformation descriptions described using the Core are therefore more verbose. In

addition, the trace models must be explicitly defined, and are not deduced from the transformation description, as is the case with Relations. The core model may be implemented directly, or simply used as a reference for the semantics of Relations, which are mapped to the Core, using the transformation language itself. The definition of the Core semantics is given in Section 11.

6.1.3 Virtual Machine Analogy

An analogy can be drawn with the Java™ architecture, where the Core language is like Java Byte Code and the Core semantics is like the behavior specification for the Java Virtual Machine. The Relations language plays the role of the Java language, and the standard transformation from Relations to Core is like the specification of a Java Compiler which produces Byte Code.

6.2 Imperative Implementations

In addition to the declarative Relations and Core Languages which embody the same semantics at two different levels of abstraction, there are two mechanisms for invoking imperative implementations of transformations from Relations or Core: one standard language, *Operational Mappings*, as well as non-standard *Black-box MOF Operation* implementations. Each relation define a class which will be instantiated to trace between model elements being transformed, and it has a one-to-one mapping to an Operation signature that the Operational Mapping or Black-box implements.

6.2.1 Operational Mappings Language

This language is specified as a standard way of providing imperative implementations, which populate the same trace models as the Relations Language. It is given in Chapter 8. It provides OCL extensions with side effects that allow a more procedural style, and a concrete syntax that looks familiar to imperative programmers.

Mappings Operations can be used to implement one or more Relations from a Relations specification when it is difficult to provide a purely declarative specification of how a Relation is to be populated. Mappings Operations invoking other Mappings Operations always involves a Relation for the purposes of creating a trace between model elements, but this can be implicit, and an entire transformation can be written in this language in the imperative style. A transformation entirely written using Mapping Operations a called an operational transformation.

6.2.2 Black Box Implementations

MOF Operations may be derived from Relations making it possible to "plug-in" any implementation of a MOF Operation with the same signature. This is beneficial for several reasons:

- It allows complex algorithms to be coded in any programming language with a MOF binding (or that can be executed from a language with a MOF binding).
- It allows the use of domain specific libraries to calculate model property values. For example, mathematical, engineering, bio-science and many other domains have large libraries that encode domain-specific algorithms which will difficult, if not impossible to express using OCL.
- It allows implementations of some parts of a transformation to be opaque.

However, it is also dangerous. The plugin implementation has access to object references in models, and may do arbitrary things to those objects. Black-box implementations do no have an implicit relationship to Relations, and each black-box must explicitly implement a Relation, which is responsible for keeping traces between model elements related by the Operation implementation. In these cases, the relevant parts of the models can be matched by a Relation, and passed out to implementations in the most relevant language for processing.

To extend the Java architecture analogy, the ability to invoke black-box and operational mapping implementations can be considered equivalent to calling the Java Native Interface (JNI).

6.3 Execution Scenarios

The semantics of the Core language (and hence the Relations language) allow for the following execution scenarios:

- Check-only transformations to verify that models are related in a specified way.
- Single direction transformations.
- Bi-directional transformations. (In fact more than two directions are possible, but two is the most common case.)
- The ability to establish relationships between pre-existing models, whether developed manually, or though some other tool or mechanism.
- Incremental updates (in any direction) when one related model is changed after an initial execution.
- The ability to create as well as delete objects and values, while also being able to specify which objects and values must not be modified.

The operational mapping and black-box approaches, even when executed in tandem with relations, restrict these scenarios by only allowing specification of transformations in a single direction. Bi-directional transformations are only possible if an inverse operational implementation is provided separately. However, all of the other capabilities defined above are available with imperative and hybrid executions.

6.4 MOF Metamodels

This specification defines three main packages, one for each of the languages defined: QVTCore (Section 9.17), QVTRelation (Section 7.11.3) and QVTOperational (Section 8.2).

The QVTBase package define common structure for transformations (Section 7.11.1).

In addition the QVTRelation package uses template pattern expressions defined in the QVTTemplateExp package (Section 7.11.2).

QVTOperational extends QVTRelation, as it uses the same framework for traces defined in that package. It uses imperative expressions defined in the ImperativeOCL Package (Section 8.2.2).

All of QVT depends on the EssentialOCL package from OCL 2.0, and all of the language packages depend on EMOF.

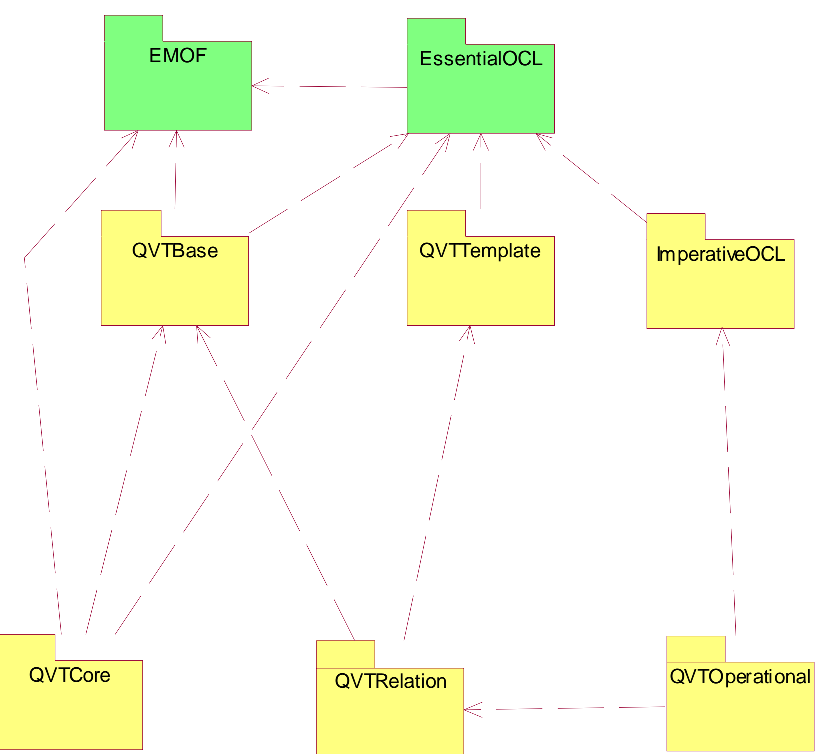


Figure 6.2 - Package dependencies in the QVT specification

7 The Relations Language

7.1 Transformations and Model Types

In the relations language, a transformation between candidate models is specified as a set of relations that must hold for the transformation to be successful. A candidate model is any model that conforms to a model type, which is a specification of what kind of model elements any conforming model can have, similar to a variable type specifying what kind of values a conforming variable can have in a program. Candidate models are named, and the types of elements they can contain are restricted to those within a set of referenced packages. An example is:

```
transformation umlRdbms (uml : SimpleUML, rdbms : SimpleRDBMS) {
```

In this declaration named "umlRdbms," there are two typed candidate models: "uml" and "rdbms". The model named "uml" declares the SimpleUML package as its metamodel, and the "rdbms" model declares the SimpleRDBMS package as its metamodel. A transformation can be invoked either to check two models for consistency or to modify one model to enforce consistency.

7.1.1 Transformation Execution Direction

A transformation invoked for enforcement is executed in a particular direction by selecting one of the candidate models as the target. The target model may be empty, or may contain existing model elements to be related by the transformation. The execution of the transformation proceeds by first checking whether the relations hold, and for relations for which the check fails, attempting to make the relations hold by creating, deleting or modifying only the target model, thus enforcing the relationship.

7.2 Relations and Domains

Relations in a transformation declare constraints that must be satisfied by the elements of the candidate models. A relation, defined by two or more domains and a pair of **when** and **where** predicates, specifies a relationship that must hold between the elements of the candidate models.

A domain is a distinguished typed variable that can be matched in a model of a given model type. A domain has a pattern, which can be viewed as a graph of object nodes, their properties and association links originating from an instance of the domain's type. Alternatively a pattern can be viewed as a set of variables, and a set of constraints that model elements bound to those variables must satisfy to qualify as a valid binding of the pattern. A domain pattern can be considered a template for objects and their properties that must be located, modified, or created in a candidate model to satisfy the relation. Pattern matching and object creation using patterns are discussed in more detail in section 9.3 and 9.4 below.

In the example below, two domains are declared which will match elements in the "uml" and "rdbms" models respectively. Each domain specifies a simple pattern ñ a package with a name, and a schema with a name, both the “name” properties being bound to the same variable “pn” implying that they should have the same value. :

```
relation PackageToSchema   /* map each package to a schema */
  {
    domain uml p:Package {name=pn}
    domain rdbms s:Schema {name=pn}
  }
```

7.2.1 When and Where Clauses

A relation can be also constrained by two sets of predicates, a **when** clause and a **where** clause, as shown in the example relation ClassToTable below. The **when** clause specifies the conditions under which the relationship needs to hold, so the relation ClassToTable needs to hold only when the PackageToSchema relation holds between the package containing the class and the schema containing the table. The **where** clause specifies the condition that must be satisfied by all model elements participating in the relation, and it may constrain any of the variables in the relation and its domains. Hence, whenever the ClassToTable relation holds, the relation AttributeToColumn must also hold.

```
relation ClassToTable   /* map each persistent class to a table */
  {
    domain uml c:Class {
        namespace = p:Package {},
        kind='Persistent',
        name=cn
     }
    domain rdbms t:Table {
         schema = s:Schema {},
          name=cn,
          column = cl:Column {
              name=cn+'_tid',
              type='NUMBER'},
                   primaryKey = k:PrimaryKey {
               name=cn+'_pk',
               column=cl}
     }
    when {
      PackageToSchema(p, s);
    }
    where {
      AttributeToColumn(c, t);
    }
  }
```

The when and where clauses may contain any arbitrary OCL expressions in addition to the relation invocation expressions.

Relation invocations allow complex relations to be composed from simpler relations.

7.2.2 Top-level Relations

A transformation contains two kinds of relations ñ top-level and non-top-level. The execution of a transformation requires that all its top-level relations hold, whereas non-top-level relations are required to hold only when they are invoked directly or transitively from the **where** clause of another relation.

```
transformation umlRdbms (uml : SimpleUML, rdbms : SimpleRDBMS) {
        top relation PackageToSchema {…}
        top relation ClassToTable {…}
        relation AttributeToColumn {…}
}
```

A top-level relation has the keyword **top** to distinguish it syntactically. In the example above, PackageToSchema and ClassToTable are top level relations, whereas AttributeToColumn is a non-top-level relation.

7.2.3 Check and Enforce

Whether or not the relationship maybe enforced is determined by the target domain, which may be marked as **checkonly** or **enforced**. When a transformation is enforced in the direction of a **checkonly** domain, it is simply checked to see if there exists a valid match in the relevant model that satisfies the relationship. When a transformation executes in the direction of the model of an **enforced** domain, if checking fails, the target model is modified so as to satisfy the relationship, i.e. a check-before-enforce semantics.

In the example below, the domain for the "uml" model is marked **checkonly** and the domain for the rdbms model is marked **enforce**.

```
relation PackageToSchema   /* map each package to a schema */
  {
    checkonly domain uml p:Package {name=pn}
    enforce domain rdbms s:Schema {name=pn}
  }
```

If we are executing in the direction of uml and there exists a schema in rdbms for which we do not have a corresponding package with same name in uml, it is simply reported as an inconsistency ñ a package is not created because the "uml" model is not enforced, it is only checked.

However, if we are executing the transformation umlRdbms in the direction of rdbms then for each package in the uml model, the relation first checks if there exists a schema with same name in the rdbms model, and if it does not, a new schema is created in that model with the given name. To consider a variation of the above scenario, if we execute in the direction of rdbms and there is not a corresponding package with the same name in uml, then that schema will be deleted from the rdbms model, thus enforcing consistency in the **enforce** domain.

These rules apply depending on the target domain only. In this execution scenario, schema deletion will be the outcome even if the uml domain is marked as enforced, because the transformation is being executed in the direction of rdbms, and object creation, modification and deletion can only take place in the target model for the current execution.

Section 7.10, "Detailed Semantics," on page 19contains a more detailed description of the checking and enforcement semantics.

7.3 Pattern Matching

Let us use an example to discuss matching of the patterns associated with domains, known as *object template expressions*. We continue to use ClassToTable above as an example.

ClassToTable defines several object template expressions which are used to match patterns in candidate models. For example the following object template expression is associated with the domain of "uml":

```
c:Class {    namespace = p:Package {},
             kind='Persistent',
             name=cn
          }
```

A template expression match results in a binding of model elements from the candidate model to variables declared by the domain. A template expression match may be performed in a context where some of the domain variables may already have bindings to model elements (e.g. resulting from an evaluation of the **when** clause or other template expressions). In this case template expression match finds bindings only for the free variables of the domain.

For example, in the case of the above template expression associated with the "uml" domain, pattern matching will bind all the variables in the expression ("c", "p", and "cn"), starting from the domain's root variable "c" of type Class. In this example the variable "p" would already have a binding resulting from the evaluation of the **when** clause expression PackageToSchema(p, s). The matching proceeds by filtering all of the objects of type Class in the "uml" model, eliminating any which do not have the same literal values for their properties as the template expression. In the example, any Class with its "kind" property not set to 'Persistent' is eliminated.

For properties that are compared to variables, such as "name=cn", two cases arise. If the variable "cn" already has a value binding then any class that does not have the same value for its name property is eliminated. If the variable "cn" is free, as in our example, then it will get a binding to the value of the name property for all classes that are not filtered out due to a mismatch with other property comparisons. The value of "cn" will be either used in another domain, or can have additional constraints placed on it in the where expression of the domain or its owning relation.

Then the matching proceeds to properties whose values are compared to nested template expressions. For example, the property pattern "namespace = p:Package {}" will match only those classes whose "namespace" property has a non-null reference to a Package. At the same time, the variable "p" will be bound to refer to the Package. However, since in our example "p" is already bound in the **when** clause, the pattern will only match those classes whose "namespace" property has a reference to the same package that is bound to "p".

Arbitrarily deep nestings of template expressions are permitted, and matching and variable binding proceeds recursively until there is a set of value tuples corresponding to the variables of the domain and its template expression. For example the three variables: "c", "p" and "cn". These make a 3 tuple, and each valid match will result in a unique tuple representing the binding.

In a given relation invocation, there may be multiple such valid matches for a given template expression. How this multiplicity is dealt with depends on the execution direction.

For instance if ClassToTable is executed with rdbms as the target model, then for each valid match (i.e. valid tuple of variable bindings) of the uml domain, there must exist at least one valid match of the rdbms domain that satisfies the where clause. If for a given valid match of the uml domain, there does not exist a valid match of the rdbms domain, then (since rdbms domain is enforced) objects are created and properties are set as specified in the template expression associated with the rdbms domain. Also, for each valid match of the rdbms domain, there must exist at least one valid match of the uml domain that satisfies the where clause (this is required since uml domain is marked checkonly); otherwise objects will be deleted from the rdbms model such that it is no longer a valid match.

Section 7.10, "Detailed Semantics," on page 19 contains a more detailed description of the pattern matching semantics.

7.4 Keys and Object Creation using Patterns

As mentioned previously, an *object template expression* also serves as a template for creating an object in a target model. When for a given valid match of the source domain pattern, there does not exist a valid match of the target domain pattern, then the object template expressions of the target domain are used as templates to create objects in the target model. For example, when ClassToTable is executed with rdbms as the target model, the following object template expression serves as a template for creating objects in the rdbms model:

```
t:Table {
    schema = s:Schema {},
    name = cn,
    column = cl:Column {name=cn+'_tid',  type='NUMBER'},
             primaryKey = k:PrimaryKey {name=cn+'_pk', column=cl}
}
```

The template associated with *Table* specifies that a table object needs to be created with properties “schema”, “name”, “column” and “primaryKey” set to values as specified in the template expression. Similarly the templates associated with *Column, PrimaryKey,* etc, specify how their respective objects should be created.

However, when creating objects we want to ensure that duplicate objects are not created when the required objects already exist. In such cases we just want to update the existing objects. But how do we ensure this? The MOF allows for a single property of a class to be nominated as identifying. However, for most metamodels, this is insufficient to uniquely identify objects. The relations metamodel introduces the concept of *Key* which defines a set of properties of a class that uniquely identify an object instance of the class in a model. A class may have multiple keys (as in relational databases).

For example, continuing with the ClassToTable relation, we might wish to specify that in simpleRDBMS models a table is uniquely identified by two properties ñ its name and the schema it belongs to. We can state this as follows:

key Table {schema, name};

Keys are used at the time of object creation ñ if an object template expression has properties corresponding to a key of the associated class, then the key is used to locate a matching object in the model; a new object is created only when a matching object does not exist.

In our example, consider the case where we have a persistent class with name “foo” in the uml model, and a table with a matching name ëfooí exists in a matching schema in the rdbms model but the table does not have matching values for properties “column” and “primaryKey”. In this case as per our pattern matching semantics, the rdbms model does not have a valid match of the pattern associated with *Table* (since two of its properties do not match) and so we need to create objects to satisfy the relation. However, since the existing table matches the specified key properties (i.e. name and schema), we do not have to create a new table; we just have to update the table to set its “column” and “primaryKey” properties.

7.5 Restrictions on Expressions

In order to guarantee executability ñ i.e. there exists a bounded algorithm to enforce a relation in the direction of a given target model ñ expressions occurring in a relation are required to satisfy the following conditions:

1. It should be possible to organize the expressions that occur in the when clause, the source domains and the where clause, into a sequential order that contains only the following kinds of expressions:

   1.1. An expression of the form

   <object>.<property> = <variable>

   Where <variable> is a free variable, and <object> is either a variable bound to an object template expression of an opposite domain pattern or a variable that gets a binding from a preceding expression in the expression order. This expression provides a binding for variable <variable>.

   1.2. An expression of the form:

   <object>.<property> = <expression>

   Where <object> is either a variable bound to an object template expression of a domain pattern or a variable that gets a binding from a preceding expression in the expression order. There are no free variable occurrences in <expression> (variable occurrences if any should all have been bound in the preceding expressions)

   1.3. No other expression has free variable occurrences (all their variable occurrence should have been bound in the preceding expressions)

2. It should be possible to organize the expressions that occur in the target domain, into a sequential order that contains

only the following kinds of expressions:

2.1. An expression of the form

<object>.<property> = <expression>

Where <object> is either a variable bound to an object template expression of the domain pattern or a variable that gets a binding from a preceding expression in the expression order. There are no free variable occurrences in <expression> (variable occurrences if any should all have been bound in the preceding expressions)

2.2. No other expression has free variable occurrences (all their variable occurrences should have been bound in the preceding expressions)

7.6 Change Propagation

In relations, the effect of propagating a change from a source model to a target model is semantically equivalent to executing the entire transformation afresh in the direction of the target model. The semantics of object creation and deletion guarantee that only the required parts of the target model are affected by the change. Firstly, the semantics of check-before-enforce ensures that target model elements that satisfy the relations are not touched. Secondly, key-based object selection ensures that existing objects are updated where applicable. Thirdly, deletion semantics ensures that an object is deleted only when no other rule requires it to exist.

An implementation is free to use any efficient algorithm for change propagation as long as it is consistent with the above semantics. The relations-to-core mapping provides one such implementation option, as core mappings support incremental change propagation more directly.

Please refer to Chapter 10 on relations to core mapping for a detailed description of how change propagation is handled in core and how that applies to relations.

7.7 In-place Transformations

A transformation may be considered in-place when its source and target candidate models are both bound to the same model at runtime. The following additional comments apply to the enforcement semantics of an in-place transformation:

- A relation is re-evaluated after each enforcement-induced modification to a target pattern instance of the model.
- A relation's evaluation stops when all the pattern instances satisfy the relationship.

7.8 Integrating Black-box Operations with Relations

A relation may optionally have an associated black-box operational implementation to enforce a domain. The black-box operation is invoked when the relation is executed in the direction of the enforced domain and the relation evaluates to false as per the checking semantics. The invoked operation is responsible for making the necessary changes to the model in order to satisfy the specified relationship. It is a runtime exception if the relation evaluates to false after the operation returns. The signature of the operation can be derived from the domain specification of the relation ñ an output parameter corresponding to the enforced domain, and an input parameter corresponding to each of the other domains.

The Relations which may be implemented by Mapping Operations and Black box Operations are restricted in the following ways:

- Their domain should be primitive or contain a simple object template (with no sub-elements)
- The when and where clause should not define variables.

These restrictions allow for a simple call-out semantics, which does not need any constraint evaluation before, and constraint checking after the operation invocation. **When** clauses, **where** clauses, patterns and other machinery can be used in a "wrapper" relation which invokes the simple relation with values constrained by the wrapper.

7.9 Executing a Transformation in Checkonly mode

A transformation can be executed in “checkonly” mode. In this mode, the transformation simply checks whether the relations hold in all directions, and reports errors when they do not. No enforcement is done in any direction, irrespective of whether the domains are marked **checkonly** or **enforced**.

7.10 Detailed Semantics

This section provides a detailed description of the semantics of relations. A more formal semantics are given in the section on “relations” to core mapping in Section 12 by specifying how a relation specification maps to the core model.

To simplify the description of semantics, we can view a relation as having the following abstract structure:

```
Relation R
{
  Var <R_variable_set>   //  declaration of variables used in the relation
  [checkonly | enforce] Domain:<typed_model_1>
             <domain_1_variable_set> // subset of <R_variable_set>
  {
    <domain_1_pattern> [<domain_1_condition>]
  }
  ...
  [checkonly | enforce] Domain:< typed_model_n>
             <domain_n_variable_set> // subset of <R_variable_set>
  {
    <domain_n_pattern> [<domain_n_condition>]
  }  // n >= 2

  [when <when_variable_set> <when_condition>]
  [where <where_condition>]
}
```

With the following properties:

- < R_variable_set > is the set of variables occurring in the relation.
- <domain_k_variable_set> is the set of variables occurring in domain k. It is a subset of <R_variable_set>, for all k = 1..n.
- <when_variable_set> is the set of variables occurring in the **when** clause. It is a subset of <R_variable_set>.
- The intersection of domain variable sets need not be null, i.e. a variable may occur in multiple domains.
- The intersection of a domain variable set and when variable set need not be null.
- The term <domain_k_pattern> refers to the set of constraints implied by the pattern of domain k. Please recall that a pattern can be viewed as a set of variables, and a set of constraints that model elements bound to those variables must

satisfy in order to qualify as a valid binding of the pattern. Please refer to Section 9.10.3 for a detailed discussion on pattern matching semantics. Given below is an example pattern and the constraint implied by it.

Pattern:

c:Class {kind=Persistent, name=cn, attribute=a:Attribute {}}

Implied constraint:

c.kind = Persistent and c.name = cn and c.attribute->includes(a)

7.10.1 Checking Semantics

Checking of a relation in the direction of model k (i.e. with model k as the target model) has the following semantics:

For each valid binding of variables of the **when** clause and variables of domains other than the target domain k, that satisfy the **when** condition and source domain patterns and conditions, there must exist a valid binding of the remaining unbound variables of domain k that satisfies domain k's pattern and **where** condition.

A more formal definition of the checking semantics in terms of a predicate calculus formula is given in Appendix B.

7.10.2 Enforcement Semantics

Enforcement of a relation in the direction of model k (i.e. with model k as the target model) has the following semantics:

For each valid binding of variables of the **when** clause and variables of domains other than the target domain k, that satisfy the **when** condition and source domain patterns and conditions, if there does not exist a valid binding of the remaining unbound variables of domain k that satisfies domain k's pattern and **where** condition, then create objects (or select and modify if they already exist) and assign properties as specified in domain k pattern. Whether an object is selected from the model or created afresh depends on whether the model already contains an object that matches the key property values, if any, specified in the object template. It is an error, if the template expression evaluation results in an assignment of a value to a property that clashes with another value set for the same property by another rule in the transformation execution, indicating an inconsistent specification. For primitive types, the values clash when they are different. An object assignment to a link of multiplicity “one” clashes if the object being assigned is different from the one that already exists.

Also, for each valid binding of variables of domain k pattern that satisfies domain k condition, if there does not exist a valid binding of variables of the **when** clause and source domains that satisfies the **when** condition, source domain patterns and **where** condition, and at least one of the source domains is marked **checkonly** (or **enforce**, which entails check), then delete the objects bound to the variables of domain k when the following condition is satisfied: delete an object only if it is not required to exist by any other valid binding of the source domains as per the enforcement semantics (i.e. avoid delete followed by an immediate create).

A more formal definition of the enforcement semantics in terms of a predicate calculus formula is given in Annex B.

7.10.3 Pattern Matching Semantics

7.10.3.1 Introduction

This section describes the semantics of the pattern specification construct supported by the relations language. To simplify the description of the semantics we introduce a simple "infrastructure" model of patterns and explain its semantics. Then Section 7.10.3.3, “Patterns Specifying Collections,” on page 22 explains how Template expressions as used within a QVT relation specification can be transformed to this simple model. The example meta-model of Figure 7.1 and its corresponding instance model of Figure 7.2 are used as examples throughout the text.

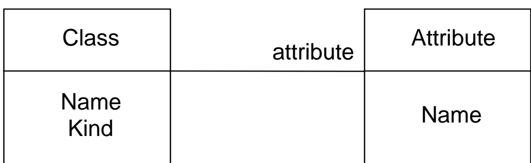


Figure 7.1 - Simple UML Model

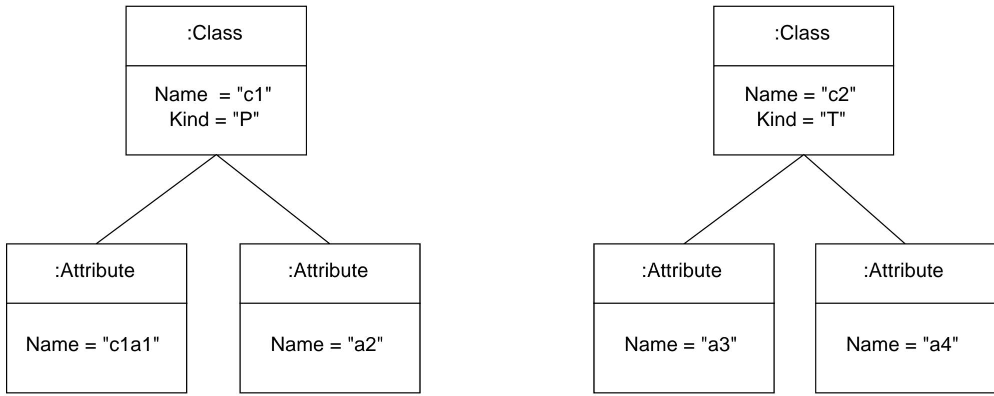


Figure 7.2 - Instance Model

7.10.3.2 Pattern Infrastructure

To simplify the semantics definition we assume a pattern to have the following abstract structure

Pattern =
{

e_1: <classname$_1$>, e_2: <classname$_2$> …. e_n:<classname$_N$>
l_1 : <assoc$_1$> (e_i, e_j) …. l_m:<assoc$_M$>(e_u, e_w)
where <predicate>

}

A pattern can be viewed as a graph where the pattern elements, e_1, e_2, …e_n, with types <$classname_1$>, <$classname_2$>…<$classname_n$> respectively, are the nodes of the graph and pattern links l_1, l_2, …l_m are the edges. The predicate is a Boolean expression that may refer to the pattern elements. The predicates may refer to variables other than the pattern elements; these are the free variables of a pattern. A pattern is used to find matching sub-graphs in a model. A sub-graph of a model consisting of objects o_1, o_2, …$o_{n,}$ matches a pattern as described above if and only if :

- o_1 is of type <$classname_1$> or one of its subtypes, and o_2 is of type <$classname_2$> or one of its subtypes, and so on…
- o_i and o_j are linked by association <$assoc_1$> and o_u and o_w are linked by association <$assoc_2$>, and so on…
- There exists one or more bindings of values to free variables such that <predicate> evaluates to true when references to e_1,e_2, …e_n are replaced by o_1, o_2, …o_n respectively.

Once a pattern has matched each e_i is bound to the corresponding o_i and each free variable v_i is bound to the value with which the <predicate> was evaluated while performing the match. For example:

```
Pattern {
c1: Class, a1: Attribute
l1: attrs (c1, a1)
   where c1.name = X and a1.name = X + Y
}
```

In the above example X and Y are free variables. The only sub-graph of the model in Figure 5 that matches the above pattern is <c1, c1a1>. This matching binds X to “c1” and Y to “a1”.

7.10.3.3 Patterns Specifying Collections

The type of elements in a pattern could be a collection such as ñ Set, OrderedSet, Bag or Sequence. If the type of e_i is a collection of type <$classname_i$> then a sub-graph of a model matches the pattern if and only if

- o_i is a collection of objects from the model of type <$classname_i$>
- There is no collection of objects from the model of type <$classname_i$>, o_j such that o_i is a sub-collection of o_j and replacement of o_i with o_j will satisfy the other pattern matching criteria
- If *lj: <assocname>(e_m, ei)* is a link in the pattern and the type of e_m is <$classname_m$> then every element of o_i must be linked to o_m by the association *<assocname>*
- If *lj: <assocname>(e_m, ei)* is a link in the pattern and the type of e_m is also set of <$classname_m$> then every element of o_i must be linked to every element of o_m by the association *<assocname>*

For example:

```
Pattern {
c1: Class, a1: Set(Attribute)
l1: attrs (c1, a1)
  where TRUE
}
```

The two sub-graphs <c1, {c1a1, a2}> and <c2, {a3, a4}> of the instance model in Figure 5 match the above pattern.

Constraints

A pattern must have at least one element. Only first order sets are allowed, i.e., Elements cannot have type set of sets.

7.10.3.4 QVT Template Expressions

Here we describe the QVT Template Expressions metamodel (Section 7.11.2, “QVTTemplate Package,” on page 30) in terms of the infrastructure described above.

An *object template expression* models a *pattern element* with its type as the *referred class* of the object template expression. Similarly a *collection template expression* models a *pattern element* with its type as the collection of the *referred class* of the collection template expression. A *property template item* connecting two *object template expressions* models a link.

The predicate part of a pattern is a conjunction of the following expressions:

- An expression of the form “*referredProperty.name = value*” derived from the *referredProperty* and *value* expression associated with a *property template item*.
- The *part expression* associated with a collection template expression.
- An expression asserting that no collection is empty
- The *where expression* associated with each *template expression*.

For example, consider the pattern specified by the concrete syntax:

Class {name = X, attribute = Attribute {name = X + Y}}

The instance model for the above example is shown in Figure 7.3. For brevity the detailed structure of OCL expressions is excluded and is replaced by a *variable expression* and a note representing the complete OCL expression. The instance model has two *object template expression* nodes corresponding to Class and Attribute. The node corresponding to Class has two *property template item* nodes corresponding to the two properties ñ name and attribute. Same is the case with the Attribute node. Each of the *property template item* nodes are associated with OCL expression nodes corresponding to the expressions ñ X and X+Y.

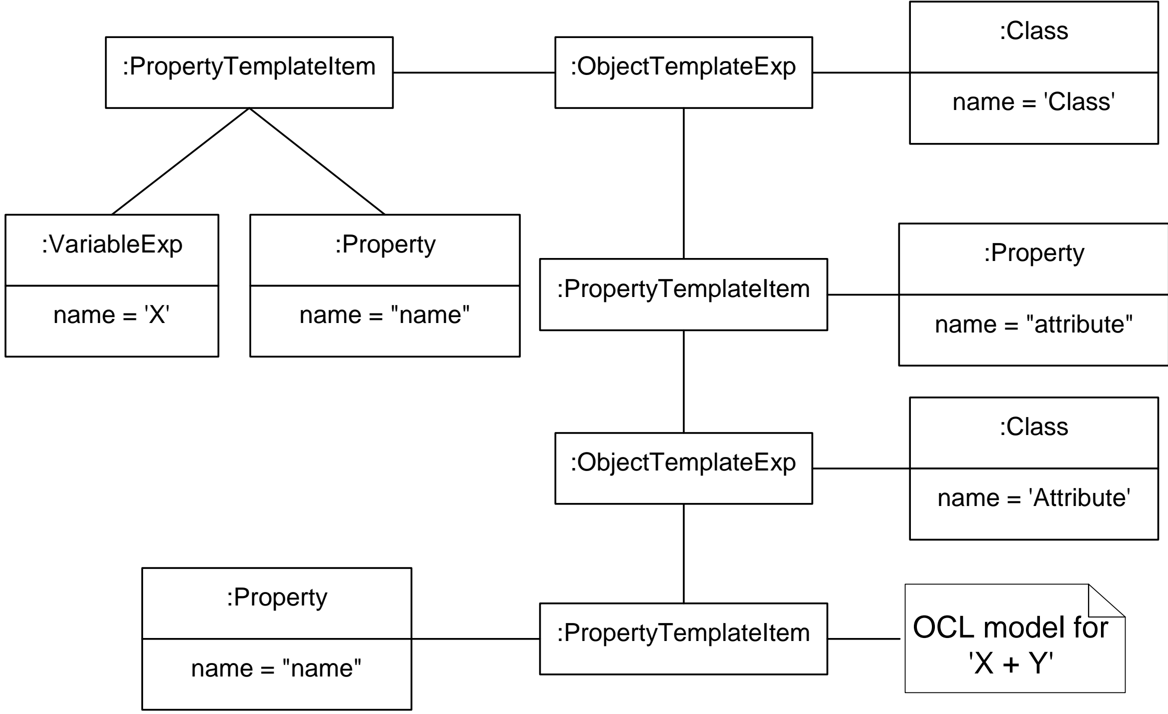


Figure 7.3 - Example Pattern Instance

The pattern structure corresponding to the above model is:

```
Pattern {
        dummy: Class, dummy1: Attribute
            dummy2: attribute (dummy, dummy1)
                freevars X, Y
    where dummy.name = X and dummy1.name = X + Y
}
```

7.11 Abstract Syntax and Semantics

The Relations metamodel is structured into three packages: QVTBase, QVTTemplate, and QVTRelation.

7.11.1 QVTBase Package

This package contains a set of basic concepts, many reused from the EMOF and OCL specifications that structure transformations, their rules, and their input and output models. It also introduces the notion of a *pattern* as a set of predicates over *variable*s in OCL expressions. These classes are extended in language-specific packages to provide language-specific semantics.

7.11.1.1 Transformation

A *transformation* defines how one set of models can be transformed into another. It contains a set of *rules* that specify its execution behavior. It is executed on a set of models whose types are specified by a set of *typed model* parameters associated with the transformation.

Syntactically, a `Transformation` is a subclass of both a `Package` and a `Class`. As a `Package` it provides a namespace for the rules contained in it. As a `Class` it can define properties and operations - properties to specify configuration values, if any, needed at runtime and operations to implement utility functions, if any, required by the transformation.

Superclasses

`Package`

`Class`

Associations

`modelParameter: TypedModel [*] {composes}`

The set of typed models which specify the types of models that may participate in the transformation.

`rule: Rule [*] {composes}`

The rules owned by the transformation, which together specify the execution behavior of the transformation.

`ownedTag: Tag [*] {composes}`

The set of tags associated with the transformation whose values may be used to configure its runtime environment.

`extends: Transformation [0..1]`

A transformation can extend another transformation. The rules of the extended transformation are included in the extending transformation to specify the latter's execution behavior. Extension is transitive.

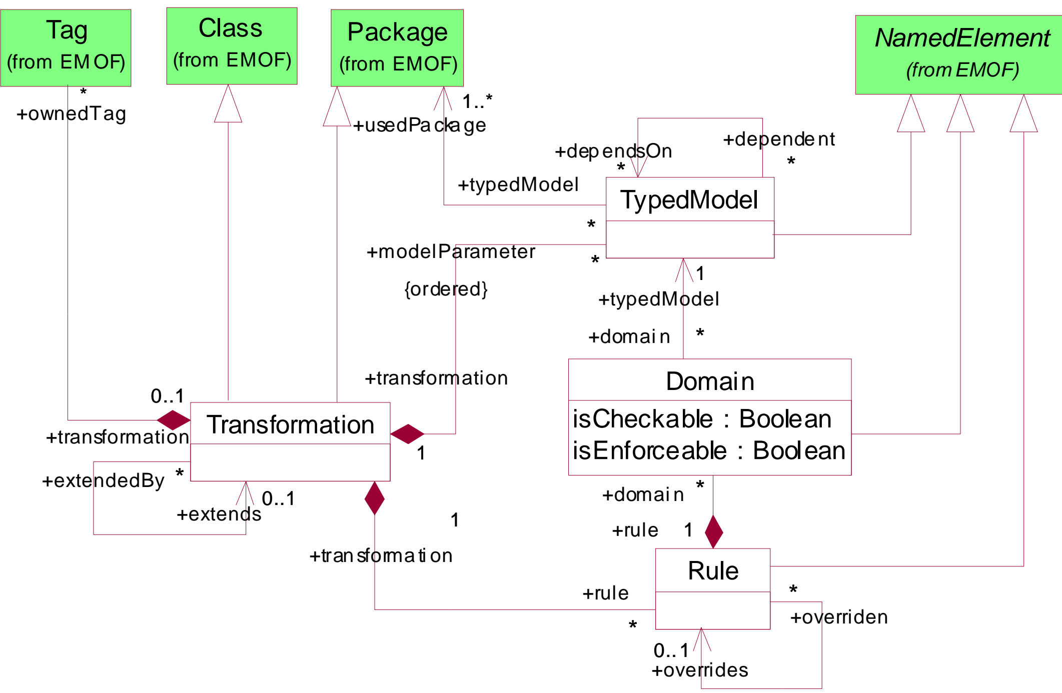


Figure 7.4 - QVTBase Package - Transformations and rules

7.11.1.2 TypedModel

A *typed model* specifies a named, typed parameter of a *transformation*. At runtime, a model which is passed to the transformation by this name is constrained to contain only those model elements whose types are specified in the set of model packages associated with the typed model. At runtime, a transformation is always executed in a particular direction by selecting one of the typed models as the target of the transformation.

A target model may be produced from more than one source model, and in such cases a transformation may require the selection of model elements from one source model to be constrained by the selection of model elements from another source model. This situation may be modelled by a typed model declaring a dependency on another typed model.

Superclasses

`NamedElement`

Associations

`transformation: Transformation [1]`

The transformation that owns the typed model.

`usedPackage: Package [1..*]`

The meta model packages that specify the types for the model elements of the models that conform to this typed model.

`dependsOn: TypedModel [*]`

The set of typed models that this typed model declares a dependency on.

7.11.1.3 Domain

A *domain* specifies a set of model elements of a *typed model* that are of interest to a *rule*. `Domain` is an abstract class, whose concrete subclasses are responsible for specifying the exact mechanism by which the set of model elements of a domain may be specified. It may be specified as a pattern graph, a set of typed variables and constraints, or any other suitable mechanism (Please see 7.11.3.2, 'RelationDomain' and 9.17.5, 'CoreDomain' for details).

A domain may be marked as *checkable* or e*nforceable*. A checkable domain declares that the owning rule is only required to check whether the model elements specified by the domain exist in the target model and report errors when they do not. An enforceable domain declares that the owning rule must ensure that the model elements specified by the domain exist in the target model when the transformation is executed in the direction of the typed model associated with the domain.

Superclasses

`NamedElement`

Attributes

`isCheckable : Boolean`

Indicates that the domain is checkable.

`isEnforceable : Boolean`

Indicates that the domain is enforceable.

Associations

`rule: Rule [1]`

The rule that owns the domain.

`typedModel: TypedModel [1]`

The typed model that contains the types of the model elements specified by the domain.

7.11.1.4 Rule

A *rule* specifies how the model elements specified by its *domains* are related with each other, and how the model elements of one domain are to be computed from the model elements of the other domains. `Rule` is an abstract class, whose concrete subclasses are responsible for specifying the exact semantics of how the domains are related and computed from one another (please see 7.11.3.1, 'Relation' and 9.17.6, 'Mapping' for details).

A rule may conditionally override another rule. The overriding rule is executed in place of the overridden rule when the overriding conditions are satisfied. The exact semantics of overriding are subclass specific.

Superclasses

`NamedElement`

Associations

`domain: Domain [*] {composes}`

The domains owned by this rule.

`transformation: Transformation[1]`

The transformation that owns this rule.

`overrides: Rule [0..1]`

The rule which this rule overrides.

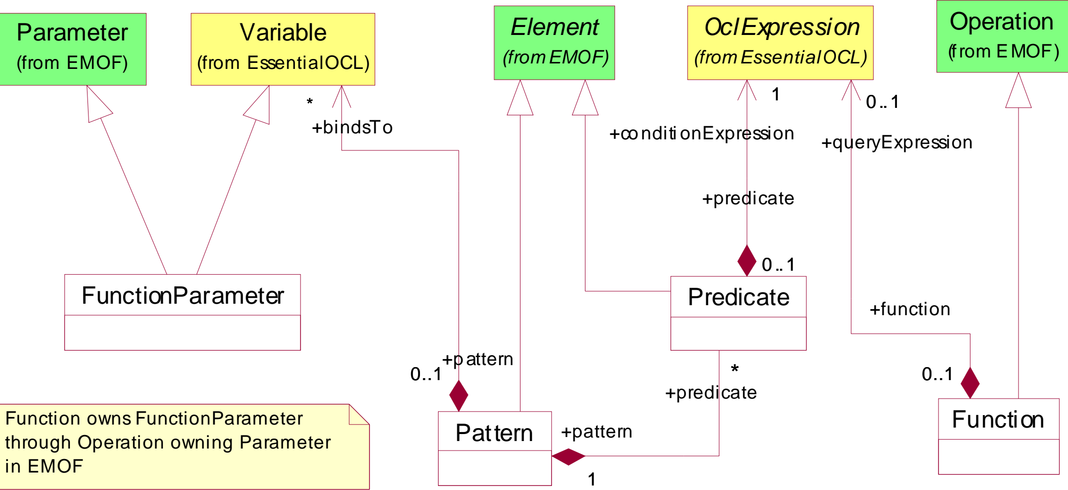


Figure 7.5 - QVTBase Package - Patterns and Functions

7.11.1.5 Function

A *function* is a side-effect-free *operation* owned by a *transformation*. A function is required to produce the same result each time it is invoked with the same arguments. A function may be specified by an OCL expression, or it may have a black-box implementation.

Superclasses

`Operation`

Associations

`queryExpression: OclExpression [0..1] {composes}`

The OCL expression that specifies the function. If this reference is absent, then a black-box implementation is assumed.

7.11.1.6 FunctionParameter

A *function parameter* specifies the parameters of a *function*.

Syntactically, it is a subclass of the classes `Parameter` and `Variable`. By virtue of it being a subclass of `Variable`, it enables the OCL expression that specifies a function to access the function parameters as named variables. A function owns its parameters through `Operation` owning `Parameters` in EMOF.

Superclasses

`Parameter`

`Variable`

7.11.1.7 Predicate

A *predicate* is a boolean-valued expression owned by a *pattern*. It is specified by an *OCL expression* which may contain references to the variables of the pattern that owns the predicate.

Superclasses

`Element`

Associations

`conditionExpression: OclExpression [1] {composes}`

The OCL expression that specifies the predicate.

`pattern: Pattern [1]`

The pattern that owns the predicate.

7.11.1.8 Pattern

A pattern is a set of variable declarations and predicates, which when evaluated in the context of a model, results in a set of bindings for the variables.

Superclasses

`Element`

Associations

`bindsTo: Variable [*] {composes}`

The set of variables which are to be bound when the pattern is evaluated.

`predicate: Predicate [*] {composes}`

The set of predicates that must evaluate to true for a binding of the variables of the pattern to be considered a valid binding.

7.11.2 QVTTemplate Package

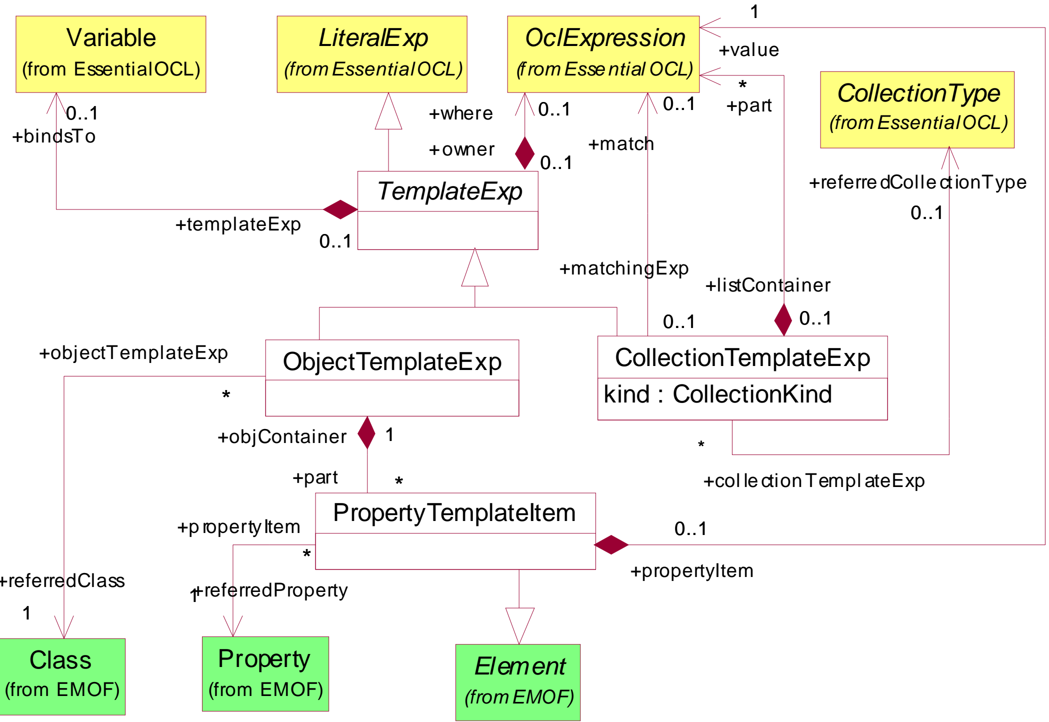


Figure 7.6 - QVT Template Package

7.11.2.1 TemplateExp

A *template expression* specifies a pattern that matches model elements in a candidate model of a transformation. The matched model element may be bound to a variable and this variable may be used in other parts of the expression. A template expression matches a part of a model only when the **where** expression associated with the template expression evaluates to true. A template expression may match either a single model element or a collection of model elements depending on whether it is an *object template expression* or a *collection template expression*.

Superclasses

`LiteralExp`

Associations

`bindsTo: Variable [0..1] {composes}`

The variable that refers to the model element matched by this template expression.

`where: OclExpression [0..1] {composes}`

A boolean expression that must evaluate to true for the template expression to match.

7.11.2.2 ObjectTemplateExp

An *object template expression* specifies a pattern that may match only single model elements. An object template has a type specified by the referred *class*. An object template is specified by a collection of *property template items* each corresponding to different attributes of the referred class.

Superclasses

`TemplateExp`

Associations

`referredClass: Class [1]`

The EMOF Class that specifies the type of objects to be matched by this expression.

`part: PropertyTemplateItem [*] {composes}`

Specification of a value expression that must be satisfied by the corresponding slot of the object that matches the object template expression.

7.11.2.3 CollectionTemplateExp

A *collection template expression* specifies a pattern that matches a collection of elements. The type of collection that a template matches is given by the *referred collection type*. A collection template expression can be specified in three different ways ñ enumeration, comprehension and member selection. The interpretation of the model in the three different cases is as follows ñ

- Enumeration - The set of *part* expressions specify exactly the elements that must comprise the collection.
- Comprehension - The *match* expression, which can be either a variable or an object template, binds to one element of the collection. If an object template is used then every element in the collection must match the pattern specified by the *object template expression*. Every element in the collection must additionally satisfy the *part* expression.
- Member Selection - The *match* expression, which can only be a variable, binds to one element of the collection. If the collection type is a sequence or an ordered set the *match* expression binds to the first element of the sequence or the ordered set. The *part* expression, which can only be a variable, binds to the rest of the collection.

Superclasses

`TemplateExp`

Attributes

`kind : CollectionKind`

A collection template expression can be of three kinds ñ `Enumeration`, `Comprehension`, and `MemberSelection`. This indicates how the collection pattern has been expressed - by enumerating each element or in the standard set comprehension style or using the first-member-and-the-rest style.

Associations

`referredCollectionType : CollectionType` [1]

The type of the collection that is being specified. It can be any of the EMOF collection types such as `Set`, `Sequence`, `OrderedSet`, etc.

`match : OclExpression` [0..1]

The interpretation of this association depends on *collection kind*. For *member selection* it specifies a variable that binds to the first element of the collection, for *comprehension* it specifies either a variable or an object template that binds to one element of the collection, and for *enumeration* it must be absent. In the case of member selec-

tion, the variable name can be "_" indicating that the variable will not be referred to in any other part of the specification. In the case of comprehension, an object template specifies a pattern that each member of the collection must match.

`part : OclExpression` [0..*]

The interpretation of this association depends on *collection kind*. When the collection kind is enumeration, *part* contains one expression per each member of the matching collection. When the collection kind is comprehension there is at most one *part* expression and it specifies a constraint that must be satisfied by each member of the matching collection. When the collection kind is member selection then there is exactly one *part* expression and it specifies a variable that binds to the rest of the collection elements.

7.11.2.4 PropertyTemplateItem

Property template items are used to specify constraints on the values of the slots of the model element matching the container object template expression. The constraint is on the slot that is an instance of the referred property and the constraining expression is given by the `value` expression.

Superclasses

`Element`

Associations

`referredProperty: Property [1]`

The EMOF Property that identifies the slot that is being constrained.

`value: OclExpression [0..1] {composes}`

The value that the slot may take.

7.11.3 QVTRelation Package

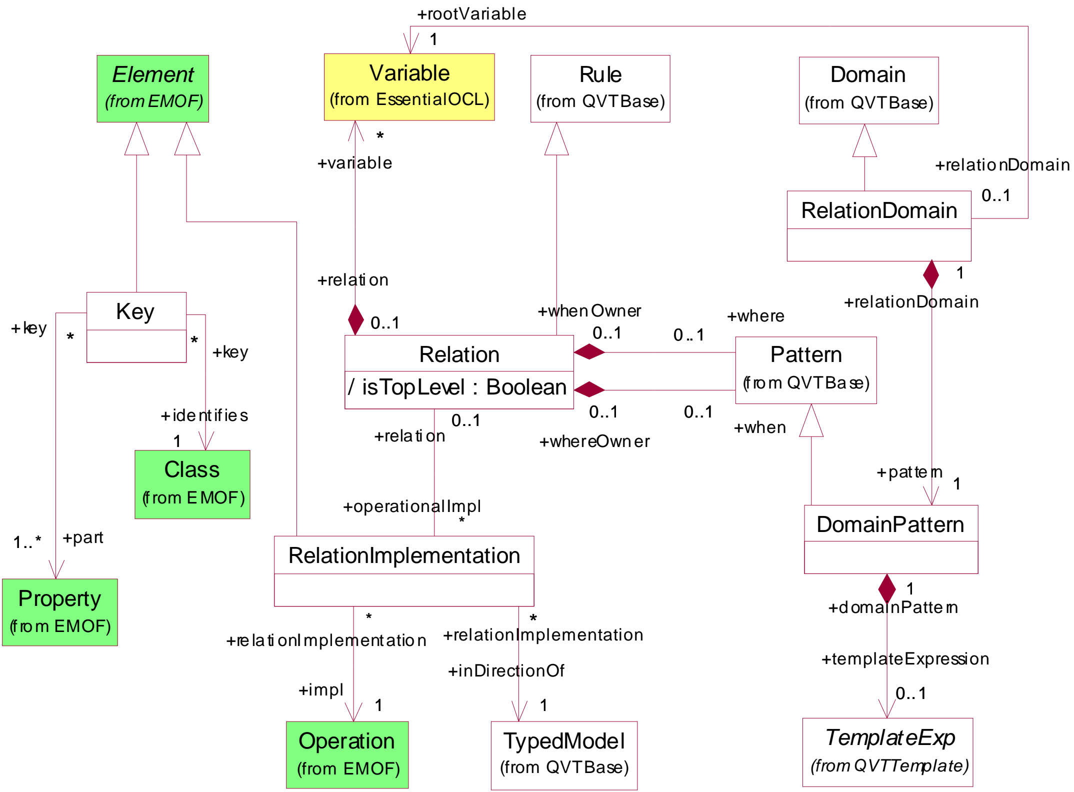


Figure 7.7 - QVT Relation Package

7.11.3.1 Relation

A *relation* is the basic unit of transformation behavior specification in the relations language. It is a concrete subclass of `Rule`. It specifies a relationship that must hold between the model elements of a set of candidate models that conform to the *typed models* of the transformation that owns the relation. A relation is defined by two or more *relation domains* that specify the model elements that are to be related, a **when** clause that specifies the conditions under which the relationship needs to hold, and a **where** clause that specifies the condition that must be satisfied by the model elements that are being related. Please refer to sections 7.1 - 7.9 for a detailed description of the semantics, and section 12 for a more formal description in terms of a mapping to the core model.

Superclasses

`Rule`

Attributes

`/isTopLevel : Boolean`

Indicates whether the relation is a top-level relation or a non-top-level relation. A top-level relation is a relation that is not invoked from the **where** clause of any other relation. A non-top-level relation is a relation that is

invoked from the **where** clause of some other relation. The execution of a transformation requires that all its top-level relations hold, whereas non-top-level relations are required to hold only when they are invoked directly or transitively from the **where** clause of another relation. This is a derived attribute.

Associations

`variable: Variable [*]  {composes}`

The set of variables occurring in the relation. This set includes all the variables occurring in its domains and **when** and **where** clauses.

`/domain: Domain [*]  {composes} (from Rule)`

The set of domains owned by the relation that specify the model elements participating in the relationship. `Relation` inherits this association from `Rule`, and is constrained to own only `RelationDomains` via this association.

`when: Pattern [0..1] {composes}`

The pattern (as a set of variables and predicates) that specifies the **when** clause of the relation.

`where: Pattern [0..1] {composes}`

The pattern (as a set of variables and predicates) that specifies the **where** clause of the relation.

7.11.3.2 RelationDomain

The class `RelationDomain` specifies the domains of a *relation*. It is a concrete subclass of `Domain`. A *relation domain* has a distinguished typed variable called the *root variable* that can be matched in a model of a given model type. A relation domain specifies a set of model elements of interest by means of a *domain pattern*, which can be viewed as a graph of object nodes, their properties and association links, with a distinguished root node that is bound to the root variable of the relation domain. Please refer to sections 7.2, 7.3 and 7.10 for a detailed discussion of the semantics of domains and domain patterns in the relationsí language.

Superclasses

`Domain`

Associations

`rootVariable: Variable [1]`

The distinguished typed variable of the relation domain that can be matched in a model of a given model type.

`/typedModel: TypedModel [1] (from Domain)`

The typed model that specifies the model type of the models in which the typed variable (*root variable*) of the domain can be matched.

`pattern: DomainPattern [1] {composes}`

The domain pattern that specifies the model elements of the relation domain. The root *object template expression* (i.e. the root node) of the domain pattern must be bound to the root variable of the relation domain.

7.11.3.3 DomainPattern

The class `DomainPattern` is a subclass of the class `Pattern`. A *domain pattern* can specify an arbitrarily complex pattern graph in terms of *template expressions* consisting of *object template expressions, collection template expressions* and *property template items*. A domain pattern has a distinguished root template expression that is required to be bound

to the root variable of the *relation domain* that owns the domain pattern. An object template expression can have other template expressions nested inside it to an arbitrary depth. Please refer to section 7.10.3 for a detailed discussion of the semantics of pattern matching involving template expressions.

Superclasses

`Pattern`

Associations

`templateExpression: TemplateExp [0..1] {composes}`

The root template expression of the domain pattern. This template expression must be bound to the root variable of the relation domain that owns this domain pattern, and its type must be the same as the type of the root variable.

`relationDomain: RelationDomain [1]`

The relation domain that owns this domain pattern.

7.11.3.4 Key

A *key* defines a set of properties of a class that uniquely identify an instance of the class in a model extent. A class may have multiple keys (as in relational databases). Please refer to section 7.4 for a detailed description of the role played by keys in the enforcement semantics of relations.

Superclasses

`Element`

Associations

`identifies: Class [1]`

The class that is identified by the key.

`part: Property [1..*]`

Properties of the class that make up the key.

7.11.3.5 RelationImplementation

A *RelationImplementation* specifies an optional black-box operational implementation to enforce a *domain* of a *relation*. The black-box operation is invoked when the relation is executed in the direction of the *typed model* associated with the enforced domain and the relation evaluates to false as per the checking semantics. The invoked operation is responsible for making the necessary changes to the model in order to satisfy the specified relationship. It is a runtime exception if the relation evaluates to false after the operation returns. The signature of the operation can be derived from the domain specification of the relation ñ an output parameter corresponding to the enforced domain, and an input parameter corresponding to each of the other domains.

Superclasses

`Element`

Associations

`impl: Operation [1]`

The operation that implements the relation in the given direction.

`inDirectionOf: TypedModel [1]`

The direction of the relation being implemented.

```
relation: Relation [1]
```

The relation being implemented.

7.12 Standard Library

The QVT standard library for Relations is the OCL standard library. No additional operation is defined.

7.13 Concrete Syntax

This section provides both the textual and graphical concrete syntaxes for the Relations Language.

7.13.1 Relations Textual Syntax Grammar

```
<topLevel>  ::= ('import' <filename> ';' )* <transformation>*
   <filename>  ::= <identifier>
   <transformation> ::= 'transformation' <identifier> '('
      <modelDecl> (; <modelDecl>)* ')'
               ['extends' <identifier> (',' <identifier>)* ]
   '{'
                        <keyDecl>* ( <relation> | <query> )*
               }'

   <modelDecl> ::= <modelId> ':' <metaModelId> (, <metaModelId>)*
   <modelId> ::= <identifier>
   <metaModelId> ::= <identifier>

<keyDecl> ::= 'key' <classId> '{' <propertyId> (, <propertyId>)* '}' ';'
<classId> ::= <identifier>
<propertyId> ::= <identifier>

<relation> ::= ['top'] 'relation' <identifier> ['overrides' <identifier>]
              '{'
<varDeclaration>*
               (<domain> | <primitiveTypeDomain>)+
               <when>? <where>?
              '}'

<varDeclaration> ::= <identifier> (, <identifier>)* ':' <typeCS> ';'

<domain> ::=  [<checkEnforceQualifier>]
             'domain' <modelId> [ <identifier> ] ':' <typeCS>
              '{' <propertyTemplate>* '}' [ '{' <oclExpressionCS> '}' ]
             ['implementedby' <OperationCallExpCS>]
             ';'

<primitiveTypeDomain> ::= 'primitive' 'domain' <identifier> ':' <typeCS>
                 ';'

<checkEnforceQualifier> ::= 'checkonly' | 'enforce'

<when>  ::= 'when' '{' <oclExpressionCS> '}'

<where>  ::= 'where' '{' <oclExpressionCS> '}'
```

```
<query>     ::= 'query' <pathNameCS> '(' [<paramDeclaration> (','
<paramDeclaration>)*] ')' ':'
[<paramDeclaration> (','
                          <paramDeclaration>)*]
                (
                  ';' | '{' <oclExpressionCS> '}'
                )
```

7.13.2 Expressions syntax (extensions to OCL)

```
<oclExpressionCS> ::=   <propertyCallExpCS>
                        |   <variableExpCS>
                        |   <literalExpCS>
                        |   <letExpCS>
                        |   <ifExpCS>
                        |   <template>
                        |   '(' <oclExpressionCS> ')'
            |   (<oclExpressionCS> ';')*

<template> ::= <objectTemplate>
             | <collectionTemplate>

objectTemplate ::= [<identifier>] ':' <typeCS> '{'
                   <propertyTemplate>* '}'

<propertyTemplate> ::= <identifier> '=' <oclExpressionCS>

<collectionTemplate> ::= <identifier> ':' <collectionTypeIdentifierCS>
            '(' <typeCS> ')'
                          '{'
                              <setComprehensionExpression>
                              | <memberSelectionExprCS>
                              | <oclExpressionCSList>
                  (',' <oclExpressionCSList>)*
                          '}'

<setComprehensionExpression> ::= (<identifier> | <objectTemplate>)
                  '|' <oclExpressionCS>

<memberSelectionExprCS> ::=  (<identifier> | <objectTemplate> | '_')
               '++' (<identifier> | '_')
```

7.13.3 Graphical Syntax

7.13.3.1 Introduction

Diagrammatic notations have been a key factor in the success of UML, allowing users to specify abstractions of underlying systems in a natural and intuitive way. Therefore our proposal contains a diagrammatic syntax to complement the textual syntax of Section 7.13.1. There are two ways in which the diagrammatic syntax is used, as a way of:

- representing transformations in standard UML class diagrams.
- representing transformations, domains and patterns in a new diagram form: transformation diagrams.

The syntax is consistent between its two uses, the first usage representing a subset of the second. Here we propose a visual notation to specify transformations. A relationship relates two or more patterns. Each pattern is a collection of objects, links and values. The structure of a pattern, as specified by objects and links between them, can be expressed using UML object diagrams. Hence, we propose to use object diagrams with some extensions to specify patterns within a relation specification. The notation is introduced through some examples followed by detailed syntax and semantics. Figure 7.8 specifies a relation, UML2Rel from UML classes and attributes to relational tables and columns. We introduce a new symbol to represent a transformation. The specifications “uml1:UML” and “r1:RDBMS” on each limb of the transformation specifies that this is a relationship between two typed candidate models “uml1” and “r1” with packages “UML” and “RDBMS” as their respective meta models. The “C” under each limb of the relation symbol specifies that both domains involved in this relation are chcckonly.

Figure 7.8 corresponds to the textual specification given below.

```
relation UML2Rel {
checkonly domain uml1 c:Class {name = n, attribute = a:Attribute{name = an}}
checkonly domain r1 t:Table {name = n, column = col:Column{name = an}}
}
```

Figure 7.8 - UML Class to Relational Table Relation

The *where* clause of a relation can be shown using a where box as shown in Figure 7.9, which specifies a relation, *PackageToSchema* that extends the above relation to specify that *UML2Rel* is to be applied on every class within a package.

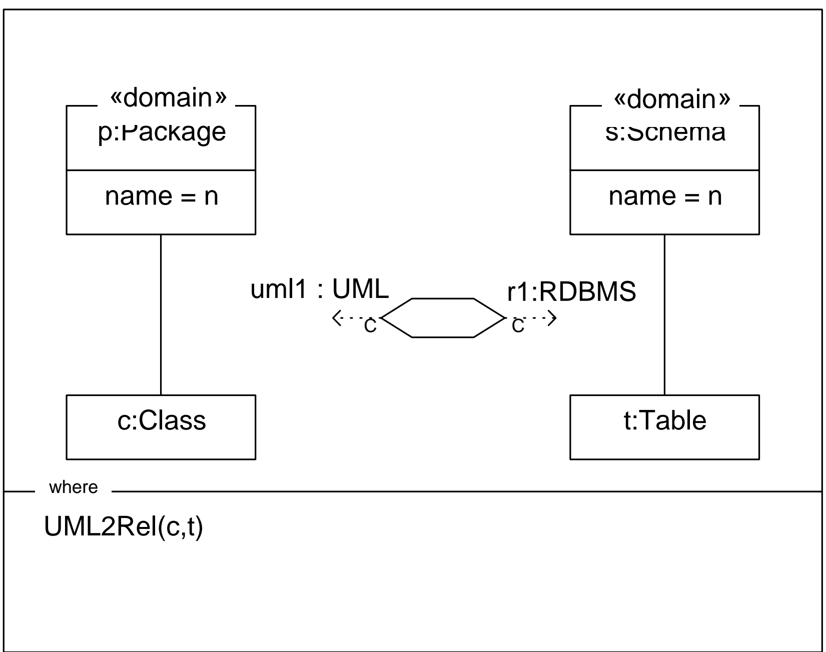


Figure 7.9 - Example showing the usage of the where clause

A similar box may be shown for the *when* clause of a relation.

An enforceable domain is shown by replacing the *C* within the relation symbol by an *E*. In the above example if the *RDBMS* side is to be made enforceable then it will be as shown in Figure 7.10.

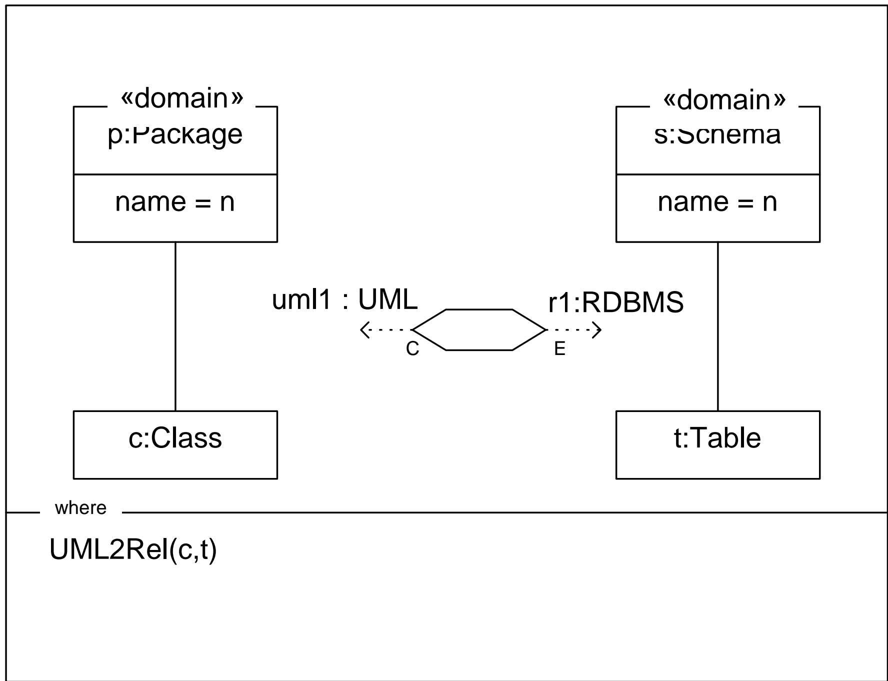


Figure 7.10 - RDBMS domain made enforceable

Constraints maybe attached to objects or a pattern. This is done as in UML by attaching a note. An example is shown in Figure 7.11. In the figure one constraint is attached to the *col* object and another one to the UML pattern.

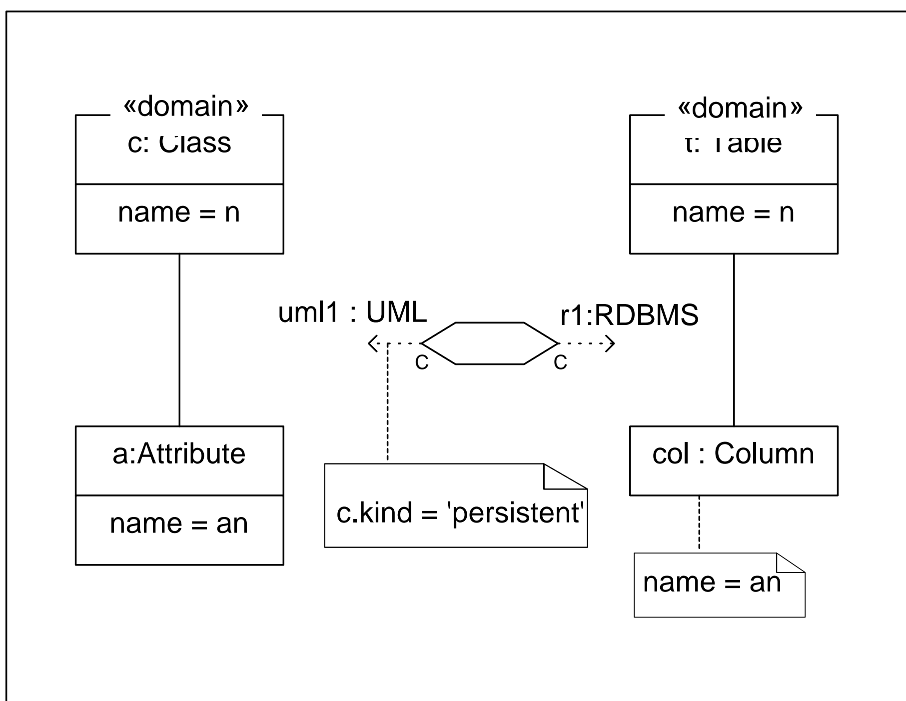


Figure 7.11 - UML2REL with constraints

In all the examples so far, the patterns comprised individual objects and links between them. The notation also supports specifications involving set of objects. Figure 7.12 shows an example where *Table* has a field totcols which is set to the number of attributes in the *Class.*

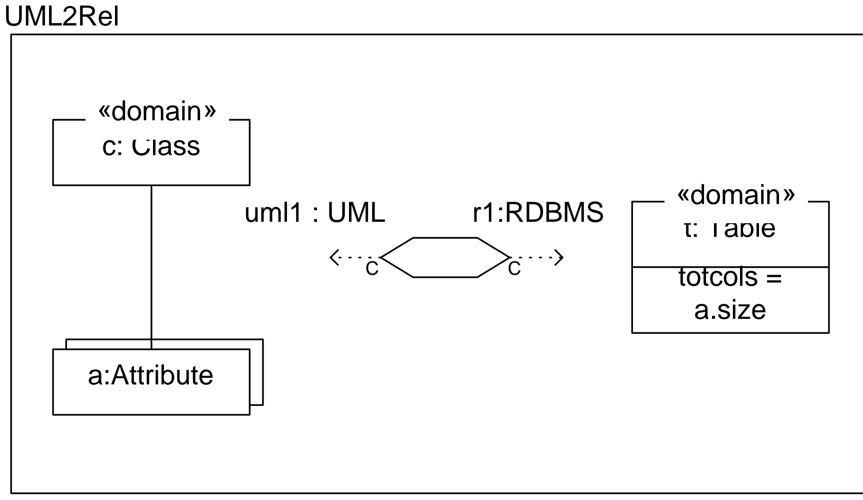


Figure 7.12 - Example using a Set

The notation also includes support for specifying the non-existence of objects and overriding of relations. Figure 16 specifies a strange relation from a class with no attributes to a table with *totcols = 0*. The *{not}* annotating Attribute indicates that this pattern matches only if there exists no Attribute linked to class c.

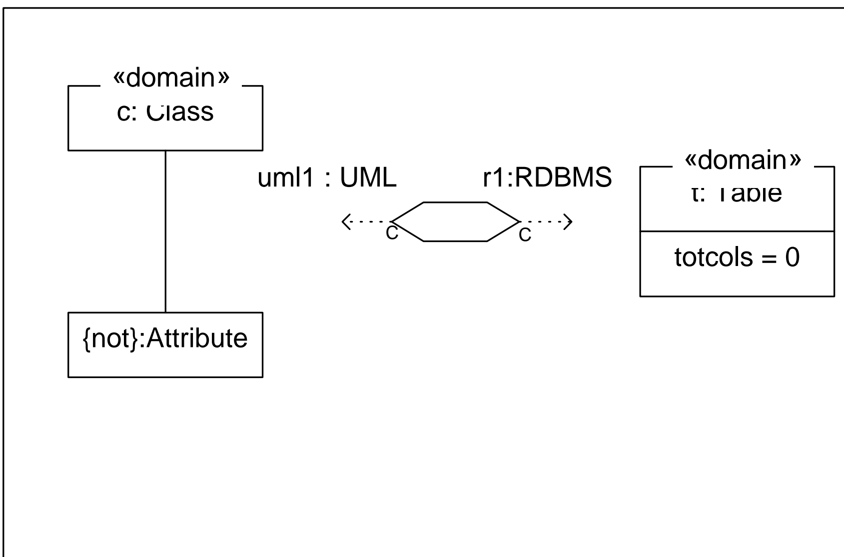


Figure 7.13 - Example using {not}

The textual specification corresponding to Figure 16 is as follows.

```
relation UML2Rel {
checkonly domain uml1 c:Class { attribute = Set(Attribute){}}{attribute->size() = 0}
checkonly domain r1 t:Table {totcols = 0 }
}
```

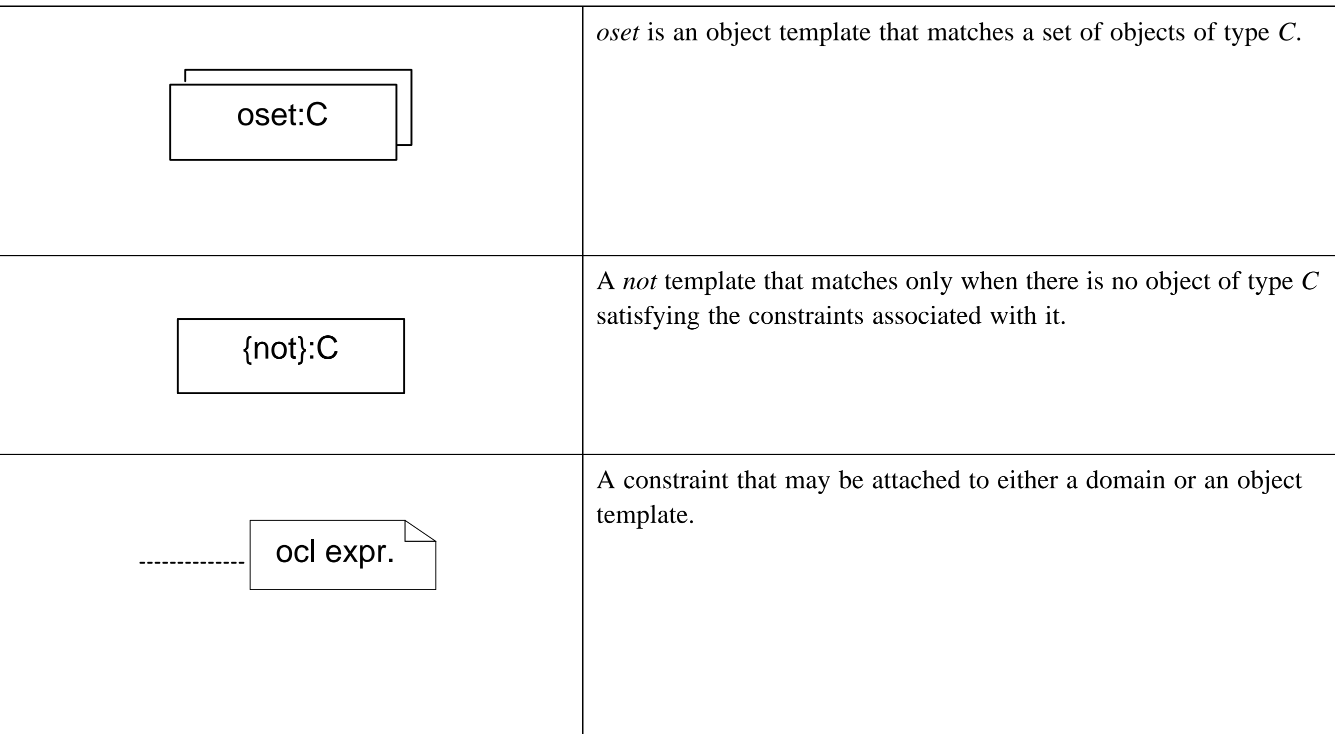
7.13.3.2 Graphical Notation Elements

Table 7.1 gives a brief description of the various visual notation elements.

Table 7.1 Diagrammatic Notations

Notation	Description
m1 : MM1 C/E ⟨relation⟩ C/E m2:MM2	A relation between models m1 having MM1 as meta-model and m2 having MM2 as meta-model. The label C/E indicates whether the domain in that direction is checkable or enforceable.
o:C	An object template having type *C* and referred to by the free variable *o*.
o:C / a = val	An object template having type *C* and a constraint that the property *a* should take the value *val*. *val* can be an arbitrary ocl expression.
«domain» o : C	The domain in a relation.

Table 7.1 Diagrammatic Notations



Notation	Description
oset:C	*oset* is an object template that matches a set of objects of type *C*.
{not}:C	A *not* template that matches only when there is no object of type *C* satisfying the constraints associated with it.
ocl expr.	A constraint that may be attached to either a domain or an object template.

7.13.3.3 Variations in Graphical Notation

In the above examples the positioning of certain elements in a diagram is only indicative and tools may choose to position these elements differently. The elements in which tools may choose to be different are.

- Name of the relation. The name of the relation may appear anywhere in the drawing page.
- *where* and *when* clause. These may appear as shown or maybe hidden and shown optionally or shown in a separate frame.
- model and meta-model names. These maybe shown optionally and when shown they maybe shown anywhere close to the relation symbol in the direction of the appropriate pattern.
- check/enforce letter. These need to be shown but may appear anywhere close to the relation symbol and in the appropriate direction.

8 Operational Mappings

The QVT Operational Mapping language allows either to define transformations using a complete imperative approach (operational transformations) or allows complementing relational transformations with imperative operations implementing the relations (hybrid approach). Section 8.1, 'Overview' provides an overview of the language, 8.2, 'Abstract Syntax and Semantics' defines its abstract syntax and its semantics, 8.3, 'Standard Library' introduces the standard library and 8.4, 'Concrete Syntax' defines the concrete syntax.

8.1 Overview

This overview presents informally salient features of the language.

8.1.1 Operational transformations

An operational transformation represents the definition of a unidirectional transformation that is expressed imperatively. It defines a signature indicating the models involved in the transformation and it defines an entry operation for its execution (named ***main***). Like a class, an operational transformation is an instantiable entity with properties and operations.

The example below shows the signature and the entry point of a transformation called *Uml2Rdbms* which transform UML class diagrams into RDBMS tables. See Appendix A for the metamodels and for a complete definition of this transformation.

```
transformation Uml2Rdbms(in uml:UML,out rdbms:RDBMS) {
   // the entry point for the execution of the transformation
   main() {
      uml.objectsOfType(Package)->map packageToSchema();
   }
….
}
```

The signature of this transformation declares that an rdbms model of type RDBMS will be produced from an *uml* model of type UML.

In the example, the **main** entry operation firstly retrieves the list of objects of type Package and then applies a mapping operation called *packageToSchema* on each Package of the list.

If the source text file only defines a unique transformation, the content of the transformation definition do not need to be placed within braces.

```
transformation Uml2Rdbms(in uml:UML,out rdbms:RDBMS);
// the entry point for the execution of the transformation
main() {
   uml.objectsOfType(Package)->map packageToSchema();
 }
```

8.1.2 Model types

UML and RDBMS symbols represent *model types*. A model type is defined by a metamodel (a set of MOF packages), a conformance kind (*strict* or *effective*) and an optional set of conditions. Model types introduce accurate flexibility for writing transformation definitions that are applicable to similar metamodels.

In our *Uml2Rdbms* example, the UML and RDBMS refer to implicit metamodels. When the textual syntax is used, the transformation writer is not obliged to indicate what MOF packages are used. However, in order to build the corresponding XMI serialization, it is mandatory to bind the model type symbol to an existing metamodel definition.

Alternatively to this approach, the transformation writer may explicitly indicate what metamodels are being used. This is done by an explicit declaration to be placed before the transformation signature:

```
modeltype UML uses SimpleUml("http://omg.qvt-examples.SimpleUml");
modeltype RDBMS "strict" uses SimpleRdbms;
```

The first declaration states that the UML model type is defined by the *SimpleUml* metamodel and assumes an *effective* conformance kind ñ which is the default. This implies that the XMI representation of the transformation definition has to be build using this *SimpleUml* metamodel definition. However, *effective* compliance allows passing models that are not necessarily "instances" of this metamodel. The input model can be checked to determine whether it is a valid input for this transformation by comparing its structure to the model elements defined by the *SimpleUml* metamodel.

When declaring explicitly a model type, an absolute URI may be given for the referred metamodel. In addition, the **where** keyword is used to define the additional constraints of the model type.

In the example below a new definition for the 'UML' model type is given: it imposes the model to contain at least a Class.

```
modeltype UML uses SimpleUml("http://omg.qvt-examples.SimpleUml")
  where { self.objectsOfType(Class)->size()>=1};
```

8.1.3 Libraries

A library contains definitions that can be reused by transformations. It may define specific types and may define operations - like query operations or metaclass constructors. A library is imported through one of the two available import facilities: *access* or *extension*. The QVT Standard library, named *Stdlib* is an example of a library. Since this library is pre-defined it does not need to be explicitly imported within transformation definitions.

The declaration below defines a library named *MyUmlFacilities* which defines a list of query operations on UML models.

```
library MyUmlFacilities(UML);
query UML::Class::getAllBaseClasses() : Set(Class);
query UML::Element::getUsedStereotypeNames() : Set(String);
```

The declaration below illustrates the usage of this library in the *UmlCleaning* transformation.

```
transformation UmlCleaning(inout umlmodel:UML14)
  extends MyUmlFacilities(UML14),
  access MathUtils;
var allSuper : Set(Class); // a global variable

main () {
   allSuper := umlmodel.objectsOfType(Class)->collect(i|i.getAllBaseClasses());
   // ….
}
```

The *UmlCleaning* transformation extends the *MyUmlFacilities* library, meaning by that that all operations defined in this library behave as if they were directly defined by the *UmlCleaning* transformation. The model type 'UML14' is a local symbol name. In our example it is bound to the 'UML' model type symbol name declared by the *UmlFacilities*. The example above also illustrates the possibility to import external libraries like the MathUtils which implementation could be given in another language than QVT.

8.1.4 Mapping operations

A mapping operation is an operation that implements a mapping between one or more source model elements into one or more target model elements.

A mapping operation is syntactically described by a signature, a guard (a *when* clause), a mapping body and a post-condition (a *where* clause). Even if it is not explicit notated in the concrete syntax, a *mapping operation* is always a refinement of a *relation* which is the owner of the *when* and *where* clauses.

The *packageToSchema* mapping operation defined below defines how a UML Package has to be transformed into a RDBMS Schema.

```
mapping Package::packageToSchema() : Schema
   when { self.name.startingWith() <> "_"}
{
   name := self.name;
   table := self.ownedElement->map class2table();
}
```

The implicit relation associated with this mapping operation has the following structure:

```
relation REL_PackageToSchema {
  checkonly domain:uml (self:Package)[]
  enforce domain:rdbms (result:Schema)[]
  when { self.name.startingWith() <> "_" }
}
```

The *packageToSchema* mapping operation behaves as any operation that would be defined on the UML Package metaclass. The *self* variable refers to the instance of the Package metaclass being passed to the operation. The **when** clause restricts the execution of the body: in our example, if a Package which name starts with underscore is passed the operation simply returns the *null* value. Otherwise, the body is executed.

The general syntax for the body of a mapping operation is:

```
mapping <dirkind0> X::mappingname
     (<dirkind1> p1:P1, <dirkind2> p2:P2)  : r1:R1, r2:R2
  when { … }
  where { … }
{
  init { … }
  population { … }
  end { … }
}
```

Where *<dirkindN>* refers to the *in/inout/out* direction kinds.

The **init** section contains some code to be executed before the instantiation of the declared outputs. The **population** section contains code to populate the result parameters and the **end** section contains additional code to be executed before exiting the operation.

Between the initialization and the population sections, there is an implicit section, named the instantiation section, which creates all output parameters that have a *null* value at the end of the initialisation section. This means that, in order to return an existing object rather than a creating a new one, one simply needs to assign the result parameter within the initialisation section.

In the *packageToSchema* mapping body the population **keyword** is omitted. The rules for interpreting the absence of the **population** keyword are given in Section 8.2.1.19 (Mapping Body: Notation).

8.1.5 Object creation and population in mapping operations

The QVT operational mappings language defines a specific "high-level" facility to create and/or update model elements: the *object expression* construct notated using the **object** keyword.

```
object s:Schema {
    name := self.name;
    table := self.ownedElement->map class2table();
}
```

In the example above, the object expression refers to an existing variable named 's' which necessarily has to be of type Schema. The semantics of this expression is as follows: if the 's' variable has a null value, a Schema instance is created and assigned to the 's' variable. After that the list of expressions of the body of the object expression are executed in sequence. The expression returns the value of the 's' variable. If the 's' value has a non null variable, no instantiation occurs, instead, the body is used to update the existing object.

Within an object expression, the properties of the target variable ('s' in our example) can be accessed directly. Thus the assignment expression "name := self.name" means "s.name := self.name".

The textual definition of the *package2Schema* mapping operation presented in Section 10.1.4 uses in fact an implicit object expression to populate the RDBMS Schema. The equivalent textual definition, without the shorthands, is:

```
mapping Package::packageToSchema() : result:Schema
   when { self.name.startingWith() <> "_"}
{
   population {
      object result:Schema {
         name := self.name;
         table := self.ownedElement->map class2table();
      }
   }
}
```

Since the *packageToSchema* mapping has no initialisation section, the *result* variable is initialized with a new created instance of Schema *before* the population section is entered. Hence, when the object expression is reached, the object expression operates on an existing object and the update semantics applies.

There is an important implicance of this instantiation policy. The *trace* associated with the execution of the mapping is created immediately after the instantiation and hence can be used within any inner mapping operation that is invoked within the population section. Usage of trace information is described in 8.1.9.

8.1.6 Inlining mapping operations

The definition below describes the *class2table* mapping operation that creates a RDBMS table from a persistent UML class. Two other mappings are used: *attr2Column* creates a column from an attribute and *class2Key* creates a Key instance attached to the "primary" attributes of the class.

```
mapping Class::class2table() : Table   when {self.isPersistent()}
{
  name := 't_' + self.name;
  column := self.attribute->map attr2Column());
  key := self.map class2key(result.column);
}
mapping Attribute::attr2Column() : Column {
   name:=self.name;type:=getSqlType(self.type);
}
mapping Class::class2Key(in cols:Sequence(Column)) : Key {
   name := 'k_'+ self.name; column := cols[kind='primary'];
}
```

Thanks to the *object expression* construct it is possible to rewrite these definitions using a unique mapping operation.

```
mapping Class::class2table() : Table  when {self.isPersistent()}
{
     name := 't_' + self.name;
     column := self.attribute->object(a) Column{
           name=a.name;
           type=getSqlType(a.type);)
     };
     key := object Key {
                name := 'k_'+ self.name;
                column := result.column[kind='primary'];
      };
}
```

The advantage of inlining mapping operations is conciseness. In the other hand, splitting of mappings may favour reusability. Also it is important to be aware that the execution of an object expression do not implies the creation of a trace.

As stated before, an object expression do not creates an instance if the target variable has a non null value. In the "object Key {…}" expression, since there is no variable reference, a Key instance is systematically created.

In the rewritten version of *class2table* an object expression is invoked on a list using the arrow symbol. As for the **map** mapping invocation operator, we are using here the "collect shorthand". The expression "self.attribute->object(a) Column {…}" is a shorthand for "self. attribute->xcollect(a| object Column {…})", where *xcollect* is an imperative variant of the OCL *collect* construct.

8.1.7 Using constructor operations

There is yet another way to express the creation of metaclass instances: we can define a *constructor* operation. A *constructor* is a specialized operation that create instances of a given type. For instance to create a UML::Operation, one may want to define a constructor that accepts a list of parameter names and create for each of them an inner UML::Parameter.

```
constructor UML::Operation::Operation(opname:String,Sequence(String));
```

This constructor operation may be used by any transformation dealing with UML models. It avoids writing various mapping for this simple purpose.

In the case of our Uml2Rdbms example, we can define constructors for Columns and for Keys:

```
constructor Column::Column (n:String,t: String)  { name:=n; type:=t; }
constructor Key::Key(n:String,primarycols:Sequence(Column))
    {name:=n;column:=primarycols;}
```

Now we have yet another variant of the *class2table* mapping operation.

```
mapping Class::class2table() : Table  when {self.isPersistent()}
{
  name := 't_' + self.name;
  column := self.attribute->new(a) Column(a.name,getSqlType(a.type))};
  key := new Key('k_'+self.name,t.column[kind='primary'];
}
```

As in Java language, the **new** keyword is used to instantiate an element. This implies looking for a matching constructor. We should note that this does not interleave with the implicit instantiation that potentially occurs within an object expression, since, in this case the constructor that is invoked is a built-in one.

All the previous examples show that there are many ways to structure a transformation definition. There is no general rule or guidance that can be provided to select the "best" solution since it really depends on a compromise between various criteria (such as reusability, readability, maintainability and conciseness).

8.1.8 Helpers

A *helper* is an operation that performs a computation on one or more source objects and provides a result. The body of a helper is an ordered list of expressions that is executed is sequence. A helper may have side-effects on the parameters ñ for instance a list may be passed and changed within the body and its effect is visible after the operation call termination. A query operation is a helper operation that has no side effects.

Helpers allow writing complex navigations operations in a comfortable way since the user is not restricted to write everything within a unique expression. The **return** is used to exit immediately from the operation.

The example below shows two queries that are written using a unique expression.

```
query Class::isPersistent() : Boolean = self.kind='persistent';
query Association::isPersistent() : Boolean =
      (self.source.kind='persistent' and self.destination.kind='persistent');
```

The example below shows a more sophisticated query defined using a block of expressions.

```
query Class::checkConsistency(typename:String) : Boolean {
   if (not typename) return false;
   if  (cl := self.namespace.lookForClass(typename) ) return false;
   return self.compareTypes(cl);
}
```

Now we have the definition of a helper with side-effects.

```
helper Package::computeCandidates(inout list:List) : List {
   if (self.nothingToAdd()) return list;
   list += self.retrieveCandidates();
   return list;
}
```

The *computeCandidates* helper operation uses a mutable List type (see type extensions section).

Query operations may be defined also on primitive types, like strings.

```
query String::addUnderscores() : String = "_".concat(self).concat("_");
```

8.1.9 Intermediate data

An operational transformation may use for its definition *intermediate classes* and *intermediate properties*.

Below an example of usage of intermediate data in a re-designed *Uml2Rdbms* transformation that treats appropriately non primitive attributes. Instead of creating a column per attribute we intend now to create as many columns as there are primitive slots in a complex data type. One approach to solve this recursive 1-to-N transformation problem is to use intermediate data.

```
intermediate class LeafAttribute {name:String;kind:String;attr:Attribute;};
intermediate property Class::leafAttributes : Sequence(LeafAttribute);
```

In the declarations above, the *LeafAttribute* class declares the structure to represent the flattened primitive attributes. The *leafAttributes* intermediate property allows a class instance to store all created intermediate objects derived from its definition. The new definition of the class2table mapping is:

```
mapping Class::class2table() : Table
   when {self.isPersistent() ;}  {
   init {
     self.leafAttributes := self.attribute->map attr2LeafAttrs();
    }
   name := 't_' + self.name;
   column := self.leafAttributes->map leafAttr2OrdinaryColumn();
   key := object Key {
        name := 'k_'+ self.name;
        column := result.column[kind='primary'];
    };
}
```

The leaf attributes are created in the initialization section through the invocation of a recursive mapping operation named *attr2LeafAttrs* (not detailed here).Then the iteration over this list is used to create the columns.

The important point is that intermediate properties are manipulated as ordinary properties of the *context* metaclass (UML::Class in our example). Intermediate properties are property extensions that do not exist outside the scope of transformation that defines it. This is similar to the query and mapping operations which conceptually are operation extensions. The addUnderscores query definition above illustrates the fact that it is even possible to add operations to the primitive types.

8.1.10 Updating objects and resolving object references

A common technique in model transformation is the usage of various passes to solve cross referencing between model elements. The language provides *resolution* constructs to access target objects created previously from source objects. These facilities implicitly use the trace records created by the execution of a mapping operation.

The *asso2table* definition below is responsible for adding a foreign key in a previously created RDBMS table. This requires retrieving the existing table.

```
mapping Association::asso2table() : Table
when {self.isPersistent()}
{ -- result is the default name for the output parameter of the rule
  init { result := self.destination.resolveone(#Table); }
  foreignKey := self.map asso2ForeignKey();
  column := result.foreignKey.column;
}
```

The **resolveone** construct inspects the trace data to see whether there is a Table instance created from an association destination class that satisfies a boolean condition. In our case the condition is to be a kind of "Table" (the notation '#Table' is a shorthand for isKindOf(Table)).

There are three orthogonal variants of this resolution construct. **invresolve** performs the reverse treatment, that is, looks for the objects that were responsible for generating the object passed as the context argument to the invresolve operator. **resolveIn** looks for target objects created from a source object by a unique mapping operation. The example below illustrates its usage: given a list of java classes (*JClass* instances) that have a *packageName* field indicating the name of the owning *Package*, the *JClass2JPackage* transformation creates a Java Package (*JPackage*) for each package name found in the list of java classes

```
transformation JClass2JPackage(inout javamodel:JAVA);
main () { javamodel->objectsOfType(JClass)->jclass2jpackage();}
mapping Class::jclass2jpackage() : JPackage () {
    init {
      result := resolveIn(jclass2jpackage,true)
            ->select(p|self.package=p.name)->first();
      if result then return;
    }
    name := self.package;
}
```

In the example above, *return* is used to avoid creating more than two packages having the same name.

The final variant of the resolution operator is the ability to postpone the retrieval of target objects until the end of the transformation (that is to say, at the end of the execution of the entry operation). Deferred resolutions may be useful to avoid defining various passes for a transformation, as in the example below that treats potential cyclic due to class inheritance dependencies.

```
transformation Uml2Java(in uml:UML,out java:JAVA)
main() : JAVA::Interface {
   uml->objectsOfType(Class)->map transformClass();
}
mapping UML::transformClass() : JAVA::Interface () {
  name := "Ifce".concat(self.name);
  base := self.superClass->late resolve(#JAVA::Interface);
}
```

We provide below a solution for this transformation problem that do not uses the **late** resolution. This solution uses two passes.

```
transformation Uml2Java(in uml:UML,out java:JAVA)
main() : JAVA::Interface {
   uml->objectsOfType(Class)->map transformClass();
   uml->objectsOfType(Class)->map transformClassInheritance();
}
mapping UML::transformClass() : JAVA::Interface {
  name := "Ifce".concat(self.name);
}
mapping UML::transformClassInheritance() : JAVA:Interface {
  base := self.superClass->resolveIn(transformClass,#JAVA::Interface);

}
```

In terms of execution, the '**late resolve**' expression is always related to an assignment. Conceptually it returns *null* but in the meantime it stores all the information that is required to re-execute later the trace inspection and the assignment.

8.1.11 Composing transformations

Composition of coarse-grained transformation is an essential feature in large "real" transformations. The language allows to instantiate and to invoke explicitly transformations.

Let's imagine that the *Uml2Rdbms* transformation requires that the source *uml* model is a "clean" model with all redundant associations removed. We will need to extend the previous transformation definition by invoking an in-place "cleaning facility" on the UML model prior to apply the *Uml2Rdbms* transformation. This can be achieved by the following definition.

```
transformation CompleteUml2Rdbms(in uml:UML,out rdbms:RDBMS)
  access transformation UmlCleaning(inout UML),
  extends transformation Uml2Rdbms(in UML,out RDBMS);

main() {
   var tmp: UML = uml.copy();
   var retcode := (new UmlCleaning(tmp))->transform(); // performs the "cleaning"
   if  (not retcode.failed())
       uml.objectsOfType(Package)->map packageToSchema()
   else raise "UmlModelTransformationFailed";
}
```

In this example we see the usage of *access* and *extension* reuse mechanisms. An access import behaves as a traditional package import, whereas extension semantics combines package import and class inheritance paradigm.

This example also illustrates the following: (i) the ability to execute *in place* transformations, like *UmlCleaning*, (ii) the ability to perform an explicit transformation instantiation (through the **new** operator) and (iii) the ability to invoke high level operations on models, like the cloning facility (*copy* operation).

8.1.12 Reuse facilities for mapping operations

The language provides two reuse facilities at the level of the mapping operations: mapping *inheritance* and mapping *merge*.

A mapping operation may *inherit* from another mapping operation. In terms of execution semantics the inherited mapping is executed *after* the initialisation section of the inheriting mapping. The example below illustrates the usage of mapping inheritance. The mapping that creates foreign RDBMS Columns reuses the mapping defined to create "ordinary" Columns.

```
mapping Attribute::attr2Column (in prefix:String) : Column {
  name := prefix+self.name;
  kind := self.kind;
  type := if self.attr.type.name='int' then 'NUMBER' else 'VARCHAR' endif;
}
mapping Attribute::attr2ForeignColumn (in prefix:String) : Column
  inherits leafAttr2OrdinaryColumn {
    kind := "foreign";
}
```

Within a transformation, a mapping operation may also declare a list of mapping operations that complements its execution: this is mapping merge. In terms of execution, the ordered list of merged mappings is executed in sequence after the **end** section. The ordinary compliance rules between a caller and the callee apply here constraining the parameters of the complementary mapping to conform to the mapping serving as the base for merging.

The example below shows an example of a transformation definition that uses mapping merging. This style of writing allows defining a modular specification where the transformation writer can try to define almost a mapping operation for each rule defined in natural language.

```
// Rule 1 (in english): A Foo should be transformed into an Atom and a Bar.
// The name of the Bar is upperized and the name of the Bar is lowerized
mapping Foo::foo2atombar () : atom:Atom, bar:Bar
    merges foo2barPersistence, foo2atomFactory
{
    object atom:{name := "A_"+self.name.upper();}
    object bar:{ name := "B_"+self.name.lower();}
}
// Rule 2: Persistent attributes of Foo are treated as volatile Bar properties
mapping Foo::foo2barPersistence () : atom:Atom, bar:Bar
   when {foo.isPersistent();} {
   object bar:{ property := self.attribute->map persistent2volatile(); }
}
// Rule 3: An Atom factory should be created for each Atom and have the name
// of the associated Bar.
mapping Foo::foo2atomFactory () : atom:Atom, bar:Bar {
   object bar:{ factory := object Factory {name := bar.name}};
}
```

A merged mapping is not invoked if the guard is not satisfied. This occurs in Rule 2 for Foo instance that are not persistent.

The code below shows an invocation of the *foo2atombar* mapping. The resulting tuple is unpacked and assigned to the 'atom' and 'bar' variables.

```
var f := lookForAFooInstance();
var (atom: Atom, bar: Bar) := f.foo2atombar();
```

We should note that conceptually a result parameter is treated as an *optional parameter* of the mapping operation. Thus, the first call of "*f.foo2atombar()*" is equivalent to invoke "*f.foo2atombar(null,null)*". In contrast, *foo2barPersistence* and *foo2atomFactory* are internally invoked with the atom and bar instances created by the *foo2atombar* mapping. This mechanism allows the merged mappings to update the instances created by the merging mapping.

8.1.13 Disjunction of mapping operations

A mapping operation may be defined as a disjunction of an ordered list of mappings. This means that an invocation of the operation results on the selection of the first mapping whose guard (type and when clause) succeeds. The *null* value is returned if no guard succeeds.

Below an example:

```
mapping UML::Feature::convertFeature () : JAVA::Element
  disjuncts convertAttribute, convertOperation, convertConstructor() {}
mapping UML::Attribute::convertAttribute : JAVA::Field {
   name := self.name;
}
mapping UML::Operation::convertConstructor : JAVA::Constructor {
  when {self.name = self.namespace.name;}
  name := self.name;
}
mapping UML::Operation::convertOperation : JAVA::Constructor {
  when {self.name <> self.namespace.name;}
  name := self.name;
}
```

8.1.14 Type extensions

The language extends the OCL and MOF type system with some general purpose data types. These are *mutable lists*, *dictionary types* and *anonymous tuples*.

A mutable list (List) contains an ordered list of elements. In contrast with an OCL collection, a List can be updated. List is a parameterized type. When no type for the elements is given, the type Any is assumed.

```
var mylist := List{1,2,3,4};  // a list literal
mylist.add(5);
```

A dictionary (Dict) is a facility to store values accessed by keys ñ also know as an hash table. It is a mutable type.

```
var mydict := Dict{"one"=1,"two"=2,"three"=3};  // a dictionary literal
mydict.put("four",1);
```

An anonymous tuple is a tuple which slots are unnamed. It can be manipulated in a more flexible way than tuples with named slots since it can be packed and unpacked more directly.

var mytuple := **Tuple**{1,2,3}; // a anonymous tuple literal
var (x,y,z) := mytuple; // unpacking the tuple into three variables

Another mechanism is defined to make the type system more flexible. The *typedef* mechanism allows attaching additional constraints to an existing type within a specific context. It also serves to define aliases to complex data types. When used in the signature of a mapping operation, the typedef constraints are added to the *guard* of the mapping operation. The condition is expressed within brackets after the type taken as reference.

typedef TopLevelPackage = Package [_parentInstance()=null];
typedef AttributeOrOperation = Any [#Attribute or #Operation];
typedef Activity = ActionState[stereotypedBy("Activity")];

The type defined by a typedef in considered to be in the scope of the model type of the type taken as reference. For instance, if *Package* exists in the context of a UML modeltype, the *TopLevelPackage* is also considered being defined in the context of the UML modeltype.

A typedef can also be used to simply define an alias to a complex type, as for tuple types:

typedef PersonInfo = *Tuple*{name:String,phone:String};

8.1.15 Imperative expressions

The QVT operational mapping language is an imperative language to define transformations. It extends OCL but includes all the necessary machinery that is needed to write in a comfortable way complex transformations. The imperative expressions in QVT realize a compromise between some nice functional features found in OCL and the more traditional constructs that we found in general purpose languages like Java.

The most relevant example of this marriage is the ability to use block expressions to compute a given value. The construct:

compute (v:T := initexp) body;

returns the value of the v variable after ending the execution of the body. The body may be a "block" expression where variables defined in outer scope can be freely accessed and changed (a block expression is not a function!).

compute (v:T := initexp) { … self.getSomething() …};

The construct can be combined with a while expression.

self.myprop **:= while(**v:T = initexp; v<>null) { … self.getSomething() …}
// "while(v;cond) body" is a shorthand for *"compute(v) while(cond) body"*

The **forEach** expression is an imperative loop that can also iterate over a block expression and make an optional filter on the elements of the list. It plays the role of a for loop in Java.

self.ownedElement->**forEach**(i|i.isKindOf(Actor)) { … }
range(2,8)->**forEach**(i) { … }

Within imperative loops, **break** and **continue** constructs are available. The combination of **compute** and **forEach** allows defining the imperative x*collect* and x*select* operations and various others high-level facilities.

The language also defines an imperative "if-then-else" construct that is not constrained as the corresponding OCL construct, since block expressions can be executed and else parts are not mandatory. The notation for this construct is Java-like:

var x:= **if** (self.name.startsWith("_")) 0
elif (self.type.isPrimitive()) **while** (res := 0; res<10) { … };

```
        else -1;
```

In general, there is no obligation to use this control expression within expressions. They can be used alone as in traditional programming.

```
if (x==0) {
   list->forEach(i) {
      if (…) continue;
      …
   }
}
```

8.1.16 Pre-defined variables: this, self and result

This represents the transformation instance being defined. It can be used implicitly to refer to the properties of the transformation class or to the operations owned by him.

Within a contextual operation (a query helper operation or a mapping operation) *self* represents the contextual parameter.

Within a contextual operation *result* represents the unique result parameter - if there is a unique declared result - or the tuple containing the list of declared result parameters.

8.1.17 Null

The literal *null* is a specific literal value that complies to any type. It can be explicitly returned by an operation or can be implicitly returned to mean the absence of value.

8.1.18 Advanced features: dynamic definition and parallelism

When dealing with a complex MDA process, it may be useful to be able to define transformations that use transformations definitions computed automatically. It is indeed possible for a transformation definition to be the output of a transformation definition ñ since a QVT model can be represented as a model. The language provides a pre-defined *'asTransformation'* operation that allows to *cast* a transformation definition (a model typed through a ModelType that accepts QVT compliant definitions) as an instance of the corresponding transformation class. The implementation of this *'asTransformation'* operation typically requires "compiling" the transformation definition on the fly.

The example below illustrates a possible usage of this facility. This *Pim2Psm* transformation transforms a PIM model into a PSM model. To this end, the initial PIM model is first of all automatically annotated using an ordered set of UML Packages that defines the transformation rules to infer the annotations, on the basis of an arbitrary proprietary UML-oriented formalism. Each UML Package defining a transformation is transformed into the corresponding QVT compliant specification. When executed in sequence, each resulting QVT transformation definition adds its own set of annotations to the PIM model. At the end, the annotated PIM is converted into a PSM model using the AnnotatedPim2Psm transformation.

```
transformation PimToPsm(inout pim:PIM, in transfSpec:UML, out psm:PSM)
  access UmlGraphicToQvt(in uml:UML, out qvt:QVT)
  access AnnotatedPimToPsm(in pim:PIM, out psm:PSM);

main() {
  transfSpec->objectsOfType(Package)->forEach(umlSpec:UML) {
     var qvtSpec : QVT;
```

```
        var retcode := new UmlGraphicToQvt(umlSpec,qvtSpec).transform();

        if (retcode.failed()) {
           log("Generation of the QVT definition has failed",umlSpec); return;};

        if (var transf := qvtSpec.asTransformation()) {
           log("Instanciation of the QVT definition has failed",umlSpec); return;}

        if ( transf.transform(pimModel,psmModel).failed()) {
          log("failed transformation for package spec:",umlSpec); return;}
   }
}
```

Another advanced feature is parallel launching of various transformations. These are useful when there are no sequencing constraints between a set of coarse-grained transformations. In terms of execution, invoking a transformation simply behaves as forking a process to accomplish the task. Synchronization is done by waiting on the variables returned by the execution of the transformation.

The example below is a "magical" requirement to psm transformation which decomposes a requirement model into two intermediate pim models (one for the GUI, another for the behavior) and then merges the two pim model into the executable psm model. The two PIM models are generated in parallel.

```
transformation Req2Psm (inout pim:REQ, out psm:PSM)
  access Req2Pimgui(in req:REQ, out pimGui:PIM)
  access Req2Pimbehavior(in req:REQ, out pimBehavior:PIM),
  access Pim2Psm(in pimGui:PIM, in pimBehavior:PIM, out psm:PSM);

main() {
   var pimGui : PIM := PIM::createEmptyModel();
   var pimBehavior : PIM  := PIM::createEmptyModel();
   var tr1 := new Req2Pimgui(req, pimGui);
   var tr2 := new Req2Pimbehavior(req, pimBehavior);
   var st1 := tr1.parallelTransform(); // forks the PIM GUI transformation
   var st2 := tr2.parallelTransform(); // forks the PIM Behavior transformation
   this.wait(Set{st1,st2}); // waits patiently
   if (st1.succeeded() and st2.succeeded())
      // creates the executable model
      new  Pim2Psm(pimGui,pimBehavior,psm).transform();
}
```

8.2 Abstract Syntax and Semantics

In this section we define the abstract syntax of the *QVT operational* formalism. Concepts are depicted firstly graphically through class diagrams, and then a description of each class is given. When applicable, additional sub-sections are included to explain in more detail the semantics of a given concept.

The QVT operational formalism is defined by two EMOF packages: `QVTOperational` and `ImperativeOCL`. The following packages are imported: `EMOF`, `EssentialOCL`, `QVTBase`, `QVTTemplate` and `QVTRelation`.

Conventions

The metaclasses imported from other packages are shaded and annotated with 'from <package-name>' indicating the original package where they are defined. The classes defined specifically by the two packages of the QVT Operational formalism are not shaded.

Within the class descriptions, metaclasses and meta-properties of the metamodel are rendered in courier font (for instance `MappingOperation`). Courier font is also used to refer to identifiers used in the examples. **Keywords** are written in bold face. *Italics* are freely used to emphasize certain words, such as specific concepts, it helps understanding. However that emphasis is not systematically repeated in all occurrences of the chosen word.

In general, inherited properties (attributes or association ends) are not repeated in class descriptions except when this is useful to understand its specific usage in the context of operational transformations. A slash prefix "/" is used to mark inherited properties in class descriptions. In addition the origin of the inherited property is indicated though a "from <class-name>" annotation.

8.2.1 The QVTOperational Package.

The QVTOperational package defines the concepts that are needed to specify transformations definitions written imperatively. It defines general structuring concepts (like *module* and *imperative operation*) and specialized ones (like *operational transformations*, *mapping operations* and so on). It uses the ImperativeOCL package which extends OCL with constructs that are common in programming languages.

For convenience, the metaclasses defined by this package grouped into two categories: concepts related to the definition of operational transformations and concepts related to the definition of operations within the transformation.

Concepts related to the definition of operational transformations

This group has seven classes: `OperationalTransformation`, `Library`, `Module`, `ModuleImport`, `ModelType`, `ModelParameter` and `VarParameter` and two enumerations: `ImportKind` and `DirectionKind`. The `EntryOperation` is described in the second group of classes.

Figure 8.1depicts the definition of all the metaclasses of this group. They are all closely related to the definition of an operational transformation.

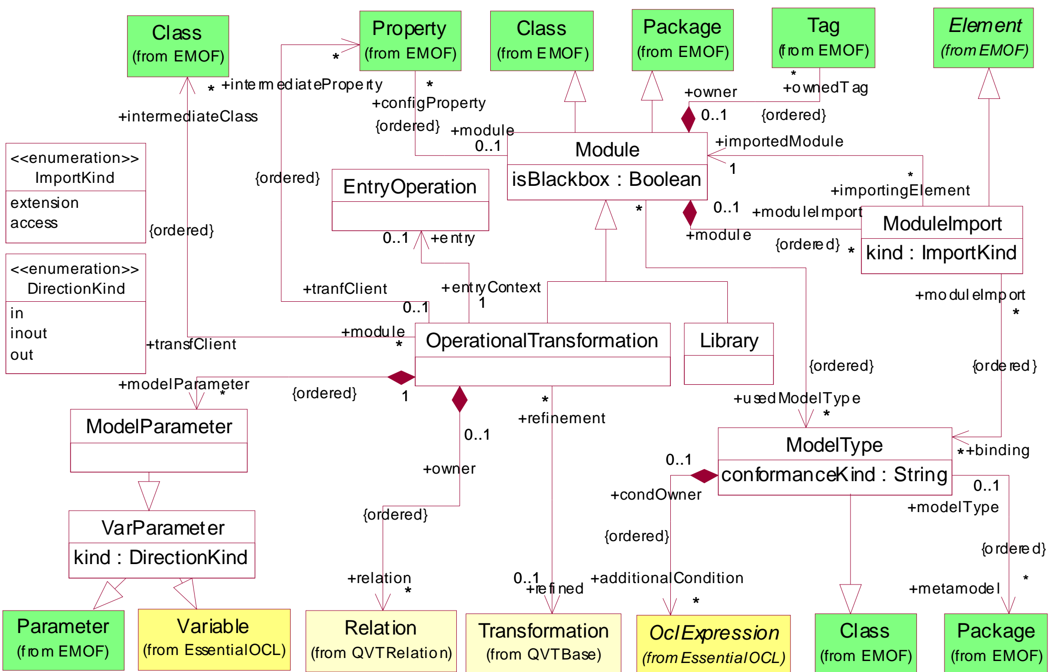


Figure 8.1 - QVT Operational Package - Operational Transformations

8.2.1.1 OperationalTransformation

An *operational transformation* represents the definition of a unidirectional transformation that is expressed imperatively. It has a model signature indicating the models involved in the transformation and it defines an entry operation, named **main**, which represents the initial code to be executed to perform the transformation. An operational transformation requires a model signature, but it does not require an implementation. This allows for black-box implementations defined outside QVT.

An operational transformation may *extend* or *access* an existing operational transformation or an existing library (see `ModuleImport` and `Library` classes).

An operational transformation may define configuration properties, that is properties which actual value is to be given at execution time. In addition it may define intermediate data properties and define explicitly new classes to store intermediate data.

When an operational transformation refines a *relational transformation* (Chapter 7: Relations), the enforced direction must be indicated (see the `refined` and `enforcedDirection` associations).

Syntactically, an `OperationalTransformation` is a subclass of `Module` (see: `ImperativeOCL` package), hence it is, by inheritance, both a `Class` and a `Package`. As a `Class` it can define properties and operations ñ like helper queries, mapping operations, and constructors and has to be instantiated to be executed (see `TransformationInstanceExp`). As a `Package` it can define and contain specific types for usage within in the transformation definition.

Superclasses

`Module`

Attributes

`/isBlackbox : Boolean (from Module)`

indicates that the whole transformation is opaque : no entry operation and no mapping operations are defined. It is typically used to reuse coarse-grained transformations defined or implemented externally.

`/isAbstract : Boolean (from Class)`

indicates that the transformation serves for the definition of other transformations. No entry operation should be defined.

Associations

`enforcedDirection : ModelParameter [0..1]`

indicates the enforced direction of a refined relational transformation. In effect, since a relational transformation is potentially multi-directional, an operational transformation ñ which is unidirectional - needs to indicate the implemented direction.

`entry : EntryOperation [0..1]`

An operation acting as the entry point for the execution of the operational transformation. It is optional since an operational transformation may serve as a base for another operational transformation.

`modelParameter: ModelParameter [*] {composes, ordered}`

The signature of this operational transformation. A model parameter indicates a direction kind (in/out/inout) and is typed by a model type (see ModelParameter class description).

`intermediateClass : Class [*] {ordered}`

the classes that are defined explicitly by the transformation writer to contain structured intermediate data used for the purpose of the transformation. These intermediate classes are to be distinguished from the trace classes that are implicitly and automatically derived from the *relations*. Instances of intermediate classes do not survive the execution of the transformation, except for ensuring trace persistence.

`intermediateProperty: Property[*]{ordered}`

refers to the properties defined to store the intermediate data used by the transformation. These properties are typically contextual properties (see `ContextualProperty` metaclass description), that is, properties that are owned by the transformation class but which are conceptually extensions of the metaclasses involved in the transformation.

Intermediate properties do not survive to the execution of the transformation ñ except for ensuring trace persistence.

`refined : Transformation [0..1]`

indicates a relational transformation (see Chapter 9: Relations) being refined by this operational transformation.

`relation : Relation [0..*] {composes, ordered}`

The ordered set of relation definitions (see Chapter 9: Relation) that are associated with the mapping operations (see `MappingOperation`) of this operational transformation.

Constraints

The `refined` and `enforcedDirection` links should be used together.

self.refined<>OclUndefined implies self.enforcedDirection<>OclUndefined
and self.enforcedDirection<>OclUndefined implies self.refined<>OclUndefined

Class inheritance should not be used for `OperationalTransformations`.

self.superClass->isEmpty()

Semantics

For convenience, the execution semantics of an operational transformation is described through comparisons with the Java language.

An operational transformation is like a Java class where all the declared properties and operations of the transformation are effectively attribute slots and methods of the class. In addition, for each model parameter there is an attribute slot. An implicit constructor also exists for the transformation: its signature corresponds to the list of the **in** and **inout** model parameters.

The code of this implicit constructor performs the following actions:

i) for each declared **out** parameter a MOF extent is created with empty contents.

ii) the attribute of the slots associated with each model parameter is assigned.

iii) loading the values of the configuration properties ñ if available.

The instantiation of the transformation is either implicit (the instance being referred though the predefined `this` variable) or explicit. In the latter case an `InstantiationExp` expression is used.

The population of configuration properties may be done using any external mechanism ñ for instance using an external configuration file. This aspect is out of the scope of this specification.

As in java, instantiation and execution of the main operation are separated things. A transformation is explicitly invoked using the `transform` pre-defined operation (see the Standard Library). An invocation of the `transform` operation provokes the execution in sequence of the list of expressions within the body of the entry operation. At the end of the execution of **main**, the deferred assignments ñ if any - are executed in sequence (see `ResolveExp`). This terminates the execution of the transformation.

For each declared model type there is, conceptually, a corresponding import of the java package representing the metamodel bound to the model type.

Import with `access` semantics is like an import of another Java class. In contrast import with extension semantics corresponds firstly to an import and secondly to class inheritance. The imported names are included in the name space of the transformation.

Notation

The notation to define a transformation uses the **transformation** keyword. The notation contains a header prefixed with the **transformation** keyword. The content of the transformation definition may be within brackets or after the header declaration. The former should be used to include various transformation definitions within a single text file.

```
// defining multiple transformations in a single file
transformation Uml2Rdbms(in uml:UML, out rdbms:RDBMS) {
    // content of the transformation definition
}
transformation Logical2PhysicalRdbms(inout rdbms:RDBMS) {
    // content of the transformation definition
}
```

When the later option is used the transformation should be the first declaration - except for comments.

The declaration below defines an operational transformation named `Uml2Rdbms` with a signature made of two model parameters `uml` and `rdbms` typed by the `UML` and `RDBMS` model types. It also defines a tag to indicate the transformer writer.

transformation Uml2Rdbms(**in** uml:UML, **out** rdbms:RDBMS);
tag "author" Uml2Rdbms = "Pepe"**;**

The imported modules (transformations or libraries) are indicated in the same statement after the model parameters using the **extends** or **access** keywords. The keywords **transformation** or **library** may be used to remind on the kind of module being imported.

transformation Uml2Rdbms(**in** uml:UML, **out** rdbms:RDBMS)

extends BasicUml2Rdbms, -- extending a transformation
extends library UMLUtilities(UML) -- extending a library
access library MathLibrary; -- accessing a math library

All access declarations may appear in a separate statement (not necessarily in the header).

An operational transformation indicates a refined relational transformation using the refined keyword.

transformation Uml2Rdbms(**in** uml:UML, **out** rdbms:RDBMS) **refines** R_UML2RDBMS;

Intermediate classes and intermediate properties of the transformation are notated using the **property** and **class** keywords prefixed by the **intermediate** keyword.

intermediate class LeafAttribute { … }
intermediate property UML::Attribute::extravalue : String;

Properties that are configuration properties are declared using the **configuration** qualifier keyword.

configuration property UML::Attribute::MAX_SIZE: String;

8.2.1.2 Library

A *library* is a module grouping a set of operations and type definitions that are put together for reuse. The QVT Standard Library is an example of a Library instance. All but the QVT Standard Library must be explicitly imported. A library may be declared as a black-box: no implementation is provided for the owned operations.

A library may declare an ordered list of model types on which it operates. This list is the signature of the library.

Syntactically, a `Library` is a subclass of `Module` and hence a subclass of `Class` and a subclass of `Package`.

Superclasses

`Module`

Semantics

For convenience, the execution semantics for a library is described through comparisons with the Java language.

A library is like a Java class where all the declared properties and declared operations of the library are effectively attribute slots and methods of the class. However, in contrast with operational transformations there is no main operation to execute and the instantiation of the class is always implicit. When a library is imported with `access` semantics, an implicit instance is created and used to access the properties and operations of the library. When a library is imported with `extension` semantics, normal class inheritance applies as for operational transformations.

For each declared model types there is conceptually a corresponding import of the java package that represents the metamodel that is bound to the model type.

Notation

The notation to define a library is similar to the notation to define transformations except that the **library** keyword is used instead of the **transformation** keyword.

The signature of the library is in this case a list of model types ñ in contrast with transformations, where the signature is a list of model parameters.

The declaration below defines a library named `UmlUtilities` extending BasicUmlUtilities with a signature made of a model type named `UML1_4`.

> **library** UmlUtilities(UML1_4)
> **extends** BasicUmlUtilities(UML1_4)
> **access** MathLibrary ;

8.2.1.3 Module

A *module* is a unit containing a set of operations and types defined to operate on models. This concept defines common features shared by operational transformations and libraries.

Syntactically, a `Module` is a subclass of a `Class` and a `Package`. As a `Class` it can define properties and operations. As a `Package` it can define and contain specific types for usage within in the module definition.

Superclasses

`Class`

`Package`

Attributes

`isBlackbox : Boolean`

indicates that the whole module is opaque : no operations are defined. It is typically used to reuse coarse-grained transformations defined or implemented externally.

`/uri : String (from Package)`

indicates an identity for the module. This is used to refer to an existing module without including its full content. This should be used in conjunction with the pre-defined tag `proxy` (see the predefined tags section in the standard library).

Associations

`moduleImport : ModuleImport [0..*] {composes, ordered}`

The list of module import elements. Each module import refers to an imported module and indicates an import kind (access or extension semantics).

`usedModelType : ModelType [0..*] {ordered}`

The list of modules being used. The semantics of module import depends on the import kind (access or extension semantics).

`configProperty : Property[0..*] {ordered}`

The list properties which value may be left undefined and assigned at execution time. Example: a MAX_SIZE property.

`ownedTag : Tag [0..*] {composes, ordered}`

All the tags that are defined within this module. Note that, in most cases, the marked element is not the owner of

the tag.

`/nestedPackage : Package [0..1] (from Package)`

package nesting is a general property of MOF packages and then applicable to modules (which are specializations of `Packages`). In this context it can be used to contain metamodel definitions referred by model types (see `ModelType` description).

`/ownedAttribute:Property[0..*](from Class){composes,ordered}`

Any property owned by this module.

`/ownedOperation:Operation[0..*](from Class){composes,ordered}`

Any operation owned by this module.

`/ownedType : Type [0..*] (from Package) {composes,ordered}`

All the types being defined by this module. Specifically this includes the model types, locally defined classes, and any composite type used to define the type of a variable or a parameter ñ for instance a 'Set(MyMetaclass)' user-defined datatype.

8.2.1.4 ModuleImport

A *module import* represents the usage of a module through one of the two import semantics. With `extension` semantics, importing the module is identical to defining the imported operations and types of the imported module in the importing module. Hence an operation or a property of the imported module is considered as being an operation or a property of the importer ñ similarly to class extension semantics. With `access` semantics, the definitions of the imported module are not inherited by the importing module. Hence an *instance* of the imported module has to be used in order to access them. In the case of accessing a *library,* an instance of the imported module is implicitly available. In the case of a non library module, the instance of the accessed module is to be created explicitly (see `InstanciationExp`).

Two identical names coming from two distinct modules can be distinguished by qualifying them. However, a symbol defined locally has always precedence in respect to the imported symbols.

Superclasses

`Element`

Attributes

`kind: ImportKind`

The semantics of the library import. Possible values are `access` and `extension`.

Associations

`binding: ModelType [0..*] {ordered}`

The model types being "passed" to the imported module. The binding is done according to the ordering of the original list of model types declared by the imported module. Note that model types are contained using the `ownedType` property link.

`importedModule: Module [1]`

The module being imported.

Notation

See the notation for operational transformations and for libraries.

8.2.1.5 ModelParameter

A *model parameter* is a parameter to a operational transformation. Hence, the ordered set of model parameters forms the signature of the transformation. Each model parameter refers implicitly to a model participating in the query or transformation.

Each model parameter contains an indication stating the effect of the module execution on a model: **in** means changes forbidden, **inout** means model updated, **out** means model being created. These model parameters are globally accessible within the transformation.

Each model parameter has a type, for which see `ModelType`.

Superclasses

`VarParameter`

Constraints

The `type` of an instance of `ModelParameter` is an instance of `ModelType`.

self.type.oclIsKindOf(ModelType)

Notation

Model parameters are notated as simple parameters within the signature of a transformation. If the direction kind is not provided, the **in** value is assumed.

8.2.1.6 ModelType

Each model parameter conforms to a model type, which is defined or referenced by the transformation or the library. A model type is defined by a metamodel, a conformance kind and an optional set of constraint expressions. The metamodel defines the set of classes and property elements that are expected by the transformation, and is captured in a set of MOF Packages.

Type conformance is defined in two ways: **strict** and **effective**. When conformance is **strict**, the objects of the model extent should necessarily be instances of the classes of the associated metamodel.

When conformance is **effective**, any object in the model extent that has a type that refers to a type of the associated metamodel must contain the properties defined in the metamodel class and have compatible property types. The binding between the types in the metamodel and the actual types in the model extent is based on name comparison, except for specific renamings expressed using the `alias` tag ñ (see the QVT standard library ñ predefined tags).

Effective compliance allows flexible transformations to be defined that can be applied to similar metamodels. For example, if a transformation is defined for UML 1.4 but uses no UML 1.4 specificities, we can manage so that it also works with UML 1.3 models.

In both cases (**strict** or **effective**), model conformance also implies conformance with the well-formedness rule of the associated metamodel. For example, if UML 1.4 is the metamodel, the well-formedness rules are the rules that are defined in the OMG formal specification document.

To restrict the set of valid participant models further, a model type may specify a list of extra conditions (expressed as OCL expressions) that need to hold for participating models. For example: a transformation that expects UML models with use cases can restrict otherwise well-formed UML models that do not contain use cases.

A model type is defined as a subclass of `Class` so that it is possible to define operations and properties on it.

Pre-defined operations defined on types can be used to inspect the objects that are part of the model at any time during the transformation. More precisely there is a MOF extent corresponding to each parameter. Any object creation occurs in an extent associated with a model parameter.

Superclasses

`Class`

Attributes

`conformanceKind: String`

indicates the kind of required compliance. Predefined values are **effective** and **strict**. The default value is **effective**. Other values could be defined but their semantics is outside the scope of this specification.

Associations

`metamodel: Package [1..*] {ordered}`

the packages defining the structural constraints for a model parameter to comply to its model type. The `Package::uri` informs the user on the "actual" metamodel that has been taken as reference ñ in the case the referred metamodel is an *effective metamodel*.

`extraCondition: OclExpression [1..*] {composes, ordered}`

additional conditions restricting the set of valid participant models for this model type.

Semantics of model compliance

When an operational transformation in instantiated, the model parameters passed as arguments should *be conformant* with the model types of the parameters of the instantiated transformation. The meaning of *compliance of models to models types* is defined below:

When the compliance kind is "strict", the objects of the model extent should necessarily be instances of the metaclasses of the referred MOF packages. If the referred metamodel defines well-formedness constraints these should also be satisfied.

When the compliance kind is "effective", any instance in the model extent that has a type that is referred in the model type metamodel need at least to contain the properties defined in the effective metamodel metaclass and have compatible types. The binding between the types in the effective metamodel and the actual types in the model extent is based on name comparison, except for specific renamings expressed using the `alias` pre-defined tag ñ see the QVT standard library.

In both cases (`strict` or `effective`), model compliance implies additionally to comply with the extra constraints of the model type.

Effective compliance allows defining flexible transformations that "work" for similar metamodels. For instance, if a transformation is defined for UML 1.4 but uses no UML 1.4 specificities, we can manage so that it also works with UML 1.3 models. An *effective metamodel* represents the metamodel that is declared at the transformation definition level whereas an *actual metamodel* represents the metamodel that is used when executing the transformation.

Note: We define here only the meaning of a model being compliant with a model type. Comparisons and classifications between model types are possible but are considered to be out of scope of this specification.

Relationship between MOF extents and model parameters

When a model element is created by a transformation it is necessary to know in what model the model element is to be created. In particular, this makes possible to use the inspection operations on model parameters like `objects`() and `objectsOfType`() to retrieve an object previously created. In MOF there is a notion of Extent which is simply

defined as a container for Objects. Here we are correlating the notion of model (represented by model parameters in a transformation definition) with the notion of MOF extent stating that that for each model parameter there is a MOF extent.

Notation

A model type is referred *by name* in the signature or in the **access** and **extends** declaration of a transformation or library. In the example below, the UML and RDBMS symbol names are necessarily model types and there is no obligation to declare them as being model types.

transformation Uml2Rdbms(**in** uml:UML, **out** rdbms:RDBMS);

When a modeltype is explicitly declared, the syntax is as follow:

modeltype <modeltypeid> "<conformance>"
uses <packageid>("<uri>") **where** {<expressions>…};

The extra conditions may use the **self** variable which is implicitly defined in the where block and refers conceptually to an instance of the model type (a model). The declaration below declares that a model type uses for its definition an existing Package named `SimpleUml` by providing its URI. The second declaration only gives an URI and indicates a `strict` compliance kind.

modeltype UML **uses** SimpleUml("http://omg.qvt-samples.SimpleUml);
modeltype RDBMS "strict" **uses** "http://omg.qvt-samples.SimpleRdbms";
transformation Uml2Rdbms(**in** uml:UML, **out** rdbms:RDBMS);

A modeltype symbol may have the same name than a metamodel definition. If no explicit definition of the model type is found, this is equivalent to declare a modeltype which "referred metamodel" is the given metamodel. The "effective" model type conformance is assumed.

metamodel SimpleUML **{ … }**;
metamodel SimpleRDBMS { … };
transformation Uml2Rdbms(**in** uml:SimpleUML, **out** rdbms:SimpleRDBMS);

8.2.1.7 VarParameter

A *variable parameter* is an abstract concept that is introduced to allow referring to parameters in the same way as variables are referred, specifically within OCL expressions.

Syntactically, a `VarParameter` is a MOF `Parameter` that is also a `Variable`.

Superclasses

`Variable`

`Parameter`

Attributes

`kind: DirectionKind`

indicates the effect of the module or the operation on the parameter. Possible values are: `in`, `inout` and `out`.

8.2.1.8 DirectionKind

A `direction kind` is an enumeration type that gives the possible values of direction kinds for parameters.

Enumeration values

`in`

`inout`

out

8.2.1.9 ImportKind

An `import kind` is an enumeration type that gives the possible values for the semantics of module import.

Enumeration values

access

extension

Concepts related with the definition of operations and properties

Figure 8.2 depicts the concepts that are related to the definition of imperative operations and contextual properties. The metaclasses described in this section are:

`ImperativeOperation`, `MappingOperation`, `Helper`, `Constructor`, `EntryOperation`, `ContextualProperty`, `OperationBody`, `MappingBody and ConstructorBody`.

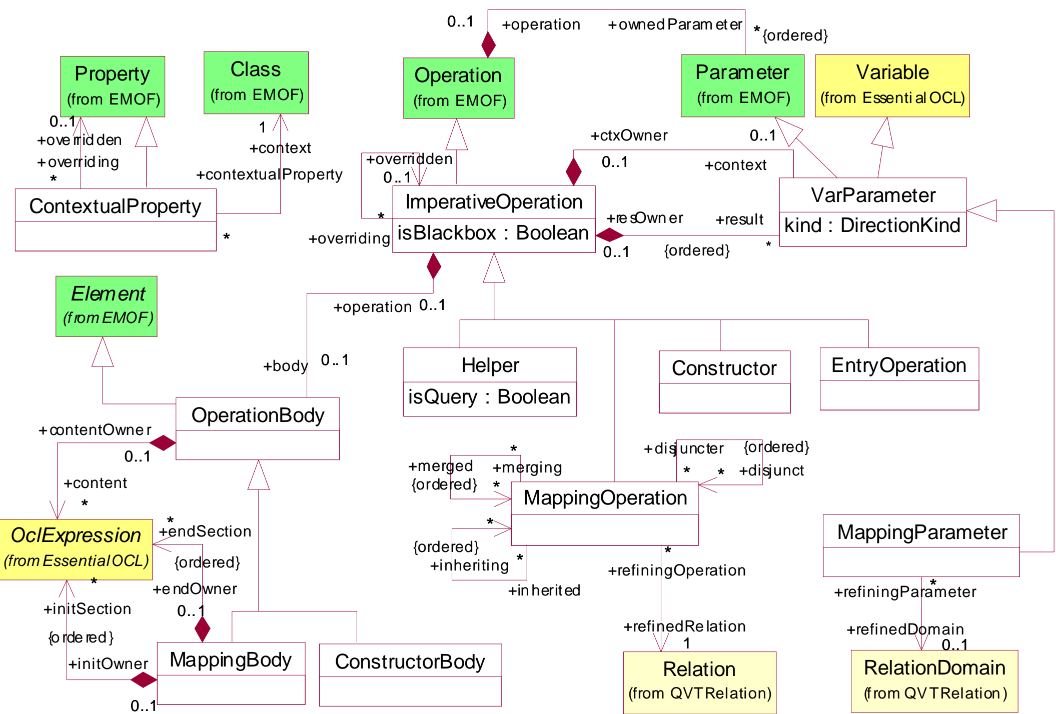


Figure 8.2 - QVT Operational Package - Defining Imperative Features

8.2.1.10 ImperativeOperation

An *imperative operation* extends the general notion of MOF operation with the ability to define an imperative body and an enriched signature for the operation. In addition to the ordinary parameters of a MOF operation, an imperative operation may declare a context parameter and zero or more result parameters. The context parameter, named **self**, has a type (called the context type of the operation). These parameters apply uniformly to helpers and mapping operations. An imperative operation is owned by an operational transformation or by a library.

An imperative operation may override an existing imperative operation defined in a higher level of the inheritance tree. In this case, the name of the overriding operation must be the same as the overridden operation and the signature must be compliant (same number of parameters, and the type of each parameter of the overriding operation should comply with the type of the corresponding parameter in the overridden operation).

An imperative operation that defines a context within its signature is known as a *contextual operation*. Within a module, two contextual operations can have the same name only if they have distinct contexts. Conceptually, a contextual operation behaves as an operation that extends the referred contextual type. For instance, within a transformation definition that deals with UML models, one may want to make usage of a query named `'getAllAbstractBaseActors'` on `Actor` instances (`Actor` is our contextual type). In order not to change the definition of the UML `Actor` metaclass by inserting "physically" a new owned operation, this query will not be owned by `Actor` class but instead will be an operation of the operational transformation. The *context* is then used to associate the query to the class being "logically" extended. In other words, making an explicit distinction between *ownership* and *context* avoids creating variants on metamodels for the sole purpose of a transformation definition.

In terms of name scoping, the definition of a contextual operation inserts a new symbol in the namespace of the owner (the transformation or the library) and in parallel inserts a symbol in the namespace of the context class. As a consequence, a contextual operation can be invoked either as a non-contextual operation (**self** is the first `argument` of the call expression) or as an operation of the context class (**self** is the `source` of the call expression).

Superclasses

`Operation`

Associations

`context: VarParameter [0..1]`

The context variable representing the object on which the operation is invoked.

`result: VarParameter [*] {composes}`

The variables containing the values to be returned by this operation.

`body: OperationBody [0..1] {composes, ordered}`

The imperative implementation for this operation.

`overridden: Operation [0..1]`

An operation of an imported transformation or library that is overridden.

`/ownedParameter:Parameter [*]{composes,ordered} (from Operation)`

The other parameters of the operation.

8.2.1.11 EntryOperation

An *entry operation* is the entry point for the execution of a transformation. Its body contains an ordered list of expressions executed in sequence. A transformation may define no more than an entry operation, which is invoked on entry.

An entry operation has no parameters but can access any globally accessible property or parameter, such as model parameters. The name of an entry operation is **main**.

Superclasses

`ImperativeOperation`

Notation

The notation uses the **main** keyword and a body with the list of expressions.

transformation UmlCleaning(**inout** uml:UML);
main() **{** uml->objectsOfType(Package)->**map** cleanPackage();}

8.2.1.12 Helper

A *helper* is an operation that performs a computation on one or more source objects and provides a result. The body of a helper is an ordered list of expressions that is executed is sequence. When more than one result is declared in the signature of the helper operation, the invocation of the operation returns a tuple.

Unless the `isQuery` property is true, a helper may have side-effects on the parameters ñ for instance a list may be passed and changed within the body and its effect is visible after the operation call termination. However it is illegal to create or update object instances within a helper operation ñ except for pre-defined types like sets, tuples and for intermediate properties.

Helpers allow writing complex queries in a comfortable way since the user is not restricted to write everything within a unique expression.

Superclasses

`ImperativeOperation`

Attributes

`isQuery: Boolean`

Indicates whether the helper operation can have side-effects on the parameters.

Constraints

The body of a helper operation is a direct instance of OperationBody.

self.body.isTypeOf(OperationBody)

Notation

The notation is the notation for any imperative operation except that the **query** or the **helper** keyword are used ñ the latter should be used when the operations provokes side-effects. The body may be introduced using a simple expression (with '=' symbol) or be defined within braces.

The declaration below declares a query to retrieve all derived classes of a UML class. In this example the body is not defined, meaning that it is a *black-box* implemented elsewhere. This query is defined within the `UmlUtilities` library.

He **self** variable to access the properties of the context argument cannot be omitted in the body of the helper operation.

library UmlUtilities(UML);
query Class::getAllDerivedClasses() : Set(Class);

8.2.1.13 Constructor

A *constructor* is an operation that defines how to create and populate the properties of an instance of a given class. This concept corresponds to the familiar notion of a constructor in object oriented languages. A constructor may be defined as an operation of the class to be constructed or may be owned by a module which plays a role of factory for the class.

A constructor may be defined within a library so that it can be reused by various transformations. A constructor is a means to factor out the code needed to create and populate an object.

A constructor does not declare result parameters. The name of the constructor is usually the name of the class to be instantiated. However this is not mandatory. Giving distinct names allow having more than one constructor.

To create an object it is not mandatory to define a constructor operation. For each class, if not defined explicitly, there is an implicit default constructor that has no parameter and that has the name of the class. Also the *object expression* specific construct allows to instantiate and to populate an object inline (see *ObjectExp*).

A constructor operation is implicitly or explicitly invoked through instantiation expression (see `InstantiationExp`).

Superclasses

`ImperativeOperation`

Associations

`/body: BlockExp [0..1](from ImperativeOperation)`

the expression serving to populate the object using the given parameters. This expression should necessarily be a `ObjectExp` instance.

Constraints

The body of a constructor operation is a direct instance of ConstructorBody.

self.body.isTypeOf(ConstructorBody)

Notation

The notation for declaring constructors is similar to the notation for declaring any imperative operation except that uses the **constructor** keyword and declares no result. The name of the constructor is necessarily the name of the context type. The body of the constructor is notated within braces. It contains directly the contents of the object expression.

The declaration below is an example of a constructor definition for a `Column` metaclass.

constructor Column::Column (n:String,t: String) { name:=n; type:=t; }

8.2.1.14 ContextualProperty

A *contextual property* is a property that is owned by a transformation or a library but is defined as an extension of the type referred by the *context*. Such properties are accessed as any other property of the referred context. This is typically used to define *intermediate properties* as class extensions of metaclasses involved in a transformation. Intermediate data is created temporarily by a transformation to perform some needed calculation but which is not part of the expected output.

Superclasses

`Property`

Associations

`context: Class [1]`

the class being extended with this property.

`overridden: Property [1]`

the imported property being overridden by this property.

Notation

The notation for a contextual property uses the "regular" **property** keyword. It may be, complemented with the **intermediate** qualifier if the property is defined as an intermediate property of the operational transformation.

intermediate property Class::leafAttributes : **Sequence**(LeafAttribute);

8.2.1.15 MappingOperation

A *mapping operation* is an operation implementing a mapping between one or more source model elements into one or more target model elements.

A mapping operation may be provided with its signature only or may additionally be provided with an imperative body definition. In the former case - where there is no body provided ñ the operation is said to be a *black-box*. Mapping black-boxes are useful to escape from the QVT language in specific sections of the transformation. An example of situation where the escape mechanism is useful is when the transformation treatment requires dedicated techniques like lexical and syntax analysis which will be hard to provide using the QVT formalism.

A mapping operation always refines a relation where each relation domain corresponds to a parameter of the operational mapping. The **when** clause in the refined relation acts either as a *pre-condition* or as a *guard*, depending on the invocation mode of the mapping operation. The **where** clause in the refined relation always acts as a post-condition for the mapping operation.

The body of a mapping operation is structured in three optional sections. The initialization section is used for computation prior to the effective instantiation of the outputs. The population section is used to populate the outputs and the finalization section is used to define termination computations that take place before exiting the body.

There are three reuse and composition facilities associated to mapping operations.

A mapping operation may *inherit* from another mapping operation. This essentially means invoking the initialization section of the inherited operation after executing its own initialization section.

A mapping operation may also *merge* other mapping operations. This essentially means invoking the merged operations after the termination section.

A mapping operation may be defined as a *disjunction* of other mapping operations. This essentially means selecting among the set of disjuncted mappings the first that satisfies the **when** clause and then invoking it. The execution semantics sub-section below provides the details of the semantics of the reuse facilities.

Syntactically, a `MappingOperation` is a subclass of the `ImperativeOperation` metaclass. As such it is owned by an `OperationalTransformation` ñ which is a specific kind of `Class`.

Superclasses

`ImperativeOperation`

Attributes

`isBlackbox: Boolean`

indicates whether the body is available. If `isBlackbox` is true this means that the definition should be provided externally in order to the transformation to be executed.

Associations

`inherited: MappingOperation [*]`

indicates the list of the mappings that are specialized.

`merged: MappingOperation [*]`

indicates the list of mapping operations that are merged.

`disjunct: MappingOperation [*]`

indicates the list of potential mapping operations to invoke.

`refinedRelation: Relation [1]`

the refined relation. The relation defines the guard (when) and the postcondition for the mapping operation.

Constraints

The body of a constructor operation is a direct instance of MappingBody.

self.body.isTypeOf(MappingBody)

Execution Semantics

We first define the semantic of the execution of a mapping operation in absence of any reuse facility (inheritance, merge and disjunction), then we describe the effect of using these facilities.

Executing a mapping operation

A mapping operation may declare a contextual parameter, in which case the operation extends the type of the contextual parameter. Resolving the mapping call implies finding the operation to call on the basis of the actual type of the source (**self** variable). This follows usual object-oriented virtual call semantics.

After call resolution, all the parameters of the mapping are passed as a tuple. The parameters includes, in this order: the context parameter, if any, the owned parameters (from `Operation::ownedParameter`) and the parameters declared as result. All `out` parameters, including result parameters, have their value initialized to `null`. All `in` or `inout` non null values, except for the primitive types are passed by reference. However it is not legal to change the value when an object is declared with `in` direction kind.

After passing of the parameters, the type compliance of the actual object parameters and the **when** clause are checked. If this fails `null` is returned.

If the guard succeeds, the relation trace is checked to find out whether the relation already holds. If so, the `out` parameters are populated using the corresponding trace tuples of the relation and the value associated to the result parameters is returned. Otherwise the body of the operation is executed in three sections.

The execution semantics of the body of a mapping operation is as follows:

i) The initialization section is entered and the expressions of the initialisation section are executed in sequence. In the initialisation section typically we find variable assignments, mapping or query invocations or explicit assignments of the output parameters.

ii) At the end of the initialization section, an implicit "instantiation section" is entered which provokes the instantiation of all the `out` parameters that are object instances and which still have a `null` value. Collection types are initialized with empty collections. By doing so, the corresponding relation trace tuple is populated. From that point the relation is considered to hold and the trace data becomes available.

iii) The population section is entered and each expression is executed in sequence. A population section typically contains a list of object expressions which correspond with the `out` and `inout` parameters.

iv) The termination section is entered provoking the execution in sequence of the list of expressions. A termination section typically contains additional computations that need to take place after population of output objects occurs, such as operation mapping invocations.

Executing mappings that inherits from other mappings

A mapping that has inherited mappings invokes first its initialisation section, including the implicit instantiation section, and then invokes the inherited mappings. Invocation of the inherited mappings follows the "standard" invocation semantics, except that the `out` parameters may now start with a non-`null` value, which would be the case if the out parameter was changed in the initialisation section of the inheriting mapping. Parameter compliance between the inheriting mapping and the inherited mapping follows the compliance constraints between a caller and a callee.

Executing mappings merging other mappings.

The merged mappings are executed at the end of the execution of the merging mapping. The parameters of the merging mapping are passed to the parameters of the merged mappings, including the actual value of the `out` parameters. Parameter compliance between the merging mapping and the merged mappings simply follows the compliance constraints between a caller and a callee.

Executing mappings defined as disjunction of other mappings.

An invocation of a mapping operation defined as a disjunction of other mapping operation is done in two steps: firstly, the guards of the disjuncted mappings are executed in sequence until one of the guards succeeds. If no guard succeeds the null value is immediately returned. Otherwise, the body of the mapping which guard has succeeded is executed. The signature of the disjuncting mapping must conform to the signature of the disjuncted mappings ñ following ordinary constraints between the caller and the callee. Specifically, the result of the disjunction needs to be a super type of the result type of the composed mappings.

Notation

A mapping operation signature is notated as any other operation signature except that it uses the **mapping** keyword, and includes various additional elements. The general form is:

> **mapping inout** **<**contexttype>::<mappingname> (**<**parameters**>**,) : <result-parameters>
> **inherits** <rulerefs>, **merges** <rulerefs>, **disjuncts** <rulerefs>,
> **refines** <rulerefs> **when** {<exprs>} **where** { <exprs>}

The `<contexttype>` appears only if the mappings declare a contextual parameter. For result parameters the direction kind is necessarily `out` and is not notated. For the other parameters, including the contextual parameter, the default direction kind is `in`.

The declaration below is an example of a mapping operation that defines a contextual parameter (of type `Package`), a result (of type `Schema`), and a guard. No body is defined (it is a black-box).

> **mapping** Package::packageToSchema() : Schema
> **when** { **self**.name.startingWith() <> "_"}**;**

8.2.1.16 MappingParameter

A *mapping parameter* is a parameter of a mapping operation. A mapping parameter has a *direction kind* which restricts the changeability of the argument passed when the mapping operation is invoked. Possible values of direction kinds are: `in`, `inout` and `out`.

A mapping operation being a refinement of a relation, a mapping parameter is associated with a *domain* of the refined relation. This correspondence is based on the order of the declared parameters and domains, where the contextual parameter ñ if any ñ is the first and the result parameters are positioned after the "regular" parameters. The type of the mapping parameter should necessarily be the same than the type specified by the object pattern of the domain (see `Domain` definition in 7.11 section).

Superclasses

`VarParameter`

Attributes

`/kind: DirectionKind (from VarParameter)`

The direction indication of the parameter. `in` value means that the actual parameter is not changed, `inout` means the actual parameter is updated, `out` means that the actual parameter will receive a new value.

Associations

`referredDomain: RelationDomain`

The relation domain that corresponds to the parameter.

Constraints

The type of the parameter is the type of the object template expression of the referred domain.

8.2.1.17 OperationBody

An *operation body* contains the implementation of an imperative operation which is made of an ordered list of expressions that are executed in sequence.

An operation body defines a scope that is contained in the scope of the operation definition. Variables and parameters defined in outer scopes are accessible.

Superclasses

`Element`

Associations

`content: OclExpression [*] {ordered}`

The list of expressions of the operation body.

Notation

An operation body is delimited by braces where each contained expression is separated by semi colons. Within the operation body the **self** variable represents the context parameter ñ if any, and the **result** variable represents the parameter to be returned ñ which is a tuple when multiple results are declared.

8.2.1.18 ConstructorBody

An *constructor body* contains the implementation of a constructor operation or the implementation of an inline constructor (see `ObjectExp`).

Superclasses

`OperationBody`

Notation

A constructor body is delimited by braces where each contained expression is separated by semi colons. However, in contrast with the general operation body notation the variable representing the instantiated object can be omitted when referring to its properties.

The example below illustrates this notation facility. A constructor for a Message class that defines two 'name' and 'type' attributes is defined.

```
constructor Message::Message(messName:String,messType:String) {
  name := messageName; // same as result.name := messageName
  type := messType:String;  // same as result.type := messType
}
```

8.2.1.19 MappingBody

A *mapping body* defines the structure of the body of a mapping operation. It consists of three non-mandatory sections: initialisation, population and termination sections. See execution semantics in the `MappingOperation` metaclass description.

Superclasses

`OperationBody`

Associations

`initSection: OclExpression [0..1] {ordered}`

The initial section containing the ordered set of expressions to be executed in sequence prior to a possible instantiation of the output parameters.

`/content: OclExpression [0..*]  {ordered}`

The population section containing the expressions used to populate the `inout` parameters and `out` parameters.

`endSection: OclExpression [0..1] {ordered}`

The termination section containing other expressions to perform final computations before leaving the mapping operations.

Notation

The notation uses firstly braces that come after the signature of the mapping operation. The general syntax which is:

```
mapping <mapping_signature>  // see MappingOperation description
{
  init { … } // init section
  population { … } // population section
  end { … } // end section
}
```

In most cases, the **population** keyword can be skipped.

On the contrary, **init** and **end** keywords cannot be skipped unless these sections are empty.

The rule for interpreting a body in which there is no **population** keyword is as follows:

(1) If the mapping operation defines a unique result, the list of expressions in the body is the list of expressions of the ñ unique - implicit *object expression* (see ObjectExp) contained by the population section.

(2) If the mapping operation defines more than one result, the list of expressions in the body is the list of expressions of the population section.

This notation convention facilitates the writing of concise specifications since the situation where there is a unique result is very common.

An explicit usage of the **population** keyword may be required is certain situations ñ such as to update inout parameters.

According to these two rules, the declaration:

```
mapping A::AtoB() : B {
  init { … }
  myprop1 := … ;
  myprop2 := …;
}
```

is equivalent to:

```
mapping A::AtoB() : B {
  init { … }
  population {
    object result:B {
        myprop1 := … ;
         myprop2 := …;
    };
  }
}
```

The same convention as for the general operation body applies to mapping bodies: **self** represents the contextual argument and **result** the declared result ñ possibly a tuple if more than a result is declared.

Concepts related with the usage of imperative operations.

Figure 8.3 depicts the specific concepts related with the usage of the imperative operations. The group of classes described in this sections consist of: `ImperativeCallExp`, `MappingCallExp`, `ObjectExp`, `ResolveExp` and `ResolveInExp`.

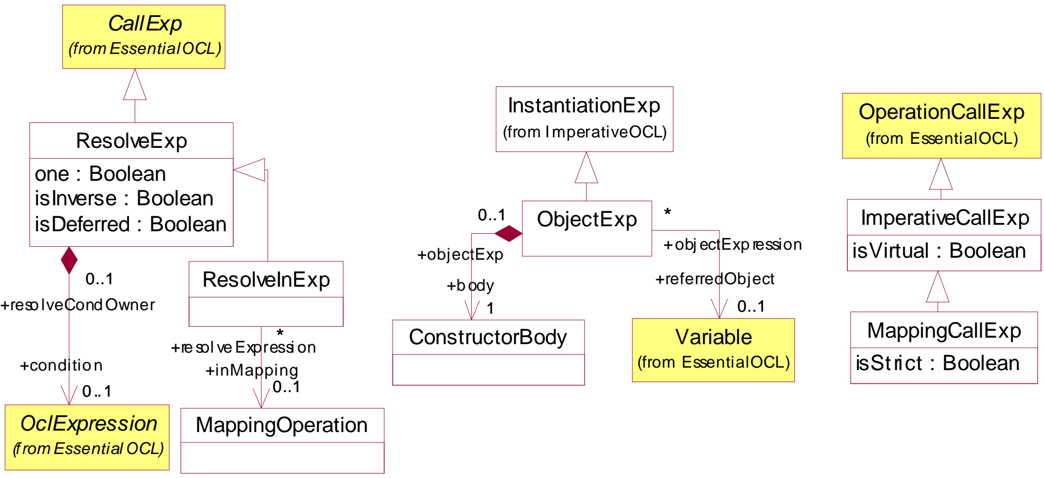


Figure 8.3 - QVT Operational Package - Usage of Imperative Operations

8.2.1.20 ImperativeCallExp

An `imperative call expression` represents the invocation of an imperative operation. Unless `isScoped` is true this invocation is *virtual* : the operation to call depends on the actual type of the context parameter (similarly as for Java and C++ languages).

Superclasses

`OperationCallExp`

Attributes

`virtual: Boolean = true`

Indicates whether the referred operation should be statically called (true value) or dynamically resolved (false value). The default value is true.

Notation

An imperative call is notated as any operation call where the source object may not be explicitly provided.

The general syntax is:

<operationreference>(<arg1>,<arg2>, …, <argN>) or
<source>.<operationreference>(<arg1>,<arg2>, …, <argN>) or
<source>-><operationreference>(<arg1>,<arg2>, …, <argN>).

The third form is used for collection operations.

Note that the omission of the source is a shorthand in terms of the execution semantics but not in terms of the metamodel representation: the `OperationCallExp::source` property may be left empty.

Qualified names in <operationreference> can be used to disambiguate operations. This is specifically needed when two *accessed* modules define operations with the same name.

8.2.1.21 MappingCallExp

A *mapping call expression* represents the invocation of a mapping operation. A mapping operation can be invoked either in strict mode or in standard mode ñ depending on the value of the `strict` Boolean property. In strict mode the when clause is evaluated as a pre-condition, that is, it provokes an exception if it evaluates to false. In contrast, when the mapping is invoked in standard mode, the execution of the mapping body is skipped and the `null` value is returned to the caller.

Superclasses

`OperationCallExp`

Attributes

`isStrict : Boolean`

Indicates the mode of the mapping invocation.

Notation

A mapping call is notated as any other imperative operation call except that is uses the **map** or **xmap** keyword. The latter keyword is used when `strict` is true.

Depending whether the invoked mapping defines a contextual parameter the call notation will require or not require a source object.

// for a mapping defined with a contextual signature: Class::class2table() : Table

```
myumlclass.map class2table();  // invocation with non strict semantics
myumlclass.xmap class2table(); // invocation with strict semantics

// for a mapping  defined with a non-contextual signature: attr2Column(Attribute) : Table
map attr2column(myattr);  // invocation with non strict semantics
xmap attr2column(myattr); // invocation with strict semantics
```

The **map** and **xmap** keywords may be called on a list and have, if needed, the iterator variable in parentheses. This is called the "collect" shorthand: the mapping operation is in fact the body of the imperative collect construct *xcollect* (see `ImperativeLoopExp`).

```
self.ownedElement->map class2table();
    // shorthand of self.ownedElement->xcollect(i) i.map class2table();
    // the iterator is implicit
self.ownedElement[#Class]->xmap(i) i.class2table();
   // the iterator variable is explicitly passed in parentheses of xmap keyword
```

It is always possible to invoke a mapping operation using a reference to a transformation instance as the source of the call.

```
// for a mapping  defined with a non-contextual signature: attr2Column(Attribute) : Table
this.map attr2column(myattr);  // the this keyword refers to the current transformation instance
```

When the invoked mapping operation has a context type, the context argument takes the position of the first argument. If the mapping declares additional parameters, the corresponding additional arguments are passed with its position shifted.

```
// for a mapping  defined with a contextual signature: Class::class2table() : Table
this.map class2table(myumlclass);  // equivalent to myumlclass.map class2table()
```

We should note that when the called mapping operation is not invoked on behalf of the current transformation (the **this** transformation instance) then the only way to make such a call is to pass the reference of the transformation instance as the source of the call. The example below depicts this situation: the `cleaningTransf` is an instance of a transformation that has been imported. An explicit call to the `removeDup` mapping is done on behalf of this `cleaningTransf` instance.

```
transformation Uml2Java(in uml:UML,out java:JAVA)
  access transformation UmlCleaning(UML);
mapping UmlCleaning::Class::removeDups(); // declaring the signature of an imported mapping
main () {
  cleaningTransf  = UmlCleaning(uml); // instantiating the imported transformation
  // first pass: cleaning the UML classes
  uml->objectsOfType(Class) // invoking the imported transformation
      ->forEach (cl) cleaningTransf.map removeDups(cl);

  // second pass: transforming all UML classes
  uml->objectsOfType(Class)->forEach (cl)
      cl.map umlclass2javaclass (); // equivalent to:  this.map umlclass2javaclass(cl)
}
mapping UML::Class::umlclass2javaclass(): JAVA::Class { … }
```

8.2.1.22 ResolveExp

A *resolve expression* is an expression that inspects trace objects of the transformation in order to retrieve target objects created or updated by mapping operation invocations executed previously on source objects. Conceptually, for each mapping invocation, the transformation records the correspondence between source and target objects participating in a given mapping invocation. A resolve expression has a conditional expression which is used to filter the target objects.

There are various variants for this construct. Firstly, instead of looking for all target objects satisfying the condition, it is possible to indicate that the first is to be returned.

Also, instead of looking for created or updated objects, the inverse operation can be requested, that is, looking for all source elements responsible for creating or updating a given target.

Finally, the last orthogonal variant, allows invoking the construct in *deferred* mode. The effect of this mode is to postpone the lookup of target objects to the end of the execution of the transformation. This facility is used in conjunction with an assignment. It is typically useful to avoid multiple passes to solve a transformation problem.

Superclasses

`CallExp`

Attributes

`one: Boolean`

Indicates whether the resolve expression should return a unique first result or all results.

`isInverse: Boolean`

If true the resolve expression looks for the source objects responsible for the "creation" of an object instead of looking for the objects "created" by the source object.

`isDeferred: Boolean`

Indicates whether the resolve expression return a *future* value containing the necessary information to compute the resolution later. See execution semantics.

Execution Semantics

The trace information for a mapping operation invocation is created after the execution of the initialisation section (see `MappingCallExp` execution semantics). The trace contains a tuple that stores a reference of the mapping operation (or, what is equivalent, a reference to the corresponding relation) and then the value for each parameter, including the context variables and the result variables. It remembers the direction kind of each parameter as well as the position of the slot referring to the context variable and the slot of the result variables. In this sense the trace for operational transformation extends the structure of the trace defined for relational transformations.

An execution engine then has sufficient information to be able to keep track of the objects that were derived from a source object and to reverse the relationship.

Deferred resolutions have implications in the execution semantics of assignment expressions (see, `AssignExp`). The effect is explained below:

A *deferred assignment* is an assignment where the value ñ necessarily a unique value ñ is a *future* value produced by a resolve expression. This assignment is not executed at the time the assignment is reached during the control flow. A `null` value is returned instead. In the meantime, the execution engine stores the following information for the future variable: the source object, the function representing the filtering expression and the property or the variable reference to

be assigned. This information is sufficient to allow the assignment to be performed later - more precisely ñ at the end of the execution of the entry operation of the transformation. The tuple storing this information is appended to an ordered list which is given to the entry operation to terminate with the execution of the transformation.

Notation

The notation uses the operation call convention where the called operation is named `resolve`, `resolveone` and `invresolve`. The condition expression is in parentheses. When `isDeferred` is true the **late** keyword is used before the operation name. We should note however that `resolve`, `resolveone` and `invresolve` are not defined as pre-defined operations ñ since they are directly represented by the `ResolveExp` metaclass. The resolution operator may be called on a list. This is a shorthand for invoking it in the body of a `ForEachExpr` expression.

myresult := mysourceobject.**resolveone**(#Table);
 // '#Table' is a shorthand for 'oclIsKindOf(Table)'
myprop := mylist->**late resolve**(#Table);
 // shorthand for mylist->**forEach**(i) i.**late** resolve(#Table)

8.2.1.23 ResolveInExp

A *resolve in expression* looks for target objects created or updated from a source object by a unique mapping operation. In contrast, a *resolve expression* performs a lookup for all mapping operations. The source object is optional. When no source object is provided, this expression inspects all the targets created or updated by the mapping operation irrespective of the source objects.

All variants described for the resolve expression are applicable to the resolve in expression.

Superclasses

`ResolveExp`

Association

`inMapping: MappingOperation [0..1]`

The mapping rule that is the target for trace inspection.

Notation

The notation uses the operation call syntax where the called operation is named `resolveIn or resolveoneIn`. The **late** keyword is used for deferred resolutions.The same notational conventions as for the `ResolveExp` apply. The first parameter is the reference of the rule and the second parameter is the condition to evaluate.

myresult := mysourceobject.**resolveIn**(myrule, mycondition);

8.2.1.24 ObjectExp

An *object expression* is a specific *inline* instantiation facility. It contains a constructor body, it refers to a class and refers to a variable, possibly `null` valued. If the variable is `null`, a new object of the given class is created, then assigned to the variable and finally the constructor body is executed. If the variable is non null, no instantiation occurs but the constructor body is used to update the existing code. In both cases the value returned is the value of the variable at the end of the execution of the body.

All visible variables of outer scopes can be accessed within the object expression.

Superclasses

`InstantiationExp`

Associations

`referredObject: Variable [0..1]`

The object to be updated or created.

`/instantiatedClass:  Class [0..1](from InstanciationExp)`

Indicates the class of the object to be created or populated.

`/extent:  Variable [0..1](from InstanciationExp)`

References a model parameter where the object should reside in case it is instantiated. This is optional.

Notation

The notation uses the **object** keyword followed by the referred variable and the referred class. There are some syntax variants depending on the availability of the referred variable and the possibility to skip the class reference.

object x:X { … } // An explicit variable here
object Y { … } // No referred variable here.
object x: { … } // the type of 'x' is skipped here when already known

When provided the model parameter is notated within brackets after the **object** keyword.

object[srcmodel] x:X { … } // x is created within the 'srcmodel'

When an object expression is the body of an imperative collect expression (see *xcollect* in `ImperativeLoopExp`), the reference to the collect construct may be skipped and the arrow symbol applies on the object keyword.

list->**object**(x) {… } // shorthand for list->**xcollect**(x) object { … }

Under certain circumstances, the **object** keyword itself is skipped and its body contents directly expanded in an outer definition. See the notation of `MappingBody`.

8.2.2 The ImperativeOCL Package

The ImperativeOCL Package extends OCL with imperative expressions to allow expressing complex treatments imperatively but in the meantime keeping some of the advantages of OCL expressivity. It also extends the OCL type system with additional facilities such as dictionaries (hashtables).

For convenience, the metaclasses defined in this package are divided in two groups. The first concerns the imperative expressions, the second one concern the additions to the type system.

Imperative expressions

Figure 8.4 depicts the class hierarchy for the side-effect expressions.

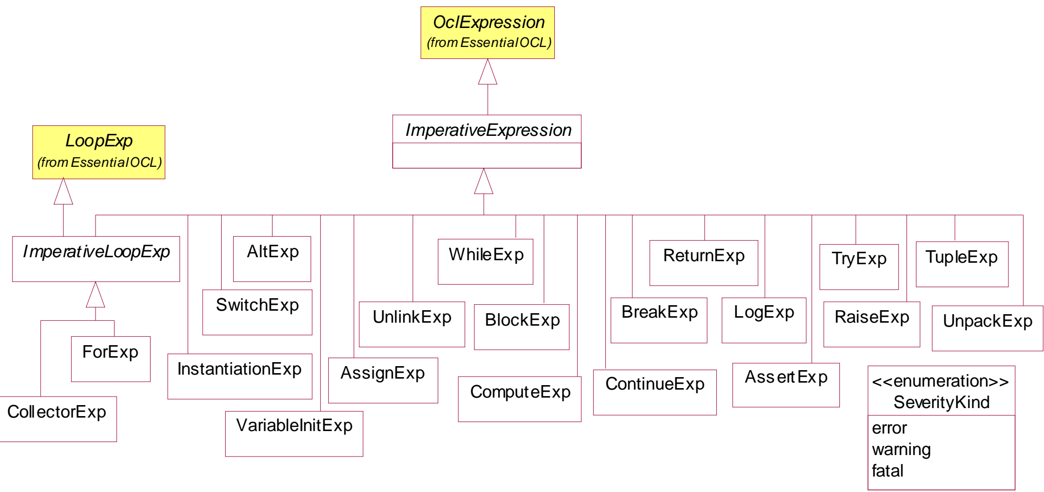


Figure 8.4 - Imperative OCL Package - Side-effect expressions hierarchy

Figure 8.5 depicts the specific control and block expressions enhancements as well as the instantiation facility.

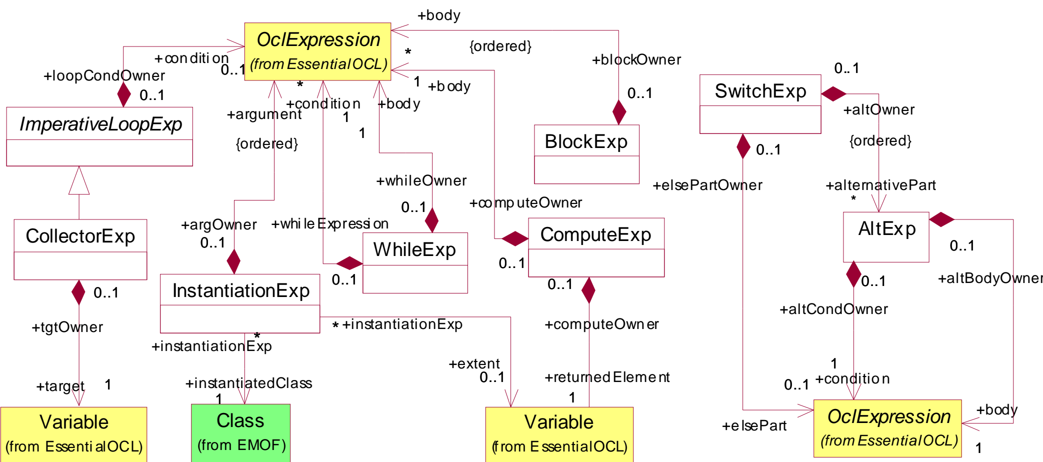


Figure 8.5 - Imperative OCL Package - Control and instantiation constructs

Figure 8.6 depicts the remaining expressions like those related with attribute manipulation, assignment, exception handling.

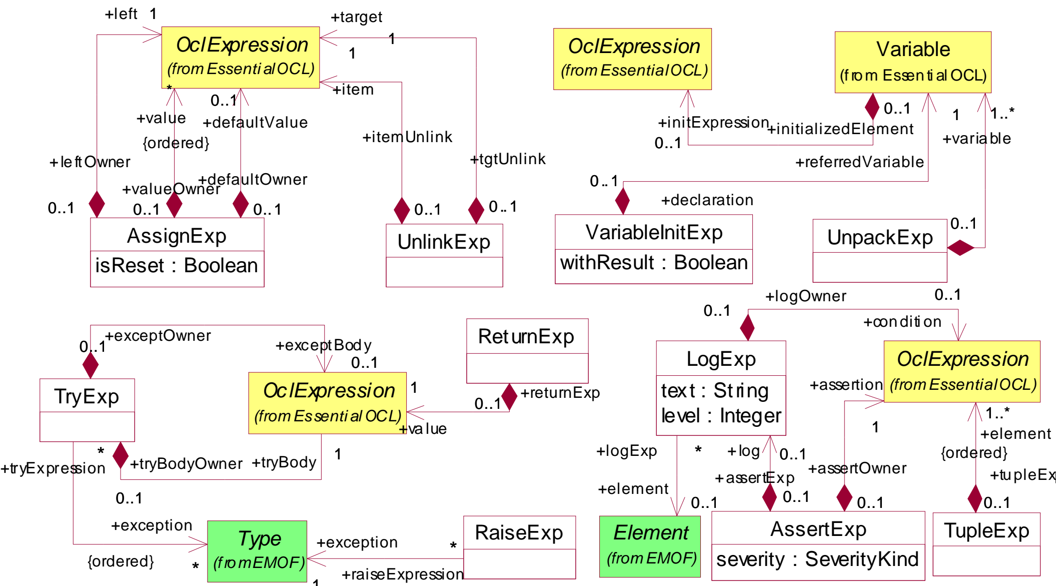


Figure 8.6 - Imperative OCL Package - Miscellaneous Facilities

8.2.2.1 ImperativeExpression

The *imperative expression* is an abstract concept serving as the base for the definition of all side-effect oriented expressions defined in this specification.

Note: In contrast with pure OCL side-effect free expressions, imperative expressions do not behave as functions. For instance executing interrupt constructs like break, continue, raise and return have an effect in the control flow of the imperative expressions that contain them.

Superclasses

`OclExpression`

8.2.2.2 BlockExp

A *block expression* is an expression that executes in sequence an ordered list of expressions. The returned value is *null*. The execution of a block expression may be interrupted by a break, a continue or a return expression.

A block expression is typically used in conjunction of other constructs like if and loop expressions.

The block creates a new scope ñ local variables to the scope are not accessible outside. Variables defined in outer scopes are accessible within the block.

Superclasses

`ImperativeExpression`

Associations

`body: OclExpression [0..*]`

The ordered list of expressions to be executed in sequence.

Notation

The notation uses the **do** keyword followed by braces to delimit the list of expressions. However, when used within the following control expressions: *if*, *switch*, *compute* and *for* expressions the **do** keyword can be skipped.

do { … } // executes the body and return null
if name.startswith("_") **then** { … } **else** null **endif** // **do** keyword being skipped

8.2.2.3 ComputeExp

A *compute expression* is an expression that defines a variable, possibly initializing it, and defines body to update the value of the variable. It returns the value of the variable at the end of the execution of the body.

Superclasses

`ImperativeExpression`

Associations

`returnedElement : TypedElement [1]`

The local Variable to hold the value for the result of the expression.

`body : OclExpression [1]`

The body to be executed to compute the value of the given variable.

Notation

The notation uses the **compute** keyword with a variable declaration in parentheses followed by the body.

compute (x:String = "_") { … } // The body is here a block expression

8.2.2.4 WhileExp

A *while expression* is a control expression that iterates on an expression until a condition becomes false. It returns `null`. A `break expression` executed within the body provokes the termination of the while expression. A `continue expression` provokes the execution of the next iteration without executing the remaining instructions in the block.

Superclasses

`ImperativeExpression`

Associations

`condition : OclExpression [1]`

The condition to be evaluated at each iteration (including the first iteration).

`body : OclExpression [1]`

The expression on which the while expression iterates.

Notation

The notation uses the **while** keyword with a condition in parentheses followed by a body.

while (not node.isFinal()) { … }
compute (x:MyClass := self.getFirstItem()) **while** (x<>null) { … }

When a compute expression is used on top of a while expression, the compute keyword can be skipped and the result variable initialization be placed within the parentheses of the header.

while (x:MyClass := self.getFirstItem(); x<>null) { … }

8.2.2.5 ImperativeLoopExp

An *imperative loop expression* is a generic concept representing an imperative loop statement that iterates on a collection ñ the *source* of the loop. It declares iterators, a body and a condition. The execution of the loop may be interrupted by a return, a break or a continue expression that is invoked within the loop.

This abstract concept serves as the base to define the pre-defined imperative loop constructs *forEach*, *forOne, collectSelect* and *collectSelectOne*.

Superclasses

`LoopExp`

`ImperativeExpression`

Associations

`condition : OclExpression [0..1]`

A Boolean condition to be evaluated. The role played by this condition depends on the concrete loop construct being instantiated.

8.2.2.6 ForExp

A *for expression* is an imperative loop expression that iterates over a source collection evaluating an expression for each element of the collection that satisfies a given condition. It returns the *null* value.

A for expression is a generic construct: it has two pre-defined variants named *forEach* and *forOne*. The name attribute is used to distinguish between the variants.

The forEach loop executes the body for all elements of the collection that satisfies the condition whereas forOne executes the body only for the first element that satisfies the condition.

Superclasses

`ImperativeLoopExp`

Attributes

`/name : String (from NamedElement)`

The name of the loop variant being used.

Associations

`/condition : OclExpression [0..1](from ImperativeLoopExp)`

The condition restricting the elements in the collection for which the body is executed.

`/body : OclExpression [1]   (from LoopExp)`

The body to execute at each iteration.

`/iterator : Variable [1..*] (from LoopExp)`

The iterator variables defined for this loop.

`/source : OclExpression [1] (from LoopExp)`

The source collection.

Semantics

The behavior of the two predefined variants is given below. All these definitions use the basic imperative constructs: *compute*, *while* and *block*.

```
Seq(T)::forEach(source, iterator, condition,body) : Seq(TT)  =
        do {
           count : Integer := 0;
           while (count <= source->size()) {
                var iterator := source->at(count+1);
                if (condition) continue;
                body;
                count += 1;
            };
        };

Seq(T)::forOne(source, iterator, condition,body) : Seq(TT)  =
        forEach (i | condition) {
           body;
           break;
        }
```

T represents the type of the source element.

Notation

```
<source>-><for-name> (<iterator-list> | <condition>) <body> ;
<source>->< for-name> (<iterator-list> ) <body> ;
```

where <for-name> is the name of the loop construct ñ for instance forEach and forOne.

```
list->forEach(i) {…}  // in this example the body is a block expression
compute (s:String = "_") { self.ownedElement->forEach(i|i.isKindOf(Actor)) {s += i.name;}}
   // in this example the for construct is embedded with a compute expression
list->forOne(i|i.isKindOf(Actor)) { …}
```

8.2.2.7 ImperativeIterateExp

An *imperative iterate expression* is an imperative loop expression that iterates over a source collection and builds a given result using iterator variables, a target variable, a body and a condition expression.

This expression is a generic construct: it has four pre-defined variants named *xcollect*, *collectselect*, *collectselectOne, xselect* and *selectOne*. The `name` attribute is used to distinguish between the variants. The behavior of these variants is similar to their OCL counterparts except that the execution of the loop can be interrupted through the usage of break, continue, raise and return expressions.

A *collectselect* behaves pretty much as an OCL collect construct composed with a select construct, except that the execution is conceptually performed within a single loop. In addition any null value is removed from the result list. See semantics section above, for a detailed definition.

Superclasses

`ImperativeLoopExp`

Attributes

`/name : String (from NamedElement)`

The name of the loop variant being used.

Associations

`target : Variable [0..1]`

The variable that holds the value computed in one iteration.

`/condition : OclExpression [0..1](from ImperativeLoopExp)`

The condition restricting the values being collected.

`/body : OclExpression [1]   (from LoopExp)`

The value that is appended to the result variable at each iteration.

`/iterator : Variable [1..*] (from LoopExp)`

The iterator variables defined for this loop.

`/source : OclExpression [1] (from LoopExp)`

The source collection.

Semantics

The behavior of the predefined variants of the collector expression is given below. Note that the approach taken is similar to the way side-effect free iterations are defined in OCL. All these definitions use the basic imperative constructs: *compute*, *forEach* and *block*.

```
Seq(T)::xcollect(source, iterator, body) : Seq(TT)  =
   compute (res:Seq(TT) := Seq{})
      source->forEach (iterator:T) {
         var target : TT := body;
         if (target<>null) res += target;
      };

Seq(T)::xselect(source, iterator, condition) : Seq(T)  =
   compute (res:Seq(T) := Seq{})
      source->forEach (iterator:T) {
         var target : T := iterator;
         if (target<>null and condition) res += target;
      };

Seq(T)::selectOne(source, iterator, condition) : Seq(T)  =
   compute (res:Seq(T) := Seq{})
      source->forEach (iterator:T) {
         var target : T := iterator;
         if (target<>null and condition) {res += target; break;}
      };

Seq(T)::collectselect(source, iterator, target, body, condition) : Seq(TT)  =
   compute (res:Seq(TT) := Seq{})
      source->forEach (iterator:T) {
         var target : TT := body;
         if (target<>null and condition) res += target;
      };

Seq(T)::collectselectOne(source, iterator, target, body, condition) : Seq(TT)  =
   compute (res:Seq(TT) := Seq{})
```

```
source->forEach (iterator:T) {
   var target : TT := body;
   if (target<>null and condition) {res += target; break;}
};
```

T and TT are respectively the type of the source elements and the type of the target elements. If the condition is not given it should be replaced by *true* in the definitions above.

Notation

The notation depends on the list of items that are explicitly passed to the construct. All possibilities are showed below:

```
<source> -> <collector-name> (<iterator-list>; <target>  = <body> | <condition>)  ;
<source> -> <collector-name> (<iterator-list> | <body_or_condition>)  ;
<source> -> <collector-name> (<body_or_condition>)  ;
```

where <collector-name> is the name of the loop construct ñ for instance collectectSelect and collectselectOne.

```
list->collectselect(i;res= i.prop | not res.startswith("_")) ;
```

When iterating on property values, the following shorthand may be used:

```
list->propertyname[res | condition]
   // represents  list->collectselect(i;res:=i.propertyname | condition)
```

Hence the previous example may be rewritten as:

```
list->prop[res| res.startswith("_")];
```

The target variable may be omitted. The example can then be rewritten as:

```
list->prop[startswith("_")]) ;
```

The target variable may be omitted. The same example can then be written as:

```
list->prop[startswith("_")]) ;
```

If a property is invoked with "->" symbol and no "bracket" follows, this means that *xcollect* construct is being used.

```
list->prop;    // same as list->xcollect(i | i.prop), the target variable is implicit here
```

Also, if a list reference is accompanied by brackets with no previous "->" symbol, this means the xselect construct is being used.

```
list[condition];    // same as list->xselect(i; condition)
```

All these defined shorthand conventions used to collect property values equally applies to operations. However, the referred operation cannot be a pre-defined collection operation since this would conflict with regular OCL call of collection operators.

```
list->foo()[startswith("_")]) ; // same as list->collectselect(i; res= i.foo() | res.startswith("_"))
```

These shorthand conventions applies also to the *collectselectOne* and *selectOne* variants except a "!" symbol should prefix the brackets used to render the condition.

```
list->prop![startswith("_")]) ; // calling collectselectOne(i;res= i.prop | not res.startswith("_"))
```

8.2.2.8 SwitchExp

A *switch expression* is an imperative expression that is used to express alternatives that depend on conditions to evaluate. Semantically it behaves almost as nested OCL *if expressions*. However, there are two important differences: the switch expression is sensitive to interrupt expressions (break, continue, raise and return expressions) which may be invoked on an inner expression. Also, it extends the corresponding OCL concept by making non mandatory the else part.

Superclasses

`ImperativeExpression`

Associations

`alternativePart : AltExp {ordered} [*]`

The alternative parts. Each alternative consists of a condition and an expression to evaluate. The alternatives are evaluated in sequence until one condition succeeds.

`elsePart : Expresion [0..1]`

The expression to evaluate if all the conditions fail.

Notation

Two distinct notation styles are available for the same construct. One may use the traditional if-then-else notation ñ using a Java-like notation instead of OCL-like. The notation pattern is:

if (cond1) exp1
elif (cond2) exp2,
…
else (condN) expN
endif;

Tre endif **keyword** can be skipped since it is needed only when the expression composed with other expressions.

The alternative notation uses the **switch** keyword with the following syntax cpattern:

switch {
  (cond1)? exp1; // alt expression
  (cond2)? exp2;
  …
  else?: expn;
}

When the **switch** keyword is used, the "collect" shorthand with the arrow symbol convention is available:

list->**switch** (i) { … } // same as list->**xcollect**(i | **switch** { … })

Remark: The concurrent usage of OCL-like syntax "if exp then body endif" and java-like syntax "if (exp) body …" produces a grammar conflict in parsers that can be solved through look ahead.

8.2.2.9 AltExp

An *alternative expression* is an expression telling that some expression is to be executed if a condition holds. It returns the executed expression if the condition is satisfied, otherwise it returns `null`. It is semantically equivalent to an if expression with a `null` in the else clause. However, its offers a more direct representation of decisions written within a switch expression.

Superclasses

`ImperativeExpression`

Associations

`condition : OclExpression [1]`

The condition to evaluate.

`body : OclExpression [1]`

The expression to evaluate if the condition is satisfied

Notation

See the notation of `SwitchExp`.

8.2.2.10 VariableInitExp

A *variable initialization expression* represents the declaration and the initialization of a variable. This expression may either return the initialisation value or return null depending on the return mode being used (see `withReturn` property).

Superclasses

`ImperativeExpression`

Attributes

`withReturn : Boolean`

Indicates whether the initialization value is returned by this expression. If false `null` is returned

Associations

`referredVariable : Variable [1]`

The variable being declared. The variable is visible within the current scope. It starts to be visible after the declaration occurs. See the execution semantic section for the definition of scope rules.

Notation

The notation uses the **var** keyword. The initialization value is notated using ':=' if `withReturn` property is `false` or uses '::=' if `withReturn` is true.

var x : String := "abracadabra";
if (**var** x::="hello") **then** …

The type of the variable can be omitted as long as it can be derived from the initialization expression. A variable may not declare an initialization value. In this case a default value is assumed (an empty collection for a collection, zero for an numeric type, the empty string for a string and null for all other elements.

Multiple variable declaration may be grouped using a unique **var** keyword.

var x:= "", i:=0;

The "=" symbol can be used instead of the ":=" to initialize a variable.

8.2.2.11 AssignExp

An *assignment expression* represents the assignment of a variable or the assignment of a Property. In this description we refer to "target field" the referred variable or property. If the variable or the property is monovalued, the effect is to reset the target field with the new value. If it is multivalued, the effect is to reset the field or to append it depending on the `isReset` property. If the `value` filed is made of more than one expression, then the assignment is said to be a *compound assignment,* otherwise it is a *simple assignment*. An expression in a compound assignment is called a *contribution*.

For a simple assignment, if the right-expression is a collection, assigning the variable or the property means adding each of the items of the collection (additive semantics). Note that this is only valid for a multivalued target field. Duplicate elements are removed if the target field are Sets ñ this is the case for property elements. In addition null values are automatically skipped.

A *compound assignment* is equivalent to perform as much *simple assignments* as there are expressions. `Null` values are skipped.

An assignment may receive a *future* variable produced by a deferred resolve expressions (see `ResolveExp`). The effect is equivalent to receive a null value except that a side-effect action occurs in order to allow re-executing the assignment at the end of the transformation. See semantics details below.

An assignment expression returns the assigned value (but null if a future value is in right-hand side).

Superclasses

`ImperativeExpression`

Attributes

`isReset : Boolean`

Indicates for a multivalued target that the list is reset before executing the assignment.

Associations

`value : OclExpression [*]`

The expression to be evaluated in order to assign the variable or the property.

`left : OclExpression   [1]`

The left hand side expression of the assignment. Should reference a variable or a property that can be updated.

`defaultValue : Expression [0..1]`

The expression to compute a value in case the evaluation of the 'value' property returns `null`.

Constraints

A deferred assignment is a simple assignment

self.value->size()=1

Local references cannot be used for the value of the assignment.

Notation

The notation uses the **':='** symbol if `isReset` is `true` and the symbol **'+='** otherwise. Composite assignments are introduced by a list of expressions delimited by braces. The default value is introduced using the **default** keyword.

```
mysimpleproperty := "hello";
mymultivaluedproperty += object Node {…}; // additive semantics
mymultivaluedproperty ::= object Node {…}; // the list is reset and re-assigned
feature := {  // a composite assignment with two contributions
  self.attribute;
  self.operation;
}
```

8.2.2.12 UnlinkExp

An *unlink expression* represents an explicit removal of a value from a multivalued property link.

Superclasses

`ImperativeExpression`

Associations

`item : [1] OclExpression`

The object to be removed from the multivalued property.

`target : [1] OclExpression`

The target expression. It should evaluate to a Property.

Notation

The notation uses a call to the `unlink` "operation" where the source argument is the `target` and the first argument is the `item` to be removed.

feature.unlink(myattribute);

8.2.2.13 TryExp

A *try expression* catches possible exceptions raised by the target expression (the `body`). It provides the list of candidate exceptions and indicates the expression to be executed if catching of the exception occurs.

Superclasses

`ImperativeExpression`

Associations

`tryBody : OclExpression [*]`

The expression being under the control of exception handling.

`exceptBody : OclExpression [1]`

The expression to execute if an exception occurs.

`exception : Type [*]`

The list of exceptions that can be potentially catched.

Notation

The notation uses the **try** and **except** keywords.

try { expression1 } **except** (exception1,exception2) {expression2};

8.2.2.14 RaiseExp

A *raise expression* is an expression that produces an exception.

Superclasses

`ImperativeExpression`

Associations

`exception: Type [1]`

The exception being raised.

Notation

The notation uses the raise keyword with the exception name as body. The exceptions can be provided as simple strings.

myproperty := self.something **default raise** "ProblemHere";

8.2.2.15 ReturnExp

A *return expression* is used within an imperative operation to exit from the operation interrupting the normal control flow. If a value is indicated and if the operation declares a unique result, the value is assigned to the result parameter of the operation. If the operation declares more than one result parameter the value is assigned to the tuple representing the collection of results. This requires that the type of the value passed to the result expressions is a tuple.

Superclasses

`ImperativeExpression`

Associations

`value: OclExpression [0..1]`

The value to return from the operation.

Notation

The notation uses the **return** keyword used alone or accompanied with the value expression.

return;
return 1+1;

8.2.2.16 BreakExp

A *break expression* is used to stop prematurely an iteration over a list of expressions. It is used in the body of imperative loop expressions (*while* and *for* expressions). A break expression cannot be directly owned by a non imperative expression ñ like the side-effect free OCL iterate expression.

Superclasses

`ImperativeExpression`

Notation

The notation uses the **break** keyword alone.

8.2.2.17 ContinueExp

A *continue expression* is used, within an iteration over a list of expressions, to jump to the next iteration without executing the remaining expressions of the current iteration. It is used within the body of imperative loop expressions (*while* and *for* expressions). A break expression cannot be directly owned by a non imperative expression ñ like the side-effect free OCL iterate expression.

Superclasses

`ImperativeExpression`

Notation

The notation uses the **continue** keyword alone.

8.2.2.18 LogExp

A *log expression* is an expression used to print a log record to the environment. It is often used for debug. A log may only be sent when a condition holds. A log expression returns `null`.

Superclasses

`ImperativeExpression`

Attributes

`text: String`

The message to be include in the log record.

`level: Integer`

Indicates the level of the log. May be used to partition the log records. By default it is equal to zero.

Associations

`condition: OclExpression [0..1] {composes}`

An optional condition to check. No log record is produced if it evaluates to false.

`element: Element [0..1]`

A model element which textual representation is included in the log record. The textual representation is implicitly obtained by calling the pre-defined `who` operation (see, QVT Standard Library).

Notation

The notation uses the syntax of an operation call where the **log** keyword is the name of the operation. The parameters are in order: the message to print, the reference to the "responsible" element and the level. Only the first parameter is mandatory.

log ("property bob is null", result) **when** result.bob=null;

8.2.2.19 AssertExp

A *assert expression* is an expression that checks whether an assertion holds. If the assertion fails an error message is generated ñ possibly accompanied with a log record. If the assertion fails with *fatal* severity, the execution terminates with the exception `AssertionFailed`. In all other cases the expression returns `null`.

Superclasses

`ImperativeExpression`

Attributes

`severity: SeverityKind`

Indicates a severity level. Possible values are `warning`, `error` and `fatal`. The default is `error`.

Associations

`assertion: OclExpression [0..1]`

The condition to check.

`log: LogExp [0..1]`

The log record to generate when the assertion fails.

Notation

The notation uses the **assert** keyword. It may be followed by a severity indication (**warning** or fatal keywords) and by the log expression introduced by a **with** keyword.

assert result.bob<>null **with log**("non null 'bob' expected", result);
assert fatal typename<>"int" **with log**("type integer expected",typename);
assert warning name.startswith("_") **with log**("special character being used", name);

8.2.2.20 SeverityKind

The *severity kind* enumeration defines all possible levels of severity for errors raised by assertion expressions.

Enumeration values

`warning`

`error`

`fatal`

8.2.2.21 TupleExp

A *tuple expression* creates an anonymous tuple literal using the values and the types of the elements (see `AnonymousTupleType`).

Superclasses

`ImperativeExpression`

Associations

`element : OclExpression [1..*]`

The list of element to be packed as a tuple.

Notation

The notation uses the syntax of an operation call where the **tuple** keyword is the name of the operation. The number of arguments corresponds with the list of expressions to pack.

8.2.2.22 UnpackExp

A *unpack expression* unpacks an anonymous tuple by assigning a list of variables with the value of the tuple elements.

Superclasses

`ImperativeExpression`

Associations

`variable : Variable [1..*]`

The list of variable receiving the values of the tuple elements.

Notation

The notation is similar to an assignment expression where the list of variables is placed in the left hand side.

x, y , z := self.foo(); // assuming foo returns a tuple of three elements.

A list of variables may be declared and initialized with an unpack expression. This is a shorthand for declaring the variables separately and then assigning them through an unique unpack expression.

var (x, y , z) := self.foo(); // shorthand for : **var** x; **var** y; **var** z; x, y, z := self.foo();

8.2.2.23 InstantiationExp

An *instantiation expression* creates an instance of a class, invokes an initialization operation on the created object and returns the created object. The initialisation operation is either implicit, either explicitly given and has necessarily the name of the class. By default, an initialisation operation with no arguments exists for all classes.

An instantiation expression may indicate the MOF extent ñ represented by a variable - where the created instance will "live".

Superclasses

`ImperativeExpression`

Notation

An instantiation expression is notated using the **new** keyword followed by the name of the class.

```
mycolumn := new Column(n,t);  // invokes a Column::Column(String,String) operation.
     // n and t are variables representing a name and a type name
```

When provided the extent is notated by adding the variable name in brackets after the **new** keyword.

```
column := new [mymodel] Column(n,t);  // mymodel is the extent for the new instance.
```

When an instantiation expression is used as the body of a *for each expression* the following shorthand can be used:

```
column := self.attribute->new(a) Column(n,t);
     // equivalent code:
    //  column := self.attribute->forEach new(a) Column(n,t);
```

Type Extensions

In this section we define the extensions to the type systems as well as the literals that are associated with the new types. Figure 8.7 depicts the four new types which are: `TemplateParameterType`, `Typedef`, `DictionaryType`, `ListType` and `AnonymousTupleType`. The DictionaryType and ListType are two mutable extensions of Collection Types.

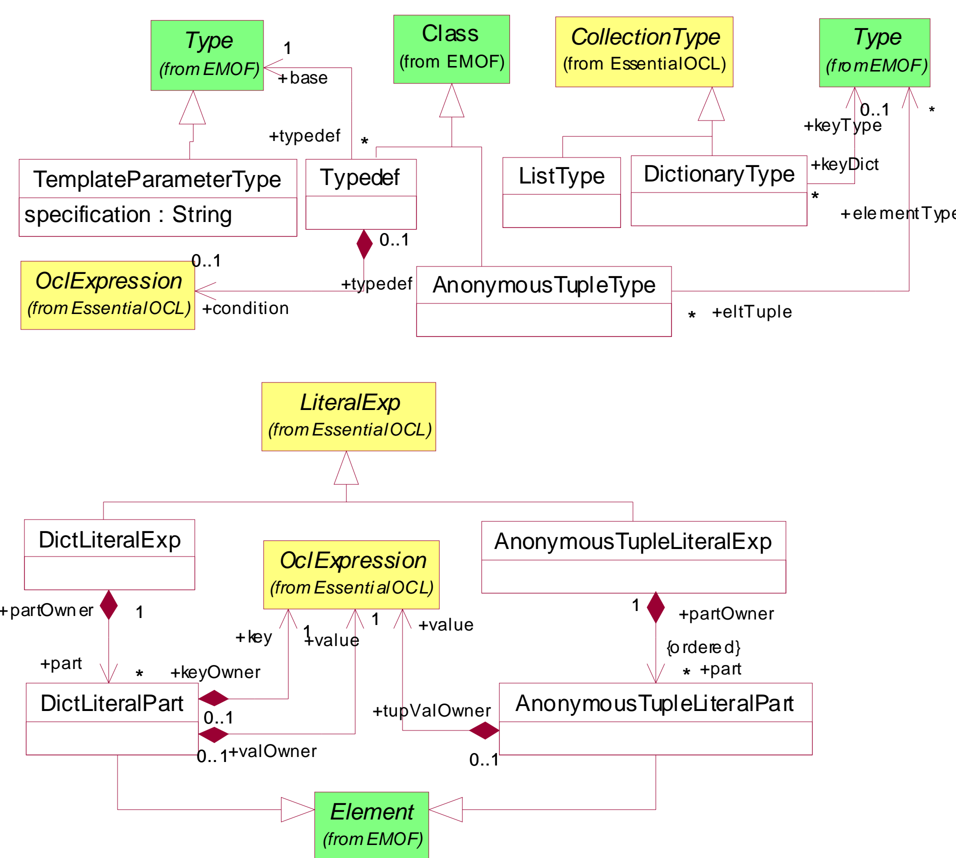


Figure 8.7 - Imperative OCL Package - Type extensions

8.2.2.24 Typedef

A *typedef* allows to constraint an existing type, called the base type. A typedef is never instantiated. An object of the base type can be checked to see whether it complies with the typedef. In the perspective of the runtime environment a typedef is never instantiated. When no condition is given a typedef simply serves as an alias to the base type. This is specifically used complex types likes tuples, collection and dictionary types.

When the base class is not a class a typedef allows attaching properties and operations to the type. This is specifically used in the QVT standard library to attach operations to pre-defined primitive types ñ like String and Collections.

Superclasses

`Class`

Associations

`base : Type [1]`

The referenced base type.

`condition: OclExpression [1]`

The expression defining the condition for a instance to comply with the typedef.

Notation

A `Typedef` is notated using the **typedef** keyword with the following syntax:

typedef MyType = MyBaseType [condition_expression];

The brackets delimiting the condition expression is omitted when the typedef is only used to rename an existing type:

typedef MyType = Tuple {x:X; y:Y};

8.2.2.25 ListType

A *list type* is a mutable parameterized collection type. It contains an ordered sequence of values. When a type for the element type is not provided Any is assumed.

A list of pre-defined operations on list is given in the QVT standard library.

Superclasses

`CollectionType`

8.2.2.26 DictionaryType

A *dictionary type* is a mutable type representing an hash-table data structure. For each key value in the table there is a unique value. The type of the key can only be a primitive type ñ such as integers and strings. A list of pre-defined operations on dictionaries is given in the QVT standard library.

A dictionary can be initialized through a literal and then freely updated (see `DictLiteralValue`).

Superclasses

`CollectionType`

Associations

`keyType : Type [0..1]`

The declared type for the key. By default the type is `String`.

`/elementType : Type [0..1] (from CollectionType)`

The declared type for the elements. By default the type is the special type `Any` (the singleton instance of the OCL `AnyType`).

Notation

A dictionary type is notated similarly to collection types except that the **Dict** type specifier is used. Also, two type informations can be indicated ñ the type of the keys and the type of the values.

The declaration below declares a variable of dictionary type.

var x:**Dict**(String,Actor);

If the types of the keys and the type of the values is not indicated, the default values is assumed (String and Any).

8.2.2.27 AnonymousTupleType

An anonymous tuple type is a type representing tuples that consist of ordered and anonymous elements. Anonymous tuples can be packed and unpacked using a single expression (see `TupleExp` and `UnpackExp`). Tuple elements can be accessed individually using the `at` pre-defined operation.

The value of a tuple may be given literally (see `AnonymousTupleLiteralExp`). A tuple is not mutable.

Superclasses

`Class`

Associations

`elementType : Type [*] {ordered}`

The types for each of the values of the tuple. The OCL Any type may be used.

Notation

An anonymous tuple type is notated using the **Tuple** type specifier. The definition below declares an anonymous tuple having an UML actor and a UML case instance.

var x: **Tuple**(Actor,UseCase);

This declaration can be distinguished from declaring "regular" named OCL tuples thanks to the absence of names.

var x: **Tuple**(a:Actor,u:UseCase); // declares a named tuple

8.2.2.28 TemplateParameterType

A *template parameter type* is used to refer to generic types in parameterized definitions. It is specifically used in defining the query operations associated with collections and dictionary types within the QVT Standard Library.

A *template parameter type* is usually named "T".

Note: The language does not provide means to write user-defined parameterized types. Hence the only parameterized types available are those defined by the QVT standard library.

Superclasses

`Type`

Attributes

`specification : String`

An uninterpreted opaque definition of the template parameter type.

8.2.2.29 DictLiteralExp

A *dictionary literal expression* is a literal definition for a dictionary. Each data stored in the dictionary consist of a key and a value (see `DictLiteralPart`).

Superclasses

`LiteralExp`

Associations

`part : OclExpression [1] {composes}`

The list of parts contained by this dictionary.

Notation

An dictionary literal expression is notated using the **Dict** type specifier followed by curly braces. The "=" symbol separates

var dic = **Dict**{'E' = 'EXPLICIT', 'I' = 'IMPLICIT'};

8.2.2.30 DictLiteralPart

A *dictionary literal part* is an element of a dictionary literal.

Superclasses

`Element`

Associations

`key : OclExpression [1] {composes}`

The key associated with this element.

`value : OclExpression [1] {composes}`

The value that corresponds to the key of this elment.

8.2.2.31 AnonymousTupleLiteralExp

An *anonymous tuple literal expression* is a literal definition for an anonymous tuple. Each tuple literal consist of an ordered list of tuple elements (see `TupLiteralPart`).

Superclasses

`LiteralExp`

Associations

`part : OclExpression [1] {composes}`

The list of parts contained by this dictionary.

Notation

An anonymous tuple literal is notated using the **Tuple** type specifier followed by curly braces containing the set of values.

The declaration below initializes a variable containing a list of tuples of type 'Tuple(String, Bag(Attribute))'.

var tuplist = self.ownedClass->collect(i|Tuple{i.name, i.ownedAttribute});

8.2.2.32 AnonymousTupleLiteralPart

An *anonymous tuple literal part* is an element of an anonymous tuple literal.

Superclasses

`Element`

Associations

`value : OclExpression [1] {composes}`

The expression defining the value of this element.

8.3 Standard Library

In this section we describe the additions to the OCL standard library.

The OCL standard library in an instance of the Library metaclass. This library is named `Stdlib` and is implicitly imported by all non black-box libraries or transformations.

8.3.1 Predefined types

MOF and OCL define pre-defined types that are usable at instance level (M1). We adopt here MOF terminology.

`Object` is a M1 type representing all types ñ including data types. The M1 Object type is an instance of the MOF::Element metatype.

`Element` is a M1 type represents class instances such as elements that are contained within a model. The M1 Element type is an instance of the MOF::Element metatype.

Some of the operations of the standard library use the `List` M1 type which is an instance of the `List(Object)` meta-type ñ the type parameter for the parameterized type `List(T)` is `Object`.

The sections below define the additional pre-defined M1 types that are specific to QVT.

8.3.1.1 Transformation

This M1 class named `Transformation` represents a base class for all instantiated OperationalTransformations. It is itself an instance of the `OperationalTransformation` metatype. It is used to define generic pre-defined operations available to any transformation instance.

8.3.1.2 Model

This M1 class named `Model` represents a base class for all instantiated ModelTypes. It is itself an instance of the `ModelType` metatype. It is used to define generic pre-defined operations available to any model parameter.

8.3.1.3 Status

This M1 class named `Status` contains information about the execution of a transformation. The M1 `Status` type is an instance of the `Class` metatype.

An *transform*() operation on a transformation returns a `Status` object.

8.3.2 Synonym types and synonym operations

All the OCL operations starting with "Ocl" prefix have a synonym operation in the QVT library where the prefix is skipped. For instance: isKindOf is a synonym of oclIsKindOf.

All the predefined OCL types M1 starting with "Ocl" prefix have a synonym type in the QVT library where the prefix is skipped.

The Ocl prefix in types and operations, can still be used but its usage not recommended since name conflicts can be avoided by qualifying with the name of the library (*StdLib*).

8.3.3 Operations on objects

The `Object` type represents all types. All MOF reflective operations applying on Objects are available.

8.3.3.1 repr

```
Object::repr() : String
```

Prints a textual representation of the object.

8.3.4 Operations on elements

All MOF reflective operations are available, in particular the Element::container() operation that is very useful to inspect a model.

8.3.4.1 _localId

```
Element::_localId() : String
```

Returns a local internal identifier for the instance (it uniquely identifies the instance in respect to the instance containing it).

8.3.4.2 _globalId

```
Element::_globalId() : String
```

Returns a global identifier for the instance ñ which identifies the instance in the used *model extent*.

8.3.4.3 metaClassName

```
Element::metaclassName() : String
```

Returns the name of the metaclass. It is an abbreviation for invoking the reflective Object::getMetaClass() method and retrieving the name.

8.3.4.4 subobjects

```
Element::subobjects() : List<Element>
```

Returns all immediate sub objects of an object.

8.3.4.5 allSubobjects

```
Element::allSubobjects(TypeType) : List<Element>
```

Returns all iteratively all sub objects of an object.

8.3.4.6 subobjectsOfType

```
Element::subobjectsOfType(TypeType) : List<Element>
```

Same as subobjects but filters according to the exact type.

8.3.4.7 allSubobjectsOfType

```
Element::subobjects(TypeType) : List<Element>
```

Same as allSubobjects but filters according to the exact type..

8.3.4.8 subobjectsOfKind

`Element::subobjectsOfKind(TypeType) : List<Element>`

Same as subobjects but filters according to the type.

8.3.4.9 allSubobjectsOfKind

`Element::subobjectsOfKind(TypeType) : List<Element>`

Same as allSubobjects but filters according to the type..

8.3.4.10 clone

`Element::clone() : Element`

Creates and returns a copy of a model element. Copy is done only at first level (sub objects are not cloned). References to non contained objects are copied only if the multiplicity constraints are not violated.

8.3.4.11 deepclone

`Element::deepclone() : Element`

Creates and returns a deep copy of a model element. Copy is done recursively on sub objects. References to non contained objects are copied only if the multiplicity constraints are not violated.

8.3.4.12 markedAs

`Element::markedAs() : Boolean`

This function has to be defined for each model type. It is used to check whether an object has been marked. For instance, for UML 1.4, this corresponds to accessing a UML::TaggedValue, while for a MOF model this corresponds to accessing a MOF::Tag.

8.3.4.13 markValue

`Element::markValue() : Object`

This function id used to return a the value associated with a mark. It is a virtual function, and should be defined specifically for each model type.

8.3.4.14 stereotypedBy

`Element::stereotypedBy(String) : Boolean`

This function id used to check whether an instance is "stereotyped". The definition depends on the metamodel.

8.3.5 Operations on models

The following operations can be invoked on any model.

8.3.5.1 objects

```
Model::objects() : Set(Element)
```

Returns the list of the objects in the model extent.

8.3.5.2 objectsOfType

```
Model::objectsOfType() : Set(Element)
```

Returns the list of the objects in the model extent that have the type given.

8.3.5.3 rootObjects

```
Model::rootobjects(): Set(Element)
```

Returns all the objects in the extent that are not contained by other objects of the extent.

8.3.5.4 removeElement

```
Model::removeElement (Element): Void
```

Removes an object of the model extent. All links that the object have with other objects in the extent are deleted.

8.3.5.5 asTransformation

```
Model::asTransformation(Model) : Transformation
```

Cast a transformation definition compliant with the QVT metamodel as a transformation instance. This is used to invoke on the fly transformations definitions created dynamically.

8.3.5.6 copy

```
Model::copy() : Model
```

Performs a complete copy of a model into another model. All objects belonging to the source extent are copied into the target extent.

8.3.5.7 createEmptyModel

```
static Model::createEmptyModel() : Model
```

Creates and initializes a model of the given type. This operation is useful when creating intermediate models within a transformation.

8.3.6 Operations on Transformations

In this section we provide the operations can be invoked on any transformation and the operation that can be invoked on Status instances (storing information on the execution of a transformation).

8.3.6.1 transform

```
Transformation::transform () : Status
```

Executes the transformation on the instance of the transformation class. Returns a Status object that can be checked using the Status::failed

8.3.6.2 parallelTransform

```
Transformation::parallelTransform () : Status
```

Executes the transformation on the instance of the transformation class. The operation returns immediately so that the invoking transformation can continue. Conceptually the transformation runs in parallel. The returned Status object can be used for synchronization (see *wait* operation).

8.3.6.3 wait

```
Transformation::wait (List(Status)) : Void
```

Waits for the termination of all transformations invoked in parallel. The Status objects are used to synchronize with the end of a transformation.

8.3.6.4 raisedException

```
Status::raisedException () : Class
```

Returns the exception raised by the transformation execution.

8.3.6.5 failed

```
Status::failed() : Boolean
```

Returns true if the transformation failed, false otherwise.

8.3.6.6 succeeded

```
Status::succeeded() : Boolean
```

Returns true if the transformation succeeded, false otherwise.

8.3.7 Operations on dictionaries

The following operations can be invoked on any dictionary. A dictionary type is a parameterized type. The symbol T denotes the type of the values and KeyT the type for the key.

8.3.7.1 get

```
Dictionary(KeyT,T)::get (k:KeyT) : T
```

Returns the value associated with the given key. The null value is returned is not present.

8.3.7.2 hasKey

```
Dictionary(KeyT,T)::hasKey (k:KeyT) : Boolean
```

Checks whether the dictionary has a value for the given key.

8.3.7.3 defaultget

```
Dictionary(KeyT,T)::defaultget (k:KeyT) : T
```

Returns the value associated with the given key or the given default value if the key does not exist in the dictionary.

8.3.7.4 put

```
Dictionary(KeyT,T)::put (k:KeyT, v:T) : Void
```

Assigns a value to a key.

8.3.7.5 clear

```
Dictionary(KeyT,T)::clear() : Void
```

Removes all values in the dictionary.

8.3.7.6 size

```
Dictionary(KeyT,T)::size() : Void
```

Returns the number of values stored in the dictionary.

8.3.7.7 values

```
Dictionary(KeyT,T)::size() : List(T)
```

Returns the list of values in a list. The order is arbitrary.

8.3.7.8 keys

```
Dictionary(KeyT,T)::size() : List(KeyT)
```

Returns the list of keys in a list. The order is arbitrary.

8.3.7.9 isEmpty

```
Dictionary(KeyT,T)::size() : List(KeyT)
```

Returns true if the dictionary is empty, false otherwise.

8.3.8 Operations on lists

All operations defined in OCL are available. In addition the following are available.

8.3.8.1 add

```
List<T>::add(T) :void
```

Adds a value at the end of the mutable list.

Synonym: **append**.

8.3.8.2 prepend

```
List<T>::prepend(T) :void
```

Adds a value at the beginning of the mutable list.

8.3.8.3 insertAt

```
List<T>::insertAt(T,int) :void
```

Adds a value at the given position.

8.3.8.4 joinfields

```
List<T>::joinfields(sep:String,begin:String,end:String) :String
```

Creates a string separated by sep and delimited with begin and end strings.

8.3.9 Operations on strings

All string operations defined in OCL are available. In addition we have:

String::format(value:Object) : String

- Print a message where the holes - marked with %s, %d, %f - are filled with the value, necessarily a tuple if more than a hole is defined. Formatting an object (with %s) implies the invocation of the *Any::repr*(). The format %d is used for integers, the %f is used for floats.

  A dictionary can be passed as the value. In this case the holes have the syntax "%(key)s" and are filled by inspecting the dictionary with the corresponding key.

String::size () : Integer

- The size operation returns the length of the sequence of characters represented by the object at hand.
- Synonym operation: length()

String::substringBefore (match : String) : String

- retrieves the substring that is before the matched string.

  - String::substringAfter (match : String) : String

  • retrieves the substring that is after the matched string.

  - String::toLower () : String

  • Converts all of the characters in this string to lowercase characters.

  - String::toUpper () : String

• Converts all of the characters in this string to uppercase characters.

- String::firstToUpper () : String

• Converts the first character in the string to an uppercase character.

- String::lastToUpper () : String

• Converts the first character in the string to a lowercase character.

- String::indexOf (match : String) : Integer

• Returns the index of the first character of the first substring if the substring provided in the parameter occurs as a substring in the object at hand. If it does not occur as a substring, the operation returns -1.

- String::endsWith (match : String) : Boolean

• Returns true if the string at hand ends with the substring provided in the parameter.

• Parameters
  .. match: the suffix to be searched for.

- String::startsWith (match : String) : Boolean

• Returns true if the string at hand starts with the substring provided in the parameter.

• o Parameters
  . match: the prefix to be searched for.

- String::trim () : String

• Returns a copy of the string where all trailing and leading white spaces have been removed. If there are no trailing or leading white spaces the operation returns the string.

- String::normalizeSpace() : String

• Removes all trailing and leading white space and replaces all internal sequences of white space with a single space.

- String::replace (String m1, String m2): String

• All occurrences of m1 in the context string are replaced by m2.

- String::equals (String match) : Boolean

• Returns true if the value of the String matches the input String ëmatchí, else return false

- String::equalsIgnoreCase (String match) : Boolean

• Returns true if the value of String, when ignoring letter casing matches the input String ëmatchí, else returns false.

- String::find (String match) : Integer

• Returns the position of the substring that starts with 'match'.

- String::rfind (String match) : Integer

• Returns the position of the substring that starts with 'match'. The search starts from the right.

- String::isQuoted (s:String) : Boolean

• Returns true if the string starts and ends with the "s" string .

- String::quotify (s:String) : String

• Adds the "s" string at the beginning and the end of the srings and returns it.

- String::unquotify (s:String) : String

• Removes the "s" string at the beginning and the end of the srings and returns the resulting string.

- String::matchBoolean (s:String) : Boolean

• Returns true if the string is "true", "false", "0" or "1". The method is not case sensitive".

- String::matchInteger (i:Integer) : Boolean

- Returns true if the string represents an integer.

- String::matchFloat (i:Integer) : Boolean

- Returns true if the string represents a float.

- String::matchIdentifier(s:String) : Boolean

- Returns true if the string represents an alphanumeric word.

- String::asBoolean() : Boolean

- Returns a Boolean value if the string can be interpreted as a string. Null otherwise.

- String::asInteger() : Boolean

- Returns a Integer value if the string can be interpreted as as integer. Null otherwise.

- String::asFloat() : Boolean

- Returns a Float value if the string can be interpreted as as float. Null otherwise.

- String::startStrCounter (String s) : Void

- Associates a counter to the string. Initializes the counter to zero.

- String::getStrCounter (String s) : Integer

- Returns the current value of the counter associated with the string.

- String::incrStrCounter (String s) : Integer

- Increments the current value of the counter associated with the string.

- String::restartAllStrCounter () : Void

- Restarts all the counters associated with strings.

- String::addSuffixNumber () : String

- Returns the string with a suffix that represents the value of the counter associated with this string. The counter is incremented. (see startStrCounter and incrStrCounter).

  This method is specifically used to generate internal names that are unique.

8.3.10 Operations on numeric types

- Integer::range (start,end) : List

- Returns a list of integers starting from 'start' position to 'end' position.

8.3.11 Predefined tags

proxy : Boolean

when present this tag indicates that the instance acts a proxy for a definition that is defined elsewhere. Used for Library and Transformations

alias : String

when marking a Class or a Property this tag provides a alternative name. This used in the concrete syntax to avoid name clashes.

topclasses : String

tag used to mark model types in order to provide the list of classes names ñ comma separated ñ that are valid as

types for the root objects of a model.

rememberChanges : Boolean

tag used to mark a MappingOperation, an ObjectExp or an AssignExp with an indication stating whether manual changes made within the properties assigned have to be restored when the transformation is executed twice.

manuallyChanged : Boolean

tag used to mark a Properties or Classes to indicate that a property or a class has been changed. A tool may automatically annotate a model with this information to implement an execution scenario that pre-serves manual changes.

8.4 Concrete Syntax

8.4.1 Files

A complete transformation specification can use various text files. A file declares the dependency on another file through an **import** statement. From the point of view of the QVT metamodel a file import statement has no representation. It is purely a concept of the textual syntax.

A file can contain the definition of the following *top entities*: transformations, libraries, model types and metamodels. If more than one element in this list is defined the import statement designates the entities that need to be visible in the importing file. This is done using the syntax:

from <filename> **import** <definition1>, <definition2>, …;

If the used file contains a unique transformation or a unique library definition, it is not always necessary for the importer file to contain an import statement. In effect; an **access** or an **extends** declaration in the header of a transformations implies looking for a file with the name of the accessed or extended module. In other words if the file exists, the import statement is implicit.

8.4.2 Comments

Three kinds of conventions are used to include comments in a text file.

1. Line comment delimited by "--" and the end of file.
2. Line comment delimited by "//" and the end of file.
3. Multi-line comments delimited by "/*" and "*/.

8.4.3 Shorthands used to invoke specific pre-defined operations

In this section we describe a list of shorthands that concern the invocation of some pre-defined operations.

1. The notation '#MyClass' involving the unary '#' operator is a shorthand for oclIsKindOf(MyClass)
2. The notation '##MyClass' involving the unary '##' operator is a shorthand for oclIsTypeOf(MyClass)
3. The notation '*"mystereotype"' involving the unary '*' operator is a shorthand for stereotypedBy("mystereotype"). Note that potential ambiguity with the integer/float multiply operation is solved thanks to the type of the argument (a string in our case).

4. The notation "blabla %s\n" % myvar involving the binary '%' operator is a shortand for invoking the pre-defined 'format' operation. The potential ambiguity with the modulo operator is solved thanks to the type of the first argument ñ which is a string in our case.
5. The binary operator "==" can replace the "=" comparison operator. Both alternatives should be available.
6. The binary operator "!=" can be used instead of the "<>" comparison operator. Both alternatives should be available.
7. The binary operator "+" can be used as a shorthand for the *concat* string operation.
8. The binary operator "+=" can be used as a shorthand for the *add* List operation.

8.4.4 Other language shorthands

We describe here additional shorthands conventions not described in the Notation sub-section of class descriptions because they do not apply to QVT metaclasses.

1. A string naming an enumeration can be used each time an enumeration value is expected. This implies that there is an implicit call to the *asEnumeration*() string operation.
2. Whenever a Boolean value is expected in an expression: (i) if an Element instance is passed there is an implicit comparison with the *null* value. (ii) if a collection value is passed there is an implicit comparison with the size of the collection.

8.4.5 Notation for metamodels

A model type may explicitly refer to a metamodel defined locally. To that end this specification defines a notation for describing MOF metamodels. The formal EBNF definition is given in 10.4.4.

A MOF Package is notated using the **metamodel** keyword followed by a name. The contents of the package are within curly braces.

metamodel SimpleUML { … }

The classes are notated using the **class** keyword followed by a name, an optional **extends** specification for class inheritance and a body delimited by curly braces.

class Class **extends** ModelElement { … }

An enumeration type is notated using the enum keyword followed by the list of enumeration values.

enum ParameterDirectionKind { "in", "inout", "out" }

The properties of a class are notated using a declarator made of: a name, a type, optional property qualifiers (derived, static, …) and a initialisation expression.

isStatic : Boolean = 0;

An operation declaration has a name, a list of formal parameters and a return type.

getAllBaseClasses() : Set(Class);

8.4.6 EBNF

We provide below a formal definition of the textual syntax in EBNF.

8.4.6.1 Syntax for module definitions

```
// start rule
<topLevel>  ::= ('import' <filename> ';' )* <file_definition>*
<import>    ::= 'from' <filename> 'import' <identifier> ';'
             |  'import' <filename>
<filename>  ::= <identifier>

// definitions in a source file
<file_definition>
            ::= <transformation>
              | <library>
              | <modeltype>
              | <metamodel>
              | <intermediate_class>
              | <property>
              | <query>
              | <constructor>
              | <entry>
              | <mapping_operation>
              | <tag>
              | <typedef>

<module_def>
          ::= <intermediate_class>
              | <property>
              | <query>
              | <constructor>
              | <entry>
              | <mapping_operation>
              | <tag>
              | <typedef>


// Transformations and library
<transformation> ::= <transformation_h> (";" | '{' <module_def> '}')
<library> ::= <library_h> (";" | '{' <module_def> '}')
// Transformations and library headers
<transformation_h> ::= 'abstract'? 'transformation' <identifier>
<trans_signature> module_usage?
    ('refines' <transf_usage> 'enforcing' <identifier>)? ';'
<library_h> ::= 'library' <identifier> lib_signature? module_usage?  ';'
<trans_signature> ::= '(' <param_list> ')'
<lib_signature> ::= '(' <id_list> ')'

// general purpose grammar rules
<declarator> ::= <direction_kind>?
             (<identifier>|<typespec>|<identifier>':'<typespec> )
             ('=' <expression>)?
<typespec> ::= <scoped_id>|<complextype>
<complextype> ::= ('Set','OrderedSet','Sequence','Bag','Dict','List')
    '(' <typespec> ')'
```

```
<direction_kind> ::= 'in' | 'inout' | 'out'
<param_list> ::= <declarator> (',' <declarator>)*
<id_list> ::= <identifier> (',' <identifier>)*
<scoped_id_list> ::= <scoped_id> (',' <scoped_id>)*
<scoped_id> ::= <identifier>  ('::'  <identifier>)*
<expression_list> ::= <expression> (';' <expression>) ';'?


// import of libraries and transformations
<module_usage> ::= ('access'|'extends') 'transformation'? transf_usage
                   | ('access'|'extends') 'library'? lib_usage ';'
<transf_usage> ::= <scoped_id>  ( '(' declarator ')' )?
<lib_usage> ::= <scoped_id> ( '(' id_list ')' )?

// model types compliance and metamodel declarations
<modeltype> ::= 'modeltype' <identifier> <compliance_kind>?
       'uses' <packageref> (',' packageref)*
       ('where' '{' <expression_list> '}' )? ';'

<packageref> ::= (<scoped_id> ( '(' <uri> ')' )? | <uri>)
<compliance_kind> ::= <STRING> // like: "strict" and "effective"
<uri> ::= <STRING>


// Syntax for defining explicitly metamodel contents
<metamodel> ::= 'metamodel' <identifier> '{' <class>* <tag>* '}'
<class> ::= 'class' <identifier> ('extends' <class>*)?
 '{' <class_feature_list>? '}'
<class_feature> ::= <class_property> | <class_operation>
<class_operation> 'operation' <scoped_id> '(' param_list? ')'
<class_property> ::= ('composes'|'derived')? <declarator>
     ('opposites' <identifier>)?
<class_property_list>::= <class_property> (';' <class_property>)* ';'?


// intermediate class definitions
<intermediate_class> ::= 'intermediate' <class>

// tags
<tag> ::= 'tag' <tagid> <scoped_id> ('=' <tagvalue>)? ';'
<tagid> ::= <STRING>
<tagvalue> :: <expression>

// typedefs
<typedef> :: 'typedef' <identifier> '=' <typespec>
                       ('[' <expression> ']')? ';'


// Syntax for properties
<property_key>+ <scoped_id> ':' <typespec>? ('=' <expression>)? ';'
<property_key> ::= ('intermediate' | 'derived'
   | 'literal' | 'configuration' | 'property')
```

```
// syntax for query operations
<query> ::= <query_decl> | <query_simple_def> | <query_compound_def>
<query_header> ::= ('query'|'helper') <scoped_id> '(' <param_list>?')'
                   ':' <param_list>
<query_decl> ::= 'blackbox'? <query_header> ';'
<query_simple_def> ::= <query_header> '=' <expression> ';'
<query_compound_def> ::= <query_header> '{' <expression_list> '}'

// syntax for constructors
<constructor> ::= <constructor_decl> | <constructor_def>
<constructor_header> ::= 'constructor' <scoped_id> '(' param_list? ')'
<constructor_decl> ::= 'blackbox'? <constructor_header> ';'
<constructor_def> ::= <constructor_header> '{' <expression_list> '}'

// syntax for entry operations
<entry> ::= "main" '(' param_list? ')'
                   (';' | '{' <expression_list> '}')

// syntax for mapping operations
<mapping_operation> ::= <mapping_decl> | <mapping_def>
<mapping_header> ::= 'abstract'? 'mapping' <direction_kind>?
   <scoped_id> '(' param_list? ')' : param_list
    mapping_composition* mapping_refinement?
    mapping_when? mapping_where?
<mapping_decl> ::= 'blackbox'? <mapping_header> ';'
<mapping_def> ::= <mapping_header> '{' <mapping_body>? '}'
<mapping_composition> ::= <composition_construct> <scoped_id_list>+
<composition_construct> ::= 'disjuncts' | 'merges' | 'inherits'
<mapping_when> ::= 'when' '{' <expression_list> '}'
<mapping_where> ::= 'where' '{ <expression_list> '}'
<mapping_body> ::= <mapping_section>* | <expression_list>
<mapping_section> ::= ('init'|'population'|'end')
     '{' <expression_list>? '}'
```

8.4.6.2 Syntax for the expressions

In this section we provide the syntax for the expressions. The syntax used extends the syntax of the OCL language (any navigation expression written in OCL can be expressed using the grammar provided by this section).

```
<expression> ::= <assignExp>
                 | <varInitExp>
                 | <letExp>
<assignExp> ::=
       impliesExp
       | unaryExp (':=' | '::=')
                      (<expression> | '{' <expression_list> '}' )
                      'default' <assignExp>
<impliesExp> ::= <inclusiveOrExp>
                 | <impliesExp> 'implies' <inclusiveOrExp>

<inclusiveOrExp> ::= <exclusiveOrExp>
```

```
                  | <inclusiveOrExp> 'or' <exclusiveOrExp>
<exclusiveOrExp> ::= <andExp>
                  | <exclusiveOrExp> 'xor' <andExp>

<andExp> ::= <equalityExp>
             | <andExp> 'and' <equalityExp>

<equalityExp> ::= <relationalExp>
                | <equalityExp> ('='|'=='|'<>') <relationalExp>
<relationalExp> ::= <additiveExp>
                 | <relationalExp> ('<','>','<=','>=') <additiveExp>
<additiveExp> ::= <multiplicativeExp>
                  | <additiveExp> ('+','-') <multiplicativeExp>

<multiplicativeExp> ::= <unaryExp>
                     | <multiplicativeExp> ('*','/','%') <unaryExp>

<unaryExp> ::= <postfixExp>
             | ('-','not','#','##','*') <unaryExp>

<postfixExp> ::= <primaryExp>
               | <postfixExp> '(' <argExp>* ')'
               | <postfixExp>
                   '!'? '[' (<declarator> '|')? <expression> ']'
               | <postfixExp> ('.'|'->'|'!->')
                   (<scoped_id> | <iteratorExp>
                    | <blockExp> | <ruleCallExp>)

<primaryExp> ::= <literal>
                  | <scopedId>
                  | <ifExp>
                  | <ruleCallExp>
                  | <quitExp>
                  | <tryExp>
                  | <raiseExp>
                  | <assertExp>
                  | <logExp>
                  | '(' <primaryExp> ')'


<literal> ::= <INT> | <FLOAT> | <STRING> | 'true' | 'false' | 'null'
             | ('Set' | 'OrderedSet' | 'Bag' | 'Sequence', 'Any')
               '{' <expression> (',' <expression>)* '}'
             | ('Tuple' '{' <declarator> (',' <declarator>)* '}'
<ifExp> ::= 'if' <condExpCS> then <ifBody>
            'else' <ifBody> 'endif'?
<ifBody>   ::=  <expression> | '{' <expression_list>? '}'
<iteratorExp> ::= ('reject', 'collect', 'exists', 'collect',
                   'forAll','xselect','xcollect',
                   'selectOne','collectOne')
          '(' declarator_list '|' <expression> ')'
                | 'iterate' '(' declarator_list ';' declarator
```

```
                                  '|' <expression> ')'
<blockExp> ::= (<objectExp>|<doExp>|<switchExp>)
<controlExp> ::= (<whileExp> | <computeExp> | <forExp>)
<objectExp> ::= 'object' (':' <modelref>)?
                <objectDeclarator> '{' <expression_list> '}'
<modelref> ::= <identifier>
<objectDeclarator> ::= (<identifier> ':')? <typespec>
                       | <identifier> ':'
<forExp> ::= ('forEach' | 'forOne')
     '(' <declarator> (';' <expression> )  ('|' <expression>)? ')'
      ( <expression> | '{' <expression_list>? '}')
<whileExp> ::= 'while' '(' <declarator> (';' <expression> ) ')'
        ( <expression> | '{' <expression_list>? '}')
<computeExp> ::= 'compute' '(' <declarator> ')' <expresión>
<doExp> ::= 'do' ('(' <declarator> ) ')')? '{' <expression_list>? '}'
<switchExp> ::= 'switch'  '(' <identifier> ) ')'
                 '{' <altExp>+ ('else' '?' <expression>)? '}'
<altExp> ::= '(' <expression> ')' '?' <expression> ';'

<ruleCallExp> ::= ('map' | 'xmap' | 'new' )
                  ('(' <declarator> ')')? <scoped_id>


<letExp> ::= 'let' <declarator_list> 'in' <expression>
<varInitExp> ::= 'var' <declarator> ('::=' | '=') <expression>
<quitExp> ::= 'break' | 'continue' | ('return' '(' <expression> ')')
<tryExp> ::= 'try' '{' expression '}'
          ('except' '(' scoped_id_list ')' '{ expression '}')+
<raiseExp> ::= 'raise' scoped_id ('(' arg_list ')')?
<assertExp> ::= 'assert' <expression> ( 'with' <logExp> )?
<logExp> ::= 'log' '(' <expression> ',' <expression> ')'
                 'when' <expressions>
```

9 The Core Language

9.1 Comparison with the Relational Language

The Core language supports pattern matching over a flat set of variables by evaluating conditions over those variables against a set of models. The Core Language is as powerful as the Relations language, though simpler. Consequently, the semantics of the Core Language can be defined more simply, though transformations described using the Core are more verbose. The Core Language may be implemented directly, or used as a reference for the semantics of Relations, which are mapped to the Core, using the transformation language itself, as described in Chapter 10.

As described in Chapter 7, the Relations Language implicitly creates trace classes and objects to record what occurred during a transformation execution. In the Core Language, these traces are explicit. By implication, the Core Language supports more class models that hold the trace than the implied class model of the Relations Language.

The following aspects, implied in the relational language, have to be defined explicitly when using the core language:

- The patterns that match against, and create the instances of the classes that are the trace objects of the transformation (e.g. the links between the transformed model elements).
- Atomic sets of model elements that are created or deleted as a whole, specified in smaller patterns than commonly specified in the Relational Language.

The trace classes (whose instances are the trace objects) are defined using MOF.

9.2 Transformations and Model Types

In the core language, a transformation is specified as a set of mappings that declare constraints that must hold between the model elements of a set of candidate models and the trace model. The candidate models are named, and the types of elements they can contain are restricted by a model type.

A transformation may be executed in one of two modes: *checking* mode or *enforcement* mode. In *checking* mode, a transformation execution checks whether the constraints hold between the candidate models and the trace model, resulting in reporting of errors when they do not. In *enforcement* mode, a transformation execution is in a particular direction, which is defined as the selection of one of the candidate models as the target model. The execution of the transformation proceeds by, first checking the constraints, and secondly attempting to make all the violated constraints hold by modifying only the target model and the trace model.

9.3 Mappings

A transformation contains mappings. A mapping has zero or more domains. A domain has an associated model type of the transformation. A domain does not have a name; it is uniquely identified by the mapping and the associated model type. Thus, in a transformation execution, each domain specifies a set of model elements of exactly one of the candidate models that is of interest to a mapping.

The following picture shows (informally) the structure of a mapping with two domains, one for model type L and one for model type R.

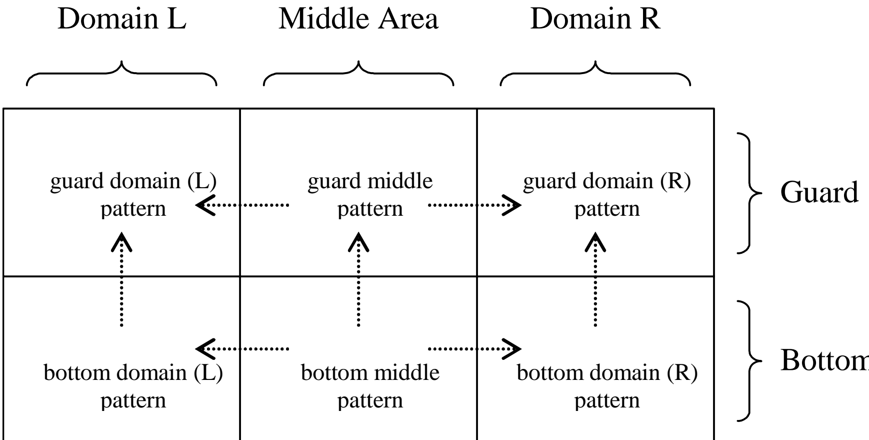


Figure 9.1 - Core Domains and Pattern Dependencies

Each rectangle in the above picture of a mapping represents a pattern. The columns in the picture are called areas. Each area consists of two patterns, the guard pattern and the bottom pattern. A mapping consists of one area for the trace (the middle area) and one area (a domain) for each model type. The domain areas consist of patterns that match the candidate models, the middle area consists of patterns that match the trace model.

A mapping defines a one-to-one relation between all bottom patterns of that mapping. This means that for any successful match of one of the bottom patterns there should exist exactly one successful match for each other bottom pattern. The one-to-one constraint between the bottom patterns is only checked or enforced if a successful match for each guard pattern of that mapping exists.

When a transformation is executed in checking mode, all the mappings of the transformation are executed, by matching the patterns, to check the one-to-one constraints.

When a transformation is executed in enforcement mode in the direction of a target model, each mapping is executed to enforce the one-to-one constraint. Firstly by matching the patterns (the same as in checking mode), secondly by enforcing the one-to-one constraint if it is violated. Enforcement will only cause changes to model elements of the trace model and the target model associated with the domain and its model type.

9.4 Patterns

A pattern is specified as a set of variables, predicates and assignments. Patterns can be matched and enforced. Matching of a pattern can result in value bindings of the variables (e.g. pattern instances), and enforcing of a pattern can result in model changes causing new value bindings for the variables during matching.

Patterns can depend on each other. A pattern that depends on another pattern may use the variables of that other pattern in its own predicates and assignments, and is matched using value bindings of the variables produced by a match of that other pattern. The dotted arrows in the picture below show the dependencies between the patterns, with the following interpretation:

More formally: If pattern C depends on pattern S, then C may use variables of S in predicates and assignments of C, and pattern C is always matched using value bindings of the variables produced by a match of pattern S.

In a mapping, bottom patterns depend on guard patterns (in the same column) and middle patterns depend on domain patterns (in the same row). In addition, dependencies between two patterns of two different domains (in the same row) also exist when one of the two associated model types is declared to depend on the other associated model type (see QVT Base in chapter 7). Dependencies between domain patterns (in the same row) are not shown in the above picture. Dependencies between model types are typically declared to define transformations between 3 or more models, where one model has model elements that refer to model elements of another model.

More formally: Only the following dependencies between patterns exist:

- A bottom pattern always depends on the guard pattern above it in the same area.
- A middle-pattern depends on all domain patterns on the same level (either between guard patterns or between bottom patterns).
- When a model type called A depends on another model type called B then each pattern of the domain that has model type A depends on the pattern, in the same mapping and on the same level, of the domain that has model type B.

9.5 Bindings

A match of a pattern results in zero or more valid bindings. A binding is a unique set of values for all variables of the pattern. A valid binding of a pattern is a binding where all the variables of the pattern are bound to a value other than *undefined*, and where all the predicates of the pattern evaluate to true. (The semantics of OCL define that (and how) OCL expressions of type Boolean evaluate to *false*, *true* or *undefined*.)

For example:

- A match of an empty pattern (no variables and no constraints) always results in one valid binding, with an empty set of values.
- A match of a pattern with one variable V and no constraints, always results in one valid binding per instance of the type of V in the model corresponding to the model type of the domain of the pattern.
- A match of a pattern with two variables V and W, and no constraints, results always in one valid binding per element of the cartesian product of all instances of the type of V and all instances of the type of W in the model corresponding to the model type of the domain of the pattern.
- A match of a pattern with two variables V and W of the same type, and one constraint that declares V and W should be equal (V=W), results always in one valid binding per element of the type of V in the model corresponding to the model type of the domain of the pattern.

A partial-valid binding is a binding where one or more variables are bound to a value other than *undefined* or one or more constraints evaluate to *true,* and no constraints evaluate to *false* (thus, either *true* or *undefined*). This implies that, any valid binding of a non-empty pattern is a partial-valid binding, and partial-valid bindings of empty patterns do not exist.

An invalid binding is a binding where at least one of the constraints evaluates to *false*.

9.6 Binding Dependencies

A pattern can depend on another pattern. Predicates in a pattern that depends on another pattern may refer to variables declared in that other pattern. Matching a pattern that depends on another pattern involves always the usage of one valid binding of the other pattern. This ensures that all variables have a value when evaluating the predicates.

More formally: If a pattern named C depends on patterns named S1 to Sn, then a valid binding of pattern C needs one valid binding for each pattern S1 to Sn. In other words: The match of pattern C takes place in the *context* of a set of valid bindings, one for each pattern S1 to Sn.

Binding dependencies commute: A valid combination of valid bindings of a set of depending patterns is a set of valid bindings, where each pattern has exactly one valid binding, and each pattern dependency has exactly one binding dependency between two valid bindings of that combination.

A partial-valid binding may depend on some other valid or partial-valid binding. However, each partial-valid binding must bind more variables or evaluate more constraints to *true* than the binding it depends upon.

9.7 Guards

Guards of a mapping narrow the selection of model elements to be considered for the mapping. The matching of bottom patterns takes place in the context of a valid combination of valid bindings of all the guard patterns. In such a combination, for each dependency between two guard patterns there must be exactly one dependency between two valid bindings of those two guard patterns.

Note that the guard patterns define neither constraints nor derivations on the transformed candidate and trace models. They are only used for defining a context in which constraints and derivations of the bottom patterns of a mapping can be computed.

9.8 Bottom Patterns

The bottom patterns of a mapping are the patterns that are checked and possibly enforced. A mapping declares essentially that all bottom patterns should relate one-to-one. That is, for each valid binding of one of the bottom patterns there must be exactly one valid binding for each other bottom pattern in that mapping. This implies that each valid binding of a bottom pattern may only be part of one unique valid combination of valid bindings for each bottom pattern.

Bottom patterns can have (in addition to variables and predicates) realized variables, assignments and black-box operations. These extra features can have side effects when executed in enforcement mode, and when a one-to-one constraint is violated. These features are used to change the trace model and the target model to create, or remove a valid-binding of a bottom pattern, to repair the one-to-one constraint. A description of these features follows.

Enforcement of the bottom patterns of a target domain and the middle area takes place when there is no valid match of these patterns for a given valid combination of valid bindings for all the guard and source domain bottom patterns.

During enforcement, all the side effect causing features of a bottom pattern are executed when a repair of a one-to-one constraint is necessary. This is different from the relational language, where enforcement of a valid binding of a pattern may result in altering only parts of the pattern.

9.9 Checking

Mappings can be checked, either as part of a transformation execution in checking mode or in the first step of a transformation execution in enforcement mode. A transformation execution in checking mode will produce an error for each violation of a mapping constraint. A transformation execution in enforcement mode will enforce a repair for each violation of a mapping constraint. The latter is described in the next paragraph called *enforcement*. In both cases we first have to check if there are any violations of the mapping constraints.

Domains may be *checkable* or not. Let us suppose that L and R are two different domains of the same mapping. Either L, or R, or both may be nominated as capable of being checked. Until now we assumed the typical situation in which both L and R are checkable.

If R is checkable and L is not checkable, this defines a one-to-one constraint between the bottom pattern of L and the middle bottom pattern, and a one-to-zero/one (i.e. 1:0..1) constraint between the bottom pattern of R and the middle bottom pattern. In other words, if there is a valid binding for the bottom pattern of L there must be one valid binding for the middle bottom pattern and one valid binding for the bottom pattern of R. If there is a valid binding for the bottom pattern of R then there does not have to be a valid binding for the bottom pattern of L nor for the middle bottom pattern.

If both R and L are checkable, this defines a one-to-one relation between the bottom pattern of L and the middle bottom pattern, and a one-to-one relation between the bottom pattern of R and the middle bottom pattern. In other words, if there is a valid binding for the bottom pattern of L there must be a valid binding for the middle bottom pattern and a valid binding for the bottom pattern of R. If there is a valid binding for the bottom pattern of R there must be a valid binding for the middle bottom pattern and a valid binding for the bottom pattern of L.

More generally and formally: There must be (exactly) one valid-binding of the bottom-middle pattern and (exactly) one valid binding of the bottom-domain pattern of a checked domain, for each valid combination of valid bindings of all bottom-domain-patterns of all domains not equal to the checked domain, and all these valid bindings must form a valid combination together with the valid bindings of all guard patterns of the mapping.

9.9.1 Checking formally defined

Following is a formal specification of the checking semantics using first order predicate logic.

- Each usage of `b1:P1` is read as "a binding `b1` resulting from a match of pattern `P1`".
- Each usage of `b1(b2,b3)` is read as "`b1` is a valid binding and `b1` uses binding `b2` and binding `b3`".

If domains L and R are given, and domain R is checked:

```
Forall  gl:Guard-L, gr:Guard-R, gm:Guard-Middle (
          ( gl() and gr() and gm(gl,gr) ) implies
          forall bl:Bottom-L (
                  bl(gl) implies
                      existsone bm:Bottom-Middle, br:Bottom-R (
                                  br(gr) and bm(bl,gm,br)
                      )
          )
)
```

If domains L, R and T are given, and domain R is checked, and the model type of L uses the model type of T:

```
forall              gl:Guard-L, gr:Guard-R, gt:Guard-T, gm:Guard-Middle (
          ( gl(gt) and gr() and gt() and gm(gl,gr,gt) ) implies
          forall bl:Bottom-L, bt:Bottom-T (
                  ( bl(gl,bt) and bt(gt) ) implies
                      existsone bm:Bottom-Middle, br:Bottom-R (
                              br(gr) and bm(bl,gm,br,bt)
                      )
          )
    )
```

If domains L and R are given and both are checked:

```
forall gl:Guard-L, gr:Guard-R, gm:Guard-Middle (
          ( gl() and gr() and gm(gl,gr) ) implies
          forall bl:Bottom-L (
                 bl(gl) implies
                    existsone bm:Bottom-Middle, br:Bottom-R (
                           br(gr) and bm(bl,gm,br)
                    )
          )
          and
          forall br:Bottom-R (
                 br(gr) implies
                    existsone bl:Bottom-L, bm:Bottom-Middle (
                           bl(gl) and bm(bl,gm,br)
                    )
          )
)
```

9.10 Enforcement

At execution time, one model type of the transformation (selecting all domains that have that model type) can be chosen as the enforcement direction (the target model). An enforcement direction thus designates the set of domains that associate with the target model type.

The target model and trace model may be changed to fulfil the (one-to-one) constraints of the mappings. The models are only changed when the constraints of a mapping are not fulfilled. The changes will lead either to the creation of new valid bindings or the removal of existing valid bindings of the target bottom patterns and the trace bottom patterns to enforce the constraints of a mapping.

A pattern contains variables, predicates and assignments. For a specific value binding of the variables, a predicate evaluates to *true*, *false* or *undefined*. Thus, predicates are only used for matching. For a specific value binding of the variables, assignments assign values to properties. These values are set to properties to repair a violation of a mappings constraint.

Assignments may be *default* assignments or not. Default assignments only assign values to satisfy the mapping constraints, they do not play a role during checking. A non-default assignment also plays the role of a predicate during checking, where the assignment operator (between the property and the value expression) is replaced by the equality operator.

Variables may be *realized* or not. When a variable is realized, a new instance of the type of that variable may be created or an existing value of that variable may be deleted. Non-realized variables are used only to bind values during matching.. Realized variables are used for both matching and enforcement.

Domains may be *enforceable* or not. Assignments and realized variables may only be defined in enforceable bottom-domain patterns and bottom-middle patterns. A non-enforceable domainís bottom pattern can not be enforced, instead only the checking semantics will apply.

If no valid binding of an enforceable target domain bottom pattern or middle bottom pattern exists and a valid binding of that pattern is required by the checking semantics, then new instances of the types of all unbound realized variables of that pattern are created and bound as values of those variables, and all assignments of the pattern are executed.

If a valid binding of an enforceable target domain bottom pattern exists and a valid binding of that pattern is required not to exist by the checking semantics, then all the properties of the assignments of the pattern are nullified and all the values of the realized variables of the pattern are deleted. A valid binding of the target pattern is required not to exist when the source domain is required to be checked and there does not exist a valid binding of the source and middle patterns in a valid combination with the valid binding of the target pattern.

9.10.1 Enforcement formally defined

Following is a formal specification of the enforcement semantics using first order predicate logic.

Each usage of `b1?(b2,b3)` is read as "`b1` is a partial-valid binding and `b1` uses binding `b2` and binding `b3`".

- Each usage of `b1>=b2` is read as "binding b1 is a superset of (or equal to) binding b2 (where `b1` and `b2` are bindings of the same pattern)".
- Each usage of `pre.b1` is read as "a binding `b1` existing before a transformation execution".
- Each usage of `post.b1` is read as "a binding `b1` existing after a transformation execution".

If domains L and R are given, and both are checked, and R is enforced:

```
forall gl:Guard-L, gr:Guard-R, gm:Guard-Middle (
          ( gl() and gr() and gm(gl,gr) ) implies
          forall bl:Bottom-L (
                 bl(gl) implies (
                        existsone post.bm:Bottom-Middle, post.br:Bottom-R (
                               post.bm(bl,gm,post.br) and post.br(gr) and
                               forall pre.bm:Bottom-Middle, pre.br:Bottom-R (
                                      ( pre.br?(gr) and pre.bm?(bl,gm,pre.br) )
                                   implies
                                   ( post.bm>=pre.bm and post.br>=pre.br )
                               )
                        )
                 )
          )
          and
          forall pre.br:Bottom-R (
              pre.br(gr) and (
                        not exists pre.bm:Bottom-Middle, bl:Bottom-L (
                               bl(gl) and pre.bm(bl,gm,pre.br)
              )
                 ) implies (
                    not exists post.br:Bottom-R, post.bm:Bottom-Middle (
                               ( post.br?(gr) and pre.br>=post.br )
                               or post.bm?(gm,pre.br)
                        )
                 )
          )
   )
```

If domains L and R are given, and R is checked and enforced:

```
forall gl:Guard-L, gr:Guard-R, gm:Guard-Middle (
```

```
        ( gl() and gr() and gm(gl,gr) ) implies
        forall bl:Bottom-L (
              bl(gl) implies (
                    existsone post.bm:Bottom-Middle, post.br:Bottom-R (
                          post.bm(bl,gm,post.br) and post.br(gr) and
                          forall pre.bm:Bottom-Middle, pre.br:Bottom-R (
                                ( pre.br?(gr) and pre.bm?(bl,gm,pre.br) )
                             implies
                             ( post.bm>=pre.bm and post.br>=pre.br )
                          )
                    )
              )
        )
  )
```

If domains L and R are given, and L is checked, and R is enforced:

```
forall gl:Guard-L, gr:Guard-R, gm:Guard-Middle (
        ( gl() and gr() and gm(gl,gr) ) implies
        forall pre.br:Bottom-R (
           pre.br(gr) and (
                    not exists pre.bm:Bottom-Middle, bl:Bottom-L (
                          bl(gl) and pre.bm(bl,gm,pre.br)
           )
              ) implies (
                 not exists post.br:Bottom-R, post.bm:Bottom-Middle (
                          ( post.br?(gr) and pre.br>=post.br )
                          or post.bm?(gm,pre.br)
                    )
              )
        )
  )
```

9.11 Realized Variables

A realized variable can be enforced to bind to a value, when enforcing the bottom pattern in which it is declared. A realized variable can be bound to a new value by creating a new instance of the type of the variable when enforcing a bottom pattern. A realized variable can also be nullified by deleting the instance that is the value of that variable. A realized variable has the same semantics as other variables when matching a pattern.

Creating and deleting instances of realized variable types are model changes that (as all other side effects) only take place when executing a transformation in enforcement mode, where a mapping has to be repaired to satisfy a one-to-one constraint.

9.12 Assignments

An assignment sets the value of a property of a target object, when enforcing the bottom pattern in which it is declared. An assignment has two associated OCL expressions and a referenced property. One of the OCL expressions, the slot expression, specifies the object whose property is to be assigned, the other OCL expression defines the value to be assigned.

An assignment is of two kinds, default and not default. Default assignments only play a role during the execution of a transformation in enforcement mode. Non-default assignments also play the role of predicates during the matching of a bottom pattern in both checking and enforcement modes.

A default assignment only sets the value of a property of an object identified by a slot expression when enforcing a target bottom pattern or a middle bottom pattern. A non-default assignment is also evaluated during the matching of a bottom pattern. If the value of the property of the object identified by the slot expression is equal to the value to be assigned, and all other predicates evaluate to true, then a valid binding is found. If the value of the property does not match the value to be assigned (or another predicate is violated), then the binding is not valid.

If an existing valid binding has to be deleted to repair the mapping constraint, then the properties of the objects identified by the slot expressions are nullified.

9.13 Enforcement Operations

In addition to realized variables and assignments, a mapping may also use black-box operations for enforcement of a domain. A domain-bottom pattern can contain operation call expressions specifically designated for the purpose of enforcement. An operation call is designated for a specific mode of enforcement: creation or deletion.

Creation semantics:

1. Check if a valid binding of the enforced domain or middle bottom pattern exists for a given valid binding of the guard and other non-enforced domain patterns.
2. If a valid binding does not exist, then execute all the enforceable elements in creation mode, i.e. create new instances of unbound realized variables, execute assignments, and invoke operations that are designated for the creation mode of enforcement of the domain.
3. Check again if a valid binding of the enforced domain pattern exists for the given valid binding of the guard and other non-enforced domain patterns. Checking is expected to succeed now and yield a valid binding (if the implementation is consistent with the mapping specification.) If the checking fails, raise a runtime exception.

Deletion semantics:

4. Check if there exists a valid binding of the enforced domain or middle bottom pattern that is required not to exist as per the checking semantics.
5. If such a binding exists, then execute all the enforceable elements in deletion mode, i.e. invoke operations that are designated for the deletion mode of enforcement, nullify properties of assignments, and delete realized variable values.

9.14 Mapping Refinement

Refinement of mappings is used to specialize mappings into more specific mappings. Refinement is similar to inheritance of features between class specializations.

To explain the semantics of refinement we need to define the meaning of pattern *correspondence* first: Each pattern in one mapping *corresponds* with a pattern in another mapping if they are both in a domain with the same associated model type or both in the middle area, and if they are both guard patterns or both bottom patterns.

If one mapping is a refinement of one or more other mappings, then all the features (variables, predicates, assignments etc.) of the patterns of the refined mappings are inherited in the corresponding patterns of the refinement, and none of the features of the patterns of the refined mappings can be overruled or removed.

To define the semantics of mapping refinement more formally, we will give the rewrite rules that will produce a new mapping that is semantically equal to, and derived from, the combination of the refined mapping and the refinement:

If a mapping named S refines a mapping named G, then the mappings S and G together are semantically equal to a mapping named R where each pattern of R has all the features of a corresponding pattern of G, extended with all the features of the corresponding pattern of S.

From the semantics of pattern matching we can now conclude: The constraints of the extended patterns in R are the conjunctions of the constraints of the corresponding patterns of S and G, applied over the union of the variables of the corresponding patterns of S and G.

9.15 Mapping Composition

Composition of mappings is used to define mappings that are checked or enforced in the context of a valid combination of valid binding for all bottom patterns of another mapping. In other words: the child mapping is matched in the context of a valid binding of the bottom-middle pattern (implying valid bindings for all other patterns) of the parent mapping.

Essentially a child mapping can be defined as a mapping whose guard patterns effectively inherit all the features (variables, predicates, assignments, etc.) of both the guard and the bottom pattern of the corresponding area (middle areas or domains associated with the same model type) of the parent mapping.

A child mapping is local to the parent mapping. They do not have a name and cannot be refined by other mappings. However, child mappings can be parents of other child mappings.

The semantics are defined more formally by giving rewriting rules that will produce a new mapping, in place of the child mapping, that is semantically equal to, and derived from, a given child mapping and its parent mapping:

If mapping C is composed by mapping P (P composes C) then the combination of C and P is semantically equal to the mappings R and P where the patterns of R have all the features of the corresponding patterns of C, and R has its guard patterns extended with all the features of the guard and bottom patterns of the corresponding area (domains with the same model type or middle areas) of P.

9.16 Functions

Functions (from the QVT Base package) can be declared in mappings. Functions are operations (as in EMOF) with an OCL expression as body. Since OCL expressions do not have side effects, functions do not have side effects either.

The parameters of the function are the variables that can be used in the body-expression. The type of the OCL expression should conform to the result type of the function.

Functions can be used in the predicates and assignments of the patterns of the mapping. If a mapping refines, or is a child of, another mapping, then a function that is defined in the refined, or parent mapping, can be used in the predicates and assignments of the refinement or child mapping.

9.17 Abstract Syntax and Semantics

In the following section we define the abstract syntax of the Core Language. The Core Language is declared the QVTCore Package that depends on the QVTBase Package, EMOF and the Expressions package of OCL. The QVTBase Package is described in chapter 9 of this document.

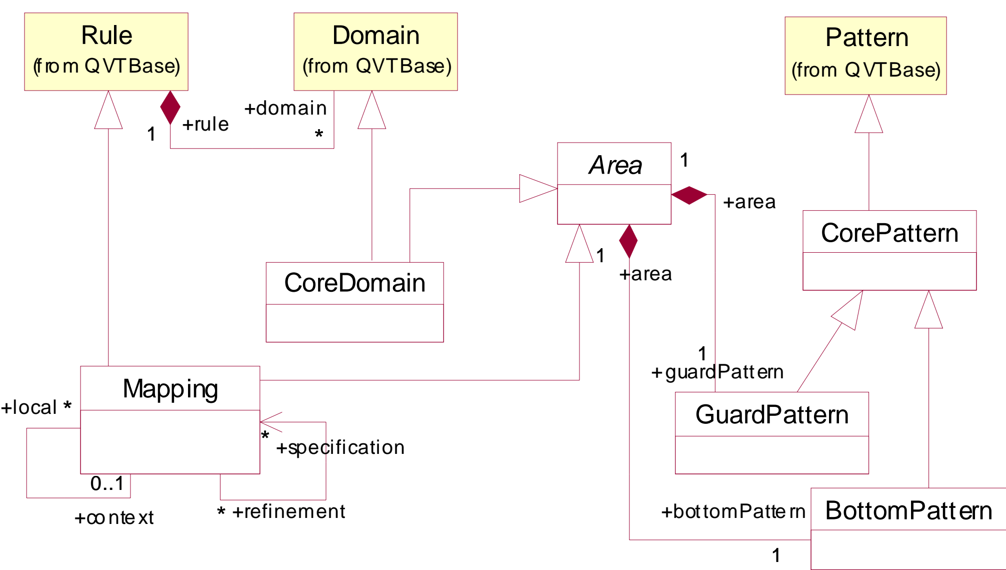


Figure 9.2 - QVTCore Package - Mappings and Patterns

9.17.1 CorePattern

A *core pattern* is specified as a set of variables, predicates and assignments. Core patterns can be matched and enforced. Matching a core pattern can result in value bindings of the variables (e.g. pattern instances), and enforcing a pattern can result in model changes causing new value bindings for the variables during matching.

A match of a core pattern results in zero or more valid bindings. A binding is a unique set of values for all the variables of the pattern. A valid binding of a pattern is a binding where all the variables of the pattern are bound to a value other than *undefined*, and where all the predicates of the pattern evaluate to true.

Core patterns are declared in *mappings*, and although there are different kinds of core patterns playing different roles in a mapping, all core patterns have the same matching semantics.

The way in which the core patterns are organized in a mapping (explained in the following class descriptions) implies *dependencies* between them. Predicates in a pattern that depends on another pattern, may refer to the variables declared in that other pattern. Matching a pattern that depends on another pattern involves always the usage of one valid binding of the other pattern. This ensures that all the variables have a value when evaluating the predicates.

Superclasses

`Pattern`

9.17.2 Area

An *area* is a pair of *core patterns,* comprising a *guard pattern* and a *bottom pattern*. A bottom pattern depends on its guard pattern in the same area. Thus, we may use variables declared in the guard pattern in predicates and assignments in the bottom pattern. The matching of the bottom pattern uses a valid binding established during the matching of the guard pattern (as well as other valid bindings, depending on which concrete subtype of `Area` is under discussion).

Associations

`guardPattern: GuardPattern [1] {composes}`

The core pattern which is evaluated as a guard to the bottom pattern.

`bottomPattern: BottomPattern [1] {composes}`

The core pattern which is evaluated if the guard pattern of the same area has a valid binding. The bottom pattern is matched using the value bindings of the variables of the guard pattern.

9.17.3 GuardPattern

A *guard pattern* is one of the two core patterns in an *area*. A guard pattern is the guard for the bottom pattern of the same area. This means that the bottom pattern is evaluated using the variable values of the valid binding of the guard pattern. The evaluation of a guard pattern does not have any side effects.

Superclasses

`CorePattern`

Associations

`area : Area [1]`

The area that owns the guard pattern.

9.17.4 BottomPattern

A *bottom pattern* can have (in addition to variables and predicates) *realized variables*, assignments and black-box operations. These features are used to change the models and to create, or remove a valid-binding of the bottom pattern in which they are defined. A description of these features is given in the following class descriptions.

Bottom patterns have side effects when executed in *enforcement* mode. Enforcing a bottom pattern for a target domain and the middle bottom pattern takes place when a constraint, that a mapping imposes, is violated. The imposed constraints of a mapping are defined in the description of the class `Mapping`.

During enforcement, all the side effect features of a bottom pattern are executed when a repair of a constraint is necessary. This is different from the relational language where enforcement of a valid binding of a pattern may result in altering only parts of the pattern

A bottom pattern is one of the two patterns declared in one area. A bottom pattern is matched or enforced using a valid binding of the guard pattern of the same area.

Superclasses

`CorePattern`

Associations

`assignment : Assignment [*] {composes}`

The assignments that specify assignment actions in enforcement mode. Some of the assignments may additionally specify equality predicates to be used in checking mode.

`enforcementOperation: EnforcementOperation [*] {composes}`

Black-box operations to enforce the bottom pattern in an opaque manner (not defined in a QVT language). See Section 11.13 for the semantics of enforcement with black-box operations.

`realizedVariable: RealizedVariable [*] {composes}`

Realized variables are the variables whose values may be created or deleted in order to enforce the bottom pattern.

9.17.5 CoreDomain

A *core domain* is an area that is associated with one *model type* (as defined for the class `Domain` in QVTBase). Patterns in a core domain are matched or enforced on the model elements of the candidate models of that model type. Each core domain of a mapping has one unique associated model type.

Domains may be *checkable* or not. If a domain is checkable, the bottom pattern of that domain is matched to check the constraint the mapping imposes. The imposed constraints of a mapping are defined in the description of the class `Mapping`.

Domains may be *enforceable* or not. A non-enforceable domainís bottom pattern can only be checked. Assignments and realized variables may only be defined in the bottom patterns of domains that are enforceable.

When a transformation is executed in enforcement mode, then one candidate model is selected as the target model. This also designates one core domain of each mapping that is associated with the model type of the target model as the target domain for the enforcement of that mapping. The candidate models corresponding to the other model types are not changed, because only the bottom patterns of the selected target core domains are possibly enforced.

A pattern of a domain has a dependency on a corresponding pattern (i.e. guard on guard and bottom on bottom) of another domain if the model type associated with the former has a dependency on the model type associated with the latter (see QVT Base in Chapter 7). Thus, in a mapping we may use the variables declared in the patterns of the domain associated with the used model type in the predicates and assignments of the patterns of the domain associated with the depending model type. The matching of the depending pattern uses a valid binding established during the matching of the used pattern. (Note that a bottom domain pattern also depends on the guard pattern in the same domain.)

Variables in the patterns of a core domain must always have types that are defined in the used packages of its associated model type.

Superclasses

`Domain`

`Area`

9.17.6 Mapping

A *mapping* has one middle area and zero or more *core domains*. The guard pattern of the middle area depends on all the guard patterns of all the domains. The bottom pattern of the middle area depends on all the bottom patterns of all the domains. Thus, we may use the variables declared in the patterns of all the domain areas in the predicates and assignments of the pattern of the middle area (provided that they are at the same level, i.e. both guard patterns or both bottom patterns). The matching of the middle pattern uses all the valid bindings established during the matching of all the domain patterns.

All the different dependencies between the patterns in one mapping commute (See Section 9.6 for semantics of binding dependencies.) For example, with respect to any domain of the mapping, matching of the bottom middle pattern uses a valid binding of its guard pattern, and a valid binding of the bottom pattern of that domain, and these two valid bindings in turn depend on the same valid binding of the guard pattern of that domain.

The constraint expressed by a mapping between its bottom patterns is only checked or enforced if a set of valid bindings, one for each guard pattern of that mapping, exists where all dependencies commute. A mapping essentially defines a one-to-one relation between all bottom patterns of the areas of that mapping. This means that for any successful match of one of the bottom patterns there should exist exactly one successful match for each other bottom pattern.

When a transformation is executed in checking mode, all the mappings of the transformation are executed, by matching the patterns, to check the one-to-one constraints.

When a transformation is executed in enforcement mode in the direction of a target model, each mapping is executed to enforce the one-to-one constraint, firstly by matching the patterns (the same as in checking mode), secondly by enforcing the one-to-one constraint if it is violated. Enforcement will only cause changes to model elements of the trace model (by the bottom middle patterns) and the target model (by the bottom patterns of the domains associated with the model type of the target model).

Superclasses

`Rule`

Associations

`domain : Domain [*] {composes} (From QVTBase)`

The domains which specify the patterns to match (and perhaps enforce) in the candidate models. All domains of a mapping must be of subtype `CoreDomain`.

`local : Mapping [*] {composes} (opposite end: context [0..1])`

The set of local mappings owned by this mapping that are evaluated in the context of this mapping as per composition semantics. Local mappings are only evaluated in the context of a set of valid bindings for all bottom patterns of the context mapping. (See Section 11.15 for semantics of composition.)

`specification : Mapping [*] (opposite end: refinement [*])`

A mapping can refine another mapping, which acts as its specification. This results in refinement semantics. The core patterns of a refinement mapping inherit the variables, predicates and assignments from the corresponding core patterns of the specification mapping. (See Section 11.14 for refinement semantics.)

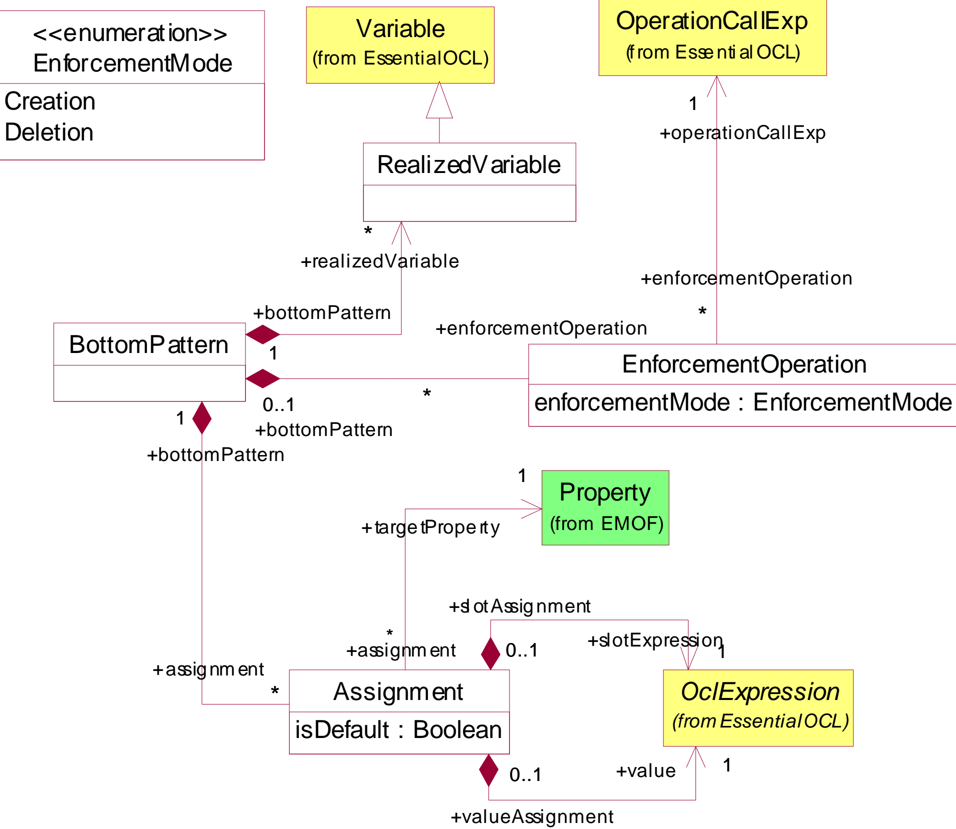


Figure 9.3 - QVT Core Package - Bottom patterns

9.17.7 RealizedVariable

A *realized variable* can be enforced to bind to a value, when enforcing the bottom pattern in which it is declared. A realized variable can be bound to a new value by creating a new instance of the type of the variable when enforcing a bottom pattern. A realized variable can also be nullified by deleting the instance that is bound as the value of that variable. A realized variable has the same semantics as other variables when matching a pattern.

Creating and deleting instances of realized variable types are model changes that (as all other side effects) take place only when executing a transformation in enforcement mode, where a mapping has to be repaired to satisfy a one-to-one constraint.

Superclasses

`Variable`

Associations

`bottomPattern : BottomPattern [*] {composes}`

The bottom pattern in which the realized variable is declared.

9.17.8 Assignment

An assignment sets the value of a property of a target object, when enforcing the bottom pattern in which it is declared. An assignment has two associated OCL expressions and a referenced target property. One of the OCL expressions, the slot expression, specifies the object whose property is to be assigned, the other OCL expression defines the value to be assigned.

An assignment is of two kinds, default and not default. Default assignments only play a role during the execution of a transformation in enforcement mode. Non-default assignments also play the role of predicates during the matching of a bottom pattern in both checking and enforcing mode.

A default assignment only sets the value of a property of an object identified by a slot expression when enforcing a target bottom pattern or a middle bottom pattern. A non-default assignment on the other hand is also evaluated during the matching of a bottom pattern. If the value of the property of the object identified by the slot expression is equal to the value to be assigned, and all other predicates evaluate to true, then a valid binding is found. If the value of the property does not match the value to be assigned (or another predicate is violated), then the binding is not valid.

If an existing valid binding has to be deleted, to repair the mapping constraint, then the properties of the objects identified by the slot expressions are nullified.

Attributes

`isDefault : Boolean`

Indicates whether the assignment is default assignment or a non-default assignment.

Associations

`slotExpression: OclExpression [1]`

An OCL expression identifying the object whose property value is to be assigned.

`value: OclExpression [1]`

An OCL expression specifying the value to be assigned to the target property. The type of the value expression must conform to the type of the target property.

`targetProperty: Property [1]`

The property whose value is to be assigned. The target property should be defined on the type of the slot expression.

`bottomPattern: BottomPattern`

The bottom pattern that contains the assignment.

9.17.9 EnforcementMode

The *enforcement mode* specifies the mode in which an operation that implements the enforcement semantics of a `mapping` is to be invoked. It is used when invoking an enforcement operation for enforcing the creation of a new valid binding of a bottom pattern or for enforcing the deletion of an existing valid binding of a bottom pattern.

Enumeration values

`Creation`

Specifies that the invoked operation is expected to alter the models to enforce a new valid binding of a bottom pattern.

`Deletion`

Specifies that the invoked operation is expected to alter the models to enforce the deletion of an existing valid binding of a bottom pattern.

9.17.10 EnforcementOperation

An *enforcement operation* is an OCL expression resulting in an *Operation Invocation* that plays a special role in the enforcement of a bottom pattern of the target domain or the middle area. It is analogous to an assignment, in that it only changes the model for valid bindings of the guard and opposite bottom patterns when the bottom pattern, to which it belongs, fails to satisfy the one-to-one constraint. The invoked operation has side-effects, which, after execution must result in a set of value bindings for variables for which all the predicates of the bottom pattern evaluate to true. It may be invoked in two modes: *Create* or *Delete*. In *Create* mode it must make objects of the types of the realized variables of the bottom pattern, and in *Delete* mode it must delete these model elements. In both modes it must also perform whatever other actions are required to fulfil the bottom pattern conditions.

Associations

`operationCallExp: OperationCallExp [1]`

An OCL Expression identifying an operation invocation and its parameters, which must include the variables which are defined in the owning bottom pattern.

`bottomPattern: BottomPattern [1]`

The bottom pattern in which the enforcement operation is declared.

9.18 Concrete Syntax

In this paragraph we define the concrete textual syntax for the QVT core language using the EBNF notation.

```
Transformation ::=
      "transformation" TransformationName "{"
            ( Direction";" )*
      "}"

Direction ::=
      DirectionName ["imports" PackageName("," PackageName)*]
                    ["uses" DirectionName("," DirectionName)*]

Mapping ::=
      "map" MappingName ["in" TransformationName] ["refines" MappingName] "{"
          ( ["check"] ["enforce"] DirectionName "("DomainGuardPattern")" "{"
                DomainBottomPattern
            "}" )*
            "where" "(" MiddleGuardPattern ")" "{"
            MiddleBottomPattern
            "}"
          ( ComposedMapping )*
      "}"

   ComposedMapping ::= Mapping

   DomainGuardPattern, MiddleGuardPattern ::= GuardPattern
```

```
   DomainBottomPattern, MiddleBottomPattern ::= BottomPattern

GuardPattern ::=
      [Variable(“,”Variable )* “|” ]
      ( Constraint “;” )*

BottomPattern ::=
      [ (Variable | RealizedVariable)
        (“,” ( Variable | RealizedVariable)* “|” ]
      ( Constraint “;” )*

Variable :=
      VariableName “:” TypeDeclaration

RealizedVariable :=
      “realized” VariableName “:” TypeDeclaration

   Constraint ::= Predicate | Assignment

   Predicate ::= BooleanOCLExpr

Assignement ::=
      [“default”] SlotOwnerOCLExpr“.”PropertyName “:=” ValueOCLExpr
```

10 Relations to Core Transformation

This Section provides the transformation that gives the Relations language its semantics in terms of the Core. The principles of the transformation are given first in an introduction. Then the full transformation specification is shown, and finally, the application of this transformation to the familiar object to relational transformation is shown to demonstrate the results.

10.1 Mapping Approach

In relations, transformation classes (or trace classes) are not explicitly specified and used. Instead, a relation directly specifies the relationship that should hold between source and target domains. Whereas in core, transformation classes and patterns over them (to query and instantiate transformation relations between source target model elements) are an essential part of the specification of mappings. Transformation classes and their instances are important for supporting efficient implementation of incremental execution scenarios, but avoiding them at specification level (as far as possible) makes a transformation writer's job easier. Since a relation is an assertion of a relationship that exists between source and target model elements, and a transformation class essentially serves to capture such assertions structurally, it is possible to derive transformation classes from relation specifications, and relation dependencies can be mapped to corresponding transformation class dependencies.

A relation ís when and where clauses map to the middle area of a mapping ñ the when clause maps to the middle-guard, and the where clause maps to the middle-bottom. A domain pattern maps to a domain area in the core ñ domain variables that occur in the when clause map to the domain-guard and the remaining domain pattern maps to the domain-bottom.

Relations can have arbitrary invocation dependencies. A relation can invoke multiple relations and a relation can be invoked by multiple relations. A relation can invoke another relation in its pre-condition (when clause) or post-condition (where clause). This style is intuitive to users and allows for complex composition hierarchies to be built. A relation invoking another relation over a set of values is semantically equivalent to a relation asserting the existence of another relation - i.e. asserting the existence of an instance of the corresponding transformation class with references to, or copies of, the values passed to the relation invocation. When mapping from relations to core we need to decompose relation invocation dependencies into simpler mapping dependencies - essentially each relation invocation chain needs to be broken down into its binary components (see the mapping rules below). Since relations are expressed in terms of patterns, relation dependencies can be translated to corresponding pattern dependencies.

Structural patterns of the domains of relations can be translated to equality constraints (or assignments depending on the “enforcement” specification) in the core.

Relation variables used in enforced domain patterns are mapped as realized variables in the bottom patterns of the corresponding core domain.

A relation domain can have a complex pattern consisting of multiple object nodes, properties and links. While translating to core an enforced relation domain’s complex pattern needs to be split into simpler patterns of multiple nested composed mappings. Naively putting the entire pattern in a single (core domain of a) core mapping can lead to duplicate object creations and unwanted object deletion-recreation cycles. This is because core has a simplified pattern matching semantics ñ during enforcement if a pattern does not match in its entirety then all the realized variables are freshly created irrespective of whether any of the objects already exist or not, and during deletion all the realized variables of a pattern binding are deleted irrespective of whether any of the objects are required in other valid pattern bindings or not. See Rule 4.3 below.

10.2 Mapping Rules

Trace class generation rule:

Rule 1:

Corresponding to each relation there exists a trace class in core. The trace class contains a property corresponding to each object node in the pattern of each domain of the relation. For example:

```
relation ClassToTable
{
  checkonly domain uml c:Class {namespace=p:Package {}, kind='Persistent',
name=cn}
  checkonly domain rdbms t:Table {schema=s:Schema {}, name=cn}
}

class TClassToTable
{
  c: Class;
  p: Package;
  t: Table;
  s: Schema;
}
```

For mapping to core we distinguish between two kinds of relations of a transformation: top-level relations and invoked relations. By a top-level relation we mean a relation that is not invoked from the where clause of any other relation; by an invoked relation we mean a relation that is invoked from the where clause of another relation.

A top-level relation maps to a single core mapping (perhaps with composed mappings), whereas an invoked relation maps to a separate mapping for each invoker-invoked combination.

For mapping to core we also distinguish between check-only relations and enforceable relations. A check-only relation maps to a single core mapping, whereas an enforceable relation may map to a composite hierarchy of core mappings.

Relation-to-mapping - common rule:

Rule 2:

The following are the common translation rules between a relation and a core mapping.

2.1: Variables of a RelationDomain that occur in the when clause become Variables of the core domain guard.

2.2: All other Variables of a RelationDomain become Variables of the core domain bottom pattern.

2.3: An instance variable corresponding to the trace class of the relation becomes part of the core mapping bottom pattern with its properties assigned to the corresponding core domain pattern variables.

2.4: A property template item in the relation domain pattern becomes an assignment (or an equality predicate in the case of check-only domains) in the core domain bottom pattern.

2.5: Predicates of the when clause become predicates of the core mapping guard.

2.6: Non RelationInvocation predicates of the where clause become predicates of the core mapping bottom.

2.6.1: RelationInvocation predicates of the where clause are ignored in this mapping, but reflected in the mapping corresponding to the invoked relation.

Top-level check-only relation to mapping:

Rule 3 (extends Rule 2):

3.1: A relation is 'checkonly' if it does not have any enforceable domains.

3.2: The only realized variable in the entire mapping is the trace class variable in the mapping bottom; there are no other realized variables in any of the mapping areas.

3.3: A property template item in a relation domain becomes an equality predicate in the core domain bottom.

3.4: A property template item in a relation domain that refers to a shared variable (i.e. a variable that also occurs in another domain) becomes an equality predicate in the mapping bottom.

3.5: Shared variables referenced in property template items of relation domains become variables of the mapping bottom. For example:

```
relation ClassToTable
{
  checkonly domain uml c:Class {namespace=p:Package {}, kind='Persistent',
name=cn}
  checkonly domain rdbms t:Table {schema=s:Schema {}, name=cn}
  when {
    PackageToSchema(p, s);
  }
  where {
    AttributeToColumn(c, t);
  }
}
```

becomes:

```
map ClassToTable in umlRdbms
{
  check uml (p:Package) {
    c: Class|
    c.namespace = p;
    c.kind = 'Persistent';
  }
  check rdbms (s:Schema) {
    t:Table|
    t.schema = s;
  }
  where (v1: TPackageToSchema| v1.p = p; v1.s = s;) {
    realize v2: TClassToTable, cn:String |
    v2.p := p;
    v2.s := s;
    v2.c := c;
    v2.t := t;
    c.name = cn;
    t.name = cn;
  }
}
```

Top-level enforceable relation to mapping:

Rule 4 (extends Rule 2):

4.1: A separate mapping is generated for each enforced domain of the relation.

4.2: In this mapping, only the enforced relation domain in question is marked as enforced in core; all its opposite domains are marked in core as checked at most (i.e. either left as they are or downgraded to checked if marked as enforced).

4.3: The enforced domain's pattern is decomposed into composed mappings as follows:

- root pattern object variable becomes a realized variable in the domain bottom pattern of the current mapping.
- all identifying property template items become assignments in the domain bottom pattern of the current mapping.
- all non identifying property template items of primitive type become assignments in the bottom pattern of a nested mapping.
- each non identifying property template item of object type results in a nested mapping which will have:
- a realized variable in the domain bottom, corresponding to the variable of the property value object.
- a property assignment from parent object variable to this variable in the domain bottom.
- and its own nested mappings recursively as described above.

4.4: Predicates of the where clause that refer to variables of the enforced domain are distributed down to the composed mappings as accumulated variable bindings become available in the nested mappings.

4.5: All other opposite domains are mapped to their respective core domain parts as described in Rule 3, i.e. their patterns are not decomposed down into nested mappings.

4.6: A black-box operational implementation, if any, that the relation has for the enforced domain becomes a pair of enforcement operations (one for creation and one for deletion) in the domain-bottom pattern, both pointing to the same operation call expression that takes its arguments from the variables corresponding to the root objects of the domains of the relation. For example:

```
key Table (name, schema); // key of class “Table”
key Key (name, owner);   // key of class “Key”; owner:Table opposite key:Key

relation ClassToTable
{
  checkonly domain uml c:Class {namespace=p:Package {}, kind='Persistent',
    name=cn, description=cd}
  enforce domain rdbms t:Table {schema=s:Schema {}, name=cn,
description=cd, key=k:Key {name=cn+'_pk'}}
  when {
    PackageToSchema(p, s);
  }
  where {
    AttributeToColumn(c, t);
  }
}
```

becomes:

```
map ClassToTable_rdbms in umlRdbms
{
  check uml (p:Package) {
    c: Class|
    c.namespace = p;
    c.kind = 'Persistent';
  }
  check enforce rdbms (s:Schema) {
    realize t:Table|
```

```
    t.schema := s;
  }
  where (v1: TPackageToSchema| v1.p = p; v1.s = s;) {
    realize v2: TClassToTable, cn:String, cd:String |
    v2.p := p;
    v2.s := s;
    v2.c := c;
    v2.t := t;
    cn := c.name;
    cd := c.description;
    t.name := cn;
  }
  map {
    where () {
      t.description := cd;
    }
  }
  map {
    check enforce rdbms () {
      realize k:Key|
      t.key := k;
    }
    where () {
      v2.k := k;
      k.name := cn+'_pk';
    }
  }
}
```

For example:

```
key Table (name, schema);
key Column (name, owner);   // owner:Table opposite column:Column
key Key (name, owner);   // key of class ëKeyí; owner:Table opposite key:Key

relation ClassToTable
{
  checkonly domain uml c:Class {namespace=p:Package {}, kind='Persistent',
name=cn}
  enforce domain rdbms t:Table {  schema=s:Schema {}, name=cn,
                                    column=cl:Column {
                                          name=cn+'_tid', type='NUMBER'},
                                    key=k:Key {name=cn+'_pk', column=cl}
                                  }
  when {
    PackageToSchema(p, s);
  }
  where {
    AttributeToColumn(c, t);
  }
}
```

becomes:

```
map ClassToTable_rdbms in umlRdbms
{
  check uml (p:Package) {
    c: Class|
    c.namespace = p;
```

```
    c.kind = 'Persistent';
  }
  check enforce rdbms (s:Schema) {
    realize t:Table |
    t.schema := s;
  }
  where (v1: TPackageToSchema| v1.p = p; v1.s = s;) {
    realize v2: TClassToTable, cn:String |
    v2.p := p;
    v2.s := s;
    v2.c := c;
    v2.t := t;
    cn := c.name;
    t.name := cn;
  }
  map {
    check enforce rdbms () {
      realize cl:Column|
      t.column := cl;
    }
    where () {
      v2.cl := cl;
      cl.name := cn+'_tid';
    }
    map {
      where () {
        cl.type := 'NUMBER';
      }
    }
  }
  map {
    check enforce rdbms (cl:Column) {
      realize k:Key|
      t.key := k;
      k.column := cl;
    }
    where (v2.cl = cl) {
      v2.k := k;
      k.name := cn+'_pk';
    }
  }
}
```

Invoked check-only relation to mapping:

Rule 5 (extends Rule 3):

5.1: An invoked relation maps to as many core mappings as the relations that invoke it, i.e. there exists a separate core mapping for each invoker-invoked pair.

5.2: The guard pattern of the mapping will have a variable corresponding to the trace class of the invoker relation, with equality predicates between root object variables of all the patterns of all the domains of the invoked relation and their corresponding properties in this trace class.

5.3: The root object variable of a relation domain's pattern becomes a pattern variable in the core domain guard (this is in addition to the variables that occur in the when clause as per rule 2.1).

For example:

```
relation ClassToTable
```

```
{
  checkonly domain uml c:Class {namespace=p:Package {}, kind='Persistent',
    name=cn}
  checkonly domain rdbms t:Table {schema=s:Schema {}, name=cn}
  when {
    PackageToSchema(p, s);
  }
  where {
    AttributeToColumn(c, t);
  }
}

relation AttributeToColumn
{
  checkonly domain uml c:Class { attribute=a:Attribute
                                        {  name=an,
                                            type=p:PrimitiveDataType {name=pn}
                                        }
                                   }
  checkonly domain rdbms t:Table {  column=cl:Column { name=an,
                                    type=sqltype}
            }
  where {
    sqltype = if (pn ='INTEGER') then 'NUMBER' else 'VARCHAR' endif
  }
}
```

becomes:

```
map AttributeColumn_ClassToTable in UmlRdbms
{
  check uml (c:Class) {
    a:Attribute, p:PrimitiveDataType|
    c.attribute = a;
    a.type = p;
  }
  check rdbms (t:Table) {
    cl:Column |
    t.column = cl;
    cl.type = sqlType;
  }
  where (v1:TClassToTable| v1.c = c; v1.t = t;) {
    realize v2:AttributeToColumn, an:String, pn:String, sqlType:String |
    v2.c := c;
    v2.a := a;
    v2.p := p;
    v2.t := t;
    v2.cl := cl;
    a.name = an;
    p.name = pn;
    cl.name = an;
    sqltype = if (pn ='INTEGER') then 'NUMBER' else 'VARCHAR' endif
  }
}
```

Invoked enforceable relation to mapping:

Rule 6 (extends Rule 4):

6.1: An invoked relation maps to as many core mappings as the relations that invoke it, i.e. there exists a separate core mapping for each invoker-invoked pair.

6.2: The guard pattern of the mapping will have a variable corresponding to the trace class of the invoker relation, with equality predicates between root object variables of all the patterns of all the domains of the invoked relation and their corresponding properties in this trace class.

6.3: The root object variable of a relation domain's pattern becomes a pattern variable in the core domain guard (this is in addition to the variables that occur in the when clause as per rule 2.1).

For example:

```
relation ClassToTable
{
  checkonly domain uml c:Class {  namespace=p:Package {},
                                   kind='Persistent',
                                   name=cn
                                 }
  enforce domain rdbms t:Table {schema=s:Schema {}, name=cn}
  when {
    PackageToSchema(p, s);
  }
  where {
    AttributeToColumn(c, t);
  }
}
relation AttributeToColumn
{
  checkonly domain uml c:Class {  attribute=a:Attribute
                                   {  name=an,
                                      type=p:PrimitiveDataType {name=pn}
                                   }
                                 }
  enforce domain rdbms t:Table {  column=cl:Column { name=an,
                               type=sqltype}
        }
  where {
    sqltype = if (pn ='INTEGER') then 'NUMBER' else 'VARCHAR' endif
  }
}

map AttributeColumn_ClassToTable_rdbms in UmlRdbms
{
  check uml (c:Class) {
    a:Attribute, p:PrimitiveDataType|
    c.attribute = a;
    a.type = p;
  }
  check enforce rdbms (t:Table) {
    realize cl:Column |
    t.column := cl;
  }
  where (v1:TClassToTable| v1.c = c; v1.t = t;) {
    realize v2:AttributeToColumn, an:String, pn:String, sqlType:String |
    v2.c := c;
    v2.a := a;
    v2.p := p;
    v2.t := t;
    v2.cl := cl;
    an := a.name;
```

```
    pn := p.name;
    cl.name := an;
    sqltype := if (pn ='INTEGER') then 'NUMBER' else 'VARCHAR' endif
  }
  map {
    where () {
      cl.type := sqlType;
    }
  }
}
```

10.3 Relational Expression of Relations To Core Transformation

```
// Rule 1: Corresponding to each relation there exists a trace class in core.
// The trace class contains a property corresponding to each object node in the
// pattern of each domain of the relation.
//
relation RelationToTraceclass
{
   checkonly domain relations r:Relation{name=rn,
                                         domain = rd:RelationDomain
                                           {
                                             pattern=t:ObjectTemplateExp
                                               {bindsTo=tv:Variable
                                                   {name=vn, type=c:Class {}}
                                               }
                                            }
                                         }
   enforce domain core rc:Class {  name='T'+rn,
                                   ownedAttribute=a:Property {name=vn, type=c}
                                }
   where {
      SubTemplateToTraceClassProps(t, rc);
   }
}

relation SubTemplateToTraceClassProps
{
   checkonly domain relations t:ObjectTemplateExp
                      { part=tp:ObjectTemplateExp
                              { bindsTo=tv:Variable
                                 { name=vn, type=c:Class {}}
                              }
                      }
   enforce domain core rc:Class {ownedAttribute = a:Property {name=vn, type=c}}
   where {
      SubTemplateToTraceClassProps(tp, rc);
   }
}

// For mapping to core we distinguish between two kinds of relations of a transformation:
//   - top-level relations and invoked relations.
// Top-level relations are not invoked by any other relation in the
// transformation.
// There exists a single mapping (with perhaps contained mappings) for a top-
// level relation, whereas for an invoked relation there exists a separate
// mapping for each invoker-invoked combination.

// For mapping to core we also distinguish between check-only relations and
// enforceable relations. A check-only relation maps to a single core mapping,
// whereas an enforceable relation typically maps to a composite hierarchy of
// mappings in core.
```

```
//

// Rule 2:
// The following are the common translation rules between
// a relation and a core mapping.
// 2.1: Variables of a RelationDomain that occur in the when clause become
// PatternVarables of the core domain guard.
// 2.2: All other Variables of a relationDomain become PatternVars
// of the core domain bottom pattern.
// 2.3: An instance variable corresponding to the trace class of the relation
// becomes part of the core mapping bottom pattern with its properties
// set(assigned or equated) to the corresponding core domain pattern variables.
// 2.4: A property template item in the relation domain pattern becomes an
// assignment (or equation in the case of check-only domains) in the core domain
// bottom pattern.
// 2.5: Predicates of the when clause become predicates of the core mapping
// guard.
// 2.6: Non relation invocation predicates of the where clause become predicates
// of the core mapping bottom.
// 2.6.1: relation invocation predicates of the where clause are ignored in this
// mapping, but are reflected in the mapping corresponding to the invoked
// relation.
//

// All Object template expressions (at the top level of the DomainPattern)
// become assignments in the core domain bottom. Nested
// ObjectTemplateExpressions become assignments in composed mappings.
//

// Rule 3 (extends Rule 2):
// 3.1: A relation is 'check-only' if it does not have any enforceable domains.
// 3.2: Only the trace class variable in the mapping bottom is a
// RealizedVariable;
// there are no other realized variables in any of the mapping areas.
// 3.3: A property template item in a relation domain becomes an equation in the
// core domain bottom.
// 3.4: A property template item in a relation domain that refers to a shared
// variable becomes an equation in the mapping bottom.
// 3.5: Shared variables referenced in property template items of relation
// domains become variables of the mapping bottom.
//
relation TopLevelRelationToMappingForChecking
{
   allDomainVars: Set(Variable);
   sharedDomainVars: Set(Variable);
   unsharedWhenVars: Set(Variable);
   domainVarsSharedWithWhen: Set(Variable);
   domainBottomUnSharedVars: Set(Variable);
   domainBottomSharedVars: Set(Variable);

   checkonly domain relations r:Relation
                     {  name=rn,
                        when=whenp:GuardPattern
                          {  variable=whenVars:Set(Variable) {}
                          },
                        where=wherep:BottomPattern {},
                        domain=rd:RelationDomain
                          { isChecked=c,
                            modelExtentName=dir:TypedModel {},
                            pattern=dp:DomainPattern
                               {
                                 variable=domainVars:Set(Variable){},
                                 templateExp=te:ObjectTemplateExp {}
                               }
                          }
                     } { r.module.transformation->
```

```
                               exists(t | t.topLevel->includes(r));
                         not r.domain->exists(d| d.isEnforced = TRUE);
                       }
    enforce domain core m:Mapping
                    { name=rn,
                      guard=mg:GuardPattern
                          { variable=mgVars:Set(RealizedVariable){}
                          },
                      bottom=mb:BottomPattern
                          { realizedVariable=vs:Set(RealizedVariable)
                              {  tcv:RealizedVariable
                                    {
                                      name=rn+'_v',
                                      type=tc}
                                    },
                                 variable = mbVars
                               },
                           }
                     }
       domain=md:CoreDomain {isChecked=c,
                             modelExtentName=dir,
                             guard=dg:GuardPattern {variable=
                                        dgVars:Set(Variable) {}},
                             bottom=db:BottomPattern {variable=
                                        dbVars:Set(Variable) {}}
                  }
    }
    when {
       RelationToTraceClass(r, tc);
    }
    where {
       allDomainVars = r.domain->iterate(d; vars: Set(Variable) = {} |
             vars->union(getVars(d)) )
       sharedDomainVars = r.domain->iterate(d; vars: Set(Variable) = {} |
             if vars->isEmpty() then vars->union(getVars(d))
                                else vars->intersection(getVars(d)));
       unsharedWhenVars = whenVars->minus(allDomainVars);
       domainVarsSharedWithWhen = domainVars->intersection(whenVars);
       domainBottomUnSharedVars =
            domainVars->minus(whenVars)->minus(sharedDomainVars);
       domainBottomSharedVars =
            domainVars->minus(whenVars)->intersection(sharedDomainVars);

       RVarSetToMVarSet(unsharedWhenVars, mgVars);
       RWhenPatternToMGuardPattern(whenp, mg);
       RVarSetToMVarSet(domainVarsSharedWithWhen, dgVars);
       RVarSetToMVarSet(domainBottomUnSharedVars, dbVars);
       RVarSetToMVarSet(domainBottomSharedVars, mbVars);
       RDomainPatternToMDBottomPattern(te, sharedDomainVars, db, mb);
       RDomainVarsToTraceClassProps(domainVars, tcv, mb);

       // Only non relation invocation predicates are copied from where
       // clause to mapping bottom.

       RSimplePatternToMPattern(wherep, mb);
    }
}

// Rule 4 (extends Rule 2):
// 4.1: A separate mapping is generated for each enforced domain of the relation.
// 4.2: In this mapping only the enforced domain in question is marked as enforced in core;
// all its opposite domains are marked in core as checked at most (i.e.
// either left as they are or downgraded to checked if marked as enforced).
// 4.3: The enforced domain's pattern gets decomposed into nested mappings as
// follows:
```

```
//         - root pattern object variable becomes a realized variable in
//           the domain bottom  pattern of the current mapping.
//         - all identifying property template items become assignments in the
//           domain bottom pattern of the current mapping.
//         - all non identifying property template items of primitive type become
//           assignments in the bottom pattern of a nested mapping.
//         - each non identifying property template item of object type results
//           in a nested mapping which will have:
//            - a realized variable in the domain bottom, corresponding to the
//              variable of the  property value object.
//            - a property assignment from parent object variable to this
//                variable in the domain bottom.
//            - and its own nested mappings as above recursively.
// 4.4: Predicates of the where clause that refer to variables of the enforced
// domain get distributed down to the nested mappings as variable bindings
// accumulate in the nested mappings.
// 4.5: all other opposite domains are mapped to their respective core domain
// parts as described in Rule 3, i.e. their patterns are not decomposed down into
// nested mappings.
// 4.6: A black-box operational implementation, if any, that the relation has for
// the enforced domain becomes a pair of enforcement operations (one for creation
// and one for deletion) in the domain-bottom pattern, both pointing to the same
// operation call expression that takes its arguments from the variables
// corresponding to the root objects of the domains of the relation.
//
relation TopLevelRelationToMappingForEnforcement
{
   allDomainVars: Set(Variable);
   oppositeDomainVars: Set(Variable);
   sharedDomainVars: Set(Variable);
   ownRdVars: Set(Variable);
   predicatesWithVarBindings: Set(Predicate);
   predicatesWithoutVarBindings: Set(Predicate);
   unsharedWhenVars: Set(Variable);
   domainTopVars: Set(Variable);

   checkonly domain relations r:Relation {name=rn,
      when=whenp:GuardPattern {variable=whenVars:Set(Variable) {}},
      where=wherep:BottomPattern
             {  predicate=rpSet:Set(Predicate){}},
                domain = rds:Set(relationDomain)
                   {  rd:RelationDomain
                         {  isEnforced = TRUE,
                            modelExtentName=dir:TypedModel {},
                            variable=domainVars:Set(Variable) {},
                            pattern=dp:DomainPattern
                              {  templateExp=te:ObjectTemplateExp
                                  {
                                       bindsTo=tev:Variable {}
                                  }
                              }
                         } ++ rOppositeDomains
                    }
              }
   enforce domain core m:Mapping
         {  name=rn+'_'+dir.name,
            guard=mg:GuardPattern {variable=mgVars:Set(Variable) {}},
            bottom=mb:BottomPattern
                {  realizedVariable=tcv:RealizedVariable
                      {  name=rn+'_v',
                         type=tc
                      },
                   predicate=predicatesWithVarBindings   // copy?
                }
            domain = cds:Set(CoreDomain)
                    {  md:CoreDomain
                         {
```

```
                            isEnforced=TRUE,
                            modelExtentName=dir,
                            guard=dg:GuardPattern
                                 {variable=dgVars:Set(Variable) {}},
                            bottom=db:BottomPattern
                                 {  realizedVariable=
                                     mtev:RealizedVariable
                                 }
                          } ++ mOppositeDomains
                     }   //TODO: add var only if tev not in whenVars
        }
   when {
      RelationToTraceClass(r, tc);
   }
   where {
      allDomainVars = r.domain->iterate(d; vars: Set(Variable) = {} |
                                          vars->union(getVars(d)) );
      oppositeDomainVars = rOppositeDomains->
            iterate(d; vars: Set(Variable) = {} |
                                          vars->union(getVars(d)) );
      sharedDomainVars = r.domain->
           iterate(d; vars: Set(Variable) = {} |
                   if vars->isEmpty() then vars->union(getVars(d))
                                      else vars->intersection(getVars(d)));
      ownrdVars = domainVars->minus(sharedDomainVars)->minus(whenVars);
      predicatesWithVarBindings =
              filterOutPredicatesThatReferToVars(rpSet, ownrdVars);   //TODO
      predicatesWithoutVarBindings = rpSet->minus(predicatesWithVarBindings);
      unsharedWhenVars = whenVars->minus(allDomainVars);
      domainTopVars = domainVars->intersection(whenVars);

      RVarSetToMVarSet(unsharedWhenVars, mgVars);
      RWhenPatternToMGuardPattern(whenp, mg);
      RVarSetToMVarSet(domainTopVars, dgVars);
      RVarToMVar(tev, mtev);
      RDomainToMDBottomForEnforcement(te, dir, predicatesWithoutVarBindings,
                                      ownrdVars, whenVars, sharedDomainVars,
                                      m, db, mb, tcv);
      RDomainVarsToTraceClassProps(oppositeDomainVars, tcv, mb);
      TROppositeDomainsToMappingForEnforcement(whenp, rOppositeDomains,
                                               sharedDomainVars,
                                               mOppositeDomains, mb);
      RRelImplToMBottomEnforcementOperation(r, rd, mb);
   }
}

// Rule 5 (extends Rule 3):
// 5.1: an invoked relation maps to as many core mappings as the relations that
// invoke it.
//   i.e. there exists a separate core mapping for each invoker-invoked pair.
// 5.2: The guard pattern of the mapping will have a variable corresponding to
// the trace class of the invoker relation, with root object variables of all the
// patterns of all the domains of the invoked relation being equated with '
// corresponding properties of this trace class .
// 5.3: The root object variable of a relation domain's pattern becomes a pattern
// variable in the core domain guard (this is in addition to the variables that
// occur in the when clause as per rule 2.1).
//
relation InvokedRelationToMappingForChecking
{
   allDomainVars: Set(Variable);
   sharedDomainVars: Set(Variable);
   unsharedWhenVars: Set(Variable);
   domainVarsSharedWithWhen: Set(Variable);
   domainBottomUnSharedVars: Set(Variable);
   domainBottomSharedVars: Set(Variable);
```

```
checkonly domain relations r:Relation
    {  name=rn,
       invokedBy= ri:RelationInvocation
          { predicate= p:Predicate
              {  pattern=pt:BottomPattern {relation=ir:Relation {name=irn}}
              }
          }
       when=whenp:GuardPattern
          {  variable=whenVars:Set(Variable) {}},
       where=wherep:BottomPattern {},
       domain=rd:RelationDomain
          {  isChecked=c,
             modelExtentName=dir:TypedModel {},
             pattern = dp:DomainPattern
                 {  variable=domainVars:Set(Variable) {},
                    templateExp=te:ObjectTemplateExp {}
                 }
          }
     } {
          not r.domain->exists(d| d.isEnforced = TRUE);
       }
enforce domain core m:Mapping
    {  name=rn+'_'+irn,
       guard=mg:GuardPattern {variable=mgVars:Set(Variable) {}},
         bottom=mb:BottomPattern
            {  realizedVariable = pvs:Set(RealizedVariable)
                  { tcv:RealizedVariable
                       { name=rn+'_v',
                         type=tc
                       }
                 },
               variable = mbVars
            },
         domain=md:CoreDomain
             {  isChecked=c,
                modelExtentName=dir,
                guard=dg:GuardPattern
                   {variable=dgVars:Set(Variable) {}},
                bottom=db:BottomPattern {}
             }
     }
when {
   RelationToTraceClass(r, tc);
}
where {
   allDomainVars = r.domain->iterate(d; vars: Set(Variable) = {} |
         vars->union(getVars(d)) );
   sharedDomainVars = r.domain->iterate(d; vars: Set(Variable) = {} |
         if vars->isEmpty() then vars->union(getVars(d))
                           else vars->intersection(getVars(d)));
   unsharedWhenVars = whenVars->minus(allDomainVars);
   domainVarsSharedWithWhen = domainVars->intersection(whenVars);
   domainBottomUnSharedVars = domainVars->minus(whenVars)
                                     ->minus(sharedDomainVars);
   domainBottomSharedVars = domainVars->minus(whenVars)
                                 ->intersection(sharedDomainVars);


   RVarSetToMVarSet(unsharedWhenVars, mgVars);
   RWhenPatternToMGuardPattern(whenp, mg);
   RInvokerToMGuard(ir, ri, r, mg);
   RVarSetToMVarSet(domainVarsSharedWithWhen, dgVars);
   RVarSetToMVarSet(domainBottomUnSharedVars, dbVars);
   RVarSetToMVarSet(domainBottomSharedVars, mbVars);
   RDomainPatternToMDBottomPattern(te, sharedDomainVars, db, mb);
   RDomainVarsToTraceClassProps(domainVars, tcv, mb);
   RSimplePatternToMPattern(wherep, mb);
```

```
   }
}

// Rule 6 (extends Rule 4):
// 6.1: an invoked relation maps to as many core mappings as the relations that
// invoke it.
//   i.e. there exists a separate core mapping for each invoker-invoked pair.
// 6.2: The guard pattern of the mapping will have a variable corresponding to
// the trace class of the invoker relation, with root object variables of all the
// patterns of all the domains of the invoked relation being equated with
// corresponding properties of this trace class .
// 6.3: The root object variable of a relation domain's pattern becomes a pattern
// variable
// in the core domain guard (this is in addition to the variables that occur in
// the when clause as per rule 2.1).
//
relation InvokedRelationToMappingForEnforcement
{
   allDomainVars: Set(Variable);
   oppositeDomainVars: Set(Variable);
   sharedDomainVars: Set(Variable);
   ownRdVars: Set(Variable);
   predicatesWithVarBindings: Set(Predicate);
   predicatesWithoutVarBindings: Set(Predicate);
   unsharedWhenVars: Set(Variable);
   domainTopVars: Set(Variable);

   checkonly domain relations r:Relation
      {  name=rn,
         invokedBy=ri:RelationInvocation
           {  predicate=p:Predicate
                {  pattern=pt:BottomPattern
                     {  relation=ir:Relation {name=irn}
                     }
                }
           },
         when=whenp:GuardPattern {variable=whenVars:Set(Variable) {}},
         where=wherep:BottomPattern {predicate=rpSet:Set(Predicate) {}},
         domain = rds:Set(relationDomain)
             {  rd:RelationDomain
                  {  isEnforced = TRUE,
                     modelExtentName=dir:TypedModel {},
                     variable=domainVars:Set(Variable) {},
                     pattern=dp:DomainPattern
                        {  templateExp=te:ObjectTemplateExp
                              {  bindsTo=tev:Variable {}
                              }
                        }
                  } ++ rOppositeDomains
             }
      }
   enforce domain core m:Mapping
     {  name=rn+'_'+irn+'_'+dir.name,
        guard=mg:GuardPattern {variable=mgVars:Set(Variable) {}},
        bottom=mb:BottomPattern
           {  realizedVariable=tcv:RealizedVariable
                { name=rn+'_v',
                  type=tc
                },
              predicate=predicatesWithVarBindings   // copy?
           }
        domain = cds:Set(CoreDomain)
          {  md:CoreDomain
               {  isEnforced=TRUE,
                  isChecked=TRUE,
                  modelExtentName=dir,
                  guard=dg:GuardPattern
```

```
                    {  variable=dgVars:Set(Variable) {}
                    },
                bottom=db:BottomPattern {}
             } ++ mOppositeDomains
         }
      }
   when {
      RelationToTraceClass(r, tc);
   }
   where {
      allDomainVars = r.domain->iterate(d; vars: Set(Variable) = {} |
                                    vars->union(getVars(d)) );
      oppositeDomainVars = rOppositeDomains->
           iterate(d; vars: Set(Variable) = {} |
                                    vars->union(getVars(d)) );
      sharedDomainVars = r.domain->iterate(d; vars: Set(Variable) = {} |
            if vars->isEmpty() then vars->union(getVars(d))
                              else vars->intersection(getVars(d)));
      ownrdVars = domainVars->minus(sharedDomainVars)->minus(whenVars);
      predicatesWithVarBindings =
         filterOutPredicatesThatReferToVars(rpSet, ownrdVars);
      predicatesWithoutVarBindings = rpSet->minus(predicatesWithVarBindings);
      unsharedWhenVars = whenVars->minus(allDomainVars);
      domainTopVars = domainVars->intersection(whenVars)->union({tev});

      RVarSetToMVarSet(unsharedWhenVars, mgVars);
      RWhenPatternToMGuardPattern(when, mg);
      RInvokerToMGuard(ir, ri, r, mg);
      RVarSetToMVarSet(domainTopVars, dgVars);
      RDomainToMDBottomForEnforcement(te, dir, predicatesWithoutVarBindings,
                              ownrdVars, whenVars, sharedDomainVars,
                              m, db, mb, tcv);
      RDomainVarsToTraceClassProps(oppositeDomainVars, tcv, mb);
      IROppositeDomainsToMappingForEnforcement(when,
                                          rOppositeDomains,
                                          sharedDomainVars,
                                          mOppositeDomains, mb);
      RRelImplToMBottomEnforcementOperation(r, rd, mb);
   }
}

// opposite domains of a top-level relation's enforced domain are mapped as per rules
// 4.2 and 4.5
//
relation IROppositeDomainsToMappingForEnforcement
{
   domainTopVars: Set(Variable);
   domainBottomUnSharedVars: Set(Variable);
   domainBottomSharedVars: Set(Variable);

   checkonly domain relations whenp:GuardPattern
     {  variable=whenVars:Set(Variable) {}
     }
   checkonly domain relations rds:Set(RelationDomain)
     {  rd:RelationDomain
         {  modelExtentName=dir:TypedModel {},
            isChecked=c,
            variable=domainVars:Set(Variable) {}
            pattern=dp:DomainPattern
              {  templateExp=te:ObjectTemplateExp
                  {  bindsTo=tev:Variable {}
                  }
              }
          } ++ _
      }
   checkonly domain relations sharedDomainVars:Set(Variable) {}
```

```
    enforce domain core cds:Set(CoreDomain)
       {  md:CoreDomain
            {  isChecked=c,
               isEnforced=FALSE,
               modelExtentName=dir,
               guard=dg:GuardPattern
                  {  variable=dgVars:Set(Variable) {}
                  },
                 bottom=db:BottomPattern
                   {  variable=dbVars:Set(Variable) {}
                   }
            } ++ _
      }
    checkonly enforce domain core mb:BottomPattern
       {  variable=mbVars:Set(Variable) {}
       }
    where {
       domainTopVars = domainVars->intersection(whenVars)->union({tev});
       domainBottomUnSharedVars = domainVars->minus(whenVars)
                                   ->minus({tev})->minus(sharedDomainVars);
       domainBottomSharedVars =
                   domainVars->minus(whenVars)->minus({tev})
                       ->intersection(sharedDomainVars);

       RVarSetToMVarSet(domainTopVars, dgVars);
       RVarSetToMVarSet(domainBottomUnSharedVars, dbVars);
       RVarSetToMVarSet(domainBottomSharedVars, mbVars);
       RDomainPatternToMDBottomPattern(te, sharedDomainVars, db, mb);
    }
}

// opposite domains of an invoked relation's enforced domain are mapped as per
// rules 4.2 and 4.5
// In addition, as per rule 6.3 the root object variable of a relation domain's
// pattern  becomes a pattern variable in the core domain guard (this is in
// addition to the variables that occur in the when clause as per rule 2.1).
//
relation TROppositeDomainsToMappingForEnforcement
{
    domainTopVars: Set(Variable);
    domainBottomUnSharedVars: Set(Variable);
    domainBottomSharedVars: Set(Variable);

    checkonly domain relations whenp:GuardPattern
       {  variable=whenVars:Set(Variable) {}
       }
    checkonly domain relations rds:Set(RelationDomain)
       {rd:RelationDomain
         {  modelExtentName=dir:TypedModel {},
            isChecked=c,
            variable=domainVars:Set(Variable) {},
            pattern=dp:DomainPattern
               {  templateExp=te:ObjectTemplateExp {}
               }
          } ++ _
        }
    checkonly domain relations sharedDomainVars:Set(Variable) {}
    enforce domain core cds:Set(CoreDomain)
       {  md:CoreDomain ]
            {  isChecked=c,
               isEnforced=FALSE,
               modelExtentName=dir,
               guard=dg:GuardPattern
                  {  variable=dgVars:Set(Variable) {}
                  },
               bottom=db:BottomPattern
```

```
                {  variable=dbVars:Set(Variable) {}
                }
            } ++ _
       }
   checkonly enforce domain core mb:BottomPattern
     {  variable=mbVars:Set(Variable) {}
     }
   where {
      domainTopVars = domainVars->intersection(whenVars);
      domainBottomUnSharedVars = domainVars->minus(whenVars)
                                 ->minus(sharedDomainVars);
      domainBottomSharedVars =
                  domainVars->minus(whenVars)->intersection(sharedDomainVars);

      RVarSetToMVarSet(domainTopVars, dgVars);
      RVarSetToMVarSet(domainBottomUnSharedVars, dbVars);
      RVarSetToMVarSet(domainBottomSharedVars, mbVars);
      RDomainPatternToMDBottomPattern(te, sharedDomainVars, db, mb);
   }
}

relation RWhenPatternToMGuardPattern
{
   checkonly domain relations whenp:GuardPattern {}
   enforce domain core mg:GuardPattern {}
   where {
      RSimplePatternToMPattern(whenp, mg);
      RInvocationInWhenToMGuard(whenp, mg);
   }
}

relation RVarSetToMVarSet(domainVarsSharedWithWhen, dgVars);
{
   checkonly domain relations rvSet:Set(Variable)
      {rv:Variable {}++_}
   enforce domain core mvSet: Variable {mv: Variable {}++_}
   where {
      RVarToMVar(rv, mv);
   }
}

relation RVarToMVar
{
   checkonly domain relations rv:Variable {name=n, type=t:Type {}}
   enforce domain core mv: Variable {name=n, type=t}
}

relation RSimplePatternToMPattern
{
   checkonly domain relations rp:Pattern
      {predicate=pd:Predicate {conditionExpression=e:OclExpression {}}}
   {
      not e.oclIsTypeOf(RelationInvocation);
   }
   enforce domain core mp:Pattern {predicate=pd}   // new predicate required.
}

// relation invocation in when clause maps to a trace class pattern in mapping
// guard.
//


relation RInvocationInWhenToMGuard
```

```
{
   checkonly domain relations rp:Pattern
     {  predicate=pd:Predicate
          {  conditionExpression=e:RelationInvocation ]
               {  relation=r:Relation
                    {  domain=dseq:Sequence(RelationDomain) {}
                    },
                  argument=aseq:Sequence(VariableExp) {}
               }
          }
     }
   enforce domain core mp:Pattern
     {  variable=vd:Variable
          {  name=tc.name+'_v',
             type=tc:Class {}
          },
     }
   when {
      RelationToTraceClass(r, tc);
   }
   where {
      RInvocationArgToTraceClassProp(aseq, dseq, vd, mp)
   }
}

// invocation argument position corresponds to the domain position in invoked
// relation.
// Domain's root pattern object var gives us the corresponding trace class prop.
//
relation RInvocationArgToTraceClassProp
{
   checkonly domain relations aseq:Sequence(VariableExp)
      {  ve:VariableExp {variable=v:Variable {}} ++ _
      }
   checkonly domain relations dseq:Sequence(RelationDomain)
      {  d:RelationDomain
           {  pattern=dp:DomainPattern
                {  templateExp = ote:ObjectTemplateExp
                     {  bindsTo=dv:Variable {name=dvn}
                     }
                }
           } ++ _
       } {
         aseq->indexOf(ve) = dseq->indexOf(d);
       }
   enforce domain core tcvd:Variable){}
   enforce domain core mp:Pattern
       {  predicate = ee:equalsExp
            {  lhs=pe:PropertyCallExpr {source=tcvd, name=dvn}, rhs=mv
            }
       }
   when {
      RVarToMVar(v, mv);
   }
}

relation RInvokerToMGuard
{
   checkonly domain relations ir:Relation {}   // invoking relation
   checkonly domain relations ri:RelationInvocation
      {argument=aseq:Sequence(VariableExp) {}}
   checkonly domain relations r:Relation
      {domain=dseq:Sequence(RelationDomain) {}} // invoked relation
   enforce domain core mg:GuardPattern
      {variable=vd:Variable {name=tc.name+'_v', type=tc:Class {}}
                        }
```

```
   when {
      RelationToTraceClass(ir, tc);
   }
   where {
      RInvokerArgToTraceClassProp(aseq, dseq, vd, mg);
   }
}

// invocation argument position corresponds to the domain position in invoked
// relation.
// Invocation argument variable name gives the invoker trace class prop name;
// Domain's root pattern object var gives us core domain guard var
//
relation RInvokerArgToTraceClassProp(aseq, dseq, vd, mg)
{
   checkonly domain relations aseq:Sequence(VariableExp)
      {ve:VariableExp {variable= v:Variable {name=vn}} ++ _}
   checkonly domain relations dseq:Sequence(RelationDomain)
      {  d:RelationDomain
           {  pattern=dp:DomainPattern
                {  templateExp = ote:ObjectTemplateExp
                      {  bindsTo=dv:Variable {}
                      }
                }
           } ++ _
       } {
         aseq->indexOf(ve) = dseq->indexOf(d);
       }
   enforce domain core tcvd:Variable){}
   enforce domain core mp:Pattern {predicate = ee:equalsExp
      {lhs=pe:PropertyCallExpr {source=tcvd, name=vn}, rhs=mdv}}
   when {
      RVarToMVar(dv, mdv);
   }
}

relation RDomainPatternToMDBottomPattern
{
   checkonly domain relations te:ObjectTemplateExp {}
   checkonly domain relations sharedDomainVars:Set(Variable) {}
   enforce domain core db:BottomPattern {}   // domain bottom
   enforce domain core mb:BottomPattern {}   // mapping bottom
   where {
      RDomainPatternToMDBottomPatternComposite(te, sharedDomainVars, db, mb);
      RDomainPatternToMDBottomPatternSimpleNonVarExpr(te, sharedDomainVars, db);
      RDomainPatternToMDBottomPatternSimpleUnSharedVarExpr(te,
                                                     sharedDomainVars,
                                                     db);
      RDomainPatternToMDBottomPatternSimpleSharedVarExpr(te,
                                                   sharedDomainVars,
                                                   mb);
   }
}

relation RDomainToMDBottomForEnforcement
{
   remainingUnBoundDomainVars: Set(Variable);
   predicatesWithVarBindings:Set(Predicate);
   remainingPredicatesWithoutVarBindings:Set(Predicate);

   checkonly domain relations te:ObjectTemplateExp
       {bindsTo=v:Variable {}}
   checkonly domain relations dir:TypedModel {}
   checkonly domain relations predicatesWithoutVarBindings:Set(Predicate) {}
   checkonly domain relations unboundDomainVars:Set(Variable) {}
```

```
    checkonly domain relations whenVars:Set(Variable) {}
    checkonly domain relations sharedDomainVars:Set(Variable) {}
    enforce domain core m:Mapping {}
    enforce domain core db:BottomPattern {}   // domain bottom
    enforce domain core mb:BottomPattern // mapping bottom
       {  predicate=predicatesWithVarBindings->
                    union({ee:equalsExp {lhs=pe:PropertyCallExpr
                                           {source=tcv, name=v.name}, rhs=mv
                                        }
                          }
       }
    enforce domain core tcv: Variable {}   // trace class variable
    when {
       RVarToMVar(v, mv)
    }
    where {
       remainingUnBoundDomainVars = unboundDomainVars->minus(v);
       predicatesWithVarBindings = filterOutPredicatesThatReferToVars(
                                     predicatesWithoutVarBindings,
                                     remainingUnBoundDomainVars);
       remainingPredicatesWithoutVarBindings =
             predicatesWithoutVarBindings->minus(predicatesWithVarBindings);

       RDomainToMDBottomForEnforcementOfIdentityProp(te,
                                              sharedDomainVars, db, mb);
       RDomainToMDBottomForEnforcementOfNonIdentityPropPrimitive(te, m);
       RDomainToMDBottomForEnforcementOfNonIdentityPropObject(te, dir,
                    remainingPredicatesWithoutVarBindings,
                    remainingUnboundDomainVars, whenVars, sharedDomainVars,
                    m, tcv);
    }
}

relation RDomainToMDBottomForEnforcementOfIdentityProp
{
    checkonly domain relations te:ObjectTemplateExp
      {  bindsTo=v:Variable
           {  type=c:Class {}
           },
         part=pt:PropertyTemplateItem
           {  referredProperty=pp:Property {},
              value=e:OclExpression {}
           }
      } {
         c.key.property->includes(pp);
      }
    checkonly domain relations sharedDomainVars:Set(Variable) {}
    enforce domain core db:BottomPattern {} // domain bottom
    enforce domain core mb:BottomPattern {}   // mapping bottom
    where {
       RDomainPatternExprToMappingDomainAssignment(v, pp, e,
                                              sharedDomainVars, db);
       // or
       RDomainPatternExprToMappingDomainVarAssignment(v, pp, e,
                                              sharedDomainVars, db);
       // or
       RDomainPatternExprToMappingBottomVarAssignment(v, pp, e,
                                              sharedDomainVars, mb);
    }
}

relation RDomainPatternExprToMappingDomainAssignment
{
    checkonly domain relations v:Variable {}
    checkonly domain relations pp:Property {name=pn}
    checkonly domain relations e:OclExpression {} {
```

```
      not e.oclIsTypeOf(VariableExp);
   }
   checkonly domain relations sharedDomainVars:Set(Variable) {}
   enforce domain core db:BottomPattern
      { assignment = a:Assignment
          {  slotOwnerExpression=pe:PropertyCallExp
               {  source=mv, name=pn
               },
             valueExpression=e
          }   //copy e
      }
   where {
      RVarToMVar(v, mv);
   }
}

relation RDomainPatternExprToMappingDomainVarAssignment
{
   checkonly domain relations v:Variable {}
   checkonly domain relations pp:Property {name=pn}
   checkonly domain relations e:VariableExp {}
   checkonly domain relations sharedDomainVars:Set(Variable) {} {
      not sharedDomainVars->includes(e);
   }
   enforce domain core db:BottomPattern
       {  realizedVariable=mv:RealizedVariable,
          assignment = a:Assignment
            {  slotOwnerExpression=pe:PropertyCallExp
                 {  source=mv, name=pn
                 },
               valueExpression=e
       }   //copy e
                    }
   where {
      RVarToMVar(v, mv);
   }
}



relation RDomainPatternExprToMappingBottomVarAssignment
{
   checkonly domain relations v:Variable {}
   checkonly domain relations pp:Property {name=pn}
   checkonly domain relations e:VariableExp {}
   checkonly domain relations sharedDomainVars:Set(Variable) {} {
      sharedDomainVars->includes(e)
   }
   enforce domain core mb:BottomPattern
      {   realizedVariable=mv:RealizedVariable,
          assignment = a:Assignment
            {  slotOwnerExpression=pe:PropertyCallExp
                  {source=mv, name=pn
                  },
               valueExpression=e
             }   //copy e
      }
   where {
      RVarToMVar(v, mv);
   }
}

relation RDomainToMDBottomForEnforcementOfNonIdentityPropPrimitive
{
   checkonly domain relations te:ObjectTemplateExp
      {  bindsTo=v:Variable {type=c:Class {}},
```

```
         part=pt:PropertyTemplateItem
           {  referredProperty=pp:Property {name=pn},
              value=e:OclExpression {}
           }
      } {
        not c.key.property->includes(pp);
        not e.oclIsTypeOf(TemplateExpr);
      }
   enforce domain core m:Mapping
     {  local=cm:Mapping
          {  name=m.name+'_forNonIdentityProp',
             bottom=bp:BottomPattern
               {  assignment=a:Assignment
                  {  slotOwnerExpression=pe:PropertyCallExp
                       {  source=mv, name=pn
                       },
                     valueExpression=e
                  }
               }   //copy e
           }
      }
   where {
      RVarToMVar(v, mv);
   }
}

relation RDomainToMDBottomForEnforcementOfNonIdentityPropObject
{
   checkonly domain relations te:ObjectTemplateExp
      {  bindsTo=v:Variable {type=c:Class {}},
         part=pt:PropertyTemplateItem
           {  referredProperty=pp:Property {name=pn},
              value=pte:ObjectTemplateExp {pv:Variable {}}
           }
       } {
         not c.key.property->includes(pp);
       }
   checkonly domain relations dir:TypedModel {}
   checkonly domain relations predicatesWithoutVarBindings:Set(Predicate) {}
   checkonly domain relations unboundDomainVars:Set(Variable) {}
   checkonly domain relations whenVars:Set(Variable) {}
   checkonly domain relations sharedDomainVars:Set(Variable) {}
   enforce domain core m:Mapping {local=cm:Mapping {name=m.name+'_for_'+v.name,
                              domain = cd:CoreDomain {modelExtentName=dir,
                                 bottom=
                                 cmdb:BottomPattern {realizedVariable=
                                       mpv:Variable,
                                       assignment = a:Assignment
                                            {slotOwnerExpression=
                                              pe:PropertyCallExp
                                                     {source=mv, name=pn},
                                                valueExpression=mpv}
                                   } //TODO: add var only if pv not in whenVars
                              }
                              bottom=cmmb:BottomPattern {}
                           }
                        }
   enforce domain core tcv: Variable {}   // trace class variable
   where {
      RVarToMVar(v, mv);
      RVarToMVar(pv, mpv);
      RDomainToMDBottomForEnforcement(pte, dir,
            predicatesWithoutVarBindings, unboundDomainVars,
            whenVars, sharedDomainVars, cm, cmdb, cmmb, tcv);
   }
}
```

```
relation RDomainPatternToMDBottomPatternComposite
{
   checkonly domain relations te:ObjectTemplateExp {bindsTo=vte:Variable {},
                      part=pt:PropertyTemplateItem {referredProperty=pp:Property {},
                           value=pte:ObjectTemplateExp {bindsTo=vpte:Variable {}}}
                     }
   checkonly domain relations sharedDomainVars:Set(Variable) {}
   enforce domain core db:BottomPattern {predicate = ee:equalsExp {lhs=
                        pe:PropertyCallExpr {source=mvte, name=<>}, rhs=mvpte}
                     }
   enforce domain core mb:BottomPattern {}   // mapping bottom
   when {
      RVarToMVar(vte, mvte);
      RVarToMVar(vpte, mvpte);
   }
   where {
      RDomainPatternToMDBottomPattern(pte, sharedDomainVars, db, mb);
   }
}

relation RDomainPatternToMDBottomPatternSimpleUnSharedVarExpr
{
   checkonly domain relations te:ObjectTemplateExp {bindsTo=vte:Variable {},
                      part=pt:PropertyTemplateItem {referredProperty=pp:Property {},
                           value=e:VariableExpression {variable=vpte:Variable {}}}
                     }
   checkonly domain relations sharedDomainVars:Set(Variable) {} {
      not sharedDomainVars->includes(vpte)
   }
   enforce domain core bb:BottomPattern {predicate = ee:equalsExp {lhs=
                        pe:PropertyCallExpr {source=mvte, name=<>}, rhs=mvpte}/* a copy of e? */
                     }
   when {
      RVarToMVar(vte, mvte);
      RVarToMVar(vpte, mvpte);
   }
}

relation RDomainPatternToMDBottomPatternSimpleSharedVarExpr
{
   checkonly domain relations te:ObjectTemplateExp {bindsTo=vte:Variable {},
                      part=pt:PropertyTemplateItem {referredProperty=pp:Property {},
                           value=e:VariableExpression {variable=vpte:Variable {}}}
                     }
   checkonly domain relations sharedDomainVars:Set(Variable) {} {
      sharedDomainVars->includes(vpte)
   }
   enforce domain core db:BottomPattern {predicate = ee:equalsExp {lhs=
                        pe:PropertyCallExpr {source=mvte, name=<>}, rhs=mvpte}/* a copy of e? */
                     }
   when {
      RVarToMVar(vte, mvte);
      RVarToMVar(vpte, mvpte);
   }
}

relation RDomainPatternToMDBottomPatternSimpleNonVarExpr
{
   checkonly domain relations te:ObjectTemplateExp {bindsTo=vte:Variable {},
                      part=pt:PropertyTemplateItem {referredProperty=pp:Property {},
                           value=e:OclExpression {}}
                     }
   checkonly domain relations sharedDomainVars:Set(Variable) {} {
      not e.oclIsTypeOf(TemplateExp);
```

```
        not e.oclIsTypeOf(VariableExpression);
     }
     enforce domain core db:BottomPattern {predicate = ee:equalsExp {lhs=
                          pe:PropertyCallExpr {source=mvte, name=<>}, rhs=e}   /* a copy of e? */
                        }
     when {
        RVarToMVar(vte, mvte);
     }
  }

  relation RDomainVarsToTraceClassProps
  {
     checkonly domain relations domainVars:Set(Variable) {dv:Variable {}++_}
     enforce domain core tcv: Variable {}
     enforce domain core mb:BottomPattern {assignment = a:Assignment {slotOwnerExpression=
                                                   pe:PropertyCallExp {source=tcv, name=dv.name},
                                                valueExpression=mdv}
                                    }
     when {
        RVarToMVar(dv, mdv);
     }
  }

  relation RRelImplToMBottomEnforcementOperation
  {
     checkonly domain relations r:Relation {operationalIml = RelationImplementation {
                                                inDirectionOf = tm:TypedModel{},
                                                impl = op:Operation{}
                                             }
                                         }
     checkonly domain relations rd:RelationDomain {modelExtentName = tm}
     enforce domain core mb:BottomPattern {enforcementOperation = Set(EnforcementOperation) {
                                             EnforcementOperation {
                                                enforcementMode='Create',
                                                operationCallExp=oce:OperationCallExp {
                                                                   referredOperation = op
                                                               }
                                             },
                                             EnforcementOperation {
                                                enforcementMode='Delete',
                                                operationCallExp=oce
                                             }
                                          }
                                       }
     where {
        RRelDomainsToMOpCallArg(r, oce);
     }
  }

  relation RRelDomainsToMOpCallArg
  {
     checkonly domain relations r:Relation {domain = RelationDomain {
                                    pattern=DomainPattern{bindsTo=rv:Variable{}}}
                                 }
     enforce domain core oce:OperationCallExp {arguments=VariableExpression {
                                    variable=mv:Variable {}}
                                 }
     where {
        RVarToMVar(rv, mv);
     }
  }

  //example identity spec syntax

  key Variable (name, pattern);
```

```
key Variable (name, relation);

function getVars(te:TemplateExp): Set(Variable)
{
   te.part->iterate(p: vset:Set(Variable) = Set{te.bindsTo} |
                vset->union(getVars(p.value))
         )
}
```

Annex A
(normative)

Additional Examples

A.1 Relations Examples

A.1.1 UML to RDBMS Mapping

A.1.1.1 Overview

This example maps persistent classes of a simple UML model to tables of a simple RDBMS model. A persistent class maps to a table, a primary key and an identifying column. Attributes of the persistent class map to columns of the table: an attribute of a primitive datatype maps to a single column; an attribute of a complex data type maps to a set of columns corresponding to its exploded set of primitive datatype attributes; attributes inherited from the class hierarchy are also mapped to the columns of the table. An association between two persistent classes maps to a foreign key relationship between the corresponding tables.

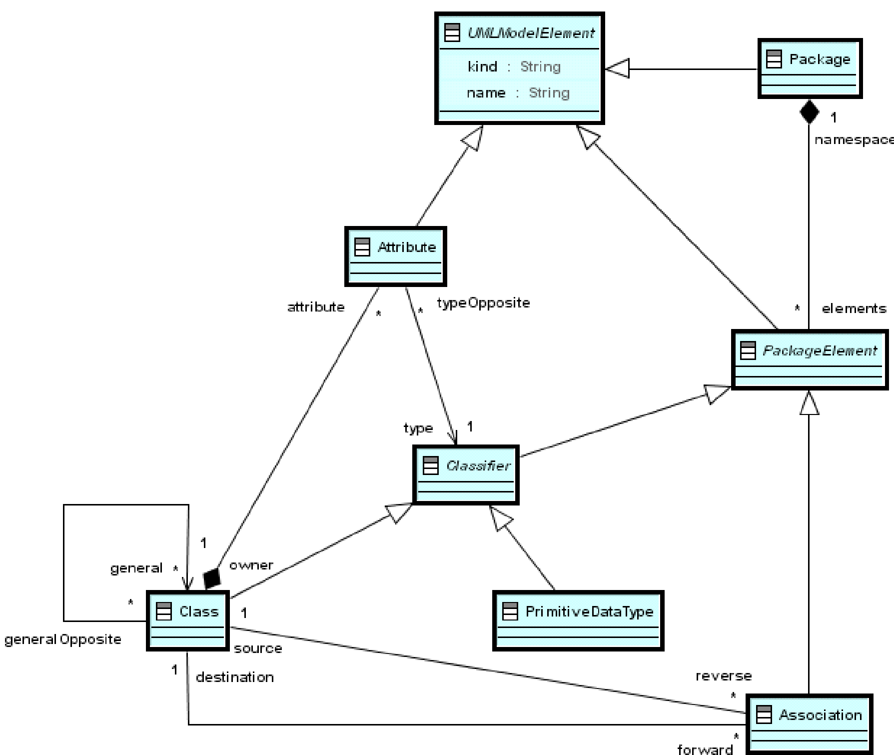


Figure A.1 - Simple UML Metamodel

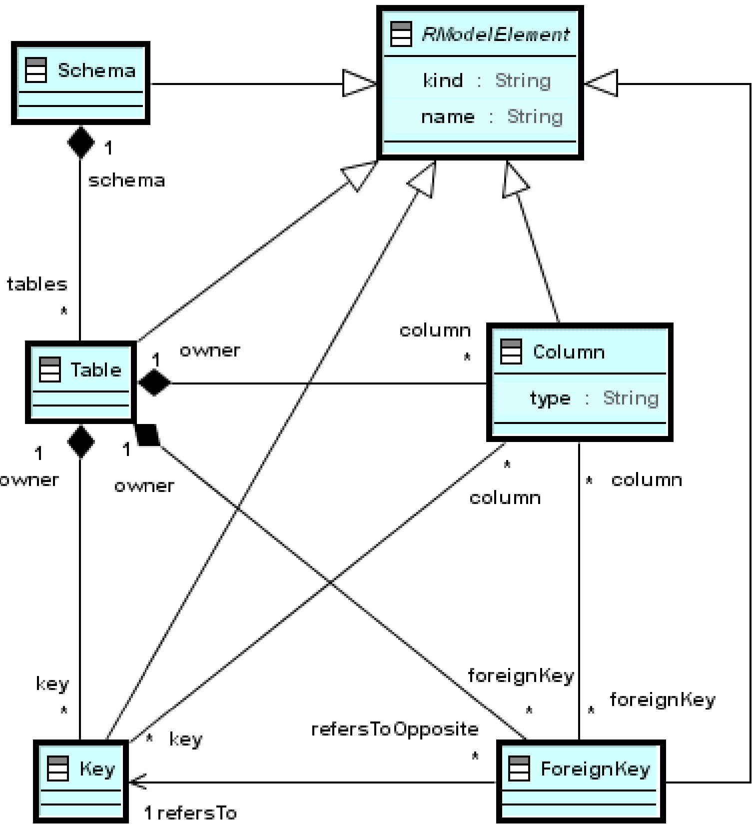


Figure A.2 - Simple RDBMS Metamodel

UML to RDBMS mapping in textual syntax

```
transformation umlToRdbms(uml:SimpleUML, rdbms:SimpleRDBMS)
{
key Table (name, schema);
key Column (name, owner);   // owner:Table opposite column:Column
key Key (name, owner); // key of class ëKeyí;
                       // owner:Table opposite key:Key

top relation PackageToSchema   // map each package to a schema
{
   pn: String;

   checkonly domain uml p:Package {name=pn};
   enforce domain rdbms s:Schema {name=pn};
}

top relation ClassToTable   // map each persistent class to a table
{
   cn, prefix: String;

   checkonly domain uml c:Class {namespace=p:Package {},
                                 kind='Persistent', name=cn};
   enforce domain rdbms t:Table {schema=s:Schema {}, name=cn,
```

```
            column=cl:Column {name=cn+'_tid', type='NUMBER'},
            key=k:Key {name=cn+'_pk', column=cl}};
    when {
        PackageToSchema(p, s);
    }
    where {
        prefix = '';
        AttributeToColumn(c, t, prefix);
    }
}

relation AttributeToColumn
{
    checkonly domain uml c:Class {};
    enforce domain rdbms t:Table {};
    primitive domain prefix:String;
    where {
        PrimitiveAttributeToColumn(c, t, prefix);
        ComplexAttributeToColumn(c, t, prefix);
        SuperAttributeToColumn(c, t, prefix);
    }
}

relation PrimitiveAttributeToColumn
{
    an, pn, cn, sqltype: String;

    checkonly domain uml c:Class {attribute=a:Attribute {name=an,
                                  type=p:PrimitiveDataType {name=pn}}};
    enforce domain rdbms t:Table {column=cl:Column {name=cn,
                                  type=sqltype}};
    primitive domain prefix:String;
    where {
        cn = if (prefix = '') then an else prefix+'_'+an endif;
        sqltype = PrimitiveTypeToSqlType(pn);
    }
}

relation ComplexAttributeToColumn
{
    an, newPrefix: String;

    checkonly domain uml c:Class {attribute=a:Attribute {name=an,
                                  type=tc:Class {}}};
    enforce domain rdbms t:Table {};
    primitive domain prefix:String;
    where {
        newPrefix = prefix+'_'+an;
        AttributeToColumn(tc, t, newPrefix);
    }
}

relation SuperAttributeToColumn
{
    checkonly domain uml c:Class {general=sc:Class {}};
    enforce domain rdbms t:Table {};
    primitive domain prefix:String;
    where {
        AttributeToColumn(sc, t, prefix);
    }
}

// map each association between persistent classes to a foreign key
top relation AssocToFKey
```

```
{
   srcTbl, destTbl: Table;
   pKey: Key;
   an, scn, dcn, fkn, fcn: String;

   checkonly domain uml a:Association {namespace=p:Package {},
            name=an,
            source=sc:Class {kind='Persistent',name=scn},
            destination=dc:Class {kind='Persistent',name=dcn}
         };
   enforce domain rdbms fk:ForeignKey {schema=s:Schema {},
            name=fkn,
            owner=srcTbl,
            column=fc:Column {name=fcn,type='NUMBER',owner=srcTbl},
            refersTo=pKey
         };
   when {   /* when refers to pre-condition */
      PackageToSchema(p, s);
      ClassToTable(sc, srcTbl);
      ClassToTable(dc, destTbl);
      pKey = destTbl.key;
   }
   where {
      fkn=scn+'_'+an+'_'+dcn;
      fcn=fkn+'_tid';
   }
}

function PrimitiveTypeToSqlType(primitiveTpe:String):String
{
   if (primitiveType='INTEGER')
   then 'NUMBER'
   else if (primitiveType='BOOLEAN')
      then 'BOOLEAN'
      else 'VARCHAR'
      endif
   endif;
}

}
```

UML to RDBMS mapping in graphical syntax

PackageToSchema

«domain»
p:Package
name = pn
uml
C
E
rdbms
«domain»
s:Schema
name = pn

Figure A.3 - PackageToSchema relation

ClassToTable

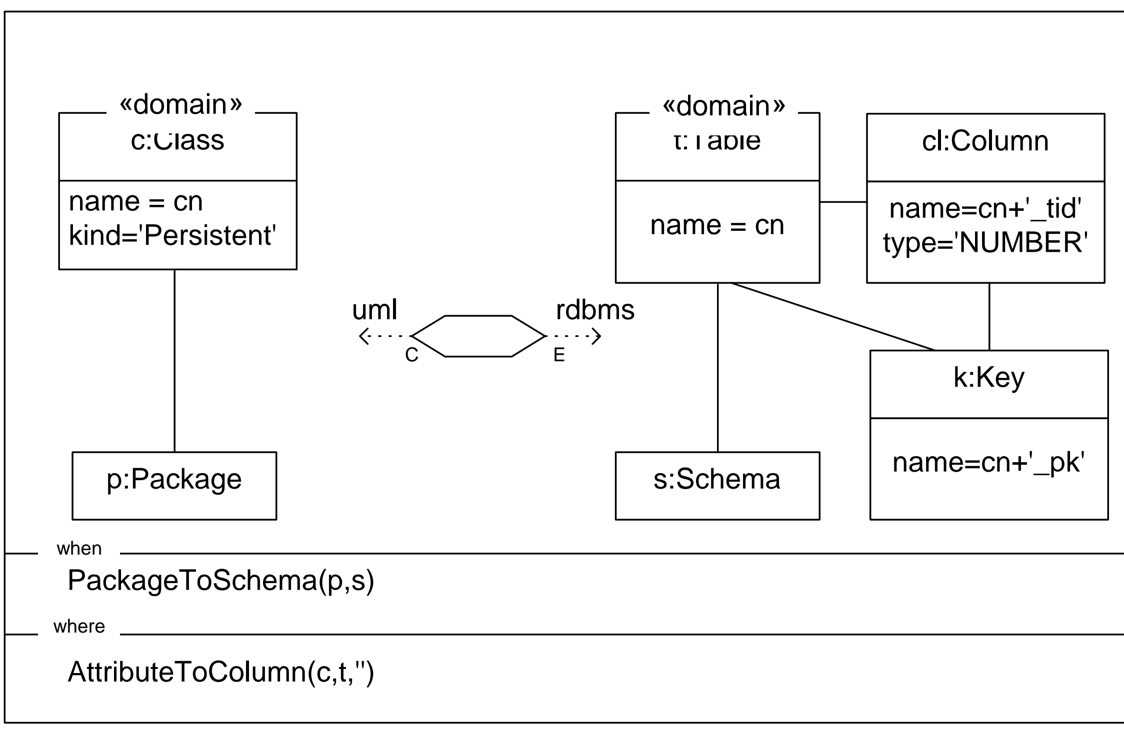


Figure A.4 - ClassToTable relation

PrimitiveAttributeToColumn

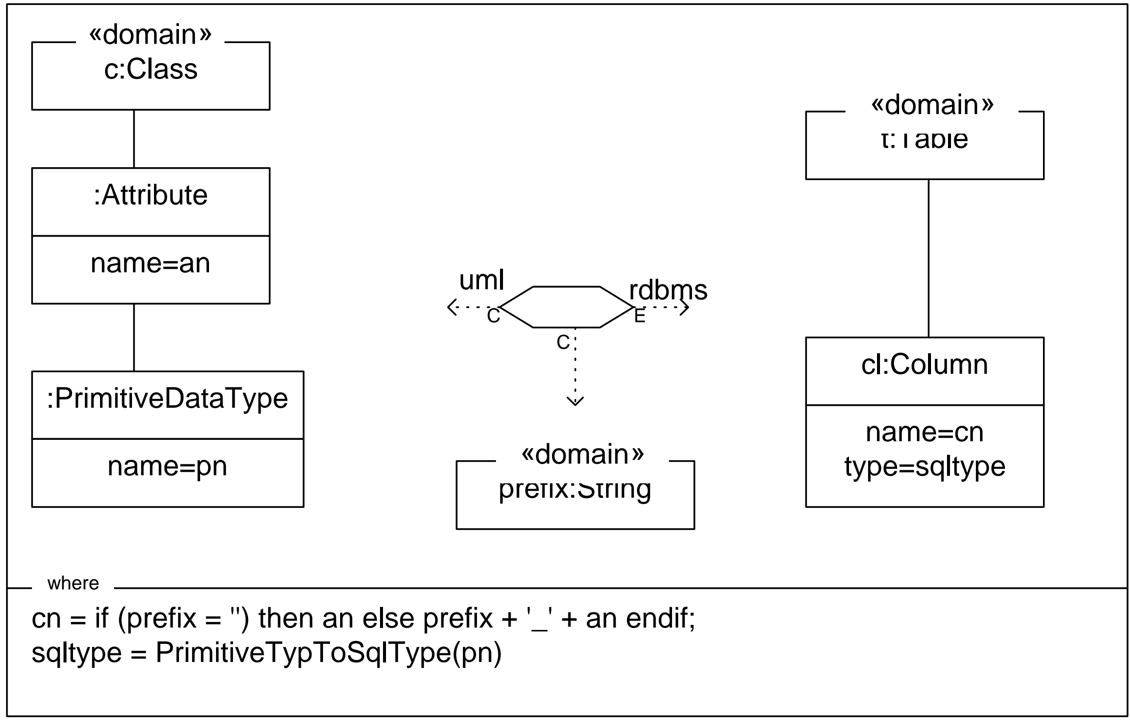


Figure A.5 - PrimitiveAttributeToColumn relation

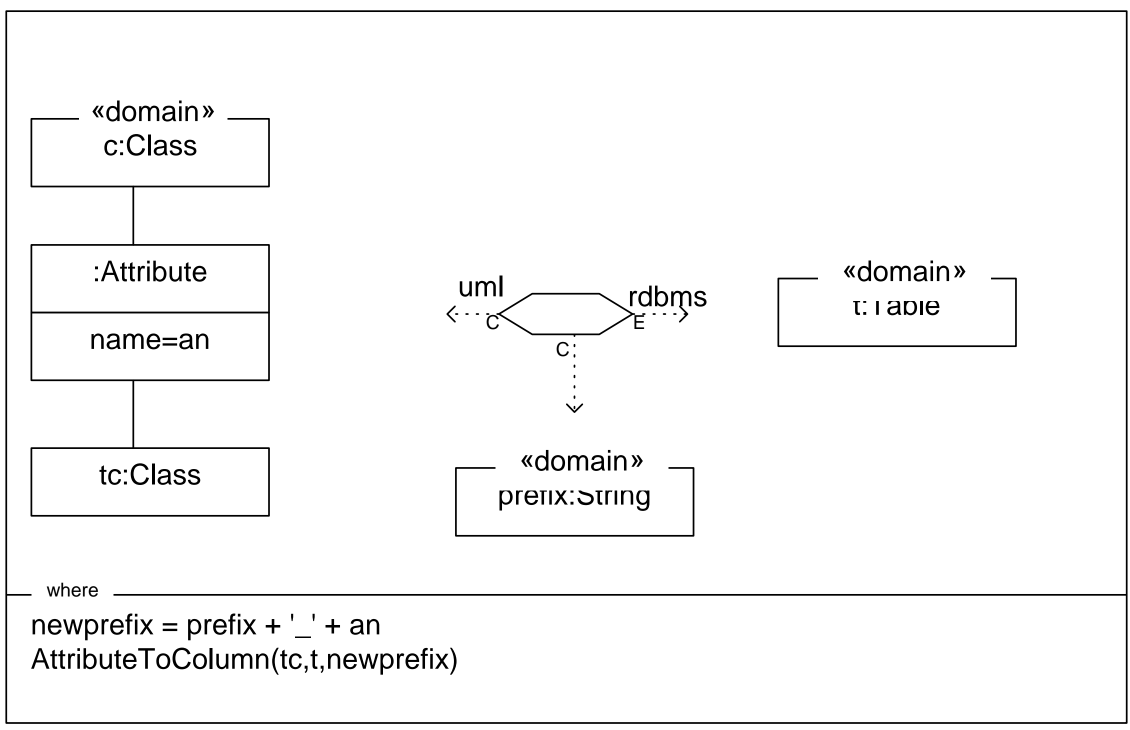


Figure A.6 - ComplexAttributeToColumn relation

SuperAttributeToColumn

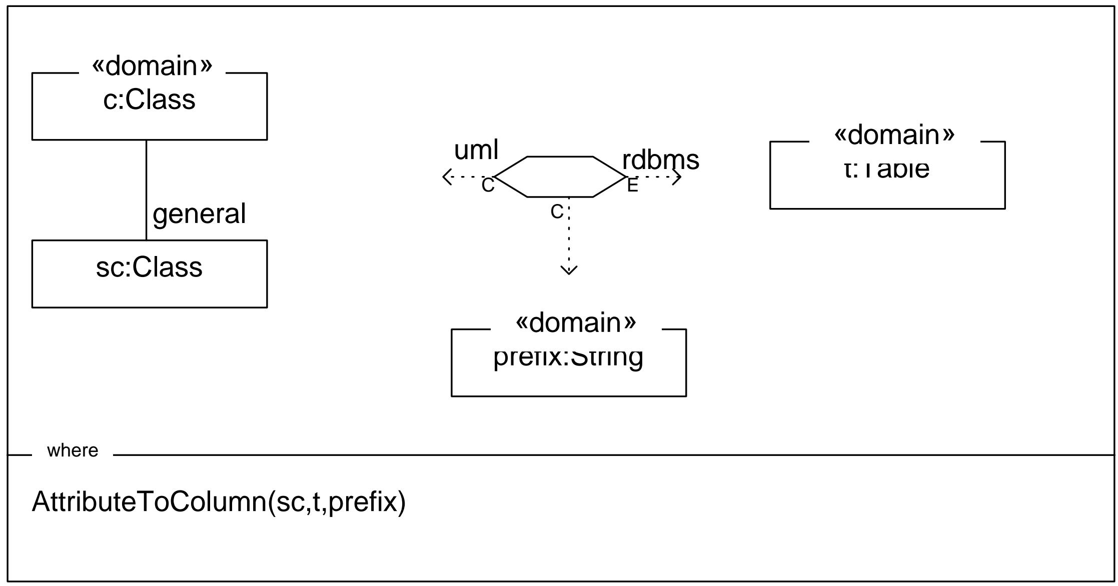


Figure A.7 - SuperAttributeToColumn relation

AssocToFKey

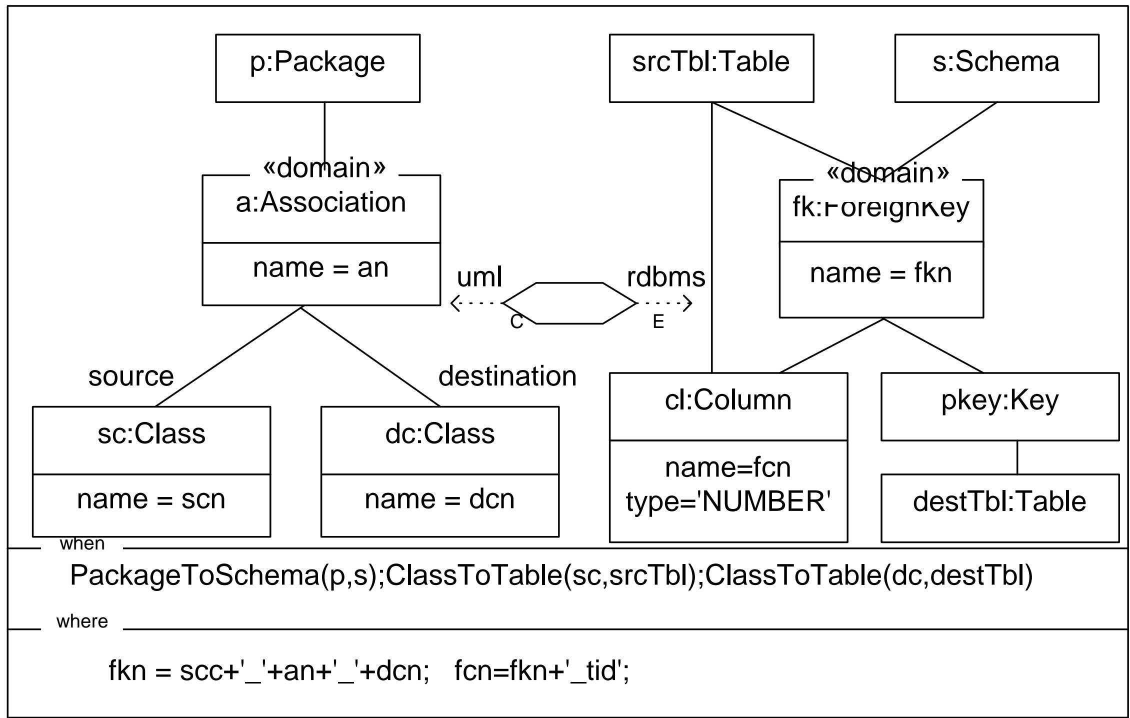


Figure A.8 - AssocToFKey relation

A.2 Operational Mapping Examples

A.2.1 Book To Publication example

```
metamodel BOOK {
  class Book {title: String; composite chapters: Chapter;}
  class Chapter {title : String; nbPages : Integer;}
}

metamodel PUB {
  class Publication {title : String; nbPages : Integer;}
}

transformation Book2Publication(in bookModel:BOOK,out pubModel:PUB)

main() {
  bookModel->objectOfType(Book)->map book_to_publication();
}

mapping Class::book_to_publication () : Publication {
  title := self.title;
  nbPages := self.chapters->nbPages->sum();
}
```

A.2.2 Encapsulation example

```
-- This QVT definition performs an in place transformation on
-- a UML class-diagram model by privatizing the attributes and
-- creating accessor methods

modeltype UML uses "omg.org.uml14"
transformation Encapsulation(inout classModel:UML);

-- entry point: selects the packages and applies the transformation
-- on each package

main() {
  classModel.objectsOfType(Package)
     ->map encapsulateAttributesInPackageClasses();
}

-- Applies the transformation to each class of the package

mapping inout Package::encapsulateAttributesInPackageClasses () {
  init {self.ownedElement->map encapsulateAttributesInClass();}
}

-- Performs the encapsulation for each attribute of the class
-- The initialization section is used to retrieve the list of attributes
-- The population section is used to add the two accessor operations
-- The end section is used to privatize each attribute

mapping inout Class encapsulateAttributesInClass ()
{
  init { var attrs := self.attribute;}
  operation := { -- assignment with additive semantics
      attrs->object(a) Operation {
           name := "get_" + self.name.upperFirst();
           visibility := "public";
           type := a.type;
         };
      attrs->object(a) Operation {
            name := "set_" + self.name.upperFirst();
            visibility := "public";
            parameter := object Parameter {
               name := 'a_'+ self.name.upperFirst();
               kind := "in";
               type := a.type;};
            };
         };
  end { attrs->map privatizeAttribute();}
}

-- in place privatization of the attribute

mapping inout Attribute::privatizeAttribute () {
   visibility := "private";
}
```

A.2.3 Uml to Rdbms

This example expresses the same transformation semantics, and uses the same metamodels shown in the Relations Examples in Section A.1.1.

```
-- declaring the transformation

modeltype UML uses SimpleUml("omg.qvt-samples.SimpleUML");
modeltype RDBMS uses SimpleRdbms("omg.qvt-samples.SimpleRDBMS");
transformation Uml2Rdb(in srcModel:UML,out RDBMS);

-- defining specific helpers and derived properties

query UML::Association.isPersistent() =
    (self.source.kind='persistent' and self.destination.kind='persistent');
intermediate property UML::Class::leafAttributes : Sequence(LeafAttribute);



-- defining intermediate data to reference leaf attributes that may
-- appear when struct data types are used

intermediate class UML::LeafAttribute {
  name:String;kind:String;attr:Attribute;
};

-- defining the default entry point for the module
-- first the tables are created from classes, then the tables are
-- updated with the foreign keys implied by the associations

main() {
  srcModel.objects()[#Class]->map class2table(); -- first pass
  srcModel.objects()[#Association]->map asso2table(); -- second pass
}

-- maps a class to a table, with a column per flattened leaf attribute

mapping Class::class2table () : Table
  when {self.kind='persistent';}
{
  init { -- performs any needed intialization
    self.leafAttributes := self.attribute->attr2LeafAttrs();
  }
  -- population section for the table
  name := 't_' + self.name;
  column := self.leafAttributes->map leafAttr2OrdinaryColumn();
  key := object Key {  -- nested population section for a 'Key'
           name := 'k_'+ self.name; column := t.column[kind='primary'];
         };
}

-- Mapping that creates the intermediate leaf attributes data.

mapping Attribute::attr2LeafAttrs (in prefix:String="",in pkind:String="")
: Sequence(LeafAttribute) {
  init {
    var k := if pkind="" then self.kind else pkind endif;
    result :=
       if self.type.isKindOf(PrimitiveDataType)
```

```
        then -- creates a sequence with a LeafAttribute instance
          Sequence {
            object LeafAttribute {attr:=self;name:=prefix+self.name;kind:=k;}
          }
        else self.type.attribute->map attr2LeafAttrs(self.name+"_",k)
        endif;
  }
}

-- Mapping that creates an ordinary colum from a leaf attribute

mapping LeafAttribute::leafAttr2OrdinaryColumn (in prefix:String=""): Column {
  name := prefix+self.name;
  kind := self.kind;
  type := if self.attr.type.name='int' then 'NUMBER' else 'VARCHAR' endif;
}

-- mapping to update a Table with new columns of foreign keys

mapping Association::asso2table() : Table
  when {self.isPersistent();}
{
  init { result := self.destination.resolveone(#Table); }
  foreignKey := self.map asso2ForeignKey();
  column := result.foreignKey.column;
}

-- mapping to build the foreign keys

mapping Association::asso2ForeignKey {
   name := 'f_' + name;
   refersTo := self.source.resolveone(Table).key;
   column := self.source.leafAttributes[kind='primary']
              ->map leafAttr2ForeignColumn(source.name+'_');
}

-- Mapping to create a Foreign key from a lef attributes
-- Inheriting of leafAttr2OrdinaryColumn has the effect to call the
-- inherited rule before entering the property population section

mapping LeafAttribute::leafAttr2ForeignColumn (in prefix:String) : Column
  inherits leafAttr2OrdinaryColumn {
     kind := "foreign";
}
```

A.2.4 SPEM UML Profile to SPEM metamodel

```
modeltype UML uses "omg.org.spem_umlprofile";
modeltype SPEM uses "omg.org.spem_metamodel";
transformation SpemProfile2Metamodel(in umlmodel:UML,out spemmodel:SPEM);

query UML::isStereotypedBy(stereotypeName:String) : Boolean;
query UML::Classifier::getOppositeAends() : Set(UML::AssociationEnd);

main () {
  -- first pass: create all the SPEM elements from UML elements
  umlmodel.rootobjects()[#UML::Model]->map createDefaultPackage();
  -- second pass: add the dependencies beyween SPEM elements
  umlmodel.objects[#UML::UseCase]->map addDependenciesInWorkDefinition();
}
```

```
mapping UML::Package::createDefaultPackage () : SPEM::Package {
  name := self.name;
  ownedElement := self.ownedElement->map createModelElement();
}

mapping UML::Package::createProcessComponent () : SPEM::ProcessComponent
  inherits createDefaultPackage
  when {self.isStereotypedBy("ProcessComponent");}
  {}

mapping UML::Package::createDiscipline () : SPEM::Discipline
  inherits createDefaultPackage
  when {self.isStereotypedBy("Discipline");}
  {}



mapping UML::ModelElement::createModelElement () : SPEM::ModelElement
  disjuncts
    createProcessRole, createWorkDefinition,
    createProcessComponent, createDiscipline
  {}

mapping UML::UseCase::createWorkDefinition () : SPEM::WorkDefinition {
  disjuncts
    createLifeCycle, createPhase, createIteration,
    createActivity, createCompositeWorkDefinition
  {}
}

mapping UML::Actor::createProcessRole () : SPEM::ProcessRole
  when {self.isStereotypedBy("ProcessRole");}
  {}

-- rule to create the default process performer singleton
mapping createOrRetrieveDefaultPerformer () : SPEM::ProcessPerformer {
  init {
    result := resolveoneByRule(createOrRetrieveDefaultPerformer);
    if result then return endif;
  }
  name := "ProcessPerformer";
}

mapping abstract UML::UseCase::createCommonWorkDefinition ()
 : SPEM::WorkDefinition
{
   name := self.name;
   constraint := {
      self.constraint[isStereotypedBy("precondition")]
         ->map createPrecondition();
      self.constraint[isStereotypedBy("goal")]->map createGoal();
   };
}

mapping UML::UseCase::createActivity () : SPEM::WorkDefinition
  inherits createCommonWorkDefinition
  when {self.isStereotypedBy("Activity");}
  {}

mapping UML::UseCase::createPhase () : SPEM::Phase
  inherits createCommonWorkDefinition
  when {self.isStereotypedBy("Phase");}
```

```
  {}

mapping UML::UseCase::createIteration () : SPEM::Iteration
  inherits createCommonWorkDefinition
  when {self.isStereotypedBy("Iteration");}
  {}

mapping UML::UseCase::createLifeCycle () : SPEM::LifeCycle
  inherits createCommonWorkDefinition
  when {self.isStereotypedBy("LifeCycle");}
  {}

mapping UML::UseCase::createCompositeWorkDefinition () : SPEM::WorkDefinition
  inherits createCommonWorkDefinition
  when {self.isStereotypedBy("WorkDefinition");}
  {}

mapping UML::Constraint::createPrecondition () : SPEM::Precondition {
  body := self.body;
}

mapping UML::Constraint::createGoal () : SPEM::Goal {
  body := self.body;
}

mapping UML::UseCase::addDependenciesInWorkDefinition ()
 : SPEM::WorkDefinition
 merging addDependenciesInActivity
{
  init {
    result := self.resolveone(WorkDefinition);
    var performers
      := self.getOppositeAends()[i|i.association
           [isStereotypedBy("perform")]->notEmpty()];
    assert (not performers->size()>1)
       with log("A unique performer is allowed",self);
  }
  subWork := self.clientDependency[*includes].supplier
     ->resolveone(WorkDefinition);
  performer := if performers then performers->first()
               else createOrRetrieveDefaultPerformer() endif;
}

mapping UseCase::addDependenciesInActivity () : WorkDefinition
  when {self.stereotypedBy("Activity");}
  {
    assistant := self.getOppositeAends()[i|i.association
       [a|a.isStereotypedBy("assist")]->notEmpty()]->resolve();
  }
```

A.3 Core Examples

A.3.1 UML to RDBMS Mapping

This example expresses the same transformation semantics, and uses the same metamodels shown in the Relations Examples in Section A.1.1.

```
-- A Transformation definition from SimpleUML to SimpleRDBMS
```

```
module UmlRdbmsTransformation imports SimpleUML, SimpleRDBMS {

    transformation umlRdbms {
        uml imports SimpleUML;
        rdbms imports SimpleRDBMS;
    }

    -- Package and Schema mapping
    class PackageToSchema {
        composite classesToTables : Set(ClassToTable) opposites owner;
        composite primitivesToNames : Set(PrimitiveToName) opposites owner;
        name : String;
        -- uml
        umlPackage : Package;
        -- rdbms
        schema : Schema;
    }

    map packageToSchema in umlRdbms {
        uml () {
            p:Package
        }
        rdbms () {
            s:Schema
        }
        where () {
            p2s:PackageToSchema|
            p2s.umlPackage = p;
            p2s.schema = s;
        }
        map {
            where () {
                p2s.name := p.name;
                p2s.name := s.name;
                p.name := p2s.name;
                s.name := p2s.name;
            }
        }
    }

    -- Primitive data type marshaling
    class PrimitiveToName {
        owner : PackageToSchema opposites primitivesToNames;
        name : String;
        -- uml
        primitive : PrimitiveDataType;
        -- rdbms
        typeName : String;
    }

    map primitiveToName in umlRdbms {
        uml (p:Package) {
            prim:PrimitiveDataType|
            prim.owner = p;
        }
        check enforce rdbms () {
            sqlType:String
        }
        where (p2s:PackageToSchema| p2s.umlPackage=p) {
            realize p2n:PrimitiveToName|
            p2n.owner := p2s;
            p2n.primitive := prim;
            p2n.typeName := sqlType;
        }
        map {
```

```
        where () {
            p2n.name := prim.name + '2' + sqlType;
        }
    }
}

map integerToNumber in umlRdbms refines primitiveToName {
    uml () {
        prim.name = 'Integer';
    }
    check enforce rdbms () {
        sqlType := 'NUMBER';
    }
}

map booleanToBoolean in umlRdbms refines primitiveToName {
    uml () {
        prim.name = 'Boolean';
    }
    check enforce rdbms () {
        sqlType := 'BOOLEAN';
    }
}

map stringToVarchar in umlRdbms refines primitiveToName {
    uml () {
        prim.name = 'String';
    }
    check enforce rdbms () {
        sqlType := 'VARCHAR';
    }
}

-- utility functions for flattening
map flattening in umlRdbms {
    getAllSupers(cls : Class) : Set(Class) {
        cls.general->collect(gen|self.getAllSupers(gen))->
            including(cls)->asSet()
    }
    getAllAttributes(cls : Class) : Set(Attribute) {
        getAllSupers(cls).attribute
    }
    getAllForwards(cls : Class) : Set(Association) {
        getAllSupers(cls).forward
    }
}

-- Class and Table mapping
class ClassToTable extends FromAttributeOwner, ToColumn {
    owner : PackageToSchema opposites classesToTables;
    composite associationToForeignKeys :
        OrderedSet(AssociationToForeignKey) opposites owner;
    name : String;
    -- uml
    umlClass : Class;
    -- rdbms
    table : Table;
    primaryKey : Key;
}

map classToTable in umlRdbms {
    check enforce uml (p:Package) {
        realize c:Class|
        c.kind := 'persistent';
```

```
        c.namespace := p;
    }
    check enforce rdms (s:Schema) {
        realize t:Table|
        t.kind <> 'meta';
        default t.kind := 'base';
        t.schema := s;
    }
    where (p2s:Package2Schema| p2s.umlPackage=p; p2s.schema=s;) {
        realize c2t:ClassToTable|
        c2t.owner := p2s;
        c2t.umlClass := c;
        c2t.table := t;
    }
    map {
        where () {
            c2t.name := c.name;
            c2t.name := t.name;
            c.name := c2t.name;
            t.name := c2t.name;
        }
    }
    map {
        check enforce rdbms () {
            realize pk:Key,
            realize pc:Column|
            pk.owner := t;
            pk.kind := 'primary';
            pc.owner := t;
            pc.key->includes(pk);
            default pc.key := Set(Key){pk};
            default pc.type := 'NUMBER';
        }
        where () {
            c2t.primaryKey := pk;
            c2t.column := pc;
        }
        map {
            check enforce rdbms () {
                pc.name := t.name+'_tid';
                pk.name := t.name+'_pk';
            }
        }
    }
}

-- Association and ForeignKey mapping
class AssociationToForeignKey extends ToColumn {
    referenced : ClassToTable;
    owner : ClassToTable opposites associationToForeignKeys;
    name : String;
    -- uml
    association : Association;
    -- rdbms
    foreignKey : ForeignKey;
}

map associationToForeignKey in umlRdbms refines flattening {
    check enforce uml (p:Package, sc:Class, dc:Class| sc.namespace = p;) {
        realize a:Association|
        getAllForwards(sc)->includes(a);
        default a.source := sc;
        getAllSupers(dc)->includes(a.destination);
        default a.destination := dc;
        default a.namespace := p;
    }
```

```
    check enforce rdbms (s:Schema, st:Table, dt:Table, rk:Key|
        st.schema = s;
        rk.owner = dt;
        rk.kind = 'primary';
    ) {
        realize fk:ForeignKey,
        realize fc:Column|
        fk.owner := st;
        fc.owner := st;
        fk.refersTo := rk;
        fc.foreignKey->includes(fk);
        default fc.foreignKey := Set(ForeignKey){fk};
    }
    where (p2s:PackageToSchema, sc2t:ClassToTable, dc2t:ClassToTable|
        sc2t.owner = p2s;
        p2s.umlPackage = p;
        p2s.schema = s;
        sc2t.table = st;
        dc2t.table = dt;
        sc2t.umlClass = sc;
        dc2t.umlClass = dc;
    ) {
        realize a2f:AssociationToForeignKey|
        a2f.owner := sc2t;
        a2f.referenced := dc2t;
        a2f.association := a;
        a2f.foreignKey := fk;
        a2f.column := fc;
    }
    map {
        where () {
            a2f.name := if a.destination=dc and a.source=sc
                        then a.name
                        else if a.destination<>dc and a.source=sc
                        then dc.name+'_'+a.name
                        else if a.destination=dc and a.source<>sc
                        then a.name+'_'+sc.name
                        else dc.name+'_'+a.name+'_'+sc.name
                        endif endif endif;
            a.name := if a.destination=dc and a.source=sc
                      then a2f.name
                      else a.name
                      endif;
            fk.name := name;
            name := fk.name;
            fc.name := name+'_tid';
        }
    }
    map {
        where () {
            fc.type := rk.column->first().type;
        }
    }
}

-- attribute mapping
abstract class FromAttributeOwner {
    composite fromAttributes : Set(FromAttribute) opposites owner;
}

abstract class FromAttribute {
    name : String;
    kind : String;
    owner : FromAttributeOwner opposites fromAttributes;
    leafs : Set(AttributeToColumn);
    -- uml
```

```
    attribute : Attribute to uml;
}

abstract class ToColumn {
    -- rdbms
    column : Column;
}

class NonLeafAttribute extends FromAttributeOwner, FromAttribute {
    leafs := fromAttributes.leafs;
}

class AttributeToColumn extends FromAttribute, ToColumn {
    type : PrimitiveToName;
}

map attributes in umlRdbms refines flattening {
    check enforce uml (c:Class) {
        realize a:Attribute|
        default a.owner := c;
        getAllAttributes(c)->includes(a);
    }
    where (fao:FromAttributeOwner) {
        fa : FromAttribute|
        fa.attribute := a;
        fa.owner := fao;
    }
    map {
        where {
            fa.kind := a.kind;
            a.kind := fa.kind;
        }
    }
}

map classAttributes in umlRdbms refines attributes {
    where (fao:ClassToTable| fao.umlClass=c) {}
    map {
        where {
            fa.name := a.name;
            a.name := fa.name;
        }
    }
}

map primitiveAttribute in umlRdbms refines attributes {
    check enforce uml (t:PrimitiveDataType) {
        a.type := t;
    }
    where (p2n:PrimitiveToName|p2n.primitive=t) {
        realize fa:AttributeToColumn|
        fa.type := p2n;
    }
    map {
        where {
            fa.leafs := Set(AttributeToColumn) {fa};
        }
    }
}

map complexAttributeAttributes in umlRdbms refines attributes {
    check uml (ca:Attribute|ca.type=c) {}
    where (fao:NonLeafAttribute | fao.attribute=ca) {}
    map {
```

```
            where {
                fa.name := fao.name+'_'+a.name;
            }
        }
    }

    map complexAttribute in umlRdbms refines attributes {
        check uml (t:Class) {
            a.type = t;
        }
        where () {
            realize fa:NonLeafAttribute
        }
        map {
            where {
                fa.leafs := fromAttributes.leafs;
            }
        }
    }

    map classPrimitiveAttributes in umlRdbms refines classAttributes, primitiveAttribute {}

    map classComplexAttributes in umlRdbms refines classAttributes, complexAttribute {}

    map complexAttributePrimitiveAttributes in umlRdbms refines complexAttributeAttributes, primitive-
Attribute {}

     map complexAttributeComplexAttributes in umlRdbms refines complexAttributeAttributes, complexAt-
tribute {}

    -- column mapping
    map attributeColumns in umlRdbms {
        check enforce rdbms (t:Table) {
            realize c:Column|
            c.owner := t;
            c.key->size()=0;
            c.foreignKey->size()=0;
        }
        where (c2t:ClassToTable| c2t.table=t;) {
            realize a2c:AttributeToColumn|
            a2c.column := c;
            c2t.fromAttribute.leafs->include(a2c);
            default a2c.owner := c2t;
        }
        map {
            check enforce rdbms (ct:String) {
                c.type := ct;
            }
            where (p2n:PrimitiveToName) {
                a2c.type := p2n;
                p2n.typeName := ct;
            }
        }
        map {
            where () {
                c.name := a2c.name;
                a2c.name := c.name;
            }
        }
        map {
            where () {
                c.kind := a2c.kind;
                a2c.kind := c.kind;
            }
```

```
          }
      }


  } -- end of module UmlRdbmsTransformation
```

Annex B
(normative)

Semantics of Relations

To simplify the description of semantics, we can view a relation as having the following abstract structure:

```
Relation R
{
   Var <R_variable_set>   //  declaration of variables used in the relation

   [checkonly | enforce] Domain:<typed_model_1>
                   <domain_1_variable_set> // subset of <R_variable_set>
   {
      <domain_1_pattern> [<domain_1_condition>]
   }
   ...
   [checkonly | enforce] Domain:< typed_model_n>
                   <domain_n_variable_set> // subset of <R_variable_set>
   {
      <domain_n_pattern> [<domain_n_condition>]
   }  // n >= 2

   [when <when_variable_set> <when_condition>]

   [where <where_condition>]
}
```

With the following properties:

- < R_variable_set > is the set of variables occurring in the relation.
- <domain_k_variable_set> is the set of variables occurring in domain k. It is a subset of <R_variable_set>, for all k = 1..n.
- <when_variable_set> is the set of variables occurring in the **when** clause. It is a subset of <R_variable_set>.
- The intersection of domain variable sets need not be null, i.e. a variable may occur in multiple domains.
- The intersection of a domain variable set and when variable set need not be null.
- The term <domain_k_pattern> refers to the set of constraints implied by the pattern of domain k. Please recall that a pattern can be viewed as a set of variables, and a set of constraints that model elements bound to those variables must satisfy in order to qualify as a valid binding of the pattern. Please refer to Section 9.10.3 for a detailed discussion on pattern matching semantics. Given below is an example pattern and the constraint implied by it.

  Pattern:

  c:Class {kind=íPersistentí, name=cn, attribute=a:Attribute { } }

  Implied constraint:

  c.kind = ëPersistentí and c.name = cn and c.attribute->includes(a)

B.1 Checking Semantics

Checking of relation R in the direction of <typed_model_k> evaluates to TRUE if the following predicate calculus formula evaluates to TRUE.

We use the following syntactic convention in the formulae:

|<variable_set>| - a binding of variables of the set <variable_set>

ForAll |<variable_set>| - for all bindings of variables of the set

Exists |<variable_set>| - there exists a binding of variables of the set

<exclusive_domain_k_variable_set> - variables occurring exclusively in domain k, i.e. those variables of domain k that do not occur in any other domain or the when clause. It is a subset of <domain_k_variable_set>.

```
CHECK(R, <TYPED_MODEL_K>)
=
FORALL |<WHEN_VARIABLE_SET>|
(
   <WHEN_CONDITION>
   IMPLIES
   (
      FORALL |(<R_VARIABLE_SET> MINUS (<WHEN_VARIABLE_SET> UNION
                                      <EXCLUSIVE_DOMAIN_K_VARIABLE_SET>))|
      (
         (
            (<DOMAIN_1_PATTERN> AND <DOMAIN_1_CONDITION>)
            AND
            ....
            (<DOMAIN_K-1_PATTERN> AND <DOMAIN_K-1_CONDITION>)
            AND
            (<DOMAIN_K+1_PATTERN> AND <DOMAIN_K+1_CONDITION>)
            AND
            ....
            (<DOMAIN_N_PATTERN> AND <DOMAIN_N_CONDITION>)
         )
         IMPLIES
         (
            EXISTS |<EXCLUSIVE_DOMAIN_K_VARIABLE_SET>|
            (
               (<DOMAIN_K_PATTERN> AND <DOMAIN_K_CONDITION>)
               AND
               <WHERE_CONDITION>
            )  // EXISTS
         )  // IMPLIES
      )  // FORALL
   ) // IMPLIES
) // FORALL
```

B.2 Enforcement Semantics

Enforcement of a relation with model k as the target model has the following semantics:

For each valid binding of variables of the guard (when) clause and variables of domains other than the target domain k, that satisfy the guard condition and source domain patterns and conditions, if there does not exist a valid binding of the remaining unbound variables of domain k that satisfies domain k's pattern and where condition, then create objects (or select and modify if they already exist) and assign properties as specified in domain k pattern. A more formal definition is given below.

Also, for each valid binding of variables of domain k pattern that satisfies domain k condition, if there does not exist a valid binding of variables of the guard clause and source domains that satisfies the guard condition, source domain patterns and where condition, and at least one of the source domains is marked 'checkonly' (or 'enforce', which entails check), then delete the objects bound to the variables of domain k when the following condition is satisfied: delete an object only if it is not required to exist by any other valid binding of the source domains as per the enforcement semantics (i.e. avoid delete followed by an immediate create). A more formal definition is given below.

```
ENFORCE(R, <TYPED_MODEL_K>)
=
CREATE(R, < TYPED_MODEL_K >)
AND      // AND IS DELIBERATE!
DELETE(R, < TYPED_MODEL_K >)

CREATE(R, < TYPED_MODEL_K >)
=
FORALL |<WHEN_VARIABLE_SET>|
(
   <WHEN_CONDITION>
   IMPLIES
   (
     FORALL |(<R_VARIABLE_SET> MINUS (<WHEN_VARIABLE_SET> UNION

                                      <EXCLUSIVE_DOMAIN_K_VARIABLE_SET>))|
    (
       (
          (<DOMAIN_1_PATTERN> AND <DOMAIN_1_CONDITION>)
          AND
          ....
          (<DOMAIN_K-1_PATTERN> AND <DOMAIN_K-1_CONDITION>)
          AND
          (<DOMAIN_K+1_PATTERN> AND <DOMAIN_K+1_CONDITION>)
          AND
          ....
          (<DOMAIN_N_PATTERN> AND <DOMAIN_N_CONDITION>)
       )
       IMPLIES
       (
           NOT EXISTS |<EXCLUSIVE_DOMAIN_K_VARIABLE_SET>|
          (
             (<DOMAIN_K_PATTERN> AND <DOMAIN_K_CONDITION>)
             AND
             <WHERE_CONDITION>
          ) // EXISTS
          IMPLIES
          (
             // ASSERT THAT THERE ARE NO REMAINING FREE VARS OTHER THAN THE OBJECT
               // NODE VARS OF THE DOMAIN PATTERN. PL SEE THE SECTION 'RESTRICTIONS ON
             // EXPRESSIONS USED IN THE RELAIONAL LANGUAGE' FOR A MORE DETAILED
             // DISCUSSION.
             ASSERT(
               (<EXCLUSIVE_DOMAIN_K_VARIABLE_SET> MINUS
                  GETOBJECTVARS(DOMAIN_K_PATTERN))
                  =
                  NULL
             )
             AND
```

```
              FORALL OBJVAR IN (GETOBJECTVARS(DOMAIN_K_PATTERN))
              (
                 CREATEORUPDATE(OBJVAR.BOUNDTEMPLATE, OBJVAR,
                         |(<R_VARIABLE_SET> MINUS
                           <EXCLUSIVE_DOMAIN_K_VARIABLE_SET>)|)
              )
               AND
               <WHERE_CONDITION> // MUST HOLD AFTER UPDATING THE TARGET MODEL
            ) // IMPLIES
          ) // IMPLIES
       ) // FORALL
    ) // IMPLIES
 ) // FORALL
```

CREATEORUPDATE is a predicate that takes as parameters an object template expression, an object variable to be bound, and a variable context as inputs. It evaluates to TRUE when it can bind an object to the object variable that conforms to the object template. The object to be bound may either be selected from the model or created afresh, and assigned properties as specified in the object template. Whether an object is selected from the model or created afresh depends on whether the model already contains an object that matches the key property values, if any, specified in the object template. It evaluates to FALSE when the template expression results in an assignment of a value to a property that clashes with another value set for the same property by another rule in the transformation execution, indicating an inconsistent specification. For primitive types, the values clash when they are different. An object assignment to a link of multiplicity ëoneí clashes if the object being assigned is different from the one that already exists.

createOrUpdate(objectTemplate, unboundObjectVar, boundVarableContext): Boolean

{

1. If the object template contains identifying properties corresponding to at least

one of the keys of the class of the object, then try to locate such an object in the

model; if there is no such object, then create a new object.

2. Bind unboundObjectVar to the object found or created in step 1.

3. Assign properties of the object as specified by the property template items of

the object template.

4. If a property had a different value set by another rule in the same transformation execution, then return FALSE.

5. Return TRUE.

}

```
DELETE(R, <DIRECTION_K>)
=
(
    NOT EXISTS OD IN R.DOMAIN ((OD.DIRECTION != <DIRECTION_K>) AND
                                  OD.ISCHECKED = TRUE)
)
OR
FORALL |<DOMAIN_K_VARIABLE_SET>|
(
   (<DOMAIN_K_PATTERN> AND <DOMAIN_K_CONDITION>)
   IMPLIES
   (
      NOT EXISTS |(<R_VARIABLE_SET> MINUS <DOMAIN_K_VARIABLE_SET>)|
      (
         <WHEN_CONDITION>
         AND
         (<DOMAIN_1_PATTERN> AND <DOMAIN_1_CONDITION>)
         AND
         ....
         (<DOMAIN_K-1_PATTERN> AND <DOMAIN_K-1_CONDITION>)
         AND
         (<DOMAIN_K+1_PATTERN> AND <DOMAIN_K+1_CONDITION>)
         AND
         ....
         (<DOMAIN_N_PATTERN> AND <DOMAIN_N_CONDITION>)
         AND
         <WHERE_CONDITION>
      ) // NOT EXISTS
      IMPLIES
      (
         FORALL OBJVAR IN MAKESET(<DOMAIN_K_VARIABLE_SET>)
         (
            // DELETE THE OBJECT ONLY IF IT IS NOT REQUIRED TO EXIST AS PER ENFORCEMENT
            // SEMANTICS FOR ANY OF THE VALID BINDINGS OF THE OPPOSITE DOMAINS
            NOT EXISTS |<DOMAIN_K_VARIABLE_SET>|   // NEW SCOPE
            (
               (<DOMAIN_K_PATTERN> AND <DOMAIN_K_CONDITION>)
               AND
               EXISTS |(<R_VARIABLE_SET> MINUS <DOMAIN_K_VARIABLE_SET>)|
               (
                  <WHEN_CONDITION>
                  AND
                  (<DOMAIN_1_PATTERN> AND <DOMAIN_1_CONDITION>)
                  AND
                  ....
                  (<DOMAIN_K-1_PATTERN> AND <DOMAIN_K-1_CONDITION>)
                  AND
                  (<DOMAIN_K+1_PATTERN> AND <DOMAIN_K+1_CONDITION>)
                  AND
                  ....
                  (<DOMAIN_N_PATTERN> AND <DOMAIN_N_CONDITION>)
                  AND
                  <WHERE_CONDITION>
               ) // EXISTS
               AND
               BELONGSTO(OBJVAR, MAKESET(<DOMAIN_K_VARIABLE_SET>))
            ) // NOT EXISTS
            IMPLIES
               DELETE(OBJVAR)
         ) // FORALL
      ) // IMPLIES
   ) // IMPLIES
) // FORALL
```