

June 2016



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

Version 1.3

OMG Document Number: formal/2016-06-03

Standard document URL: <http://www.omg.org/spec/QVT/1.3>

Machine Consumable Files:

Normative :

<http://www.omg.org/cgi-bin/doc?ptc/15-10-05>

<http://www.omg.org/spec/QVT/20151201/EMOF.xmi>

<http://www.omg.org/spec/QVT/20151201/EssentialOCL.xmi>

<http://www.omg.org/spec/QVT/20151201/ImperativeOCL.xmi>

<http://www.omg.org/spec/QVT/20151201/PrimitiveTypes.xmi>

<http://www.omg.org/spec/QVT/20151201/QVTBase.xmi>

<http://www.omg.org/spec/QVT/20151201/QVTCore.xmi>

<http://www.omg.org/spec/QVT/20151201/QVTOperational.xmi>

<http://www.omg.org/spec/QVT/20151201/QVTRelation.xmi>

<http://www.omg.org/spec/QVT/20151201/QVTTemplate.xmi>

Informative:

<http://www.omg.org/cgi-bin/doc?ptc/15-10-06>

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 © 2016 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, 109 Highland Avenue, Needham, MA 02494, U.S.A.

TRADEMARKS

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

COMPLIANCE

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

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

OMG's Issue Reporting Procedure

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

Table of Contents

	Preface.....	xix
1	Scope.....	1
2	Conformance	1
2.1	Conformance Points	1
2.2	Language Dimension.....	2
2.3	Interoperability Dimension	2
2.4	EMOF and CMOF Compliance.....	2
2.5	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.....	6
6	QVT Overview.....	9
6.1	Two Level Declarative Architecture	9
6.1.1	QVTr - Relations Language.....	9
6.1.2	QVTc - Core Language	9
6.1.3	Virtual Machine Analogy.....	10
6.2	Imperative Implementations.....	10
6.2.1	QVTo - Operational Mappings Language.....	10
6.2.2	Black Box Implementations	10
6.3	Execution Scenarios	11
6.4	MOF Metamodels	11
6.5	OCL usage in QVT	12

7	The Relations Language	13
7.1	Transformations and Model Types	13
7.1.1	Import statements	13
7.1.2	Transformation Execution Direction	13
7.2	Relations and Domains	14
7.2.1	When and Where Clauses	14
7.2.2	Top-level Relations	15
7.2.3	Check and Enforce	15
7.2.4	Primitive Domains	16
7.3	Pattern Matching	16
7.4	Keys and Object Creation Using Patterns	17
7.5	Restrictions on Expressions	18
7.6	Change Propagation	19
7.7	In-place Transformations	19
7.8	Integrating Black-box Operations with Relations	19
7.9	Executing a Transformation in Checkonly mode	20
7.10	Detailed Semantics	20
7.10.1	Checking Semantics	21
7.10.2	Enforcement Semantics	21
7.10.3	Pattern Matching Semantics	21
7.10.3.1	Introduction	21
7.10.3.2	Pattern Infrastructure	22
7.10.3.3	Patterns Specifying Collections	23
7.10.3.4	QVT Template Expressions	23
7.11	Abstract Syntax and Semantics	24
7.11.1	QVTBase Package	24
7.11.1.1	Transformation	25
7.11.1.2	TypedModel	27
7.11.1.3	Domain	27
7.11.1.4	Rule	28
7.11.1.5	Function	28
7.11.1.6	FunctionParameter	29
7.11.1.7	Predicate	29
7.11.1.8	Pattern	29
7.11.2	QVTTemplate Package	29

7.11.2.1	TemplateExp	29
7.11.2.2	ObjectTemplateExp	30
7.11.2.3	CollectionTemplateExp	31
7.11.2.4	PropertyTemplateItem	31
7.11.3	QVTRelation Package	33
7.11.3.1	RelationalTransformation	33
7.11.3.2	Relation	33
7.11.3.3	RelationDomain	35
7.11.3.4	DomainPattern	36
7.11.3.5	Key	36
7.11.3.6	RelationImplementation	37
7.11.3.7	RelationDomainAssignment	37
7.11.3.8	RelationCallExp	37
7.12	Standard Library	38
7.13	Concrete Syntax	38
7.13.1	Compilation Units	38
7.13.2	Keywords	38
7.13.3	Comments	38
7.13.4	Relations Textual Syntax Grammar	38
7.13.5	Expressions Syntax (extensions to OCL)	39
7.13.6	Graphical Syntax	40
7.13.6.1	Introduction	40
7.13.6.2	Graphical Notation Elements	44
7.13.6.3	Variations in Graphical Notation	45
7.13.7	Concrete textual syntax to abstract syntax mapping	45
8	Operational Mappings	65
8.1	Overview	65
8.1.1	Operational transformations	65
8.1.2	Model Types	66
8.1.3	Extents, Models and Model Parameters	66
8.1.3.1	Implementation responsibilities	66
8.1.3.2	Extents	66
8.1.3.3	Models	67
8.1.3.4	Object Containment and Extent Residence	67
8.1.3.5	Orphans	68
8.1.4	Libraries	68
8.1.5	Mapping operations	69
8.1.6	Object Creation and Population in Mapping Operations	70
8.1.7	Inlining Mapping Operations	71
8.1.8	Using Constructor Operations	72

8.1.9	Helpers	73
8.1.10	Intermediate Data	73
8.1.11	Tracing and Resolving	74
8.1.11.1	Trace Records	74
8.1.11.2	The inhibition and instantiation sections	75
8.1.11.3	resolve() - Resolution of target objects by Type	76
8.1.11.4	resolveIn() - Resolution of target objects by Mapping	77
8.1.11.5	invresolve() - Resolution of source objects by Type or Mapping	77
8.1.11.6	resolveOne() - Resolution of a single source or target object by Type or Mapping	78
8.1.11.7	Late resolution	78
8.1.11.8	Persisted Trace Data	79
8.1.12	Updating Objects and Resolving Object References	79
8.1.13	Composing Transformations	81
8.1.14	Mapping Overloading	81
8.1.14.1	Explicit Disjuncts	82
8.1.14.2	Implicit Disjuncts	82
8.1.14.3	Disjunct candidates	83
8.1.15	Reuse Facilities for Mapping Operations	83
8.1.16	Type Extensions	85
8.1.17	Imperative Expressions	85
8.1.18	Pre-defined Variables: this, self, and result	86
8.1.19	Null	86
8.1.20	Invalid	86
8.1.21	Advanced Features: dynamic definition and parallelism	87
8.2	Abstract Syntax and Semantics	88
8.2.1	The QVTOperational Package	88
8.2.1.1	OperationalTransformation	89
8.2.1.2	Library	95
8.2.1.3	Module	95
8.2.1.4	ModuleImport	97
8.2.1.5	ModelParameter	97
8.2.1.6	ModelType	98
8.2.1.7	VarParameter	100
8.2.1.8	DirectionKind	100
8.2.1.9	ImportKind	100
8.2.1.10	ImperativeOperation	101
8.2.1.11	EntryOperation	101
8.2.1.12	Helper	102
8.2.1.13	Constructor	103
8.2.1.14	ContextualProperty	104
8.2.1.15	MappingOperation	104
8.2.1.16	MappingParameter	107
8.2.1.17	OperationBody	108
8.2.1.18	ConstructorBody	109

8.2.1.19 MappingBody	109
8.2.1.20 ImperativeCallExp	111
8.2.1.21 MappingCallExp	111
8.2.1.22 ResolveExp	113
8.2.1.23 ResolveInExp	115
8.2.1.24 ObjectExp	116
8.2.2 The ImperativeOCL Package	117
8.2.2.1 ImperativeExpression	117
8.2.2.2 BlockExp	121
8.2.2.3 ComputeExp	121
8.2.2.4 WhileExp	121
8.2.2.5 ImperativeLoopExp	122
8.2.2.6 ForExp	122
8.2.2.7 ImperativeIterateExp	124
8.2.2.8 SwitchExp	127
8.2.2.9 AltExp	127
8.2.2.10 VariableInitExp	128
8.2.2.11 AssignExp	129
8.2.2.12 UnlinkExp	130
8.2.2.13 TryExp	131
8.2.2.14 CatchExp	131
8.2.2.15 RaiseExp	132
8.2.2.16 ReturnExp	132
8.2.2.17 BreakExp	133
8.2.2.18 ContinueExp	133
8.2.2.19 LogExp	133
8.2.2.20 AssertExp	134
8.2.2.21 SeverityKind	134
8.2.2.22 InstantiationExp	135
8.2.2.23 Typedef (deprecated)	135
8.2.2.24 ListType	136
8.2.2.25 DictionaryType	136
8.2.2.26 TemplateParameterType	137
8.2.2.27 DictLiteralExp	137
8.2.2.28 DictLiteralPart	137
8.2.2.29 ListLiteralExp	138
8.3 Standard Library	138
8.3.1 Predefined types	138
8.3.1.1 Transformation	138
8.3.1.2 Model	138
8.3.1.3 Status	139
8.3.1.4 Exception	139
8.3.1.5 StringException	139
8.3.1.6 AssertionFailed	139
8.3.2 Synonym types and synonym operations	139
8.3.3 Operations on objects	140
8.3.3.1 repr	140
8.3.4 Operations on Elements	140

8.3.4.1	_localId	140
8.3.4.2	_globalId	140
8.3.4.3	metaClassName	140
8.3.4.4	subobjects	140
8.3.4.5	allSubobjects	140
8.3.4.6	subobjectsOfType	141
8.3.4.7	allSubobjectsOfType	141
8.3.4.8	subobjectsOfKind	141
8.3.4.9	allSubobjectsOfKind	141
8.3.4.10	clone	141
8.3.4.11	deepclone	141
8.3.4.12	markedAs	141
8.3.4.13	markValue	142
8.3.4.14	stereotypedBy	142
8.3.4.15	stereotypedStrictlyBy	142
8.3.5	Operations on Models	142
8.3.5.1	objects	142
8.3.5.2	objectsOfKind	142
8.3.5.3	objectsOfType	142
8.3.5.4	rootObjects	142
8.3.5.5	addElement	142
8.3.5.6	removeElement	143
8.3.5.7	asTransformation	143
8.3.5.8	copy	143
8.3.5.9	createEmptyModel	143
8.3.6	Operations on Transformations	143
8.3.6.1	transform	143
8.3.6.2	parallelTransform	143
8.3.6.3	wait	143
8.3.7	Operations on Status	144
8.3.7.1	raisedException	144
8.3.7.2	failed	144
8.3.7.3	succeeded	144
8.3.8	Operations on Dictionaries	144
8.3.8.1	get	144
8.3.8.2	hasKey	144
8.3.8.3	defaultget	144
8.3.8.4	put	144
8.3.8.5	clear	145
8.3.8.6	size	145
8.3.8.7	values	145
8.3.8.8	keys	145
8.3.8.9	isEmpty	145
8.3.9	Operations on Lists	145
8.3.9.1	=	145
8.3.9.2	<>	145
8.3.9.3	add	146
8.3.9.4	append	146
8.3.9.5	asBag	146

8.3.9.6	asList	146
8.3.9.7	asOrderedSet	146
8.3.9.8	asSequence	146
8.3.9.9	asSet	147
8.3.9.10	at	147
8.3.9.11	clone	147
8.3.9.12	count	147
8.3.9.13	deepclone	147
8.3.9.14	excludes	147
8.3.9.15	excludesAll	147
8.3.9.16	excludesAll	147
8.3.9.17	excluding	148
8.3.9.18	first	148
8.3.9.19	flatten	148
8.3.9.20	includes	148
8.3.9.21	includesAll	148
8.3.9.22	includesAll	148
8.3.9.23	including	148
8.3.9.24	indexOf	149
8.3.9.25	insertAt	149
8.3.9.26	insertAt	149
8.3.9.27	isEmpty	149
8.3.9.28	joinfields	149
8.3.9.29	last	149
8.3.9.30	max	150
8.3.9.31	min	150
8.3.9.32	notEmpty	150
8.3.9.33	prepend	150
8.3.9.34	product	150
8.3.9.35	remove	150
8.3.9.36	removeAll	151
8.3.9.37	removeAll	151
8.3.9.38	removeAt	151
8.3.9.39	removeFirst	151
8.3.9.40	removeLast	151
8.3.9.41	reverse	151
8.3.9.42	selectByKind	152
8.3.9.43	selectByType	152
8.3.9.44	size	152
8.3.9.45	subSequence	152
8.3.9.46	sum	152
8.3.9.47	union	152
8.3.10	Iterations on Lists	153
8.3.11	Operations on Collections	153
8.3.11.1	asList	153
8.3.11.2	clone	153
8.3.11.3	deepclone	154
8.3.12	Operations on Bags	154
8.3.12.1	asList	154

8.3.12.2 clone	154
8.3.12.3 deepclone	154
8.3.13 Operations on OrderedSets	154
8.3.13.1 asList	154
8.3.13.2 clone	154
8.3.13.3 deepclone	154
8.3.14 Operations on Sequences.....	155
8.3.14.1 asList	155
8.3.14.2 clone	155
8.3.14.3 deepclone	155
8.3.15 Operations on Sets.....	155
8.3.15.1 asList	155
8.3.15.2 clone	155
8.3.15.3 deepclone	155
8.3.16 Operations on Strings.....	155
8.3.16.1 format	156
8.3.16.2 length.....	156
8.3.16.3 substringBefore	156
8.3.16.4 substringAfter	156
8.3.16.5 toLower.....	156
8.3.16.6 toUpper.....	156
8.3.16.7 firstToUpper.....	156
8.3.16.8 lastToUpper	156
8.3.16.9 indexOf	157
8.3.16.10 endsWith	157
8.3.16.11 startsWith	157
8.3.16.12 trim	157
8.3.16.13 normalizeSpace.....	157
8.3.16.14 replace.....	157
8.3.16.15 match.....	157
8.3.16.16 equalsIgnoreCase	157
8.3.16.17 find.....	158
8.3.16.18 rfind	158
8.3.16.19 isQuoted	158
8.3.16.20 quotify.....	158
8.3.16.21 unquotify.....	158
8.3.16.22 matchBoolean	158
8.3.16.23 matchInteger	158
8.3.16.24 matchFloat.....	158
8.3.16.25 matchReal	158
8.3.16.26 matchIdentifier	159
8.3.16.27 asBoolean	159
8.3.16.28 asInteger	159
8.3.16.29 asFloat.....	159
8.3.16.30 asReal	159
8.3.16.31 startStrCounter	159
8.3.16.32 getStrCounter	159
8.3.16.33 incrStrCounter	159
8.3.16.34 restartAllStrCounter.....	159

8.3.16.35	addSuffixNumber	160
8.3.17	Operations on numeric types	160
8.3.18	Operations on Classifiers	160
8.3.18.1	Classifier::allInstances() : Set(T)	160
8.3.19	Predefined tags	160
8.4	Concrete Syntax	161
8.4.1	Files	161
8.4.2	Comments	161
8.4.3	Strings	161
8.4.4	Shorthands used to invoke specific pre-defined operations	162
8.4.5	Other Language Shorthands	163
8.4.6	Notation for Metamodels	163
8.4.7	EBNF	164
8.4.7.1	Deprecated syntax	171
8.4.8	Scoped Identifiers	171
8.4.8.1	Type Identifiers	171
8.4.8.2	Transformation Identifiers	172
8.4.8.3	Mapping Identifiers	172
9	The Core Language	173
9.1	Comparison with the Relational Language	173
9.2	Transformations and Model Types	173
9.3	Mappings	173
9.4	Patterns	174
9.5	Bindings	175
9.6	Binding Dependencies	175
9.7	Guards	176
9.8	Bottom Patterns	176
9.9	Checking	176
9.9.1	Checking Formally Defined	177
9.10	Enforcement	178
9.10.1	Enforcement formally defined	179
9.11	Realized Variables	180
9.12	Assignments	180

9.13	Enforcement Operations	181
9.14	Mapping Refinement.....	181
9.15	Mapping Composition	182
9.16	Functions	182
9.17	Abstract Syntax and Semantics	182
9.17.1	CorePattern	183
9.17.2	Area.....	184
9.17.3	GuardPattern.....	185
9.17.4	BottomPattern	185
9.17.5	CoreDomain	185
9.17.6	Mapping.....	186
9.17.7	RealizedVariable	187
9.17.8	Assignment.....	187
9.17.9	PropertyAssignment.....	188
9.17.10	VariableAssignment	188
9.17.11	EnforcementMode	189
9.17.12	EnforcementOperation	189
9.18	Concrete Syntax	189
10	Relations to Core Transformation	191
10.1	Mapping Approach.....	191
10.2	Mapping Rules.....	192
10.3	Relational Expression of Relations To Core Transformation	198
11	QVT For CMOF.....	229
11.1	The QVT Metamodel for CMOF.....	229
11.2	Semantics Specificities	229
Annex A:	Additional Examples	231
A.1	Relations Examples	231
A.1.1	UML to RDBMS Mapping	231
A.1.1.1	Overview	231
A.2	Operational Mapping Examples.....	237
A.2.1	Book To Publication example	237
A.2.2	Encapsulation example	237

A.2.3 Uml to Rdbms.....	238
A.2.4 SPEM UML Profile to SPEM metamodel	241
A.3 Core Examples	243
A.3.1 UML to RDBMS Mapping	243
Annex B: Semantics of Relations.....	251
B.1 Checking Semantics	251
B.2 Enforcement Semantics.....	252
INDEX.....	257

Preface

About the Object Management Group

OMG

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

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

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

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. All OMG Specifications are available from the OMG website at:

<http://www.omg.org/spec>

Specifications within the Catalog are organized by the following categories:

Business Modeling Specifications

Middleware Specifications

- **CORBA/IIOP**
- **Data Distribution Services**
- **Specialized CORBA**

IDL/Language Mapping Specifications

Modeling and Metadata Specifications

- **UML, MOF, CWM, XMI**
- **UML Profile**

Modernization Specifications

Platform Independent Model (PIM), Platform Specific Model (PSM), Interface Specifications

- CORBAServices
- CORBAFacilities

OMG Domain Specifications

CORBA Embedded Intelligence Specifications

CORBA Security Specifications

Signal and Image Processing

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

OMG Headquarters
109 Highland Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: pubs@omg.org

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

Typographical Conventions

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

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

Courier - 10 pt. Bold: Programming language elements.

Helvetica/Arial - 10 pt: Exceptions

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

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to <http://issues.omg.org/issues/create-new-issue>.

1 Scope

This specification provides the architecture, languages, operational mappings, and core language for the MOF 2.0 Query, View, and Transformation (QVT) specification. The specification defines three related transformation languages: Relations, Operational Mappings, and Core.

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 “Conformance Points.”

2.1 Conformance Points

Any combination of two named levels, one from each dimension, constructs a conformance point. Figure 2.1 specifies the different possible conformance points. A tool can claim to be conformant according to one or more of these conformance points.

		Interoperability			
		Syntax Executable	XMI Executable	Syntax Exportable	XMI Exportable
Language	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:

1. **Core:** The Core language is described in Clause 9. This includes the ability to insert black-box implementations via MOF operations as specified.
2. **Relations:** The Relations language is described in Clause 7. This includes the ability to insert black-box implementations via MOF operations as specified.
3. **Operational:** The Operational Mappings language is described in Clause 8.

2.3 Interoperability Dimension

The interoperability dimension has four named interoperability levels:

1. **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.
2. **XMIExecutable:** An implementation shall provide a facility to import or read, and then execute an **XMI serialization** of a transformation description that 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.
3. **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.
4. **XMIExportable:** An implementation shall provide a facility to export a model-to-model transformation into its **XMI serialization** that conforms to the MOF meta-model of the language given by the language dimension.

2.4 EMOF and CMOF Compliance

A QVT tool may declare to be EMOF-compliant or CMOF-compliant (possibly both) depending on the kind of models that it is capable of working with. The same dimensions serving to characterize QVT-EMOF compliant implementations (in Figure 2.1) are applicable to QVT CMOF-compliant implementations.

Note however that the XMI for an EMOF-compliant QVT tool is not the same as the XMI for a CMOF-compliant QVT tool since the XMI generation rules for CMOF are distinct from the corresponding generation rules for EMOF.

2.5 Conformance of QVT Definitions

Figure 2.1 defines compliance points for tools. We address here conformance of transformation definitions written in QVT.

The authors of a QVT transformation definition shall indicate:

1. The language dimension being used, and
2. whether black-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 2.5 Core Specification: <http://www.omg.org/spec/MOF/2.5/PDF>
- OCL 2.4 Specification: <http://www.omg.org/spec/OCL/2.4/PDF>

4 Definitions and Terms

4.1 Glossary

Area	In the context of a <i>core mapping</i> an area is a pair of <i>patterns</i> , consisting of a <i>guard pattern</i> and a <i>bottom pattern</i> .
Bottom Pattern	A pattern that is checked or enforced for the bindings generated by the <i>guard pattern</i> of the same <i>area</i> of a <i>mapping</i> , and other patterns that this area is related to for execution in a particular direction.
Core Domain	A specialized kind of <i>domain</i> that forms part of a <i>mapping</i> . A core domain is also an <i>area</i> that defines a pair of <i>patterns</i> , consisting of a <i>guard pattern</i> and a <i>bottom pattern</i> .
Core Transformation	A transformation definition formalized by a list of <i>core mappings</i> .
Domain	<p>A domain is a distinguished set of variables to be matched in a <i>typed model</i>. It is related to other domains by a transformation <i>rule</i>.</p> <p>Domain is an abstract type in the QVTBase package that has concrete types RelationDomain and CoreDomain.</p> <p>Domains have flags to indicate whether they are <i>checkonly</i> or <i>enforced</i>. 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.</p>
Guard Pattern	A <i>pattern</i> that must hold as a precondition to the application of the <i>bottom pattern</i> related to it in an <i>area</i> 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 <i>trace instances</i>) 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 <i>relations</i> 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 <i>identifying properties</i> , and are used when matching <i>template patterns</i> 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 <i>rule</i> in a Core transformation description. It is an <i>area</i> , consisting of a pair of <i>patterns</i> that are designed to locate or create instances of the <i>trace classes</i> that store the relationships between models. It also owns a set of core domains that 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 <i>operational transformation</i> a model type represents the type of the models involved in the transformation. A model type is defined by a metamodel, a conformance kind - 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	<p>A relation is a subset of an N-ary product of sets, $A_1 \times A_2 \times \dots \times A_N$, and may be represented as a set of N-tuples (a_1, a_2, \dots, a_N). In the context of MOF, each set A_K, called a <i>domain</i>, is a MOF type, and a relation will be populated by tuples referring to model elements of those types that exist in MOF extents.</p> <p>A Relation in the QVT specification also defines the <i>rules</i> by which the exact subset of model elements to be related is determined. These <i>rules</i> comprise variables of additional MOF types, <i>template pattern</i> matches on the structure of the <i>relation domains</i> that bind values to the variables, OCL constraints over the <i>relation domains</i> and variables of the relation, and assertions that other relations hold.</p> <p>Relations imply the existence of equivalent <i>trace classes</i> that have properties for each of its domains, and whose set of <i>trace instances</i> are equivalent to the relation's population of tuples.</p>
Relational Transformation	A transformation definition that is formalized by a list of relations.
Relation Domain	<p>A specialization of the concept of a <i>domain</i>. In a Relation a domain is a type that may be the root of a <i>template pattern</i>, which can match any model element navigable from that type.</p> <p>A domain implies the existence of a property of the same type in a <i>trace class</i> derived from the relation to which it belongs.</p>
Template Pattern	A template pattern is a combination of a literal as defined in OCL that 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 <i>relation domain</i> . They provide a terse, user-friendly expression of what can be quite verbose expressions in ordinary OCL. Template Patterns rely on <i>identifying properties</i> of classes, defined by <i>keys</i> 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 (<i>trace instances</i>) 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* that 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.

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

Clause 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.

Clause 8 - Operational Mappings, provides the details of this language in terms of imperative mappings, which extend the QVTRelation package introduced in the previous clause, 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.

Clause 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.

Clause 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.

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
- Nomos Software
- Open Canarias
- 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), Radek Dvorak (Borland), Ed Willink (Willink Transformations Ltd).

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, Karl Frank, Michael Wagner, Grégoire Dupé, Florian Guillard, Nicolas Rouquette, Victor Sanchez, Adolfo Sanchez-Barbudo Herrera, Sergey Boyko, Chistopher Gerking.

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 that 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.

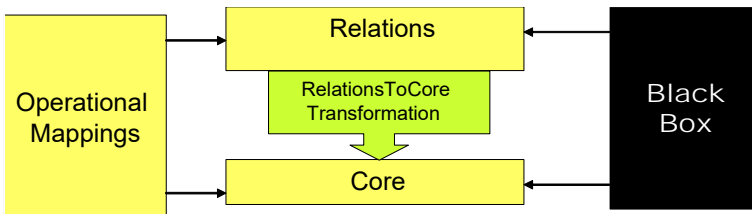


Figure 6.1 - Relationships between QVT metamodels

6.1.1 QVTr - Relations Language

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 Clause 9.10 “Enforcement” on page 178, 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 Clause 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 QVTc - Core Language

This is a small model/language that 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 Clause 9.

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 that 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 defines a class that 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 QVTo - 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 Clause 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 is 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 that will be 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, possibly breaking encapsulation. Black-box implementations do not 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 through 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 (Clause 9.17), QVTRelation (Clause 7.11.3), and QVTOperational (Clause 8.2).

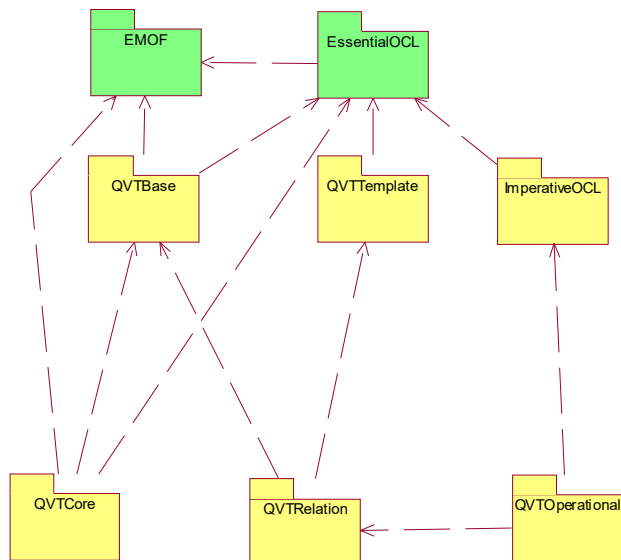


Figure 6.2 - Package dependencies in the QVT specification

The QVTBase package defines common structure for transformations (Clause 7.11.1).

In addition the QVTRelation package uses template pattern expressions defined in the QVTTemplateExp package (Chapter 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 (Clause 8.2.2).

All of QVT depends on the EssentialOCL package from OCL 2.4, and all of the language packages depend on EMOF.

6.5 OCL usage in QVT

The QVT languages introduce controlled mechanisms for object mutation that conflict with OCL's expectations of stability. These conflicts are clarified in this subclause.

Essential OCL is used as the expressions language for QVT Relations, Core and Operational Mappings. It is a side effect free language that supports evaluation of constraints on the unchanging state of the objects in a model.¹

The declarative QVTc and QVTtr languages may cascade mappings in which OCL evaluations access intermediate objects. These evaluations occur predictably for either an old or a new state of an object. An old state is inherently stable. The new state is stabilized by the declarative computation of values before usage. An exception however arises for *allInstances()* for which the declarative mapping execution order is difficult for a programmer to predict with certainty. *allInstances()* is therefore defined to return the final state of instances in the domain within which *allInstances()* is invoked.

Imperative OCL is an extension of Essential OCL with additional facilities to realize the side effects required by QVT Operational Mappings.

The imperative QVTo language performs object mutations as it advances from one program state to another in a predictable order. OCL evaluation may be used within each state. The functionality of *allInstances()* is clarified in the Clause 8.3.18.1.

1.Operation pre- and post-conditions and @pre extend this to two states.

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:

```
import SimpleUML : 'http://www.omg.org/spec/UML/20131001/UML.xmi';  
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.

A declarative transformation is not restricted to a fixed number of models. Collections of input models may be provided and transformed as a single multi-rooted input model.

7.1.1 Import statements

An import statement introduces definitions of unqualified, or first qualifier names for resolution as if defined at the root package of the transformation.

The import of a document URI (such as `http://www.omg.org/spec/UML/20131001/UML.xmi`) makes the root element (the UML Package) available to resolve references (to UML). Additionally, the optional alias (`SimpleUML`) also makes the root element available to resolve references to the alias name.

Alternatively the import of a namespace URI (such as `http://www.omg.org/spec/UML/20131001`) makes the referenced element available in a similar way. It is not specified how an implementation locates elements corresponding to namespace URIs or how it distinguishes namespace URI access from document URI access. (A plausible implementation maintains a catalog of known namespace URIs to support their resolution leaving everything else to be interpreted as a document URI.)

The import may use further qualification such as `'http://www.omg.org/spec/UML/20131001'::Activities` to provide a finer grained import. A trailing wildcard as in `'http://www.omg.org/spec/UML/20131001'::*` imports all names defined by the reference preceding the wild card. For UML, this is the Packages Actions, Activities, Classification, An alias and a wildcard cannot be used together.

7.1.2 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 Clause 7.3 and Clause 7.4 below.

In the following example two domains are declared that will match elements in the “uml” and “rdbms” models respectively. Each domain specifies a simple pattern - 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 also can be 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: 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

When a transformation is evaluated, the transformation executes in the direction of the domain specified as the target. Whether or not a relation is enforced is determined by the target domain, which may be marked as **checkonly** or **enforced**. When a relation executes in the direction of a **checkonly** domain, the domain is simply checked to see whether there exists a valid match in the relevant model that satisfies the relationship. When a relation executes in the direction of an **enforced** domain, if checking fails, the target model is modified so as to satisfy the relation.

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 - 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.

Clause 7.10 “Detailed Semantics” on page 20 contains a more detailed description of the checking and enforcement semantics.

7.2.4 Primitive Domains

Simple data such as configuration information or constants may be passed as parameters to a relation using primitive domains. A primitive domain is identified by primitive keyword and no domain name. A primitive domain is neither checkable nor enforceable.

```
relation Outer {
  checkonly domain source s:Source {};
  enforce domain target t:Target {};
  where {
    Inner(s,t,'target');
  }
}
relation Inner {
  checkonly domain source s:Source {name=pn};
  enforce domain target t:Target {name=separator + pn};
  primitive domain separator:String;
}
```

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 that 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 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.

Sometimes it may be necessary to use a non-navigable opposite role in the specification of object templates.

E.g.

```
domain myModel a:Attribute {name = n, opposite(Class.attribute) = c:Class{}};
```

This specifies that the pattern will only match if the class the attribute belongs to is equal to ‘c.’ This is equivalent to the following template definition with condition:

```
domain myModel a:Attribute {name = n} {c.attribute.includes(a)};
```

But the ‘opposite’ notation allows this to be specified in the template itself, leading to more compact specifications.

Clause 7.10 “Detailed Semantics” on page 20 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 - 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; 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 “foo” 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.

Sometimes it may be necessary to use a non-navigable opposite role in the specification of a key. For instance, an attribute is identified by a combination of its name and the owning class. Suppose we have a meta model in which the 'Class' to 'Attribute' association 'attribute' is only navigable from Class to Attribute. We should still be able to use this association in the key specification of Attribute.

E.g.

key Attribute {name, opposite(Class.attribute)};

This specifies that an attribute is uniquely identified by its name and the class it belongs to.

7.5 Restrictions on Expressions

In order to guarantee executability (i.e., there exists a bounded algorithm to enforce a relation in the direction of a given target model) 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 occurrences 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 Clause 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 one or more source models are bound to one or more target models at runtime. Execution proceeds as if the source and target models are distinct with an atomic update of non-distinct models occurring on completion of the transformation. This implies that an implementation that operates in-place must take copies of the old state to avoid confusion with updated new state.

Execution of a transformation should return a Boolean status to indicate whether any changes were made to target models. This status enables transformation applications to repeat execution until no changes occur where that is appropriate for the application.

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 - an output parameter corresponding to the enforced domain, and an input parameter corresponding to each of the other domains.

The Relations that 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 that 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 clause provides a detailed description of the semantics of relations. A more formal semantics are given in Clause 10 “Relations to Core Transformation” 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 Clause 7.2.3 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 Annex 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 clause 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 Clause 7.10.3.3 "Patterns Specifying Collections" on page 23 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.

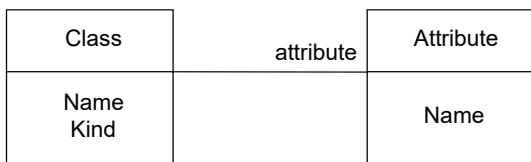


Figure 7.1 - Simple UML Model

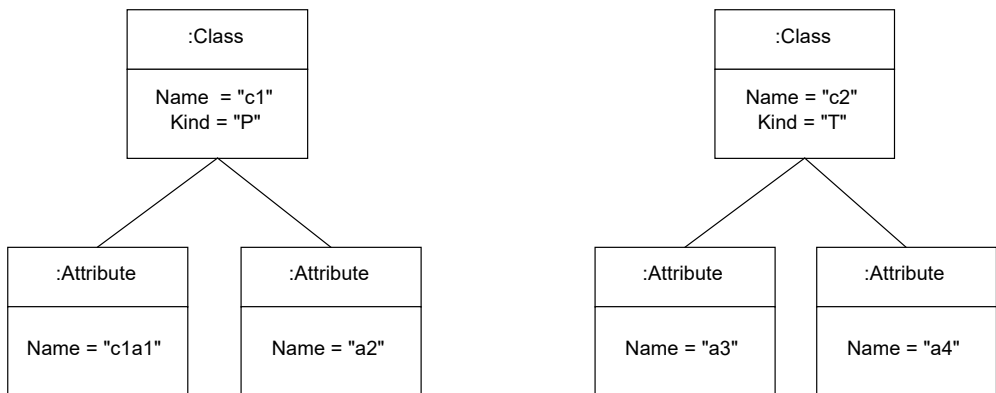


Figure 7.2 - Instance Model

7.10.3.2 Pattern Infrastructure

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

```

Pattern =
{
  e1: <classname1>, e2: <classname2> .... en:<classnameN>
  l1: <assoc1> (ei, ej) .... lm:<assocM>(eu, ew)
  where <predicate>
}

```

A pattern can be viewed as a graph where the pattern elements, e_1, e_2, \dots, e_n , with types $\langle \text{classname}_1 \rangle, \langle \text{classname}_2 \rangle, \dots, \langle \text{classname}_n \rangle$ respectively, are the nodes of the graph and pattern links l_1, l_2, \dots, 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, \dots, o_n , matches a pattern as described above if and only if:

- o_1 is of type $\langle \text{classname}_1 \rangle$ or one of its subtypes, and o_2 is of type $\langle \text{classname}_2 \rangle$ or one of its subtypes, and so on
- o_i and o_j are linked by association $\langle \text{assoc}_1 \rangle$ and o_u and o_w are linked by association $\langle \text{assoc}_2 \rangle$, and so on
- There exists one or more bindings of values to free variables such that $\langle \text{predicate} \rangle$ evaluates to true when references to e_1, e_2, \dots, e_n are replaced by o_1, o_2, \dots, 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 $\langle \text{predicate} \rangle$ 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 7.2 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 - 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 7.3 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 (Clause 7.11.2 “QVTTemplate Package” on page 29) 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 - 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 - X and X+Y.

Instance diagrams omit the usual UML decorations, so the reader should note that a top to bottom composition layout is used in Figure 7.3. Thus the PropertyTemplateItem for 'attribute' is composed by the ObjectTemplateItem for 'Class' and composes the ObjectTemplateItem for 'Attribute'.

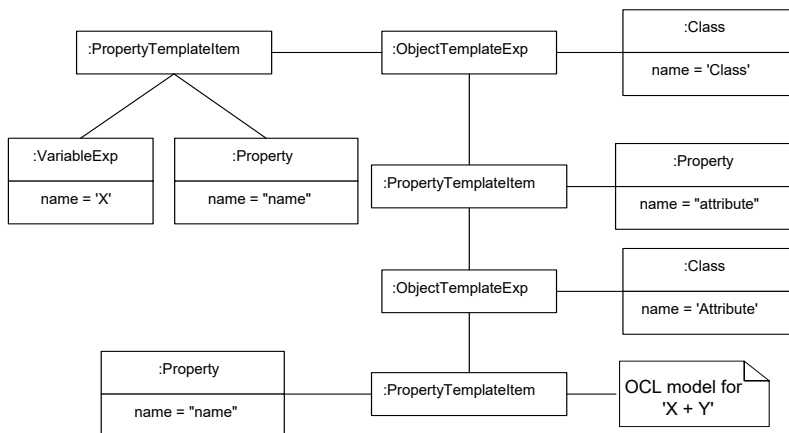


Figure 7.3 - Example Pattern Instance

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 *variables* in OCL expressions. These classes are extended in language-specific packages to provide language-specific semantics.

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 packages of the QVT Relation formalism are not shaded. Within the class descriptions, metaclasses and meta-properties of the metamodel are rendered in courier font. 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.

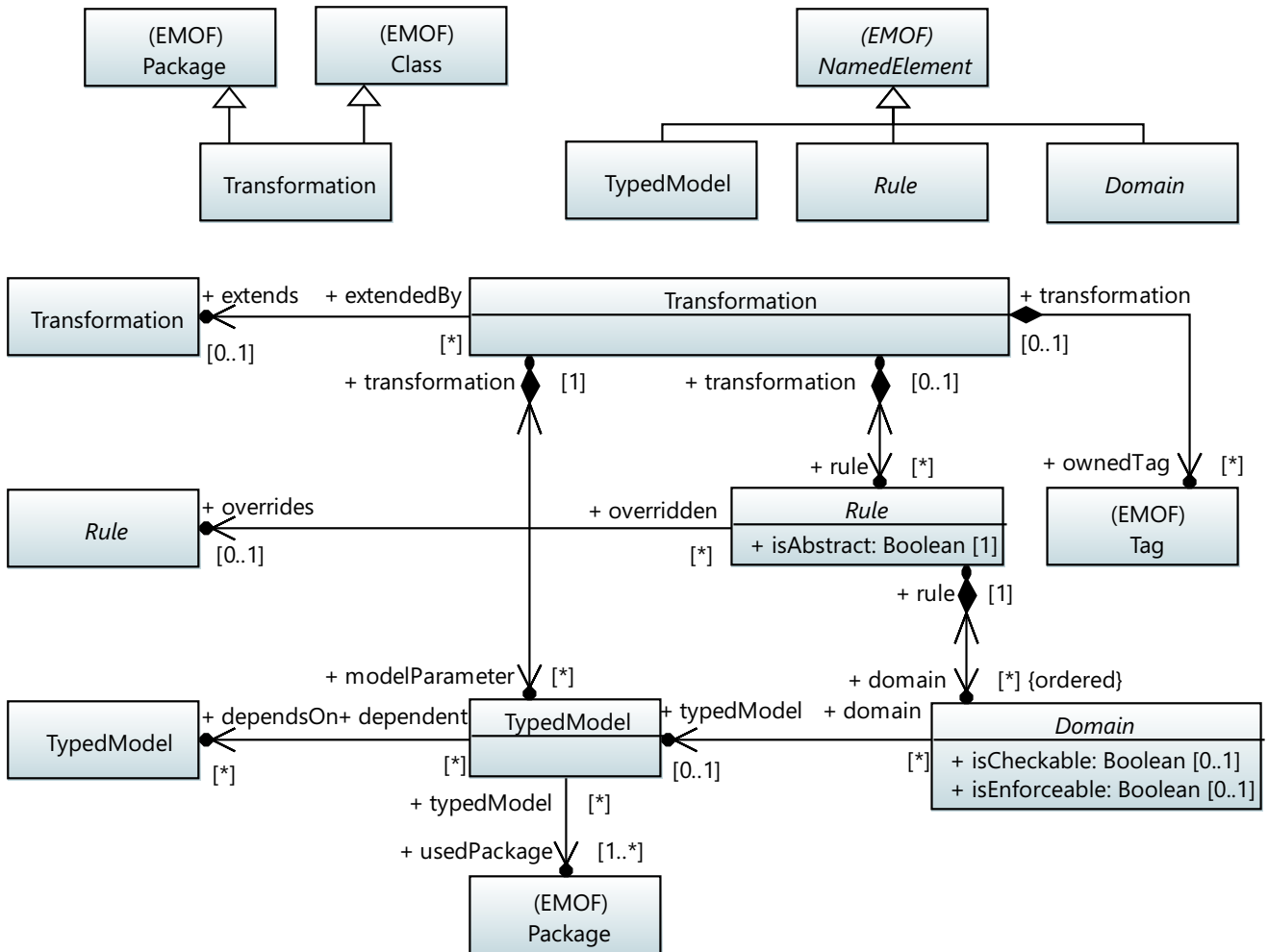


Figure 7.4 - QVTBase Package - Transformations and Rules

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.

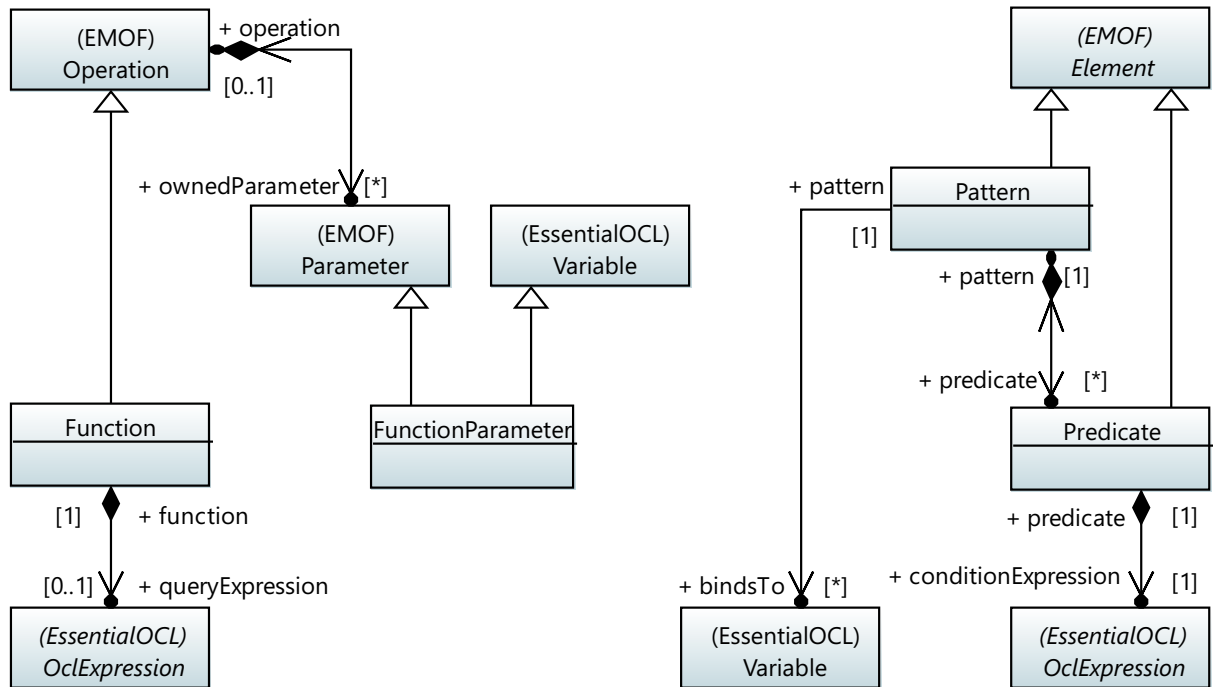


Figure 7.5 - QVTBase Package - Patterns and Functions

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.

Remark: Instances of parameterized types defined in different transformations denote the same type if the base types are the same: for instance ‘Tx1::Set(Integer)’ and ‘Tx2::Set(Integer)’ denote the same type ‘Set(Integer).’

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.

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

All the types being defined by this transformation. Specifically this includes any composite type used to define the type of a variable or a parameter, for instance a ‘Set(MyMetaclass)’ parameterized type instance.

7.11.1.2 TypedModel

A *typed model* specifies a typed parameter of a *transformation*. Explicit external parameters have a non-null name. An implicit parameter such as the QVTc middle model may have a null name. At runtime, a model that 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 modeled 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 Clause 7.11.3.3 “RelationDomain” and Clause 9.17.5 “CoreDomain” for details).

A domain may be marked as *checkable* or *enforceable*. 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 [0..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 Clause 7.11.3.2 “Relation” and Clause 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.

An abstract rule provides functionality that can be exploited by refined rules. An abstract rule is never matched directly and so never executes directly.

Superclasses

NamedElement

Associations

domain: Domain [*] {composes}

The domains owned by this rule.

transformation: Transformation[0..1]

The transformation that owns this rule.

overrides: Rule [0..1]

The rule that this rule overrides.

isAbstract: Boolean [1]

indicates that the rule is abstract. Default is false.

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. Since a function is side-effect free, it is often called a query. 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* that 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 [*]`

The set of variables that 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

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*.

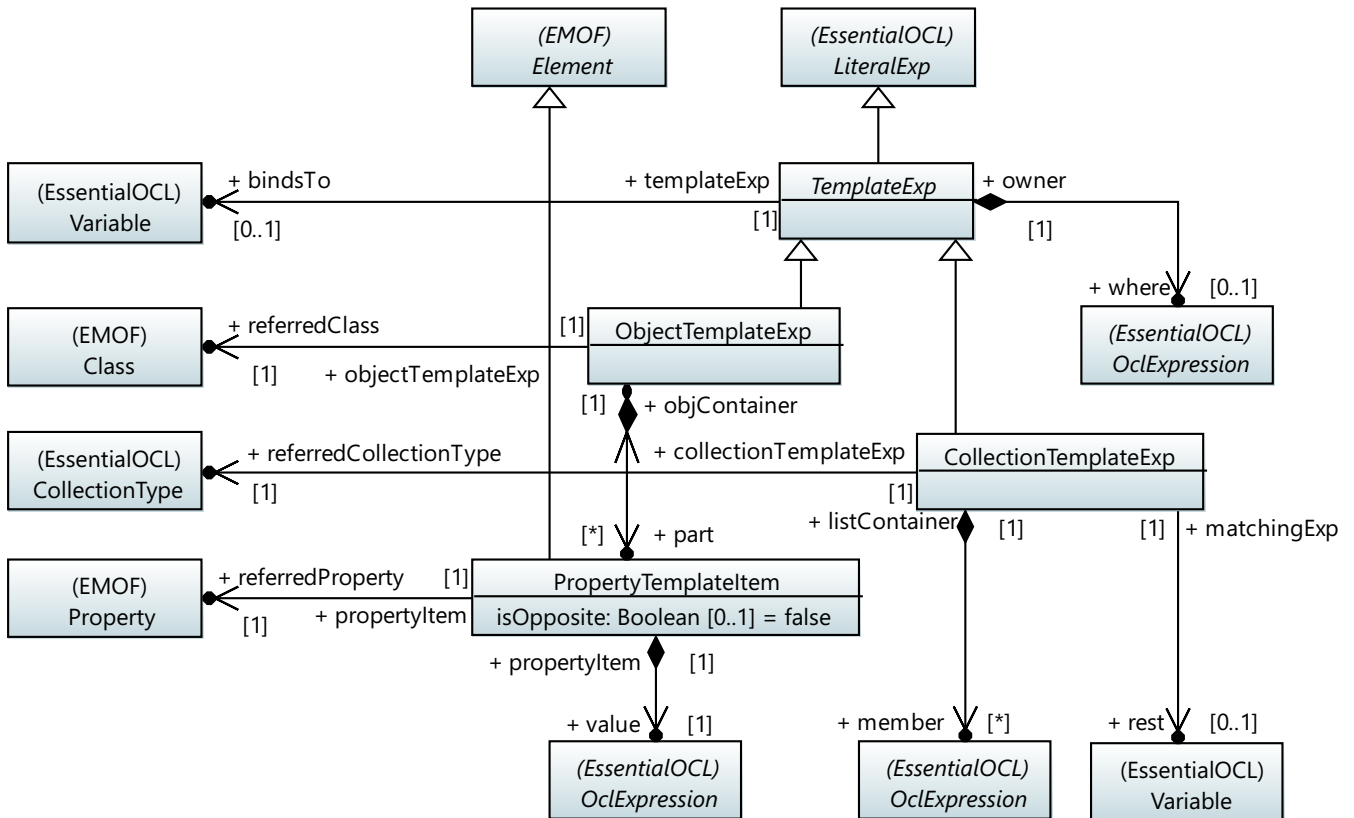


Figure 7.6 - QVT Template Package

Superclasses

LiteralExp

Associations

bindsTo: Variable [0..1]

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.

It specifies a set of member expressions that match individual elements, and a collection-typed variable that matches the rest of the collection. The type of collection that a template matches is given by the referred collection type. If the referred collection type is a sequence, then the matching member elements must be present at the head of the sequence in the order specified by the *member* expressions.

Superclasses

TemplateExp

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.

member : OclExpression [*] {composes}

The expressions that the elements of the collection must have matches for. A special variable `_` may be used to indicate that any arbitrary element may be matched and ignored. The expression may be omitted to restrict a match to an empty collection.

rest : Variable [0..1]

The variable that the rest of the collection (i.e., excluding elements matched by member expressions) must match. A special variable `_` may be used to indicate that any arbitrary collection may be matched and ignored. The variable may be omitted to restrict a match to a collection with no elements other than those matched by the member expressions.

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.

A property template item has a valid match when the value of the referred property matches the value specified by the value expression. The following rules apply when the value is specified by a template expression:

- If the value expression is an object template expression and the referred property is single-valued, then the property value must match the object template expression.
- If the value expression is an object template expression and the referred property is multi-valued, then at least one of the property values must match the object template expression.
- If the value expression is a collection template expression and the referred property is single-valued, then the property value is treated as a singleton collection that must match the collection template expression.
- If the value expression is a collection template expression and the referred property is multi-valued, then the collection of property values must match the collection template expression. The following rules apply:
 - If the property value is a set, then the collection type of the collection template expression must be a set.

- If the property value is an ordered set, then the collection type of the collection template expression can be one of set, sequence, or ordered set.
- If the property value is a sequence, then the collection type of the collection template expression can be one of set, sequence, or ordered set. When the collection type is a set or an ordered set, a valid match requires that each value of the property be unique.

Superclasses

Element

Attributes

`isOpposite`: Boolean

Specifies whether the referred property is owned by the opposite end class. An opposite property template item selects an object of the opposite end class that has this object as the value of the specified property. The default value of this property is false. Please refer to Clause 7.3 for an example that uses an opposite property in template specification.

Associations

`referredProperty` : Property [1]

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

`value` : OclExpression [1] {composes}

The value that the slot may take.

7.11.3 QVTRelation Package

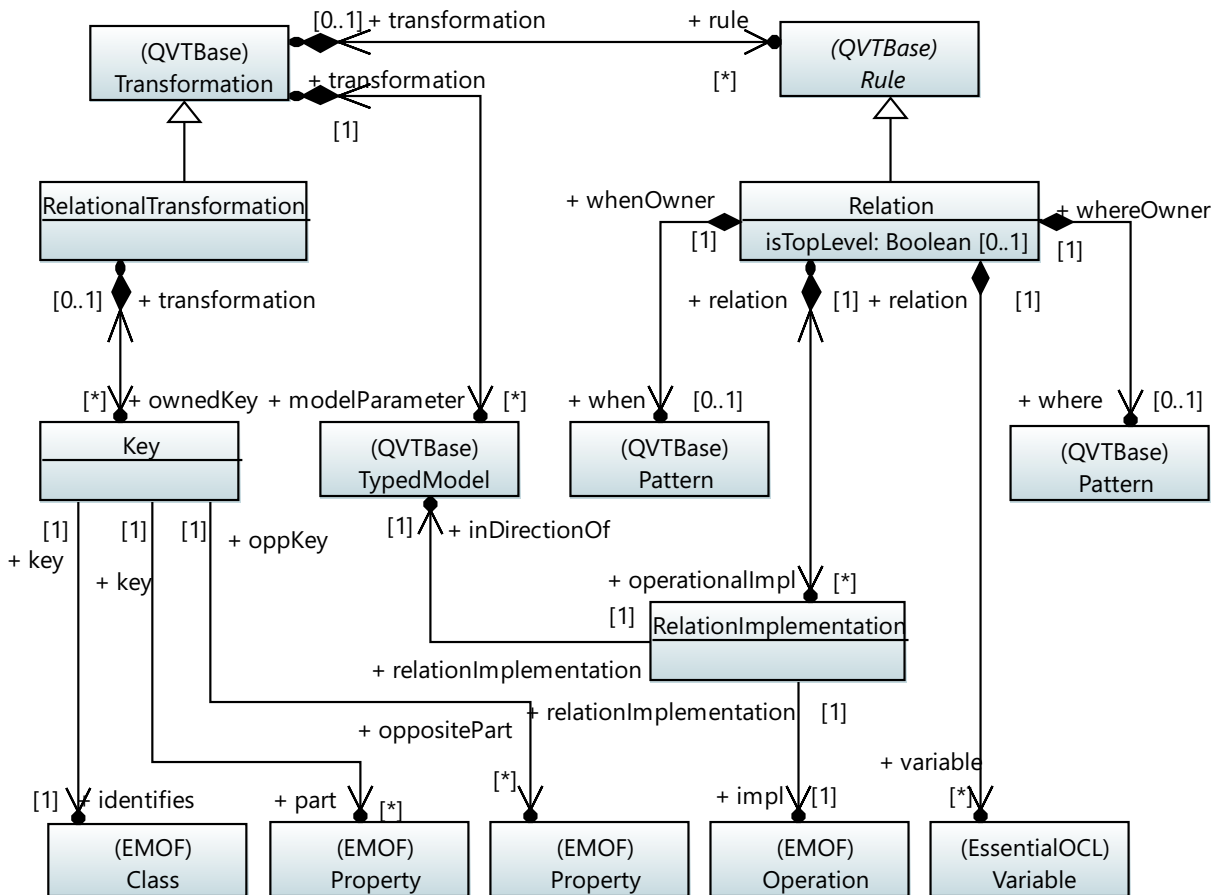


Figure 7.7 - QVT Relation Package - Transformations and Relations

7.11.3.1 RelationalTransformation

A RelationalTransformation is a specialization of a Transformation representing a transformation definition that uses the QVT-Relation formalism.

Superclasses

Transformation (from QVTBase)

Associations

ownedKey: Key [*] {composes}

The keys defined within the transformation.

7.11.3.2 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

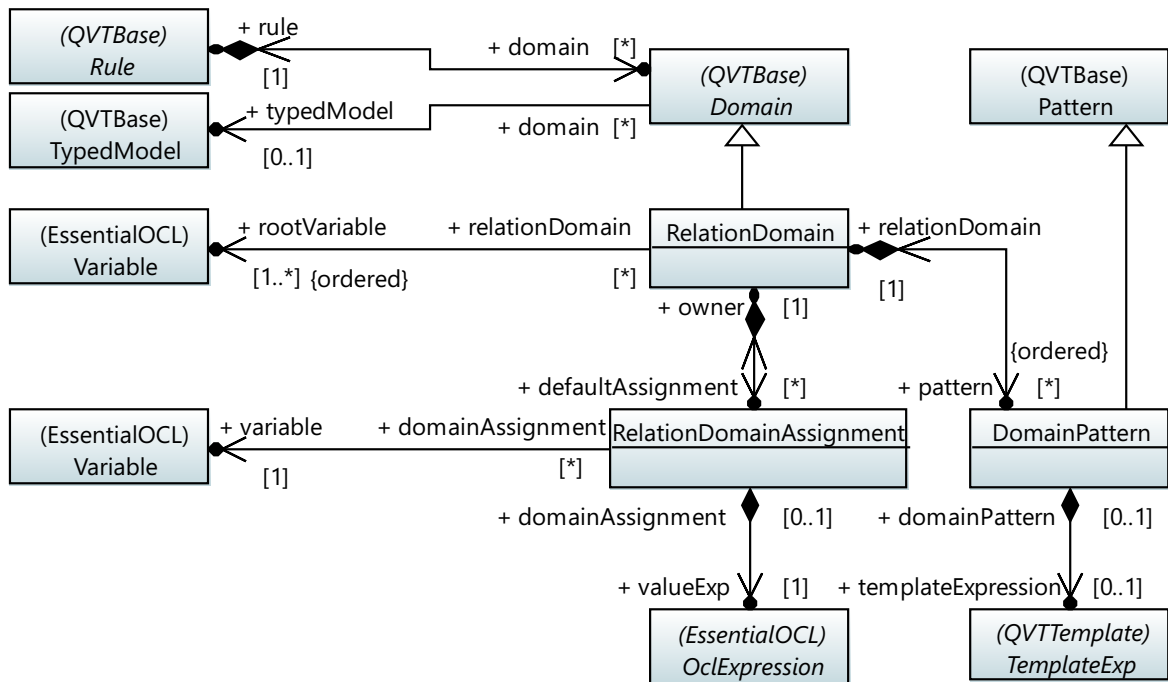


Figure 7.8 - QVT Relation Package - Domains and patterns

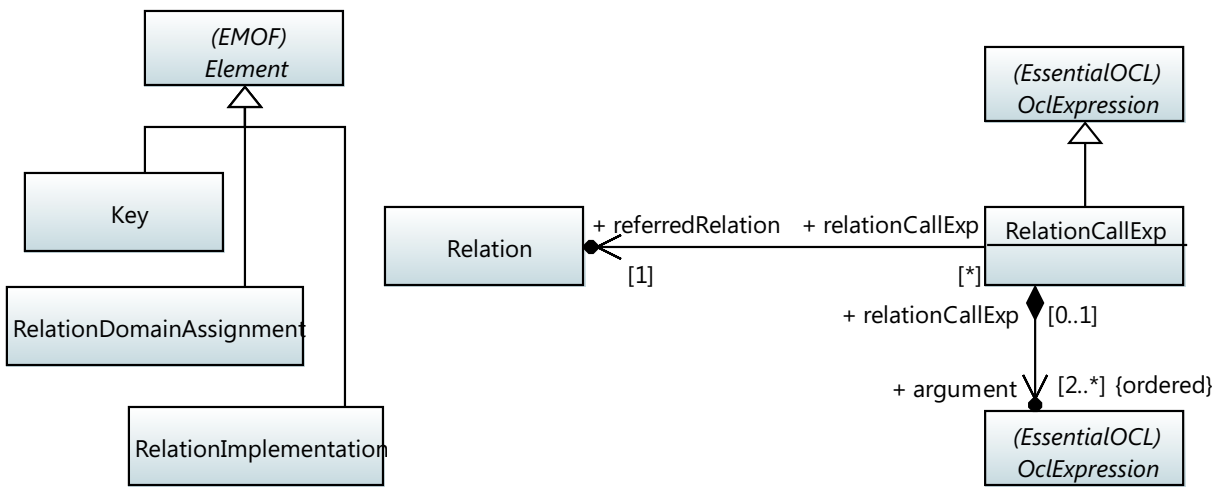


Figure 7.9 - QVT Relation Package - Expressions and miscellaneous inheritance

needs to hold, and a **where** clause that specifies the condition that must be satisfied by the model elements that are being related. A relation may optionally have an associated black-box operational implementation to enforce a domain. Please refer to Clause 7.1 to Clause 7.9 for a detailed description of the semantics, and Clause 10 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. The execution of a transformation requires that all its top-level relations must hold, whereas a non-top-level relation is required to hold only when it is invoked from another relation.

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: RelationDomain [*] {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.

operationalImpl: RelationImplementation [*] {composes}

The set of black-box operational implementations, if any, that are associated with the relation to enforce its domains.

7.11.3.3 RelationDomain

The class RelationDomain specifies the domains of a *relation*. It is a concrete subclass of Domain. A *relation domain* has one or more distinguished typed variables called the *root variables* 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 one or more distinguished root nodes that are bound to the corresponding root variable of the relation domain. Please refer to Clause 7.2, Clause 7.3 and Clause 7.10 for a detailed discussion of the semantics of domains and domain patterns in the relations language.

In bidirectional relations, sometimes it is not possible to automatically compute the values required to enforce a relation in the reverse direction. This can be addressed by allowing a domain to specify default value assignments for its variables. The default assignments are executed only during enforcement and the values they assign must be capable of satisfying the relationship condition. Default assignments do not play a role in checking.

Superclasses

Domain

Associations

rootVariable: Variable [+] {ordered}

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

/typedModel: TypedModel [0..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 [+] {composes, ordered}

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

defaultAssignment: RelationDomainAssignment [0..*] {composes}

The assignments that set default values for the variables of the domain that are required for its enforcement.

7.11.3.4 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 Clause 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.

7.11.3.5 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). Sometimes it may be necessary to specify a key in terms of opposite properties that are not navigable from the class. Please refer to Clause 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 [0..*]

Properties of the class that make up the key.

oppositePart: Property [0..*]

Opposite properties of the class that make up the key.

7.11.3.6 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 - 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.

7.11.3.7 RelationDomainAssignment

A *relation domain assignment* sets the value of a domain variable by evaluating the associated expression in the context of a relation.

Associations

variable: Variable [1]

The variable being assigned.

valueExp: OclExpression [1] {composes}

The expression that provides the value of the variable.

7.11.3.8 RelationCallExp

A *relation call expression* specifies the invocation of a relation. A relation may be invoked from the when or where clause of another relation. The expression contains a list of argument expressions corresponding to the domains of the relation. The number and types of the arguments must match the total number and types of the root variables of the domains. The arguments are ordered firstly by the order of the domains in the called relation, and then by the order of the patterns of each domain.

Superclasses

OclExpression

Associations

argument: OclExpression [2..*] {composes, ordered}

Arguments to the relation call.

referredRelation: Relation [1]

The relation being invoked.

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

7.13.1 Compilation Units

Within an import statement a compilation unit is referenced by means of a simple unqualified alphanumeric identifier - with no special characters and blank characters allowed - or is referenced by means of a qualified one. In the latter case, the qualification is given by a list of namespaces separated by the dot character. These namespaces have no representation in the metamodel. It is up to an implementation to make a correspondence between the namespace hierarchy and the hierarchy of the file system.

7.13.2 Keywords

checkonly, domain, enforce, extends, implementedby, import, key, overrides, primitive, query, relation, top, transformation, when, where.

All these keywords are reserved words, they cannot be used as identifiers.

Reserved words and awkward characters may be used within identifiers using OCL's underscore-prefixed-string-literal escape. e.g.

```
_key' _new\nline'
```

A simple underscore prefix may also be used but is now deprecated.

7.13.3 Comments

The syntax supports two forms of comments; a line comment, and a paragraph comment.

The line comment starts with the string '--' and ends with the next newline. The paragraph comment starts with the string '/*' and ends with the string '*/.' Paragraph comments may be nested.

7.13.4 Relations Textual Syntax Grammar

Production names ending CS are defined by OCL.

```
<identifier> ::= <simpleNameCS>
<topLevel> ::= <import>* <transformation>*
<import> ::= 'import' [<identifier> ':' ] <URI> ( ':' <identifier> )* [ ':'* ] ';'
<URI> ::= #x27 StringChar* #x27 //from OCL StringLiteralExpCS
<transformation> ::= 'transformation' <transformationId>
    '(' <modelDecl> ( ',' <modelDecl> )* ')' [ 'extends' <transformationId> ]
    '{' <keyDecl>* ( <relation> | <query> )* '}'
<transformationId> ::= <pathNameCS>
```

```

<modelDecl> ::= <modelId> ':' ( <metaModelId> |
    '{' <metaModelId> (',' <metaModelId>)* '}' )
<modelId> ::= <identifier>
<metaModelId> ::= <identifier>
<keyDecl> ::= 'key' <classId> '{' <keyProperty> (, <keyProperty>)* '}' ';'
<classId> ::= <pathNameCS>
<keyProperty> ::= <identifier> | 'opposite' '(' <classId> ':' <identifier> ')'
<relation> ::= ['top'] ['abstract'] 'relation' <identifier>
    ['overrides' <identifier>]
    '{'
    <varDeclaration>*
    (<domain> | <primitiveTypeDomain>)+
    [<when>] [<where>]
    '}'
<varDeclaration> ::= <identifier> (, <identifier>)* ':' <typeCS> ['=' <OclExpressionCS>] ';'
<domain> ::= [<checkEnforceQualifier>] 'domain' <modelId> <template> (',' <template>)*
    ['implementedby' <OperationCallExpCS>]
    ['default_values' '{' (<assignmentExp>)+ '}' ]
    ';'
<primitiveTypeDomain> ::= 'primitive' 'domain' <identifier> ':' <typeCS> ';'
<checkEnforceQualifier> ::= 'checkonly' | 'enforce'
<template> ::= (<objectTemplate> | <collectionTemplate>)
    ['{' <OclExpressionCS> '}' ]
<objectTemplate> ::= [<identifier>] ':' <pathNameCS>
    '{' [<propertyTemplateList>] '}'
<propertyTemplateList> ::= <propertyTemplate> (',' <propertyTemplate>)*
<propertyTemplate> ::= <identifier> '=' <OclExpressionCS> |
    'opposite' '(' <classId> ':' <identifier> ')' '=' <OclExpressionCS>
<collectionTemplate> ::= [<identifier>] ':'
    <CollectionTypeIdentifierCS> '(' <typeCS> ')' '{' [<memberSelection>] '}'
<memberSelection> ::= (<identifier> | <template> | '_' )
    (',' (<identifier> | <template> | '_'))*
    ['++' (<identifier> | '_')]
<assignmentExp> ::= <identifier> '=' <OclExpressionCS> ';'
<when> ::= 'when' '{' (<OclExpressionCS> ';')* '}'
<where> ::= 'where' '{' (<OclExpressionCS> ';')* '}'
<query> ::= 'query' <queryId>
    '(' [<paramDeclaration> (',' <paramDeclaration>)*] ')'
    ':' <typeCS>
    (';' | '{' <OclExpressionCS> '}' )
<queryId> ::= <PathNameCS>
<paramDeclaration> ::= <identifier> ':' <typeCS>

```

7.13.5 Expressions Syntax (extensions to OCL)

```

<OclExpressionCS> ::= <PropertyCallExpCS>
    | <VariableExpCS>
    | <LiteralExpCS>
    | <LetExpCS>
    | <IfExpCS>
    | '(' <OclExpressionCS> ')'
    | <template>

```

7.13.6 Graphical Syntax

7.13.6.1 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 this specification contains a diagrammatic syntax to complement the textual syntax of Clause 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. A visual notation is suggested 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. Using object diagrams with some extensions to specify patterns within a relation specification is suggested. The notation is introduced through some examples followed by detailed syntax and semantics. Figure 7.10 specifies a relation, UML2Rel from UML classes and attributes to relational tables and columns. A new symbol is introduced $\langle\langle c \rangle\rangle$ 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 checkonly.

Figure 7.10 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}}
}
```

UML2Rel

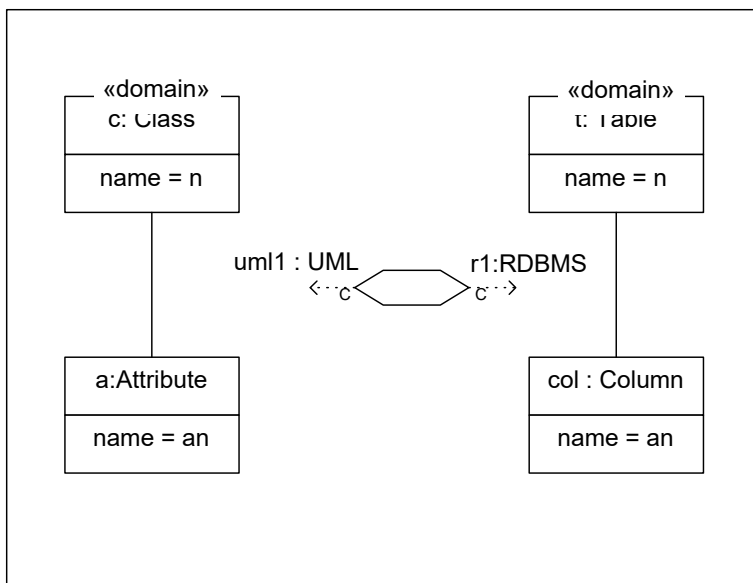


Figure 7.10 - UML Class to Relational Table Relation

The *where* clause of a relation can be shown using a where box as shown in Figure 7.11, which specifies a relation, *PackageToSchema* that extends the above relation to specify that *UML2Rel* is to be applied on every class within a package.

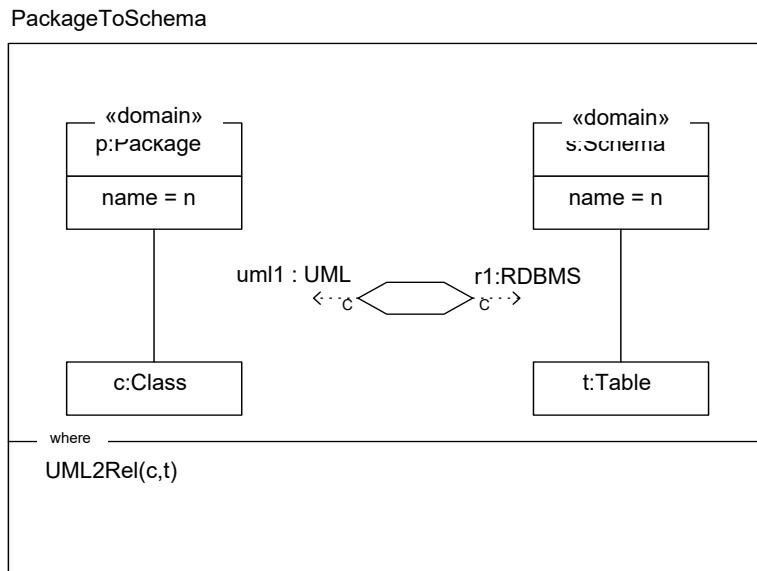


Figure 7.11 - Example showing the usage of the where clause

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 7.12.

PackageToSchema

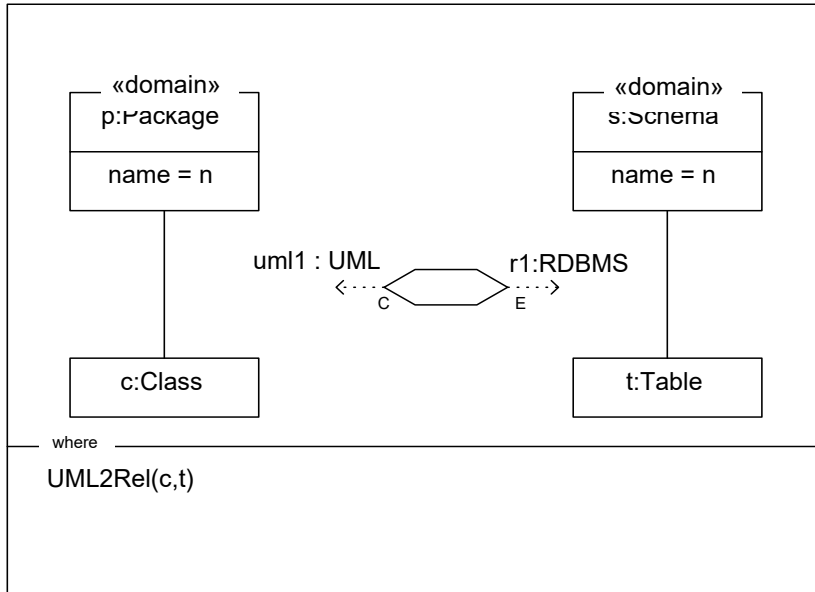


Figure 7.12 - RDBMS domain made enforceable

Constraints may be attached to objects or a pattern. This is done as in UML by attaching a note. An example is shown in Figure 7.13. In the figure, one constraint is attached to the `col` object and another one to the UML pattern.

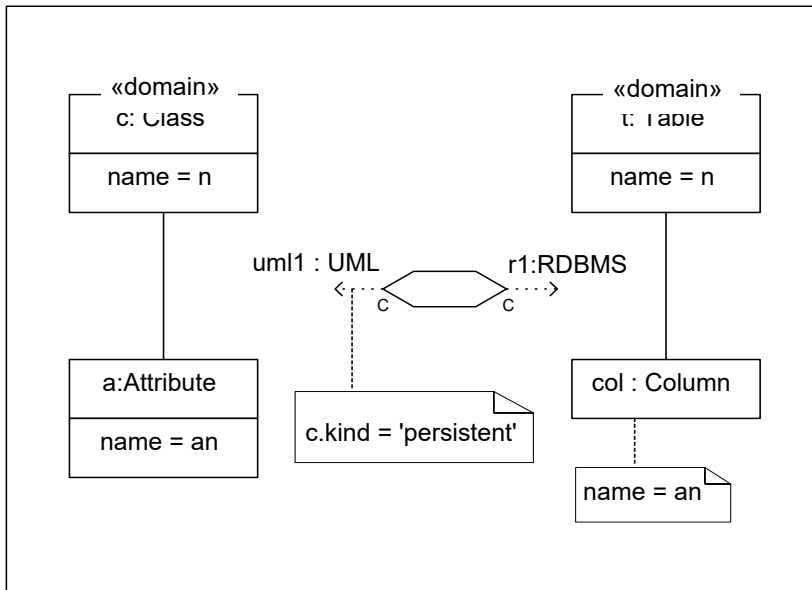


Figure 7.13 - UML2REL with constraints

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 7.14 shows an example where *Table* has a field *totcols* that is set to the number of attributes in the *Class*.

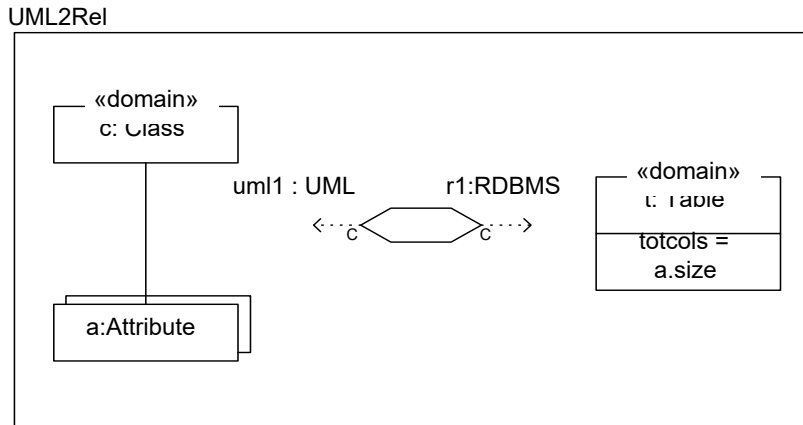


Figure 7.14 - Example using a Set

The notation also includes support for specifying the non-existence of objects and overriding of relations. Figure 7.15 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*.

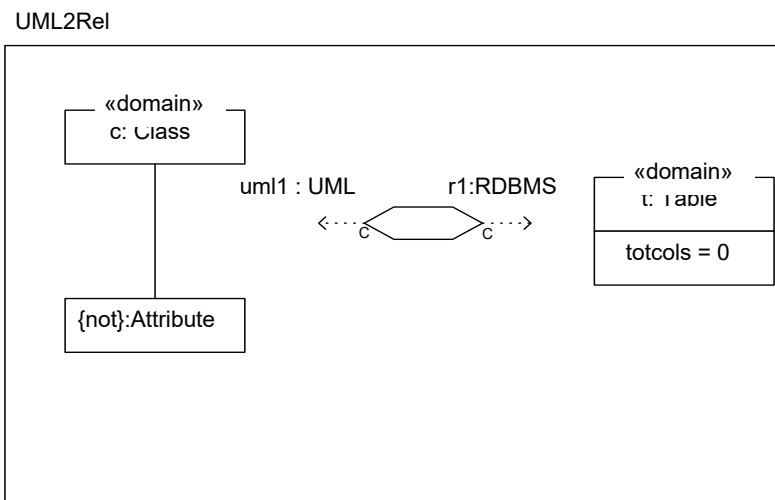


Figure 7.15 - Example using {not}

The textual specification corresponding to Figure 7.15 is as follows.

```
relation UML2Rel {
  checkonly domain um1 c:Class {attribute = Set(Attribute){}}{attribute->size()=0}
  checkonly domain r1 t:Table {totcols = 0 }
}
```

7.13.6.2 Graphical Notation Elements

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

Table 7.1 - Diagrammatic Notations

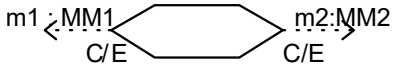
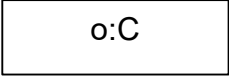
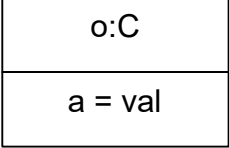
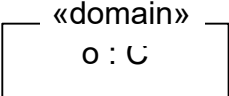
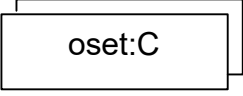
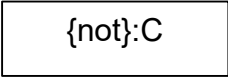
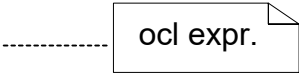
Notation	Description
	<p>A relation between models <i>m1</i> having MM1 as meta-model and <i>m2</i> having MM2 as meta-model. The label C/E indicates whether the domain in that direction is checkable or enforceable.</p>
	<p>An object template having type <i>C</i> and referred to by the free variable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

Table 7.1 - Diagrammatic Notations

	<p><i>oset</i> is an object template that matches a set of objects of type <i>C</i>.</p>
	<p>A <i>not</i> template that matches only when there is no object of type <i>C</i> satisfying the constraints associated with it.</p>
	<p>A constraint that may be attached to either a domain or an object template.</p>

7.13.6.3 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 may be hidden and shown optionally or shown in a separate frame.
- model and meta-model name: These may be shown optionally and when shown they may be 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.

7.13.7 Concrete textual syntax to abstract syntax mapping

The mapping is specified using attribute grammar similar to how OCL' concrete to abstract syntax mapping is specified.

topLevelCS

```
topLevelCS ::= importListCS? transformationListCS?
```

Abstract syntax mapping

```
topLevelCS.ast : Set(RelationalTransformation)
```

Synthesized attributes

```
topLevelCS.ast = TransformationListCS.ast
```

Inherited attributes

```
topLevelCS.ast = TransformationListCS.ast
```

Inherited attributes

```
transformationListCS.env =  
  if (importListCS.ast.notEmpty())  
  then  
    Environment.EMPTY_ENV.addElement('imported transformation', importListCS.ast)  
  else  
    Environment.EMPTY_ENV  
  endif
```

importListCS

```
importListCS[1] ::= 'import' unitCS ';' importListCS[2]?
```

Abstract syntax mapping

```
importListCS[1].ast : Set(RelationalTransformation)
```

Synthesized attributes

```
importListCS[1].ast = unitCS.ast->union(importListCS[2].ast)
```

unitCS

```
unitCS ::= identifierCS ('.' identifierCS)*
```

Abstract syntax mapping

```
unitCS.ast : Set(RelationalTransformation)
```

Synthesized attributes

```
<left unspecified>  
The dot separated identifiers identify a compilation unit that is expected  
to contain transformation specifications. As explained in Clause 7.13.1 of  
the specification, how the dot separated identifiers are mapped to, say, a file  
system hierarchy is implementation specific.
```

transformationListCS

```
transformationListCS[1] ::= transformationCS transformationListCS[2]?
```

Abstract syntax mapping

```
transformationListCS[1].ast : Set(RelationalTransformation)
```

Synthesized attributes

```
transformationListCS[1].ast =  
  Set{transformationCS.ast}->union(transformationListCS[2].ast)
```

Inherited attributes

```
transformationCS[1].env = transformationListCS[1].env  
transformationListCS[2].env = transformationListCS[1].env
```

transformationCS

```
transformationCS ::= 'transformation' identifierCS[1] '(' modelDeclListCS ')'  
  ('extends' identifierCS[2])?  
  '{' keyDeclListCS? relQueryListCS? '}'
```

Abstract syntax mapping

```
transformationCS.ast : RelationalTransformation
```

Synthesized attributes

```
transformationCS.ast.name = identifierCS[1].ast  
transformationCS.ast.modelParameter = modelDeclListCS.ast  
if (not identifierCS[2].ast.oclIsUndefined())  
then transformationCS.ast.extends =  
  let importedTransformationSet:Set(RelationalTransformation) =  
    transformationCS.env.lookup('imported transformations').referred  
    Element.oclAsType(SetType)  
  in  
    importedTransformationSet->any(t | t.name = identifierCS[2].ast)  
  endif  
transformationCS.ast.rule = relQueryListCS->select(r | r.oclIsTypeOf (Relation))  
transformationCS.ast.ownedOperation = relQueryListCS->select(r | r.  
  oclIsTypeOf(Function))  
transformationCS.ast.ownedKey = keyDeclListCS.ast
```

Inherited attributes

```
identifierCS[1].env = transformationCS.env  
modelDeclListCS.env = transformationCS.env  
identifierCS[2].env = transformationCS.env  
let  
  env:Environment =  
    transformationCS.env.addElement('context transformation',  
    transformationCS.ast)  
  in keyDeclListCS.env =  
    env and relQueryListCS.env = env  
modelDeclListCS  
modelDeclListCS[1] ::= modelDeclCS (',' modelDeclListCS[2])?
```

Abstract syntax mapping

```
modelDeclListCS[1].ast : Sequence(TypedModel)
```

Synthesized attributes

```
modelDeclListCS[1].ast = Sequence{modelDeclCS.ast}->union(modelDeclListCS[2].ast)
```

Inherited attributes

```
modelDeclCS.env = modelDeclListCS[1].env modelDeclListCS[2].env =  
modelDeclListCS[1].env
```

keyDeclListCS

```
keyDeclListCS[1] ::= keyDeclCS keyDeclListCS[2]?
```

Abstract syntax mapping

```
keyDeclListCS[1].ast : Set(Key)
```

Synthesized attributes

```
keyDeclListCS[1].ast = Set{keyDeclCS.ast}->union(keyDeclListCS[2].ast)
```

Inherited attributes

```
keyDeclCS.env = keyDeclListCS[1].env keyDeclListCS[2].env =  
keyDeclListCS[1].env
```

relQueryListCS

```
relQueryListCS[1] ::= relQueryCS relQueryListCS[2]?
```

Abstract syntax mapping

```
relQueryListCS[1].ast : Set(ModelElement)
```

Synthesized attributes

```
relQueryListCS[1].ast = Set{relQueryCS.ast}->union(relQueryListCS[2].ast)
```

Inherited attributes

```
relQueryCS.env = relQueryListCS[1].env relQueryListCS[2].env =  
relQueryListCS[1].env
```

relQueryCS

```
[A] relQueryCS ::= relationCS  
[B] relQueryCS ::= queryCS
```

Abstract syntax mapping

```
relQueryCS.ast : ModelElement
```

Synthesized attributes

```
[A] relQueryCS.ast = relationCS.ast  
[B] relQueryCS.ast = queryCS.ast
```

Inherited attributes

```
[A] relationCS.env = relQueryCS.env  
[B] queryCS.env = relQueryCS.env
```

modelDeclCS

```
modelDeclCS ::= modelIdCS ':' metaModelIdCS
```

Abstract syntax mapping

```
modelDeclCS.ast : TypedModel
```

Synthesized attributes

```
modelDeclCS.ast.name = modelIdCS.ast  
modelDeclCS.ast.usedPackage = metaModelIdCS.ast
```

Inherited attributes

```
modelIdCS.env = modelDeclCS.env  
metaModelIdCS.env = modelDeclCS.env
```

modelIdCS

```
modelIdCS ::= identifierCS
```

Abstract syntax mapping

```
modelIdCS.ast : String
```

Synthesized attributes

```
modelIdCS.ast = identifierCS.ast
```

metaModelIdCS

```
metaModelIdCS ::= identifierCS
```

Abstract syntax mapping

```
metaModelIdCS.ast : Package
```

Synthesized attributes

```
metaModelIdCS.ast = loadMetaModelPackage(identifierCS.ast)  
// How the package is located and loaded is tool specific and so left unspecified.
```

keyDeclCS

```
keyDeclCS ::= 'key' classIdCS '{' keyPropertyListCS '}' ';' ;
```

Abstract syntax mapping

```
keyDeclCS.ast : Key
```

Synthesized attributes

```
keyDeclCS.ast.identifies = classIdCS.ast  
keyDeclCS.ast.part->union(keyDeclCS.ast.oppositePart) = keyPropertyListCS.ast
```

Inherited attributes

```
classIdCS.env = keyDeclCS.env  
keyPropertyListCS.env = keyDeclCS.env.addElement('context key', keyDeclCS.ast)
```

keyPropertyListCS

```
keyPropertyListCS[1] ::= keyPropertyCS (',' keyPropertyListCS[2])?
```

Abstract syntax mapping

```
keyPropertyListCS[1].ast : Set(Property)
```

Synthesized attributes

```
keyPropertyListCS[1].ast : Set{keyPropertyCS.ast}->union(keyPropertyListCS[2].ast)
```

Inherited attributes

```
keyPropertyCS.env = keyPropertyListCS[1].env  
keyPropertyListCS[2].env = keyPropertyListCS[1].env
```

classIdCS

```
classIdCS ::= pathNameCS
```

Abstract syntax mapping

```
classIdCS.ast : Class
```

Synthesized attributes

```
classIdCS.ast =  
  let  
    trans:RelationalTransformation =  
      classIdCS.env.lookup('context transformation').  
        referredElement.oclAsType(RelationalTransformation)  
  in  
    trans.lookupClassName(pathNameCS.ast)
```

keyPropertyCS

```
[A] keyPropertyCS ::= identifierCS  
[B] keyPropertyCS ::= 'opposite' '(' classIdCS '.' identifierCS ')'
```

Abstract syntax mapping:

```
keyPropertyCS.ast : Property
```

Synthesized attributes:

```
[A] keyPropertyCS.ast =
```

```

let
    cls:Class = keyPropertyCS.env.lookup('context key').
                referredElement.oclAsType(Key).identifies
in
    cls.lookupProperty(identifierCS.ast)
keyPropertyCS.env.lookup('context key').referredElement.oclAsType(Key).
part->includes(keyPropertyCS.ast)
[B] keyPropertyCS.ast = classIdCS.ast.lookupProperty(identifierCS.ast)
keyPropertyCS.env.lookup('context key').referredElement.oclAsType(Key).
oppositePart->includes(keyPropertyCS.ast)

```

Inherited attributes:

```
[B] classIdCS.env = keyPropertyCS.env
```

relationCS

```

relationCS ::= topQualifierCS? 'relation' identifierCS[1]
              ('overrides' identifierCS[2])?
              '{'
              varDeclListCS?
              domainListCS
              whenCS?
              whereCS?
              '}'

```

Abstract syntax mapping

```
relationCS.ast : Relation
```

Synthesized attributes

```

relationCS.ast.isTopLevel = if (not topQualifierCS.ast.oclIsUndefined() and
                                topQualifierCS.ast = 'top')
                            then
                                true
                            else
                                false
                            endif

relationCS.ast.name = identifierCS[1].ast

if (not identifierCS[2].ast.oclIsUndefined())
then
    relationCS.ast.overrides =
        let
            currTrans:RelationTransformation =
                relationCS.env.lookup('context transformation').
                    referredElement.oclAsType(RelationalTransformation)
        in
            currTrans.extends.rule->any(r | r.name = identifierCS[2].ast)
        endif
    relationCS.ast.variable =
        varDeclListCS.ast->union(
            domainListCS->iterate(d:RelationDomain; domainVars:Set(Variable)={} |
                domainVars->including(d.rootVariable))
        )
    relationCS.ast.domain = domainListCS.ast
    relationCS.ast.when = whenCS.ast

```

```
relationCS.ast.where = whereCS.ast
```

Inherited attributes

```
let
  env:Environment = relationCS.env.addElement('context relation', relationCS.ast)
in
  varDeclListCS.env = env
  domainListCS.env = env
  whenCS.env = env
  whereCS.env = env
```

topQualifierCS

```
topQualifierCS ::= 'top'
```

Abstract syntax mapping

```
topQualifierCS.ast : String
```

Synthesized attributes

```
topQualifierCS.ast = 'top'
```

varDeclListCS

```
varDeclListCS[1] ::= varDeclarationCS varDeclListCS[2]?
```

Abstract syntax mapping

```
varDeclListCS[1].ast : Set(Variable)
```

Synthesized attributes

```
varDeclListCS[1].ast ::= varDeclarationCS.ast->union(varDeclListCS[2].ast)
```

Inherited attributes

```
varDeclarationCS.env = varDeclListCS[1].env
varDeclListCS[2].env = varDeclListCS[1].env
```

varDeclarationCS

```
varDeclarationCS ::= varListCS ':' TypeCS ';' 
```

Abstract syntax mapping

```
varDeclarationCS.ast : Set(Variable)
```

Synthesized attributes

```
varDeclarationCs.ast->size() = varListCS.ast->size()
varListCS.ast->forall(vn |
  varDeclarationCS.ast->exists(v | v.name = vn and v.type = TypeCS.ast)
)
```


Inherited attributes

```
TypeCS.env = varDeclarationCS.env
```

varListCS

```
varListCS[1] ::= identifierCS (',' varListCS[2])?
```

Abstract syntax mapping

```
varListCS[1].ast : Set(String)
```

Synthesized attributes

```
varListCS[1].ast : Set{identifierCS.ast}->union(varListCS[2].ast)
```

domainListCS

```
domainListCS[1] ::= domainCS domainListCS[2]?
```

Abstract syntax mapping

```
domainListCS[1].ast : Sequence(RelationDomain)
```

Synthesized attributes

```
domainListCS[1].ast = Sequence{domainCS.ast}->union(domainListCS[2].ast)
```

Inherited attributes

```
domainCS.env = domainListCS[1].env  
domainListCS[2].env = domainListCS[1].env
```

domainCS

```
domainCS[A] ::= modelDomainCS  
domainCS[B] ::= primitiveDomainCS
```

Abstract syntax mapping

```
domainCS.ast : RelationDomain
```

Synthesized attributes

```
[A] domainCS.ast = modelDomainCS.ast  
[B] domainCS.ast = primitiveDomainCS.ast
```

Inherited attributes

```
[A] modelDomainCS.env = domainCS.env  
[B] primitiveDomainCS.env = domainCS.env
```

modelDomainCS

```
modelDomainCS ::= checkEnforceQualifierCS? 'domain' modelIdCS templateCS
  'implementedby' OperationCallExpCS)?
  ('default_values' '{' assignmentExpListCS '}')? ';' ;
```

Abstract syntax mapping

```
modelDomainCS.ast : RelationDomain
```

Synthesized attributes

```
if (checkEnforceQualifierCS.ast.oclIsUndefined())
then
  modelDomainCS.ast.isCheckable = false
  and
  modelDomainCS.ast.isEnforceable = false
else
  if (checkEnforceQualifierCS.ast = 'checkonly')
  then
    modelDomainCS.ast.isCheckable = true
    and
    modelDomainCS.ast.isEnforceable = false
  else
    modelDomainCS.ast.isCheckable = true
    and
    modelDomainCS.ast.isEnforceable = true
  endif
endif

modelDomainCS.ast.typedModel =
  modelDomainCS.env.lookup('context transformation').modelParameter->
    any(t | t.name = modelIdCS.ast)

modelDomainCS.ast.pattern.templateExpression = templateCS.ast
modelDomainCS.ast.rootVariable = templateCS.ast.bindsTo

if (not OperationCallExpCS.ast.oclIsUndefined())
then
  let
    rel:Relation = modelDomainCS.env.lookup('context relation').
      referredElement.oclAsType(Relation)
  in
    rel.operationalImpl.impl = OperationCallExpCS.ast.referredOperation
    and
    rel.operationalImpl.inDirectionOf = modelDomainCS.ast.typedModel
endif

if (assignmentExpListCS.ast.notEmpty())
then
  modelDomainCS.ast.defaultAssignment = assignmentExpListCS.ast
endif
```

Inherited attributes

```
let
  env:Environment = modelDomainCS.env.addElement('context domain', modelDmainCS.ast)
in
  templateCS.env = env
  OperationCallExpCS.env = env
  assignmentExpListCS.env = env
```

primitiveTypeDomainCS

```
primitiveTypeDomainCS ::= 'primitive' 'domain' identifierCS ':' TypeCS ';' ;
```

Abstract syntax mapping

```
primitiveTypeDomainCS.ast : RelationDomain
```

Synthesized attributes

```
primitiveTypeDomainCS.ast.rootVariable.name = identifierCS.ast  
primitiveTypeDomainCS.ast.rootVariable.type = TypeCS.ast
```

Inherited attributes

```
TypeCS.env = primitiveTypeDomainCS.env
```

checkEnforceQualifierCS

```
[A] checkEnforceQualifierCS ::= 'checkonly'
```

```
[B] checkEnforceQualifierCS ::= 'enforce'
```

Abstract syntax mapping

```
checkEnforceQualifierCS.ast : String
```

Synthesized attributes

```
[A] checkEnforceQualifierCS.ast = 'checkonly'
```

```
[B] checkEnforceQualifierCS.ast = 'enforce'
```

templateCS

```
[A] templateCS ::= objectTemplateCS ('{' OclExpressionCS '})?
```

```
[B] templateCS ::= collectionTemplateCS ('{' OclExpressionCS '})?
```

Abstract syntax mapping

```
templateCS.ast : TemplateExp
```

Synthesized attributes

```
[A] templateCS.ast = objectTemplateCS.ast  
    if (not OclExpressionCS.ast.oclIsUndefined())  
    then  
        templateCS.ast.where = OclExpressionCS.ast  
    endif
```

```
[B] templateCS.ast = collectionTemplateCS.ast  
    if (not OclExpressionCS.ast.oclIsUndefined())  
    then  
        templateCS.ast.where = OclExpressionCS.ast  
    endif
```

Inherited attributes

```
[A] objectTemplateCS.env = templateCS.env  
    OclExpressionCS.env = templateCS.env
```

```
[B] collectionTemplateCS.env = templateCS.env  
    OclExpressionCS.env = templateCS.env
```

objectTemplateCS

```
objectTemplateCS ::= identifierCS? ':' pathNameCS '{' propertyTemplateListCS? '}'
```

Abstract syntax mapping

```
objectTemplateCS.ast : ObjectTemplateExp
```

Synthesized attributes

```
let
  cls:Class = objectTemplateCS.env.lookup('context domain').
              referredElement.typedModel.usedPackage.getEnvironmentWithoutParents().
              lookupPathName(pathNameCS.ast).referredElement.oclAsType(Class)
in
  if (not identifierCS.ast.oclIsUndefined)
  then
    objectTemplateCS.ast.bindsTo.name = identifierCS.ast
    and
    objectTemplateCS.ast.bindsTo.type = cls
    and
    objectTemplateCS.env.lookup('context relation').
      referredElement.oclAsType(Relation).variable->
        exists(v | v = objectTemplateCS.ast.bindsTo)
  endif
  and
  objectTemplateCS.ast.referredClass = cls
objectTemplateCS.ast.part = propertyTemplateListCS.ast
```

Inherited attributes

```
propertyTemplateListCS.env =
  objectTemplateCS.env.addElement('context template', objectTemplateCS.ast)
```

propertyTemplateListCS

```
propertyTemplateListCS[1] ::= propertyTemplateCS (',' propertyTemplateListCS[2])?
```

Abstract syntax mapping

```
propertyTemplateListCS[1].ast : Set(PropertyTemplateItem)
```

Synthesized attributes

```
propertyTemplateListCS[1].ast = Set{propertyTemplateCS.ast}->
  union(propertyTemplateListCS[2].ast)
```

Inherited attributes

```
propertyTemplateCS.env = propertyTemplateListCS[1].env
propertyTemplateListCS[2].env = propertyTemplateListCS[1].env
```

propertyTemplateCS

```
[A] propertyTemplateCS ::= identifierCS '=' ExtOclExpressionCS
```

```
[B] propertyTemplateCS ::= 'opposite' '(' classIdCS '.' identifierCS ')' '='
    ExtOclExpressionCS
```

Abstract syntax mapping:

```
propertyTemplateCS.ast : PropertyTemplateItem
```

Synthesized attributes:

```
[A] propertyTemplateCS.ast.referredProperty =  
    propertyTemplateCS.env.lookup('context template').  
        referredElement.oclAsType(ObjectTemplateExp).  
        referredClass.lookupProperty(identifierCS.ast)  
propertyTemplateCS.ast.isOpposite = false  
propertyTemplateCS.ast.value = ExtOclExpressionCS.ast  
  
[A] propertyTemplateCS.ast.referredProperty =  
    classIdCS.ast.lookupProperty(identifierCS.ast)  
propertyTemplateCS.ast.isOpposite = true  
propertyTemplateCS.ast.value = ExtOclExpressionCS.ast
```

Inherited attributes:

```
[A] ExtOclExpressionCS.env = propertyTemplateCS.env  
[B] ExtOclExpressionCS.env = propertyTemplateCS.env  
    classIdCS.env = propertyTemplateCS.env
```

collectionTemplateCS

```
collectionTemplateCS ::= identifierCS? ':' CollectionTypeIdentifierCS '(' TypeCS ')' '  
    '{' (memberListCS '++' restCS)? '}'
```

Abstract syntax mapping

```
collectionTemplateCS.ast : CollectionTemplateExp
```

Synthesized attributes

```
if (CollectionTypeIdentifierCS.ast = CollectionKind::Set)  
then  
else  
    collectionTemplateCS.ast.referredCollectionType.oclIsTypeOf(SetType)  
else  
    if (CollectionTypeIdentifierCS.ast = CollectionKind::Sequence)  
    then  
        collectionTemplateCS.ast.referredCollectionType.oclIsTypeOf(SequenceType)  
    else  
        if (CollectionTypeIdentifierCS.ast = CollectionKind::Bag)  
        then  
            collectionTemplateCS.ast.referredCollectionType.oclIsTypeOf(BagType)  
        else  
            if (CollectionTypeIdentifierCS.ast = CollectionKind::OrderedSet)  
            then  
                collectionTemplateCS.ast.referredCollectionType.  
                    oclIsTypeOf(OrderedSetType)  
            endif  
        endif  
    endif  
endif  
collectionTemplateCS.ast.referredCollectionType.elementType = TypeCS.ast  
if (not identifierCS.ast.oclIsUndefined)  
then  
    collectionTemplateCS.ast.bindsTo.name = identifierCS.ast  
    and  
    collectionTemplateCS.ast.bindsTo.type =
```

```

        collectionTemplateCS.ast.referredCollectionType
    and
    collectionTemplateCS.env.lookup('context relation').
        referredElement.oclAsType(Relation).variable->
        exists(v | v = collectionTemplateCS.ast.bindsTo)
endif
if (memberListCS->notEmpty())
then
    collectionTemplateCS.ast.member = memberListCS.ast
endif
if (not restCS.oclIsUndefined())
then
    collectionTemplateCS.ast.rest = restCS.ast
endif

```

Inherited attributes

```

TypeCS.env = collectionTemplateCS.env
memberListCS.env = collectionTemplateCS.env
restCS.env = collectionTemplateCS.env

```

memberListCS

```
memberListCS[1] ::= memberCS (',' memberListCS[2])?
```

Abstract syntax mapping

```
memberListCS[1].ast : Sequence(OclExpression)
```

Synthesized attributes

```
memberListCS[1].ast = Sequence{memberCS.ast}->union(memberListCS[2].ast)
```

Inherited attributes

```
memberCS.env = memberListCS[1].env
memberListCS[2].env = memberListCS[1].env

```

restCS

```
[A] restCS ::= identifierCS
[B] restCS ::= '_'
```

Abstract syntax mapping

```
restCS.ast : Variable
```

Synthesized attributes

```
[A] restCS.ast = restCS.env.lookup('context relation').
    referredElement.oclAsType(Relation).variable->any(v | v.name = identifierCS.ast)
[B] restCS.ast.name = '_'
```

memberCS

```
[A] memberCS ::= identifierCS
[B] memberCS ::= templateCS
```

[C] memberCS ::= '_'

Abstract syntax mapping

memberCS.ast : OclExpression

Synthesized attributes

[A] memberCS.ast = memberCS.env.lookup('context relation').
referredElement.oclAsType(Relation).variable->any(v | v.name = identifierCS.ast)

[B] memberCS.ast = templateCS.ast

[C] memberCS.ast.oclIsTypeOf(Variable)
memberCS.ast.oclAsType(Variable).name = '_'

Inherited attributes

[B] templateCS.env = memberCS.env

assignmentExpListCS

assignmentExpListCS[1] ::= assignmentExpCS assignmentExpListCS[2]?

Abstract syntax mapping

assignmentExpListCS[1].ast : Set(RelationDomainAssignment)

Synthesized attributes

assignmentExpListCS[1].ast = Set{assignmentExpCS.ast}->
union(assignmentExpListCS[2].ast)

Inherited attributes

assignmentExpListCS[2].env = assignmentExpListCS[1].env
assignmentExpCS.env = assignmentExpListCS[1].env

assignmentExpCS

assignmentExpCS ::= identifierCS '=' OclExpressionCS ';'

Abstract syntax mapping

assignmentExpCS.ast : RelationDomainAssignment

Synthesized attributes

assignmentExpCS.ast.variable = assignmentExpCS.env.lookup('context relation').
referredElement.oclAsType(Relation).variable->
any(v | v.name = identifierCS.ast)
assignmentExpCS.ast.valueExp = OclExpressionCS.ast

Inherited attributes

OclExpressionCS.env = assignmentExpCS.env

whenCS

```
whenCS ::= 'when' '{ predicateListCS }'
```

Abstract syntax mapping

```
whenCS.ast : Pattern
```

Synthesized attributes

```
whenCS.ast.predicate = predicateListCS.ast
```

Inherited attributes

```
predicateListCS.env = whenCS.env
```

whereCS

```
whereCS ::= 'where' '{ predicateListCS }'
```

Abstract syntax mapping

```
whereCS.ast : Pattern
```

Synthesized attributes

```
whereCS.ast.predicate = predicateListCS.ast
```

Inherited attributes

```
predicateListCS.env = whereCS.env
```

predicateListCS

```
predicateListCS[1] ::= predicateCS ';' predicateListCS[2]?
```

Abstract syntax mapping

```
predicateListCS[1].ast : Set(Predicate)
```

Synthesized attributes

```
predicateListCS[1].ast = Set{predicateCS.ast}->union(predicateListCS[2].ast)
```

Inherited attributes

```
predicateCS.env = predicateListCS[1].env  
predicateListCS[2].env = predicateListCS[1].env
```

predicateCS

```
predicateCS ::= ExtOclExpressionCS
```

Abstract syntax mapping

```
predicateCS.ast : Predicate
```


Synthesized attributes

```
predicateCS.ast.conditionExpression = ExtOclExpressionCS.ast
```

Inherited attributes

```
ExtOclExpressionCS.env = predicateCS.env
```

queryCS

```
queryCS ::= 'query' identifierCS '(' paramDeclListCS? ')' ':' TypeCS  
        queryBodyCS
```

Abstract syntax mapping

```
queryCS.ast : Function
```

Synthesized attributes

```
queryCS.ast.name = identifierCS.ast  
queryCS.ast.ownedParameter = paramDeclListCS.ast  
queryCS.ast.type = TypeCS.ast  
queryCS.ast.queryExpression = queryBodyCS.ast
```

Inherited attributes

```
paramDeclListCS.env = queryCS.env  
queryBodyCS.env = queryCS.env
```

paramDeclListCS

```
paramDeclListCS[1] ::= paramDeclarationCS (',' paramDeclListCS[2])?
```

Abstract syntax mapping

```
paramDeclListCS[1].ast : Sequence(Parameter)
```

Synthesized attributes

```
paramDeclListCS[1].ast = Sequence{paramDeclarationCS.ast}->  
    union(paramDeclListCS[2].ast)
```

Inherited attributes

```
paramDeclarationCS.env = paramDeclListCS[1].env  
paramDeclListCS[2].env = paramDeclListCS[1].env
```

paramDeclarationCS

```
paramDeclarationCS ::= identifierCS ':' TypeCS
```

Abstract syntax mapping

```
paramDeclarationCS.ast : Parameter
```

Synthesized attributes

```
paramDeclarationCS.ast.name = identifierCS.ast  
paramDeclarationCS.ast.type = typeCS.ast
```

Inherited attributes

```
typeCS.env = paramDeclarationCS.env
```

queryBodyCS

```
[A] queryBodyCS ::= ';' ;  
[B] queryBodyCS ::= '{' OclExpressionCS '}'
```

Abstract syntax mapping

```
queryBodyCS.ast : OclExpression
```

Synthesized attributes

```
[B] queryBodyCS.ast = OclExpressionCS.ast
```

Inherited attributes

```
[B] OclExpressionCS.env = queryBodyCS.env
```

ExtOclExpressionCS

```
[A] ExtOclExpressionCS ::= OclExpressionCS  
[B] ExtOclExpressionCS ::= templateCS  
[C] ExtOclExpressionCS ::= RelationCallExpCS
```

Abstract syntax mapping

```
ExtOclExpressionCS.ast : OclExpressionCS
```

Synthesized attributes

```
[A] ExtOclExpressionCS.ast = OclExpressionCS.ast  
[B] ExtOclExpressionCS.ast = templateCS.ast  
[C] ExtOclExpressionCS.ast = RelationCallExpCS.ast
```

Inherited attributes

```
[A] OclExpressionCS.env = ExtOclExpressionCS.env  
[B] templateCS.env = ExtOclExpressionCS.env  
[C] RelationCallExpCS.env = ExtOclExpressionCS.env
```

RelationCallExpCS

```
RelationCallExpCS ::= (identifierCS[1] '.')? identifierCS[2] '(' argumentsCS? ')'
```

Abstract syntax mapping

```
RelationCallExpCS.ast : RelationCallExp
```

Synthesized attributes

```
RelationCallExpCS.ast.referredRelation =
  let
    trans:RelationalTransformation =
      if (not identifierCS[1].ast.oclIsUndefined)
      then
        RelationCallExpCS.env.lookup('imported transformation').
          referredElement.oclAsType(CollectionType)->
            any(t | t.name = identifierCS[1].ast)
      else
        RelationCallExpCS.env.lookup('context transformation').
          referredElement.oclAsType(RelationalTransformation)
      endif
    in
      trans.rule->any(r | r.name = identifierCS[2].ast).
        oclAsType(RelationalTransformation)
  RelationCallExpCS.ast.argument = argumentsCS.ast
```

Inherited attributes

```
argumentsCS.env = RelationCallExpCS.env
```

argumentsCS

```
argumentsCS[1] ::= OclExpressionCS (',' argumentsCS[2])?
```

Abstract syntax mapping

```
argumentsCS[1].ast : Sequence(OclExpression)
```

Synthesized attributes

```
argumentsCS[1].ast = Sequence{OclExpressionCS.ast}->union(argumentsCS[2].ast)
```

Inherited attributes

```
OclExpressionCS.env = argumentsCS[1].env
argumentsCS[2].env = argumentsCS[1].env
```

Operations

```
context RelationalTransformation::lookupClassName(names: Sequence(String)) : Class
post:
result =
let
  typesPackage:Package =
    self.modelParameter.usedPackage->any(p |
      not p.getEnvironmentWithoutParents().lookupPathName(names).oclIsUndefined())
in
  typesPackage.getEnvironmentWithoutParents().lookupPathName(names).oclAsType(Class)
```


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). Clause 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 Annex 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 a *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 defines a unique transformation, the content of the transformation definition does 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();  
}
```

A transformation is not restricted to a fixed number of models. Collections of *in* and *inout* models may also be transformed.

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

The multiple models can be distinguished using collection operations.

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 model types 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 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, which is the default. This implies that the XMI representation of the transformation definition has to be built 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 Extents, Models and Model Parameters

The inputs and output of a transformation are identified by Model Parameters which have a direction, model name and ModelType.

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

8.1.3.1 Implementation responsibilities

A transformation is executed by a QVTo implementation which must load an external model for each *in* and *inout* parameter and identify a future external model for each *out* parameter. When the transformation completes successfully, the implementation must save the *inout* models and the no longer future *out* models.

8.1.3.2 Extents

The interface between implementation and specified behavior uses a distinct MOF Extent associated with each Model Parameter. A MOF Extent identifies the root objects of its Model. The root objects are members of the Extent and are said to reside in the Extent.

The initial members of *in* and *inout* Extents are the roots of loaded external models. There are no initial members of *out* Extents.

Each Extent for an *in* Model Parameter is immutable. It may be shared when the same external model is loaded by multiple *in* Model Parameters. Other Extents are mutable and must be distinct even when the same external model is bound to more than one Model Parameter. There are therefore no shared model conflicts within the QVTo engine. Resolution of the conflict between multiple Extents that need to be saved to the same external model is a problem for the implementation to solve, perhaps by prohibiting the conflict in the first place.

8.1.3.3 Models

A Model comprises a forest of model elements (objects) with each tree defined by MOF containment relationships. For many applications, a single root, and its containment tree, rather than a forest is sufficient. However during the course of a transformation, model elements are created for use by the model and these may form additional roots until the model elements are assigned to their intended container.

Each Model Parameter has an Extent whose members are the root objects of a Model. The ModelParameter's ModelType identifies the permitted content of the Model, The ModelParameter's name may be used with the operations of the Model library type.

```
uml.objectsOfKind(Class);
```

The name may be used to assign all members of the Extent.

```
rdbms := aModel.copy();
```

A model assignment displaces any previous members associated with the extent in favor of the root model elements of the replacement Model. The previous members cease to reside in any Extent and so become orphans.

Additional Models and corresponding Extents may be created by the Model::copy() and Model::createEmptyModel() operations. These Extents have no associated external model and so their contents will be lost unless assigned to Extents with external models or unless passed to other transformations. The Models passed to nested transformation calls by Transformation::transform() or Transformation::parallelTransform() may not be modified after the call. This may require that a copy is created.

8.1.3.4 Object Containment and Extent Residence

A MOF object can be contained by at most one other object, consequently the containment relationship of a model form a forest. The roots of the forest have no containers, rather they are members of at most one Extent and are therefore said to reside in that Extent. The non-roots of the forest are transitively contained by a root that is a member of the Extent; the non-roots are therefore also said to reside in the Extent. Objects that do not reside in an Extent are orphans.

```
parent.child := firstChild; // firstChild contained by parent  
parent.child := secondChild; // secondChild contained by parent, displaced firstChild is a root object
```

Assignment of a container relationship, or opposite containment relationship, establishes a new parent-child containment relationship. This may displace the previous child so that it has no container, however the displaced child retains its residence in its models' extent; the displaced child is therefore added to the extent's members and is an additional root object for the model.

```
model.addElement(anObject); // anObject resides in model
```

Similarly assignment of the residence of a root object, establishes or updates the residence of the root object and transitively of all its contained objects. Any previous residence is eliminated.

When a transformation consistently assigns all residences and containments for all objects, the result is often a tree for the Extent of each *inout* and *out* parameter. However if any residence or containment is left unassigned, some orphan objects are lost.

Direct object creation or construction typically results in orphans whose containment and residence is assigned later since there is no inference of a default residence. The object may be explicitly assigned to the root of some extent when it is created:

```
column := new Column@mymodel(n,t); // mymodel is the extent for the new instance.  
object x:X@srcmodel { ... }      // x is created within the 'srcmodel'
```

Objects returned as *out* or *inout* parameters of a mapping are assigned to an explicit or inferred extent. If no extent can be inferred, typically because two ModelParameters use the same ModelType, the objects are orphans. This is a programming hazard that can be diagnosed by tooling.

An object may be completely removed from its extent by invoking Model::removeElement(). This makes the object an orphan with no references to or from any other object.

Cloned and deepcloned objects are added to the Extent of the source object; they are therefore new root objects until assigned to some container.

8.1.3.5 Orphans

An orphan is a potentially lost object; it resides in no container and no extent and so will not contribute to any *out* or *inout* model.

Orphans may arise as the result of:

- Model::removeElement() explicitly orphaning an object
- model assignment displacing an obsolete model
- non-mapping object construction

Creation of orphans by object construction is discouraged in two ways. An @extent may form part of the ObjectExp, or Mapping out parameter declaration. In the case of mappings, a default extent is inferred by metamodel compatibility. However objects constructed within helpers may be orphans.

An orphan may cease to be an orphan when a container is assigned, or when Model::addElement() is used to add the orphan element to the members of the Extent that identifies the root objects of the Model.

Although an orphan may have no container and may reside in no extent, it may still be referenced by objects that reside in an out extent. At the end of the transformation, when out extents are saved, out extents referencing orphans are not well-formed since the orphans are missing; the saved extent is incomplete and difficult to use. Practical implementations may assist diagnosis of the bad result by rescuing the referenced orphans as members of one or more extents or providing an additional lifeboat output model.

8.1.4 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|.getAllBaseClasses());
  // ....
}
```

The *UmlCleaning* transformation extends the *MyUmlFacilities* library, meaning 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.

8.1.5 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) or precondition, a mapping body, and a post-condition (a *where* clause). Even if it is not explicitly notated in the concrete syntax, a *mapping operation* is always a refinement of an implicit *relation*.

In strict invocation mode, a precondition (a *when* clause) is executed as an assertion before any other execution. Failure of the assertion causes a fatal execution failure. A postcondition (a *where* clause) is similarly executed as an assertion after all other execution has completed.

The *packageToSchema* mapping operation below defines how a UML Package has to be transformed into an 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 whose 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 { ... }
  {
    // inhibition section - see Clause 8.1.11.2
    init { ... }
    // instantiation section - see Clause 8.1.11.2
    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.

Before the **init**, there is an implicit section, named the inhibition section, which suppresses redundant re-execution of a mapping.

Following the **init** and before 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 initialization section. The null value tests mean that, in order to return an existing object rather than creating a new one, one simply needs to assign the result parameter within the **init** section. The instantiation section also creates the trace record of the execution.

In the *packageToSchema* mapping body the population keyword is omitted. The rules for interpreting the absence of the **population** keyword are given in Clause 8.2.1.19 “MappingBody”.

8.1.6 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. The list of expressions of the body of the object expression are then 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 Clause 8.1.5 uses in fact an implicit object expression to populate the RDBMS Schema. The equivalent textual definition, without the shorthand, 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 initialization 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 outcome 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 Clause 8.1.10.

8.1.7 Inlining Mapping Operations

The definition below describes the *class2table* mapping operation that creates an 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. On the other hand, splitting of mappings may favor reusability. Also it is important to be aware that the execution of an object expression does not imply the creation of a trace.

As stated before, an object expression, such as `'object x:X { ... }'`, does not create an instance if the target variable - 'x' in the example - has a non null value. In the `'object Key { ... }'` expression, since there is no variable reference, a Key instance will be 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 *imperative 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.

8.1.8 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 creates 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 mappings 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).

8.1.9 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 in sequence. A helper may have side-effects on the parameters; 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->add(self.retrieveCandidates());  
    return list;  
}
```

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

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

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

8.1.10 Intermediate Data

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

Below is 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 the transformation that defines it. This is similar to the query and mapping operations that conceptually are operation extensions. The *addUnderscores* query definition above illustrates the fact that it is even possible to add operations to the primitive types.

8.1.11 Tracing and Resolving

Execution of a mapping establishes a *trace-record* to maintain the relationship between its context object and the result object or objects. This relationship can be queried using one of the eight *resolve* expressions.

8.1.11.1 Trace Records

Execution of a transformation builds the overall *trace-data* which comprises a sequence of *trace-records*; one for each mapping execution in execution order. This includes every mapping executed by, accessed or extended, transformations or libraries, but not those by transformation invocations. Each *trace-record* comprises:

- *context-parameter* - the context parameter object
- *in-parameters* - the in and inout parameter objects or values
- *invoked-mapping* - the invoked mapping operation
- *executed-mapping* - the executed mapping operation
- *out-parameters* - the out parameter objects or values
- *result-parameters* - the result parameter object or objects

The *invoked-mapping* and *executed-mapping* operations may differ when a *disjuncting mapping* is executed. The *invoked-mapping* is the *disjuncting mapping*. The *executed-mapping* is the successfully selected *candidate mapping* or *null*.

inout parameters are traced once as *in-parameters*; they cannot change during production of the *trace-record*.

The *trace-record* is created during the instantiation section of the selected *candidate mapping*, which is after the initialization section and before the population section.

A *trace-record* is created by every mapping execution, unless predicates or initialization section fail.

A *trace-record* is created or re-used by every mapping invocation, unless predicates or initialization section fail. In the case of a standard mode execution for which no candidate mapping is selected, the *executed-mapping*, *out-parameters* and *result-parameters* are *null*.

The following example mapping declarations

```
mapping X::disjunctingMapping(in p1:P1, inout p2:P2, out p3:P3) : r1:R1, r2:R2
  disjuncts mappingName, ... {}

mapping X::mappingName(in p1:P1, inout p2:P2, out p3:P3) : r1:R1, r2:R2 .. {...}
```

may be invoked as

```
var t := anX.map disjunctingMapping(aP1, aP2, aP3);
var anR1 := t.r1;
var anR2 := t.r2;
```

to create the following *trace-record*. (Hyphens are used in names that are helpful for exposition, but which are not accessible to program code.)

```
object Trace-Record {
  context-parameter := anX;
  in-parameters := Sequence{aP1, aP2};
  invoked-mapping := X::disjunctingMapping;
  executed-mapping := X::mappingName;
  out-parameters := Sequence{aP3};
  result-parameters := Sequence{anR1, anR2};
}
```

The *trace-record* traces the relationship between mapping inputs and outputs; between the inputs {anX, aP1, aP2} and the outputs {aP3, anR1, anR2}.

Note that there is no trace for object construction in helpers or for nested object construction in mappings. If a trace of all objects is needed, a mapping must be used to create each object as a mapping output.

8.1.11.2 The inhibition and instantiation sections

The auto-generated inhibition and instantiation sections surround the initialization section of a mapping, whose general structure is:

```
mapping X::mappingName(in p1:P1, inout p2:P2, out p3:P3) : r1:R1, r2:R2
  when { ... }
  where { ... }
  {

    // inhibition section
    var trace-records := trace-data
      ->select(executed-mapping=X::mappingName)
      ->select(context-parameter=self)
      ->select(in-parameters->at(1)=p1)
      ->select(in-parameters->at(2)=p2);

    if (trace-records->notEmpty()) {
      var trace-record := trace-records->at(1);
      p3 := trace-record.out-parameters->at(1);
    }
  }
```

```

    r1 := trace-record.result-parameters->at(1);
    r2 := trace-record.result-parameters->at(2);
    return;
};

init { ... }

// instantiation section
if (p3 == null) p3 := object P3 {};
if (r1 == null) r1 := object R1 {};
if (r2 == null) r2 := object R2 {};

trace-data += object Trace-Record{
    context-parameter:=self,
    in-parameters:=Sequence{p1, p2},
    invoked-mapping:=X::disjunctingMapping,
    executed-mapping:=X::mappingName,
    out-parameters:=Sequence{p3},
    result-parameters:=Sequence{r1,r2}
};

population { ... }
end { ... }
}

```

In the inhibition section, the *trace-data* is consulted to locate all *trace-records* whose *executed-mapping*, *context-parameter* and *in-parameters* match the new candidate mapping invocation. If a match is found, the previous *out-parameters* and *result-parameters* are used as the mapping return values and the mapping re-execution is suppressed.

Then the initialization section provides an opportunity for the default *null* values of outputs to be replaced by something more useful. If the *null* values are unchanged, the instantiation section constructs an object for each object. *inout* parameters may not be changed during the initialization section. Finally the execution of the mapping is recorded by adding a *trace-record* to the *trace-data*.

A candidate mapping execution is suppressed to avoid creating a *trace-record* whose *context-parameter*, *in-parameters*, *invoked-mapping* and *executed-mapping* fields duplicate another *trace-record* already in the *trace-data*. When comparing *trace-record* fields, Class instances are compared as objects without regard for their content, and *DataType* values are compared by deep value equality. Traced object instances may therefore be modified between mapping executions without inhibiting detection of re-execution since only the object references are traced. However any modification of a traced *DataType* value such as a *List* inhibits detection of a re-execution since the entire value is traced.

When a re-execution attempt is detected, the re-execution is suppressed without any additional *trace-record* being created. Note that traced Class instances are mutable and so the re-used value may have changed in the interim.

8.1.11.3 resolve() - Resolution of target objects by Type

The *trace-data* may be queried to identify all target objects using the *resolve* operation without a context object or argument.

```
resolve()
```

The query may be restricted to identifying all target objects conforming to a given type by adding a type argument.

```
resolve(Table)
```


The returned target objects may be restricted to those mapped from a particular source object by supplying the source object as the context object.

```
source.resolve()
```

Additionally, or alternatively, the returned target objects may be restricted by an OCL condition.

```
source.resolve(t : Table | t.name.startsWith('_'))
```

These queries return a sequence of target objects in mapping invocation order.

An equivalent OCL-like query for *SOURCE.resolve(T : TYPE | CONDITION)* is

```
let selectedRecords = trace-data->select(in-parameters->including(context-parameter)->includes(SOURCE)) in
let selectedTargets = selectedRecords->collect(out-parameters->union(result-parameters)) in
selectedTargets->selectByKind(TYPE)->select(T | CONDITION)
```

8.1.11.4 resolveIn() - Resolution of target objects by Mapping

The trace data may be queried to identify all target objects produced by a given invoked disjuncting mapping or executed candidate mapping using the *resolveIn* expression.

```
resolveIn(Class2Table)
```

The returned target objects may be restricted to those mapped from a particular source object by supplying the source object as the context object.

```
source.resolveIn(Class2Table)
```

Additionally, or alternatively, the returned target objects may be restricted by an OCL condition.

```
source.resolveIn(Class2Table, t : Table | t.name.startsWith('_'))
```

These queries return a sequence of target objects in mapping invocation order.

An equivalent OCL-like query for *SOURCE.resolveIn(MAPPING, T : TYPE | CONDITION)* is

```
let selectedRecords1 = trace-data->select(in-parameters->including(context-parameter)->includes(SOURCE)) in
let selectedRecords2 = selectedRecords1->select((invoked-mapping = MAPPING)
or (executed-mapping = MAPPING)) in
let selectedTargets = selectedRecords2->collect(out-parameters->union(result-parameters)) in
selectedTargets->selectByKind(TYPE)->select(T | CONDITION)
```

8.1.11.5 invresolve() - Resolution of source objects by Type or Mapping

The corresponding inverse queries are available using the *invresolve* or *invresolveIn* expressions. These identify source objects mapped to a given type or by a given mapping.

```
invresolve()           // all sources
invresolve(Class)     // all sources of kind Class
target.invresolve()   // all sources mapped to target
target.invresolve(c : Class | c.name.startsWith('_'))
```

```

invresolveIn(Class2Table)           // all sources for Class2Table mappings
target.invresolveIn(Class2Table)     // all sources mapped to target by a Class2Table mapping
target.invresolveIn(Class2Table, c : Class | c.name.startsWith('_'))

```

8.1.11.6 resolveone() - Resolution of a single source or target object by Type or Mapping

The four *resolveone* variants of the four *resolve* expressions modify the return to suit the common case where only a single object is expected. The return is therefore the first resolved object or *null*.

```

resolveone()                       // the first target
resolveone(Table)                   // the first target of kind Table
source.resolveone()                 // the first target mapped from source
source.resolveone(t : Table | t.name.startsWith('_'))

resolveoneIn(Class2Table)           // the first target of a Class2Table mapping
source.resolveoneIn(Class2Table)     // the first target mapped from source by a Class2Table mapping
source.resolveoneIn(Class2Table, t : Table | t.name.startsWith('_'))

invresolveone()                    // the first source
invresolveone(Class)                // the first source of kind Class
target.invresolveone()              // the first source mapped to target
target.invresolveone(c : Class | c.name.startsWith('_'))

invresolveoneIn(Class2Table)       // the first source for a Class2Table mapping
target.invresolveoneIn(Class2Table) // the first source mapped to target by a Class2Table mapping
target.invresolveoneIn(Class2Table, c : Class | c.name.startsWith('_'))

```

8.1.11.7 Late resolution

The *resolve* expressions query the prevailing state of the *trace-data*. *resolve* cannot of course return results from mappings that have yet to execute and so careful programming of mapping invocations may be required to ensure that required results are available when needed. Alternatively a *late* keyword may prefix *resolve* when the resolution occurs within an assignment. This defers the execution of the assignment and the partial computation involving *late resolves* until all mapping executions have completed. The deferred assignment cannot return the future value; it therefore returns a null value.

More precisely, mappings execute, assignment right hand sides involving late resolutions are computed, then finally deferred assignments are made. The ordering in which late resolutions occur does not matter, since each late resolution can influence only its own deferred assignment.

```

myprop := mylist->late resolve(Table);
           // shorthand for mylist->xcollect(late resolve(Table))

```

This last example also demonstrates that an implicit imperative *xcollect* of resolutions may be performed, in this case requiring the collection to be performed after all mappings have executed.

8.1.11.8 Persisted Trace Data

The *trace-data* may be persisted and reloaded to support a re-execution. However the *trace-record* does not trace configuration data, transformation properties or intermediate data, and does not involve a deep clone of every traced object. It is therefore not possible to use a persisted form of the *trace-data* to support incremental re-execution of an arbitrary QVTo transformation since the required object state may not be present in persisted trace. A well-behaved transformation that avoids dependence on mutable object properties or other untraced facilities may be re-executable.

8.1.12 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.”

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.
- **resolveIn** looks for target objects created from a source object by a unique mapping operation.

```
JClass2JPackage.qvto
```

```
modeltype JAVA uses 'http://www.omg.org/qvt/examples/javamodel';
```

```
transformation JClass2JPackage(inout javamodel : JAVA);
```

```
main () {
  javamodel.objectsOfType(JClass)->jclass2jpackage();
}
```

```
mapping JClass::jclass2jpackage() : JPackage {
  init {
    result := resolveoneIn(JClass::jclass2jpackage, p : JPackage | self.packageName = p.name);
  }
  if (name = null) {
    name := self.packageName;
  }
}
```

In the example above, the *JClass::jclass2jpackage* mapping is invoked for each *JClass*. Within the mapping, *resolveoneIn*, without a source expression, is used to examine all *JPackages* created by the *JClass::jclass2jpackage* mapping. The *p* iterator traverses the *JPackages* and the condition selects only those whose name matches the *JClass*'s *packageName*. Using *resolveoneIn* rather than *resolveIn* ensures that at most one match is returned. The match is assigned to *result* for re-use avoiding the creation associated with a null *result*. A redundant re-assignment of *name* in the re-used *JPackage* is avoided by testing whether it has already been assigned.

- 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.

UML2JavaLate.qvto

```

modeltype JAVA uses 'http://www.omg.org/qvt/examples/javamodel';
modeltype UML uses 'http://www.omg.org/qvt/examples/umlmodel';

transformation Uml2Java(in uml:UML,out java:JAVA);

main() {
    uml.objectsOfType(Class)->map transformClass();
}

mapping UML::Class::transformClass() : JAVA::Interface {
    name := "Ifce" + self.name;
    base := self.superClass->late resolve(JAVA::Interface);
}

```

In the example above, the *UML::Class::transformClass()* mapping is invoked for each *Class* in the *uml* model. The mapping causes a corresponding *JAVA::Interface* to be created with *name* and *base* property initialization. The *name* is initialized by a simple String concatenation. The *base* requires the *JAVA::Interfaces* corresponding to each *self.superClass* to be resolved and referenced. The *resolve*, with a source expression, locates the *JAVA::Interface* created by any mapping from the object referenced by the source expression. Application of this on a collection resolves all creations from each of the collection elements.

Since it is awkward to ensure that the order of mapping execution is shallowest-superClass-first, a *late resolve* is used thereby deferring the assignment of the base property until all possible *superClasses* have been created by all possible *UML::Class::transformClass()* executions.

A *late resolve* expression is always executed in conjunction with an assignment. The assignment initially returns null, and the context that is required to execute the assignment later is stored as part of the trace state.

Below is an equivalent solution for this transformation problem that does not use the *late* resolution. This solution uses two passes.

UML2JavaTwoPass.qvto

```

modeltype JAVA uses 'http://www.omg.org/qvt/examples/javamodel';
modeltype UML uses 'http://www.omg.org/qvt/examples/umlmodel';

transformation Uml2Java(in uml:UML, out java:JAVA);

main() {
    uml.objectsOfType(Class)->map transformClass();
    uml.objectsOfType(Class)->map transformClassInheritance();
}

```

```

mapping UML::Class::transformClass() : JAVA::Interface {
    name := "Ifce" + self.name;
}

mapping UML::Class::transformClassInheritance() : JAVA::Interface {
    init {
        result := self.resolveoneIn(UML::Class::transformClass);
    }
    base := self.superClass->resolveIn(UML::Class::transformClass);
}

```

The main program now invokes the *UML::Class::transformClass()* mapping for each *Class* in the *uml* model to perform the first pass, then it invokes the *UML::Class::transformClassInheritance()* mapping for each *Class* to perform the second pass. The first mapping just creates a corresponding *JAVA::Interface* with a *name* initialized by a simple String concatenation. The second mapping relocates the *JAVA::Interface* created by the first mapping using an *init* section within which *resolveoneIn*, with *self* as a source expression, locates the result of the *UML::Class::transformClass()* mapping when applied to *self*. *base* is initialized by assigning the results of each of the *superClass* resolutions.

8.1.13 Composing Transformations

Composition of coarse-grained transformation is an essential feature in large and complex transformations. To this end, 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).

8.1.14 Mapping Overloading

Invocation of a mapping selects a disjunction of one or more *candidate mappings* at compile time. At run-time, the first matching *candidate mapping* is selected and invoked. The disjunction may be specified explicitly using the **disjuncts** keyword or implicitly by an overloaded mapping.

8.1.14.1 Explicit Disjuncts

In the following example, the explicit disjunction defines *convertFeature* as a *disjuncting mapping* name that may be invoked on a *UML::Feature* with a *Boolean* argument. *convertAttribute*, *convertConstructor* and *convertOperation* are *candidate mapping* names.

```
mapping UML::Feature::convertFeature(asUpper: Boolean) : JAVA::Element
  disjuncts convertAttribute, convertOperation, convertConstructor {}
mapping UML::Attribute::convertAttribute(asUpper: Boolean) : JAVA::Field {
  name := if asUpper then name.toUpper() else name endif;
}

mapping UML::Operation::convertConstructor(asUpper: Boolean) : JAVA::Constructor
  when {self.name = self.namespace.name;} {
  name := if asUpper then name.toUpper() else name endif;
}

mapping UML::Operation::convertOperation(asUpper: Boolean) : JAVA::Constructor
  when {self.name <> self.namespace.name;} {
  name := if asUpper then name.toUpper() else name endif;
}
```

The explicit **disjuncts** causes the mapping invocation to successively assess the implicit and explicit predicates of *convertAttribute*, *convertConstructor* and *convertOperation* to identify the first match. If no match is found the mapping invocation returns *null*.

The explicit predicates are provided by arbitrary constraints specified in *when* clauses. Implicit predicates are provided by the type signatures; each source and argument must conform to the type of the corresponding *candidate mapping* parameter.

The *candidate* return type must be covariant, that is the same as, or derived from that of, the *disjuncting* return type to ensure that no result incompatibility arises.

Since the argument types contribute to implicit predicates, the *candidate* parameter types may be supertypes or subtypes of the *disjuncting* mapping parameter type. The number of *candidate* and *disjuncting* parameter types must be the same.

An explicit *candidate mapping* is identified by its *mapping identifier* which may contribute to more than one disjunction.

8.1.14.2 Implicit Disjuncts

An implicit disjunction groups overloaded mappings. One mapping overloads another when the overloading source type extends the overloaded source type and when the overloading and overloaded mappings have same name and parameter count. Every mapping declaration establishes an implicit disjunction of itself and all its overloads, which may be in an extending transformation.

When *UML::Attribute* and *UML::Operation* extend *UML::Feature*, the previous example may be simplified to use an implicit disjunction.

```
mapping UML::Feature::convertFeature(asUpper: Boolean) : JAVA::Element {}
mapping UML::Attribute::convertFeature(asUpper: Boolean) : JAVA::Field {
  name := if asUpper then name.toUpper() else name endif;
}

mapping UML::Operation::convertFeature(asUpper: Boolean) : JAVA::Constructor
```

```

    when {self.name = self.namespace.name;} {
        name := if asUpper then name.toUpper() else name endif;
    }

    mapping UML::Operation::convertFeature(asUpper: Boolean) : JAVA::Constructor
    when {self.name <> self.namespace.name;} {
        name := if asUpper then name.toUpper() else name endif;
    }

```

The explicit disjuncts provides distinct names and so facilitates explicit calls direct to the *candidate mappings*. The implicit disjuncts requires no *disjuncting* declaration and so facilitates extension by addition of further contributions.

8.1.14.3 Disjunct candidates

All mappings with the required name, parameter count and matching or derived source type are *candidate mappings* for the invocation of a *disjuncting mapping*. This includes mappings inherited from extended transformations. The *candidate mappings* referenced in a *disjuncting mapping* may introduce new names and consequently a further disjunction of *candidate mappings*; the explicit disjunct is transitive.

For instance invocation of *convertFeature* for a *Property* in the explicit disjuncts example should consider a *Property::convertOperation(Boolean)* inherited from an extended transformation since the explicit disjunct adds *convertOperation* to the transitive candidates. Conversely, the implicit disjunct example considers only candidates whose signature is *convertFeature(Boolean)*.

For standard evaluation, a deterministic selection order for evaluation of the predicates of the candidates as guards is established by sorting using the following prioritized criteria. A distinction by an earlier criteria overrules all later criteria.

- directly invoked explicitly disjuncted candidate mappings are evaluated in declaration order
- mappings in the current transformation are selected before those in an extended transformation, then mappings in an extended transformation before those in an extended extended transformation, and so forth
- mappings for a more derived type are selected before those for a less derived type
- mappings are prioritized by alphabetical mapping name order
- mappings are prioritized by alphabetical context type name order
- mappings are prioritized by alphabetical context type containing package name order, then by containing package containing package name order, and so forth

The ordering above ensures that an extending transformation can occlude a mapping in an extended transformation and that a mapping for a derived type occludes that for a base type. (An implementation may use static analysis of the predicates to eliminate occluded candidates completely and to provide reduced candidate lists according to the source type of the mapping invocation.)

For strict evaluation, the same ordering applies but the first candidate for which the source type conforms is selected without evaluating the predicate as a guard. The predicate is instead evaluated as a pre-condition giving an assertion failure when not satisfied.

8.1.15 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 initialization 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 mapping operations 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 the *Foo* instances 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 t := f.foo2atombar();           // Tuple(atom: Atom, bar: Bar)
var atom := t.atom;
var bar := t.bar;
```

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.

8.1.16 Type Extensions

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

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 generic MOF type Object 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 - also known as a hash table. It is a mutable type.

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

A typedef can be used to define an alias for a complex type such as a tuple type:

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

8.1.17 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 realizes a compromise between functional features 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.

```
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 first. It plays the role of a loop instruction in Java language.

```
self.ownedElement->forEach(i|i.ocIsKindOf(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 *xcollect* and *xselect* operations and various others high-level facilities.

The language also defines an imperative “if-then-else” construct that is less 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(" ") "PROTECTED"
        elif (self.type.isPrimitive() "NORMAL"
        else "UNEXPECTED"
```

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;
    ...
  }
}
```

8.1.18 Pre-defined Variables: this, self, and result

The variable *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* is the name of the unique result parameter, which is a tuple when there are multiple result parameters. When the name of a single result parameter is specified explicitly, no pre-defined *result* variable exists.

8.1.19 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.

8.1.20 Invalid

When an OCL evaluation fails it does not raise an Exception, rather it returns the *invalid* value. The *invalid* may be the result of a hard error such as divide-by-zero, or an ordered-collection-index-out-of-bounds or a programming problem such as navigation of a *null* object.

Every use of a source value by an *ImperativeExpression* has an implicit assertion that the source value is not *invalid*, therefore using *invalid* is an assertion failure.

Initialization of, or assignment to, a variable stores rather than uses the value. *invalid* may be stored in a variable, however subsequent access of the variable will probably use it and encounter an assertion failure. An *invalid* value may be tested by the *oclIsInvalid()* operation without causing a failure.

8.1.21 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 transformation definitions computed automatically. It is indeed possible for a transformation definition to be the output of a transformation definition - 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 *PimToPsm* 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 *AnnotatedPimToPsm* 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 achieved through the invocation of the *wait* operation on the status variable returned by the '*transform*' operation.

The example below is a '*Requirement To Psm*' transformation that decomposes a requirement model into two intermediate PIM models (one for the GUI, another for the behavior) and then merges the two pim models 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

This clause defines the abstract syntax of the *QVT operational* formalism. Concepts are depicted first graphically through class diagrams, and then a description of each class is given. When applicable, additional sub-clauses 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 through a “from <class-name>” annotation.

8.2.1 The QVTOperational Package

The `QVTOperational` package defines the concepts that are needed to specify transformation 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 that extends OCL with constructs that are common in programming languages.

For convenience, the metaclasses defined by this package are 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 8.1 depicts the definition of all the metaclasses of this group. They are all closely related to the definition of an operational transformation.

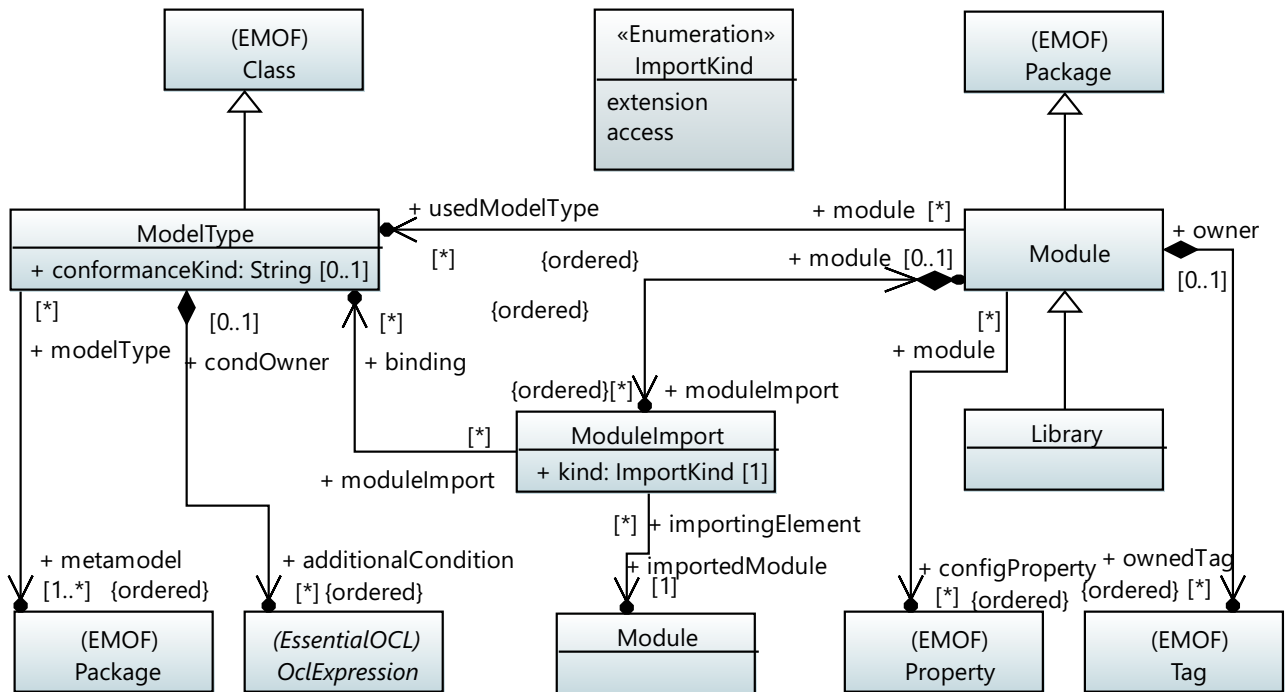


Figure 8.1 - QVT Operational Package - Modules and ModelTypes

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).

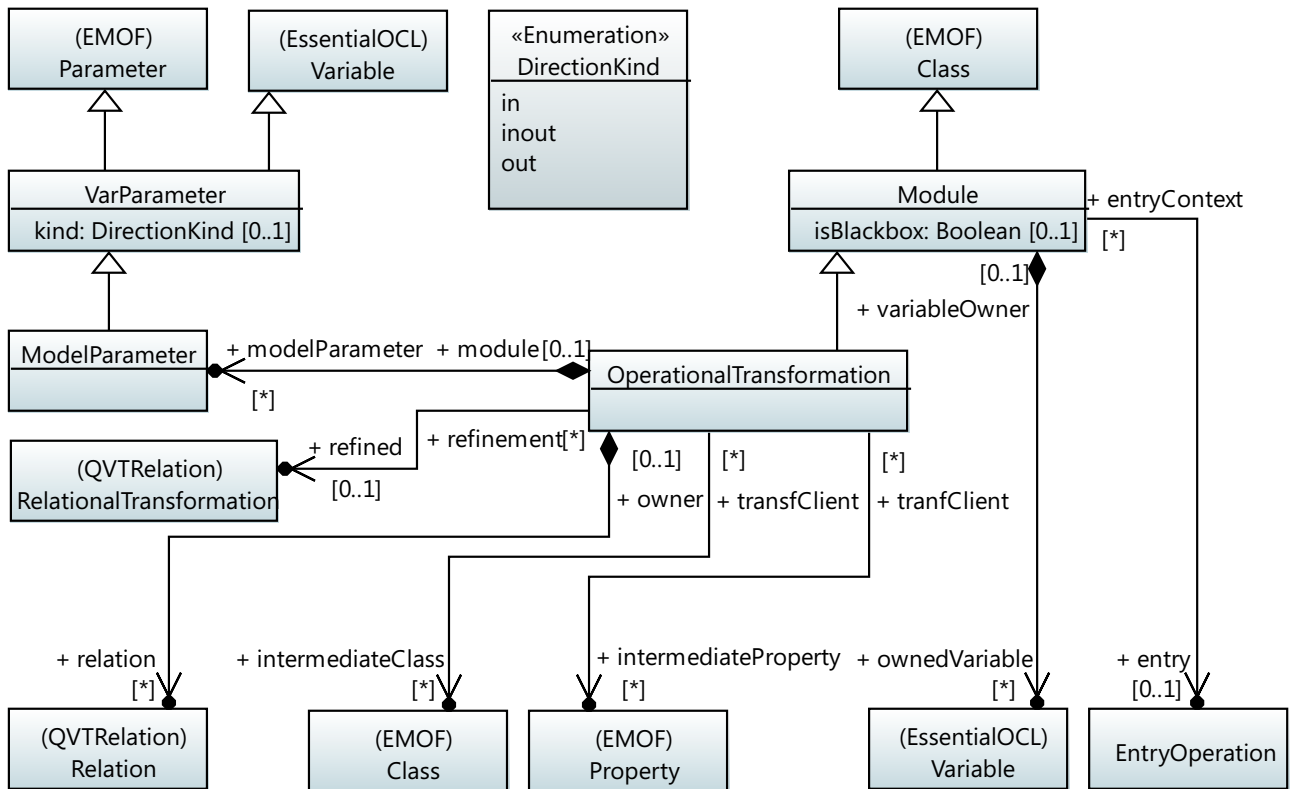


Figure 8.2 QVT Operational Package - OperationalTransformation and ModelParameter

An operational transformation may define configuration properties, that is, properties that 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.

An operational transformation may be explicitly declared as the ‘refinement’ of a relational transformation. In that case each model parameter in the operational transformation corresponds to a typed model in the relational transformation. The enforced direction is the one corresponding to the output parameter of the operational transformation.

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

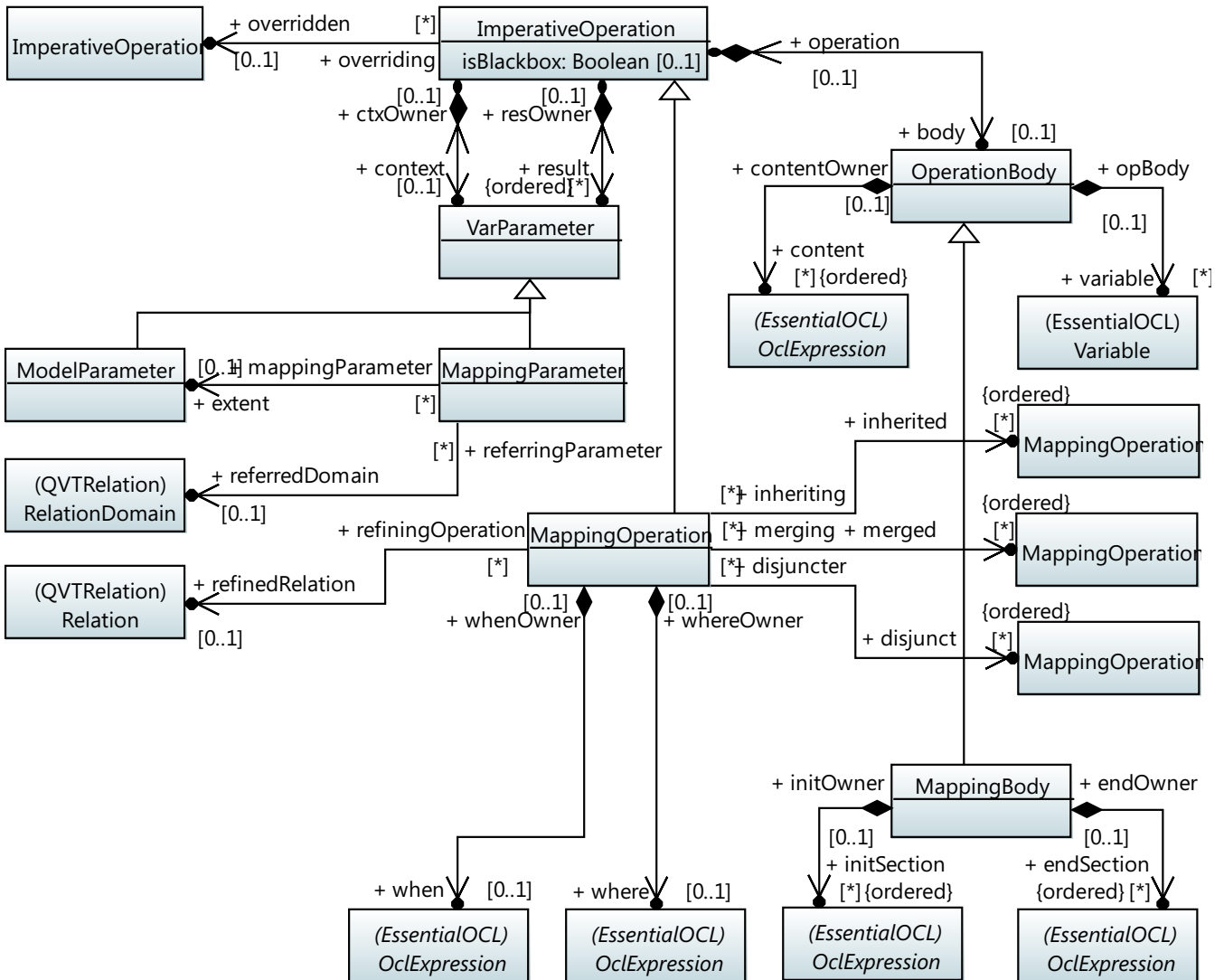


Figure 8.3 QVT Operational Package - Operations and Bodies

Associations

```
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). If the transformation defines intermediate classes this contains a parameter named `'_intermediate'` of the type `'_INTERMEDIATE'` (see `'Module::ownedType'` property 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.

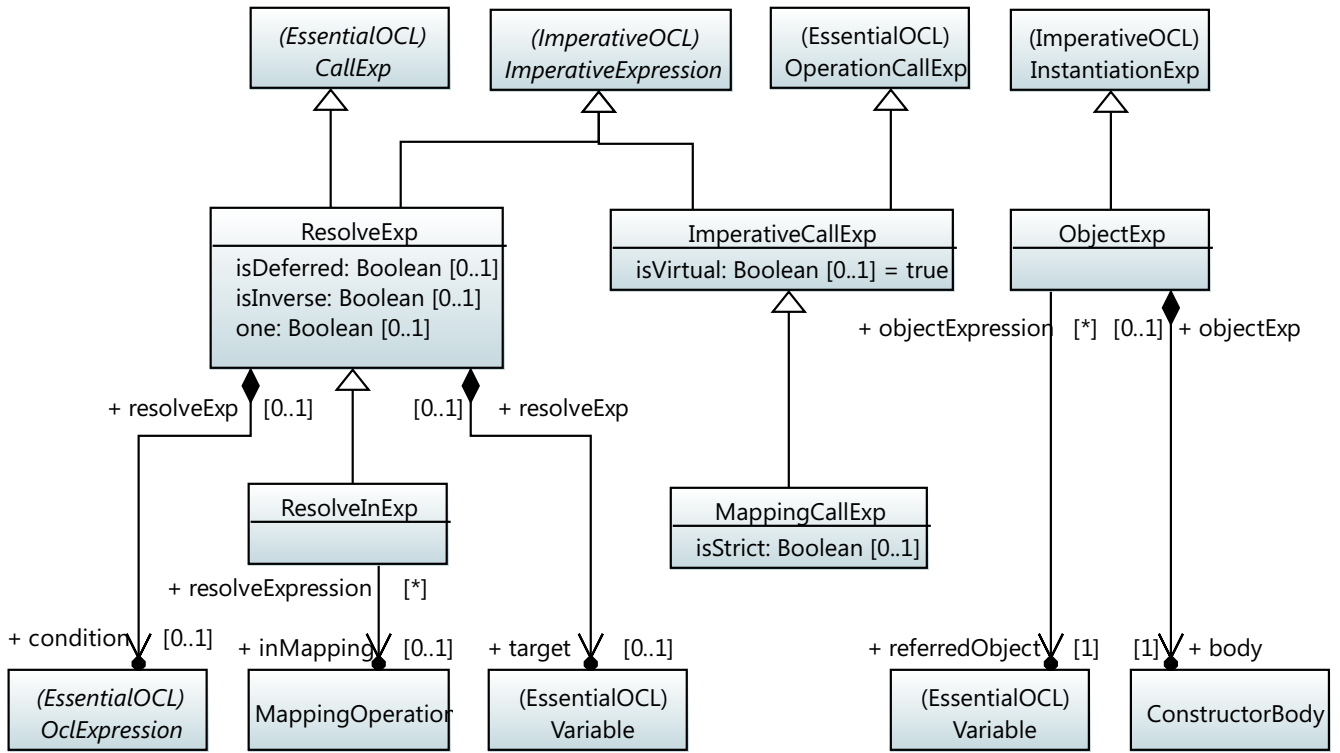


Figure 8.4 QVT Operational Package - Imperative Expressions

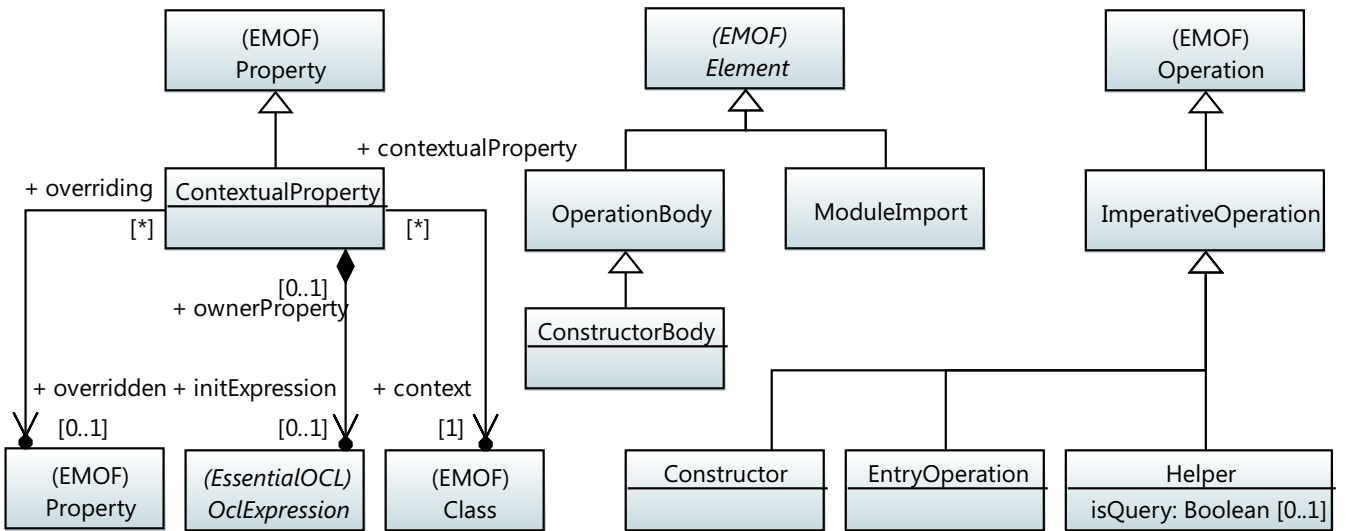


Figure 8.5 - QVT Operational Package - Miscellaneous Elements

`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 - except for ensuring trace persistence.

`refined : Transformation [0..1]`

Indicates a relational transformation (see Clause 7: Relations) being refined by this operational transformation.

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

The ordered set of relation definitions (see Clause 7: Relation) that are associated with the mapping operations (see `MappingOperation`) of this operational transformation.

Constraints

Class inheritance should not be used for `OperationalTransformations`.

`self.superClass->isEmpty()`

Semantics

The execution semantics of an operational transformation is described below 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:

1. For each declared **out** parameter a MOF extent is created with empty contents.
2. The attribute of the slots associated with each model parameter is assigned.
3. Loading the values of the configuration properties, if available.

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

The implicit “this” variable is represented by a `Variable` instance named “this” having the transformation as its type and owned by the transformation through the ‘ownedVariable’ property.

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

As in java, instantiation and execution of the main operation are separate 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, 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 braces 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 latter 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 **refines** 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 may have hierarchical scope. They 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: this means that 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

The execution semantics for a library is described below 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 type 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; 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 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 clause in the standard library).

Associations

`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.

`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 model types being used. This includes the implicit ‘`_INTERMEDIATE`’ model type when available (see `ownedType` description).

`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.

`ownedVariable` : Variable [0..*] {composes}

The list of variables owned by this module.

`/nestedPackage` : Package [0..*] (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 - for instance a ‘`Set(MyMetaclass)`’ user-defined datatype.

If the module contains the declaration of intermediate classes (see `OperationalTransformation::intermediateClass` definition) a model type named ‘`_INTERMEDIATE`’ is automatically defined and inserted in the list of owned types. The model type package is also named ‘`_INTERMEDIATE`.’ This package is nested by the transformation (by means of the inherited `Package::nestedPackage` property).

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 - 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 Clause 8.2.2.22 “InstantiationExp”).

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

Superclasses

Element

Attributes

kind: ImportKind [1]

The semantics of the library import. Possible values are *access* and *extension*. *access* is the default.

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 an operational transformation. Hence, the ordered set of model parameters forms the signature of the transformation. Each model parameter refers implicitly to a model or collection of models 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 a `ModelParameter` is a `ModelType` or `Collection` of a `ModelType`.

```
self.type.ocIsKindOf(ModelType)
or (self.type.ocIsKindOf(CollectionType)
and self.type.ocAsType(CollectionType).elementType.ocIsKindOf(ModelType))
```

Notation

Model parameters are notated as simple or collection parameters within the signature of a transformation.

transformation ManyToOne(**in** many:Sequence(Mine),**out** one:Mine)

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. See semantics sub-clause for the binding rules between actual and effective types.

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.

To restrict further the set of valid participant models, 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 each `ModelParameter` has a `ModelType` for a MOF Extent whose members are the root objects of the corresponding Model.

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 - in the case the referred metamodel is an *effective metamodel*.

additionalCondition: OclExpression [*] {composes, ordered}

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

Semantics of model compliance

When an operational transformation is 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 - see the QVT standard library.

In both cases - strict or effective - model compliance implies that all the extra conditions (see `extraCondition` property) are satisfied as well as any pre-existing well-formedness rule that exists for the associated metamodel.

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 beyond the scope of this specification.

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 follows:

```
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 a 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 declaration may have the same name as 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 a 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 to 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 clause are:

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

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] {composes}

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

result: VarParameter [*] {composes,ordered}

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

body: OperationBody [0..1] {composes}

The imperative implementation for this operation.

overridden: ImperativeOperation [0..1]

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

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

The list of parameters of the operation excluding the context and result parameters.

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 when the transformation execution starts.

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

Helpers and queries are operations that perform a computation on one or more source objects and provide a result. The body is an ordered list of expressions that are executed in sequence.

A *query* has no side-effects. The `isQuery` property is true. All parameters are implicitly *in*.

A *helper* may have side-effects; for instance, a list may be passed and changed within the body and its effect is visible after the operation call termination. The `isQuery` property is false. Parameters may be *in*, *inout* or *out*. A *helper* may create or modify Class instances or mutable DataType values such as List or Dict.

A *helper* or *query* may create mutable DataType values such as List or Dict or immutable DataType values such as Tuple, Set or String.

When more than one result is declared in the signature of a *helper* operation, the invocation of the operation returns a tuple.

All Class parameters are passed or returned by reference. An *in* Class parameter may not be modified.

in DataType parameters are passed by value and may be modified by a *helper* without affecting the caller.

result DataType parameters are returned by value.

inout and *out* DataType parameters are passed by reference-to-variable, that is the caller passes a reference to a variable whose value may be updated zero or more times by the *helper*. These updates are visible to the caller.

The initial value of each *out* and *result* parameter is *null*.

The value of *out*, *inout* and *result* parameters may be updated many times by assignments.

The final value of each *out*, *inout* and *result* parameter is returned to the caller.

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.oclIsTypeOf(OperationBody)
```

Notation

The notation uses the standard convention for notating operations in UML except that the **query** or **helper** keywords are used to prefix the declaration. The latter ‘helper’ prefix is used when the operation 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.

The **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 that 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 allows 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 can be implicitly or explicitly invoked through an instantiation expression (see *InstantiationExp*).

Superclasses

```
ImperativeOperation
```

Associations

```
/body : ConstructorBody [0..1] {composes} (from ImperativeOperation)
```

The imperative implementation for this constructor.

Constraints

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

```
self.body.oclIsTypeOf(ConstructorBody)
```

Notation

The notation for declaring constructors is similar to the notation for declaring any imperative operation except that it uses the **constructor** keyword and declares no result. The name of the constructor is usually 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 [0..1]

The imported property being overridden by this property.

initExpression : OclExpression [0..1] {composes}

An optional OCL Expression to initialize the contextual 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 and 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, 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 a situation where the escape mechanism is useful is when the transformation treatment requires dedicated techniques like lexical and syntax analysis that will be hard to provide using the QVT formalism.

A mapping operation always refines conceptually a relation where each relation domain corresponds to a parameter of the operational mapping.

The **when** clause acts either as a *pre-condition* when invoked with strict semantics, or as a *guard*, when invoked with standard semantics. The **where** clause always acts as a post-condition for the mapping operation. *pre-conditions* and *post-conditions* are assertions; any execution failure is fatal.

The body of a mapping operation is structured in three optional sections. The **init** (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 **end** (finalization) section is used to define termination computations that take place before exiting the body.

A mapping operation may be explicitly defined as a disjunction of *candidate mapping* operations. Every mapping operation is also an implicit disjunction of all mappings that are overloads as a consequence of matching name and argument count. Execution of a disjunction involves selecting the first candidate mapping whose when clause and other predicates are satisfied and then invoking it. This is described in Clause 8.1.14. The empty body of the disjuncting mapping is not executed.

Additionally, there are two extension mechanisms associated to mapping operations:

1. A mapping operation may *inherit* from another mapping operation. This means invoking the initialization section of the inherited operation after executing its own initialization section.
2. A mapping operation may also *merge* other mapping operations. This means invoking the merged operations after the termination section.

The execution semantics sub-section below provides the details of these mapping extension mechanisms.

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

Superclasses

`ImperativeOperation`

Attributes

`/isBlackbox: Boolean (from ImperativeOperation)`

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

Associations

`inherited: MappingOperation [*] {ordered}`

Indicates the list of the mappings that are specialized.

`merged: MappingOperation [*] {ordered}`

Indicates the list of mapping operations that are merged.

`disjunct: MappingOperation [*] {ordered}`

Indicates the list of potential mapping operations to invoke.

`refinedRelation: Relation [0..1]`

The refined relation, if any.

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

The pre-condition or the guard of the operational mapping. It acts as a pre-condition when the operation is called with strict semantics, it acts as a guard otherwise (see `MappingCallExp`).

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

The post condition for the operational mapping.

Constraints

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

```
self.body.oclIsTypeOf(MappingBody)
```

The body of a *disjuncting mapping* must be empty.

```
disjunct->notEmpty() implies body = null
```

Execution Semantics

We firstly define the semantic of the execution of a mapping operation in the absence of any inheritance or merge reuse facility.

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 building a prioritized list of *candidate mappings* then selecting the first *candidate mapping* for which, using standard invocation mode, all predicates are satisfied. For strict invocation mode, the first *candidate mapping* for which the source object (self variable) conforms to the context type is selected and predicates are executed as pre-conditions. If no candidate mapping is identified, *null* is returned to all results. Building the prioritized *candidate mapping* list is described as part of the Mapping Overloading description in Clause 8.1.14.

After call resolution, the parameters of the mapping include: the context parameter (if any), the owned parameters (from `Operation::ownedParameter`), and the parameters declared as result.

All Class parameters are passed or returned by reference. An *in* Class parameter may not be modified.

in DataType parameters are passed by value and may be modified by the mapping without affecting the caller.

result DataType parameters are returned by value.

inout and *out* DataType parameters are passed by reference-to-variable, that is the caller passes a reference to a variable whose value may be updated zero or more times by the mapping. These updates are visible to the caller.

The initial value of each *out* and *result* parameter is *null*.

The value of *out*, *inout* and *result* parameters may be updated many times by assignments.

The final value of each *out*, *inout* and *result* parameter is returned to the caller.

After passing the parameters, the type compliance of the actual object parameters as well as 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:

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

3. 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.
4. 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 inherit from other mappings

A mapping that has inherited mappings invokes first its initialization 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 were changed in the initialization 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.

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 <mappingids>, merges <mappingids>, disjuncts <mappingids>,
refines <mappingids> when {<exprs>} where {<exprs>}
```

where `<mappingids>` is one or more comma-separated mapping identifiers.

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* that restricts the changeability of the argument passed when the mapping operation is invoked. Possible values of direction kinds are: `in`, `inout`, and `out`.

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

`extent: ModelParameter [0..1]`

The extent of the mapping parameter. If not provided, the extent is inferred by inspecting the model types of the transformation. See the inference rules below. Should be explicitly provided when there is an ambiguity on the extent to own the potential created element corresponding to this parameter.

`referredDomain: RelationDomain [0..1]`

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.

Extent inference rule for mapping parameters

If the mapping parameter direction is “in,” inspect the input model types of the transformation to find the one that contains the type of the parameter. A model type “contains” a type if the type is directly or indirectly contained by the package defining the model type.

If the model parameter direction is “inout” or “out,” inspect the inout or output model types of the transformation to find the one that contains the type of the parameter.

In both cases there should be a unique model type found.

Notation

The extent of a mapping parameter can be provided explicitly using the '@' symbol after the type of the mapping parameter. In that case the declarator has the form:

```
mymappingparameter : myextent::MyType@mymodelparameter
```

It is not mandatory to provide the extent when it can be inferred from the type.

Example:

```
transformation T(in src:S, out dest1:D, out dest2:D);  
mapping X::foo(inout Y@dest1) : Y@dest2;  
// 'X' is a class of 'S' metamodel and 'Y' is a class of 'D' metamodel
```

8.2.1.17 OperationBody

An *operation body* contains the implementation of an imperative operation that is 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 [*] {composes, ordered}`

The list of expressions of the operation body.

`variable: Variable [*] {composes}`

The variables defined implicitly within this operation body. This concerns implicit variables in object expressions (ObjectExp).

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, if any, and the **result** variable represents the parameter to be returned, which is a tuple when multiple results are declared.

8.2.1.18 ConstructorBody

A *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: initialization, population, and termination sections. See execution semantics in the MappingOperation metaclass description.

Superclasses

OperationBody

Associations

`initSection: OclExpression [0..*] {composes,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..*] {composes,ordered} (from OperationBody)`

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

```
endSection: OclExpression [0..*] {composes,ordered}
```

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

Notation

The notation uses braces that come after the signature of the mapping operation. The general syntax 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. The **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 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, however, be required in certain situations, 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, possibly a tuple if more than a result is declared.

Concepts related to the usage of imperative operations.

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

8.2.1.20 ImperativeCallExp

An imperative call expression represents the invocation of an imperative operation. Unless `isVirtual` 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

`isVirtual: 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. 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 - the source is the transformation class itself, denoted by 'this,' 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, depending on the value of the `isStrict` Boolean property. In strict mode, failure to evaluate the *when* clause as a pre-condition causes the mapping execution to fail. In contrast in standard mode, failure to evaluate the *when* clause as a guard causes execution of the mapping body to be skipped and *null* to be returned to the caller.

Superclasses

`ImperativeCallExp`

Attributes

`isStrict : Boolean`

Indicates whether the mapping invocation mode is strict or standard.

Notation

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

If the invoked mapping defines a contextual parameter, the call notation will 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 collection as source and have, if needed, the iterator variable in parentheses. This is called the “imperative 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()
```

Invoking a mapping operation with a transformation instance as the source argument may be useful when a transformation make usage of other coarse-grained transformations (transformation composition). 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 that is used to filter the target objects.

The source object is optional. When no source object is provided, this expression inspects all the targets created or updated by all mapping operations irrespective of the source 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.

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 may be 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 returns a *future* value containing the necessary information to compute the resolution later. See execution semantics.

Associations

target: Variable [0..1] {composes}

A variable whose type indicates the primary condition for filtering the potential result objects. The extra condition (see ‘condition’ property) may use the variable to express a complementary filtering condition. This variable also has an influence in the type returned by the resolve expression (see type returned by the resolution expression).

condition : OclExpression [0..1] {composes}

An optional additional Boolean condition to be evaluated to filter the potential results.

Type of a resolve expression

The type of a *ResolveExp* expression depends on the type of the ‘target’ variable and on the multiplicity indication (the ‘one’ property). If ‘one’ is true, the returned type is the type of the ‘target’ variable. Otherwise, the returned type is a Sequence of the type of the ‘target’ variable. If no target variable is provided, the type is either *Object* (the type representing all types, see Clause 8.3.1) either a Sequence of Objects - depending on the multiplicity.

Execution Semantics

The trace information for a mapping operation invocation is created after the execution of the initialization 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 implicit 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 - necessarily a unique value - 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 at the end of the execution of the entry operation of the transformation. The tuple storing this information is appended to an ordered list that is given to the entry operation to terminate with the execution of the transformation.

Notation

The notation uses one of these three forms:

```
<resolve_op> '(' (<identifier> ':')? <typespec> ')' // no extra condition
| <resolve_op> '(' (<identifier> ':')? <typespec> '|' <expression> ')' // with extra condition
| <resolve_op> '(' ')' // no target, no extra condition
```

where the `<resolve_op>` is one of the following: `resolve`, `resolveone`, `invresolve`, and `invresolveone`.

When *isDeferred* is true the `late` keyword is used before the `<resolve_op>`.

The resolution operator may be called on a collection. This is a shorthand for invoking it in the body of an *xcollect* `ImperativeIterateExp` expression.

```
myresult := mysourceobject.resolveone(Table);
myresult := mysourceobject.resolveone(t:Table | t.name.startsWith("_"));
myprop := mylist->late resolve(Table);
// shorthand for mylist->xcollect(late resolve(Table))
```

See also the third and fourth examples of Clause 8.1.12.

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.

A resolve in expression may be applied with an empty source argument. In that case it inspects all objects created or updated by the rule (instead of inspecting only the objects created or updated from the source object).

Type of a resolveIn expression

The type of a *ResolveInExp* expression depends on the type of the ‘target’ variable, the ‘inMapping’ operation and on the multiplicity indication (the ‘one’ property). The overall returned type is specified in terms of an intermediate resolved type.

If a ‘target’ variable is provided, the resolved-type is the type of the ‘target’ variable
Otherwise if an ‘inMapping’ is provided, the resolved-type is the type of the ‘inMapping’.
Otherwise the resolved-type is *Object* (the type representing all types, see Clause 8.3.1).

If ‘one’ is true, the returned type is the resolved-type. Otherwise, the returned type is a Sequence of the resolved-type.

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 notation uses the same syntax as *ResolveExp* except that the operation names are one of the following **resolveIn**, **resolveoneIn**, **invresolveIn**, or **invresolveoneIn**. The two variants starting with “inv” prefix correspond to the “inverse” variant (`ResolveExp::isInverse == true`).

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.

The reference to the rule is given by a qualified identifier (context class and name). As a limitation of the concrete syntax, it is not possible to provide a reference to a rule if there is an ambiguity (having the same name and context but different parameters).

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

See also the second and third examples of Clause 8.1.12.

8.2.1.24 ObjectExp

An *object expression* is an *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.

Object creation, initialization and residence are separate activities. Object creation occurs when the referredObject has a null value; it is skipped if the referredObject variable references an existing object. Object initialization always occurs, but may be trivial if the body is empty. Object residence is left unchanged when the extent is omitted; it will be established as soon as the created object is put at the target end of some composition relationship. An explicit object residence may be established by specifying the model parameter for the required extent as the extent.

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

Superclasses

InstantiationExp

Associations

referredObject: Variable [1]

The object to be updated or created.

body: ConstructorBody [1] {composes}

The inline constructor body for the object to be instantiated or updated.

/initializationOperation : Constructor [0..1] (from InstantiationExp)

The constructor that uses the arguments to initialize the object after creation. The constructor may be omitted when implicit construction occurs with no arguments.

/instantiatedClass: Class [1] (from InstantiationExp)

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

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

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 an explicit extent is provided, the model parameter variable name postfixes the type of the object expression using the “@” separator symbol.

object x:X@srcmodel { ... } // x is created within the ‘srcmodel’

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

list->**object**(x) X{... } // shorthand for list->**xcollect**(x | object X{ ... })

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 concerns the additions to the type system.

Imperative expressions

Figure 8.6 depicts the class hierarchy for the side-effect expressions.

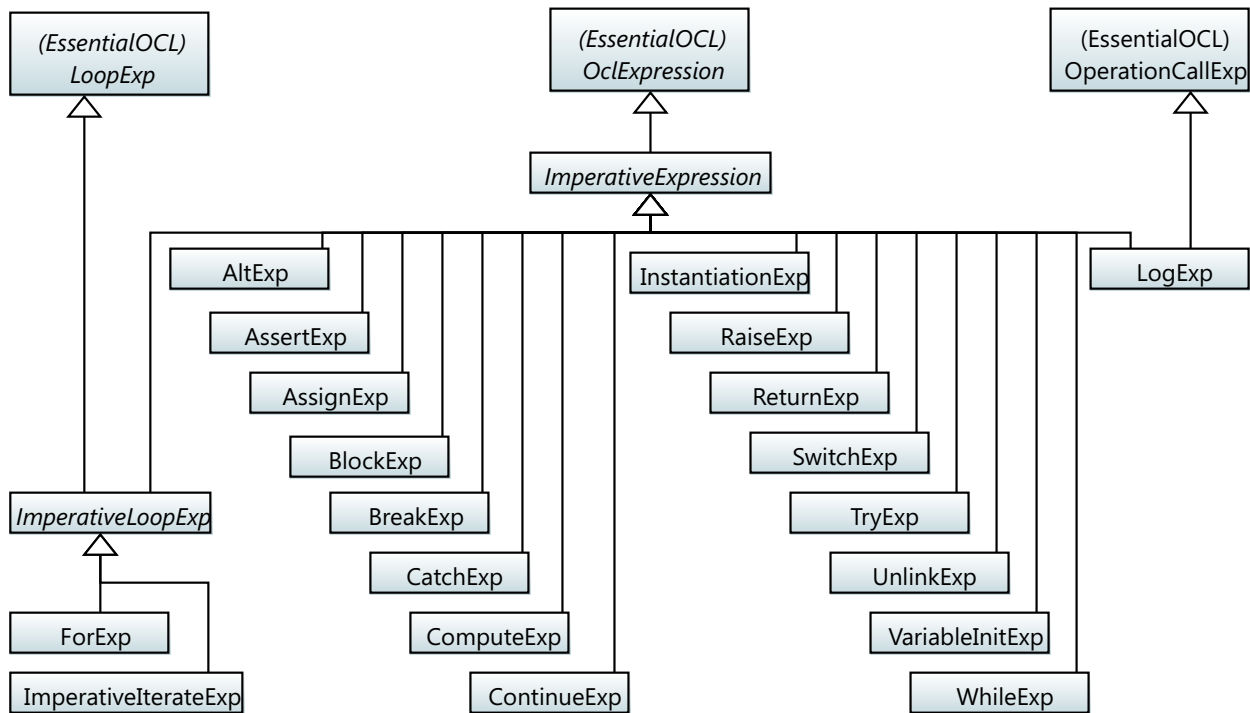


Figure 8.6 - Imperative OCL Package - Side-effect expressions hierarchy

Figure 8.7 depicts the specific control and block expressions enhancements as well as the instantiation facility.

Figure 8.8, Figure 8.9 and Figure 8.10 depict the remaining expressions like those related with attribute manipulation, assignment, exception handling.

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.

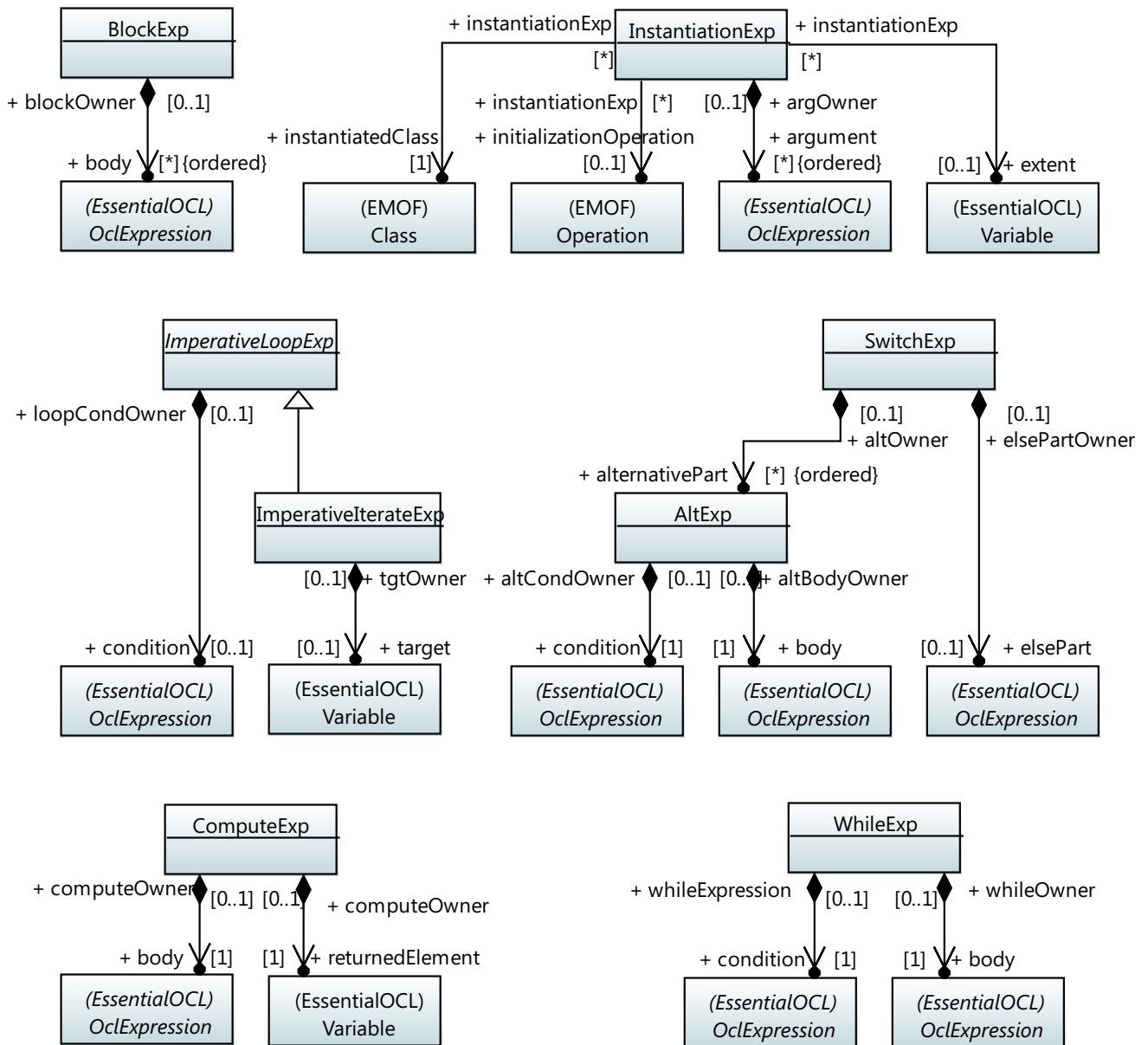


Figure 8.7 - Imperative OCL Package - Control and instantiation constructs

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.

Note also that since an *ImperativeExpression* can introduce side effects, it may not be used in a side effect free *OclExpression* even though its inheritance from *OclExpression* might suggest that it can. When side effects are required, imperative constructs such as *ImperativeLoopExp*, *SwitchExp* or *VariableInitExp* should be used in place of *LoopExp*, *IfExp* or *LetExp*.

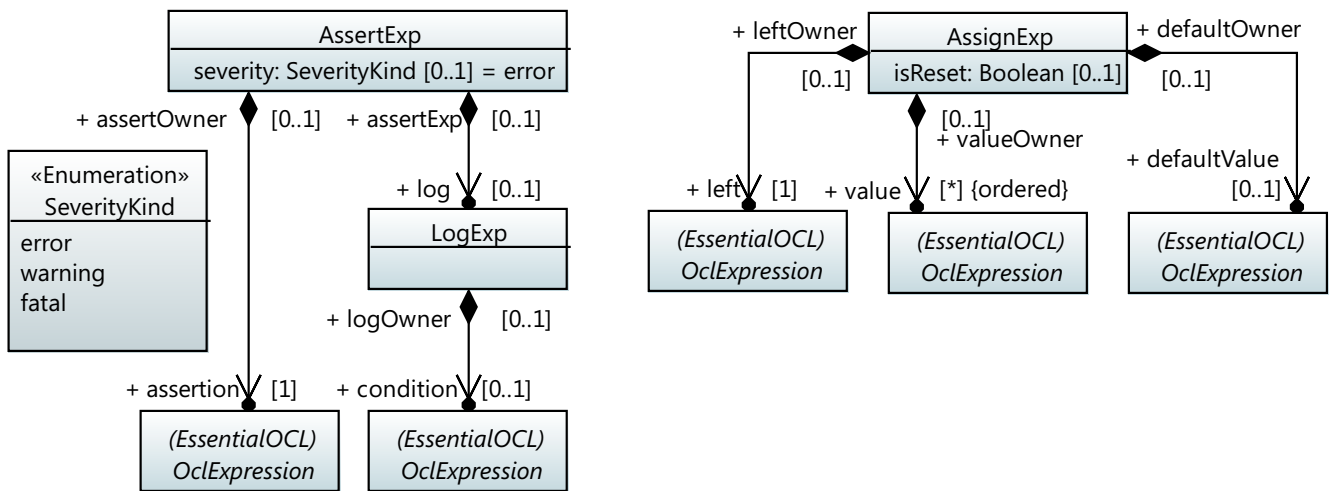


Figure 8.8 Imperative OCL Package - Assert and Assign Expressions

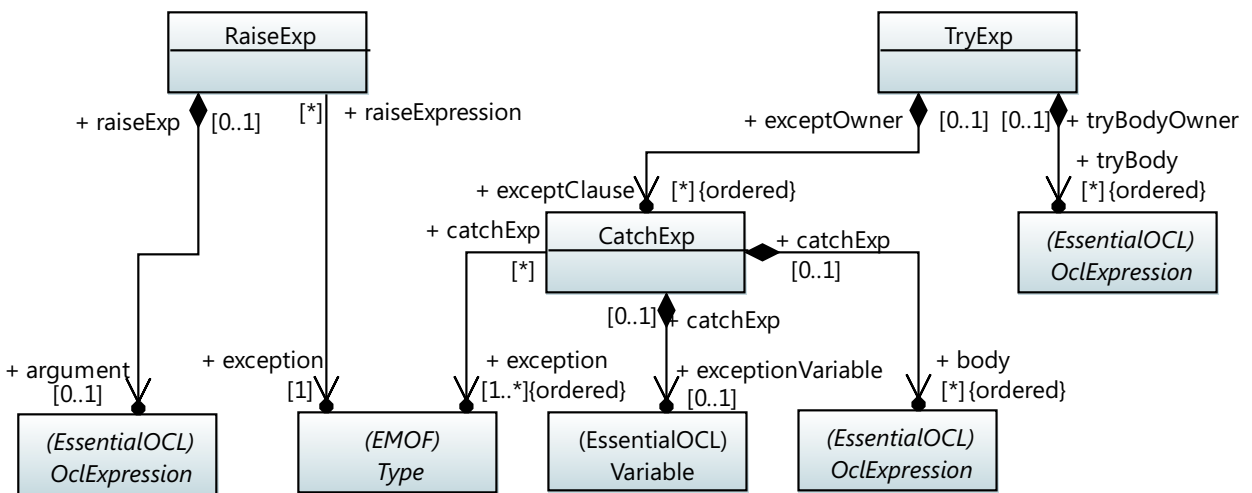


Figure 8.9 - Imperative OCL Package - Exception Expressions

Superclasses

OclExpression

Constraints

Every containment ancestor of an *ImperativeExpression* that is an *OclExpression* must also be an *ImperativeExpression*.

```

context ImperativeExpression
inv IsInImperativeContext: let ancestors = self->closure(oclContainer())
in ancestors->forall(oclIsKindOf(OclExpression) implies oclIsKindOf(ImperativeExpression))
  
```

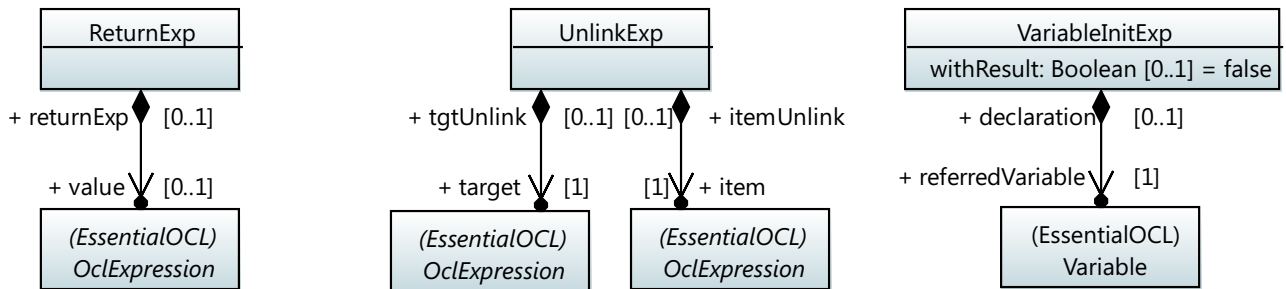


Figure 8.10 - Imperative OCL Package - Return Unlink and VariableInit Expressions

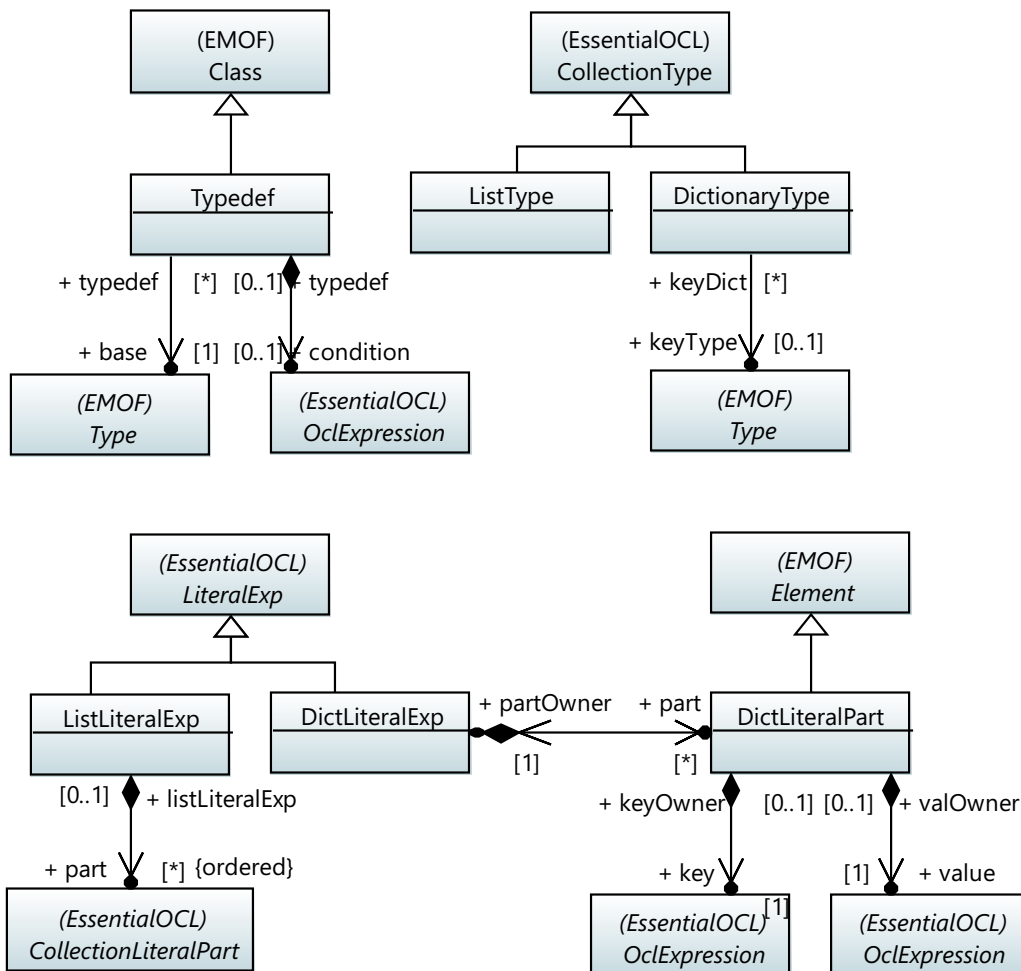


Figure 8.11 - Imperative OCL Package - Type extensions

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 with other constructs like *if* and *loop* expressions.

The block creates a new scope, 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..*] {composes, ordered}

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 returns 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 a 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 : Variable [1] {composes}

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

body : OclExpression [1] {composes}

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]{composes}`

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

`body : OclExpression [1] {composes}`

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, 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 such as *forEach*, *forOne*, *xcollectselect*, and *xcollectselectOne*.

Superclasses

LoopExp

ImperativeExpression

Associations

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

An optional additional Boolean condition to be evaluated to filter the potential results. 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] {composes}(from ImperativeLoopExp)
The condition restricting the elements in the collection for which the body is executed.

/body : OclExpression [1] {composes} (from LoopExp)
The body to execute at each iteration.

/iterator : Variable [1..*] {composes,ordered}(from LoopExp)
The iterator variables defined for this loop.

/source : OclExpression [1] {composes} (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*.

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

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

T represents the type of the source element.

Collection(T) is any collection type. When applying the for expression, if the source collection is not ordered it is implicitly converted into the corresponding ordered collection (Set and Bag become respectively OrderedSet and Sequence).

Notation

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

where <for-name> is the name of the loop construct - 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 within a compute expression
list->forOne(i|i.isKindOf(Actor)) { ... }
```

When using a foreach expression in conjunction with a compute expression the following shorthand can be used:

```
mylist->forEach(i;x:X=...|cond) { ... }
```

Which is equivalent to:

```
compute (x:X=...) mylist->forEach(i|cond) { ... }
```

This is similar to the shorthand notation for while expression (see Clause 8.2.2.4).

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 five pre-defined variants named *xcollect*, *xcollectselect*, *xcollectselectOne*, *xselect*, and *xselectOne*. 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. Also a specific implicit type casting rule applies depending on the condition expression that is associated with the iteration (see implicit type casting rules sub-clause).

A *xcollectselect* 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 clause 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] {composes}
```

The variable that holds the value computed in one iteration.

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

The condition restricting the values being collected.

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

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

```
/iterator : Variable [1..*]{composes, ordered} (from LoopExp)
```

The iterator variables defined for this loop.

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

The source collection.

Type re-casting

The type of the sequence or the object returned by the ImperativeIterateExp construct depends on the usage of the ‘condition’ expression: if the condition is an instance of TypeExp, the condition is firstly re-interpreted as a Boolean expression of the form ‘oclIsKind(TypeExp).’ Additionally, the returned sequence (resp. the returned single object) is re-casted as a sequence of the type denoted in the type expression (resp. as an instance of the denoted type). If the ‘condition’ expression is not used or is not a TypeExp instance no implicit re-casting semantic applies.

Example:

```
self.mysequence[MyType]
  // the type of this expression is a Sequence of 'MyType'

self.mysequence[oclIsKind(MyType) and name=="foo"]
  //the type is the type of the self.mysequence source expression
```

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*.

```
Collection(T)::xcollect(BODY) : BagOrSequence(TT) =
  compute (res : List(TT) := List{ }) {
    self->forEach(COLLECTOR) {
      BODY->flatten()->forEach(target) {
        if (target <> null) res += target;
      };
    };
  }->asBagOrSequence();

Collection(T)::xselect(CONDITION) : BagOrOrderedSetOrSequenceOrSet(T) =
  compute (res : List(T) := List{ }) {
    self->forEach(SELECTOR) {
      if (SELECTOR <> null and CONDITION) res += SELECTOR;
    };
  }->asBagOrOrderedSetOrSequenceOrSet();

Collection(T)::xselectOne(CONDITION) : T =
  compute (res : T := null) {
    self->forEach(SELECTOR) {
      if (SELECTOR <> null and CONDITION) { res := SELECTOR; break; }
    };
  };

Collection(T)::xcollectselect(BODY, CONDITION) : BagOrSequence(TT) =
  compute (res : List(TT) := List{ }) {
    self->forEach(COLLECTOR) {
      BODY->flatten()->forEach(SELECTOR) {
        if (SELECTOR <> null and CONDITION) { res += SELECTOR; }
      };
    };
  }->asBagOrSequence();
```

```

Collection(T)::xcollectselectOne(BODY, CONDITION) : TT =
  compute (res : TT := null) {
    self->forEach(COLLECTOR) {
      BODY->flatten()->forEach(SELECTOR) {
        if (SELECTOR <> null and CONDITION) { res := SELECTOR; break; }
      };
      if (res <> null) { break; }
    };
  };

```

where

- *BagOrOrderedSetOrSequenceOrSet* denotes the Bag, OrderedSet, Sequence or Set kind of the source Collection.
- *BagOrSequence* denotes either Bag or Sequence according to whether the source Collection is unordered or ordered.
- *BODY* is a *TT*-valued Imperative OCL expression that may use the *COLLECTOR* variable.
- *CONDITION* is a Boolean-valued Imperative OCL expression that may use the *SELECTOR* variable.

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 - for instance *xcollectselect* and *xcollectselectOne*.

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

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

```
list->propertyname[res | condition]
// represents list->xcollectselect(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 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 iterator 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 apply 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->xcollectselect(i; res= i.foo() | res.startswith("_"))
```

These shorthand conventions apply also to the *xcollectselectOne* and *xselectOne* variants, except a “!” symbol should prefix the brackets used to render the condition.

```
list->prop![startswith("_")] ; // calling xcollectselectOne(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) that 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 [*] {composes,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 : OclExpression [0..1] {composes}
```

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, using a Java-like notation instead of OCL-like. The notation pattern is:

```
if (cond1) exp1
elif (cond2) exp2,
...
else expN
endif;
```

The **endif** keyword may be skipped. It is needed only when the expression composed with other expressions.

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

When the **switch** keyword is used, the “imperative 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 ...” may produce a grammar conflict in parsers which can, however, be solved through appropriate 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, it offers a more direct representation of decisions written within a switch expression.

Superclasses

ImperativeExpression

Associations

condition : OclExpression [1] {composes}

The condition to evaluate.

body : OclExpression [1] {composes}

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 of a variable with an optional initialization value. This expression may either return the initialization value or return null depending on the return mode being used (see `withResult` property).

Initialization of a multiple valued *Variable* uses the value of the variable's initializer without conversion or modification. This is unlike an *AssignExp* which flattens multiple values and optionally replaces nulls. The initializer must therefore conform to any explicit *Variable* type.

Superclasses

ImperativeExpression

Attributes

withResult : Boolean

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

Associations

referredVariable : Variable [1] {composes}

The variable being declared. The variable is visible within the current scope. It starts to be visible after the declaration occurs.

Notation

The notation uses the **var** keyword. The initialization value is notated using `:=` if `withResult` property is false or uses `::=` if `withResult` 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 omit an initialization value. In this case a default value is assumed (an empty collection for a collection, zero for a numeric type, the empty string for a string, and null for all other elements).

Multiple variable declarations 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* assigns or appends one or more right hand side values to a left hand side Variable or to a Property.

A simple single valued assignment assigns the single value RHS to the LHS, optionally replacing a null value by a *defaultValue*.

A simple multiple valued assignment assigns or appends the multiple flattened RHS values to the LHS optionally replacing null values by a *defaultValue*. In the case of an assignment to a Property any residual null values are omitted.

A *complex assignment* is equivalent to a simple assignment comprising a Sequence of the compound expression values.

A *deferred assignment* is an assignment in which one or more of the RHS expressions involves a deferred resolve expression (see *ResolveExp*). The entire assignment is deferred until the late resolution has been performed. Any premature access to the yet-to-be-assigned Variable or Property yields a null value.

For all assignments, the type of the LHS is unchanged by the assignment.

The return value of an *AssignExp* is the value of the RHS in the following equivalent assignments that clarify the various possibilities.

append is one of the following operations:

Bag::including, List::append, OrderedSet::append, Sequence::append, Set::including,

defaultValue is the *AssignExp::defaultValue*

late denotes the **late** keyword for a deferred assignment or nothing otherwise.

left denotes the *AssignExp::left* property identifying the LHS Variable or Property

LEFT denotes the type of *left* such as **Set(String)**

value is the *AssignExp::value* property identifying the RHS expression value or values

Equivalent Single valued Property or Variable assign: *left := value*

```
left := late if value->at(1) <> null
    then value->at(1)
    else defaultValue
endif
```

Equivalent Multiple valued Property assign: *left := value*

```
left := late value->flatten()->iterate(c; acc :LEFT = LEFT{ } |
    let v = if c <> null then c else defaultValue endif
    in if v <> null then acc->append(v) else acc endif)
```

Equivalent Multiple valued Variable assign: *left := value*

```
left := late value->flatten()->iterate(c; acc LEFT = left |
    let v = if c <> null then c else defaultValue endif
    in acc->append(v))
```

Equivalent Single valued Property or Variable append: *left += value*

invalid – append is not possible for single values.

Equivalent Multiple valued Property append: *left += value*

```
left := late value->flatten()->iterate(c; acc : LEFT = left |
  let v = if c <> null then c else defaultValue endif
  in if v <> null then acc->append(v) else acc endif)
```

Equivalent Multiple valued Variable append: *left += value*

```
left := late value->flatten()->iterate(c; acc : LEFT = left |
  let v = if c <> null then c else defaultValue endif
  in acc->append(v))
```

Superclasses

ImperativeExpression

Attributes

isReset : Boolean

Indicates, for a multivalued target, that the list is reset before executing the assignment.

Associations

value : OclExpression [*] {composes, ordered}

The expression to be evaluated in order to assign the variable or the property.

left : OclExpression [1] {composes}

The left hand side expression of the assignment. Should reference a variable or a property that can be updated.

defaultValue : OclExpression [0..1] {composes}

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 an object from a multivalued property link.

The residence of the removed object is unaffected. If the unlink affects a containment relationship, the no longer contained object becomes a root object of its model. If total removal is required the Model::removeElement() operation may be used.

Superclasses

ImperativeExpression

Associations

item : [1] OclExpression {composes}

The object to be removed from the multivalued property.

target : [1] OclExpression {composes}

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 an exception occurs.

The `exceptClauses` are searched in order to select the first `exceptClause` that provides an exception type to which the raised exception conforms. If an `exceptClause` is selected, its body is executed.

A nested exception within an `exceptClause` terminates the `exceptClause` unless caught by a nested `TryExp`.

Superclasses

ImperativeExpression

Associations

tryBody : OclExpression [*] {composes,ordered}

The expression being under the control of exception handling.

exceptClause : CatchExp [*] {composes,ordered}

The exception clauses providing the code to execute in case of failure.

Notation

The notation uses the **try** and **except** keywords.

```
try { expression1 } except (exception1,exception2) {expression2};
```

8.2.2.14 CatchExp

A *catch expression* represents the code to be executed when an exception matching a given list of exception types is fired during the execution of the containing try expression.

The caught expression may be accessed in the body expression using the `exceptionVariable` whose apparent (static) type is the most derived common super type of all catchable exception types.

Superclasses

ImperativeExpression

Associations

exception: Type [+] {ordered}

The list of exceptions being treated by this catch handler.

body: OclExpression [*] {composes,ordered}

The list of expressions to execute in sequence.

exceptionVariable: Variable [0..1]

The variable through which the caught exception may be accessed.

Notation

The notation uses the **except** keyword with the list of exception types in parenthesis and the body in braces.

```
except (exception1,exception2) {expression2};
```

8.2.2.15 RaiseExp

A *raise expression* is an expression that produces an exception.

Superclasses

ImperativeExpression

Associations

exception: Type [1]

The exception being raised.

argument: OclExpression [0..1] {composes}

The argument accompanying the raised exception.

Notation

The notation uses the **raise** keyword followed by the exception type name and arguments for one of the expression type name constructors.

```
myproperty := self.something default raise StringException("ProblemHere");
```

The exceptions can be provided as simple strings. This is a shorthand for raising a `StringException` with the string as the constructor argument

```
myproperty := self.something default raise "ProblemHere";
```

8.2.2.16 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 return parameter, the result is a tuple with a tuple part for each return parameter. When the return expression value is omitted, this tuple is created automatically from the assignable result parameters. Alternatively the return expression value may be an explicitly constructed tuple with one part for each return parameter.

Superclasses

ImperativeExpression

Associations

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

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;
```

8.2.2.17 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, like the side-effect free OCL iterate expression.

Superclasses

ImperativeExpression

Notation

The notation uses the **break** keyword alone.

8.2.2.18 ContinueExp

Within an iteration over a list of expressions, a *continue expression* is used 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 *continue expression* cannot be directly owned by a non-imperative expression.

Superclasses

ImperativeExpression

Notation

The notation uses the **continue** keyword alone.

8.2.2.19 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 is a kind of operation call expression where the first argument contains the message to be print, the second argument gives the model element to be print (using an implicit call to the ‘repr’ operation from the QVT Standard Library) , and the third argument gives a level number for the log. Only the first argument is mandatory.

A log expression returns null.

Superclasses

OperationCallExp

ImperativeExpression

Associations

condition: OclExpression [0..1] {composes}

An optional condition to check. No log record is produced if it evaluates to false.

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;
```

8.2.2.20 AssertExp

An *assert expression* is an expression that checks whether a condition holds. If the assertion fails, an error message is generated - 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 [1]{composes}

The condition to check.

log: LogExp [0..1] {composes}

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** identifiers - 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);
```

8.2.2.21 SeverityKind

The *severity kind* enumeration defines all possible levels of severity for errors raised by assertion expressions.

Enumeration values

warning

error

fatal

8.2.2.22 InstantiationExp

An *instantiation expression* creates an instance of a class, invokes an initialization operation on the created object, and returns the created object. The initialization operation is either implicit, either explicitly given and has necessarily the name of the class. By default, an initialization operation with no arguments exists for all classes.

An instantiation expression may indicate the MOF extent, represented by a variable, where the created instance will “live.”

Superclasses

ImperativeExpression

Associations

argument : OclExpression [*] {composes, ordered}

The arguments of the instantiation expression. Should correspond with the initialization operation for the class (which by default is implicit and has no arguments).

extent : Variable [0..1]

The extent on which the new object is created.

instantiatedClass : Class [1]

The type of the object to be created.

initializationOperation : Operation [0..1]

The initialization operation that uses the arguments to initialize the object after creation. The initialization operation may be omitted when implicit initialization occurs with no arguments.

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 an explicit extent is provided, the variable name postfixes the type of the instantiation expression using the “@” separator symbol.

```
column := new Column@mymodel(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(a.name,a.type.name);  
// equivalent code:  
// column := self.attribute->forEach { new(a) Column(a.name,a.type.name); }
```

Type Extensions

This clause defines the extensions to the type systems as well as the literals that are associated with the new types. Figure 8.11 depicts the four new types, which are: TemplateParameterType, Typedef(deprecated), DictionaryType, and ListType. The DictionaryType and ListType are two mutable collection types.

8.2.2.23 Typedef (deprecated)

The *Typedef* class was underspecified and unnecessary.

A type alias may be defined by using a typedef construct.

A Constrained Type may be defined by adding an invariant to a derived class.

Superclasses

Class

Associations

```
base : Type [1]
condition : OclExpression [0..1] {composes}
```

8.2.2.24 ListType

A *list type* is a mutable parameterized collection type. A value conforming to this type contains an ordered sequence of values. When a type for the element type is not provided Object is assumed.

A list of pre-defined operations on list is given in the QVT standard library.

Superclasses

CollectionType

Notation

An initialized list literal may be created in the same way as an initialized sequence literal.

```
List{1,2,3}      List{1..10,12,14..16}
```

Implementation Note

ListLiteralExp will become obsolete once OCL eliminates the prohibition on extension of CollectionLiteralExp imposed by the redundant CollectionLiteralExp::kind attribute.

8.2.2.25 DictionaryType

A *dictionary type* is a mutable type representing a 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 - 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 DictLiteralExp).

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 Object.
```

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 - 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 are assumed (String and Object).

8.2.2.26 TemplateParameterType

A *template parameter type* is used to refer to generic types in parameterized definitions. It is specifically used when 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 a 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.

8.2.2.27 DictLiteralExp

A *dictionary literal expression* is a literal definition for a dictionary. Each data stored in the dictionary consists of a key and a value (see DictLiteralPart).

Superclasses

LiteralExp

Associations

```
part : DictLiteralPart [*] {composes}
```

The parts contained by this dictionary.

Notation

A dictionary literal expression is notated using the **Dict** type specifier followed by curly braces. The “=” symbol separates

```
var dic = Dict{'E' = 'EXPLICIT', 'I' = 'IMPLICIT'};
```

8.2.2.28 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 element.

8.2.2.29 ListLiteralExp

A *list literal expression* is a literal definition for a mutable list type (see 8.2.2.25).

Superclasses

LiteralExp (From EssentialOCL)

Associations

part : CollectionLiteralPart [*] {composes,ordered}

The values of the literal list.

8.3 Standard Library

This clause describes the additions to the OCL standard library to form the QVT Operational Mappings Library, which is an instance of the Library metaclass. This library is named `StdLib` and is imported implicitly 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). MOF terminology is adopted here.

`Object` is an M1 type representing all types, including data types. The M1 `Object` type is an instance of the `MOF::Element` metatype.

`Element` is an M1 type and 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)` metatype, the type parameter for the parameterized type `List(T)` is `Object`.

The clauses 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.

A `transform()` operation on a transformation returns a `Status` object.

8.3.1.4 Exception

This M1 class named `Exception` represents the base class for all exceptions. The M1 `Exception` type is an instance of the `Class` metatype.

8.3.1.5 StringException

The `StringException` supports the simple string-valued shorthand of a `RaiseExp`.

Superclasses

`Exception`

Constructor

```
StringException(reason : String)
```

Attributes

```
reason : String [1]
```

The reason provided on construction.

8.3.1.6 AssertionFailed

The `AssertionFailed` exception supports a fatal `AssertExp`.

Superclasses

`Exception`

Constructor

```
AssertionFailed(reason : LogExp)
```

Attributes

```
reason : LogExp [1]
```

The reason provided on construction.

8.3.2 Synonym types and synonym operations

QVT 1.0, 1.1 and 1.2 encouraged the use of the simpler "asType", "isKindOf" and "isTypeOf" synonyms for "ocl"-prefixed operation names such as "oclAsType", "oclisKindOf" and "oclisTypeOf" . Use of "Any" and "Void" synonyms were similarly encouraged in preference to "Ocl"-prefix types such as "OclAny" and "OclVoid".

Since these synonyms may introduce conflicts and since few QVTo users can avoid also being OCL users, the use of these synonyms is deprecated. A QVTo implementation may support these deprecated synonyms by translating them to their prefixed form before using them in any Abstract Syntax representation. Any conflicting usage of the synonyms must ignore the synonym in favor of the user declaration.

8.3.3 Operations on objects

The `Object` type represents all types.

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, 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 `Element::getMetaClass()` method and retrieving the name.

8.3.4.4 subobjects

```
Element::subobjects() : Set(Element)
```

Returns all immediate sub objects of an object.

8.3.4.5 allSubobjects

```
Element::allSubobjects() : Set(Element)
```

Returns iteratively all sub objects of an object.

8.3.4.6 subobjectsOfType

`Element::subobjectsOfType(type : Classifier) : Set(T)`

Same as `subobjects` but filters according to the exact type. The returned Set element type *T* is the type specified as *type*.

8.3.4.7 allSubobjectsOfType

`Element::allSubobjectsOfType(type : Classifier) : Set(T)`

Same as `allSubobjects` but filters according to the exact type. The returned Set element type *T* is the type specified as *type*.

8.3.4.8 subobjectsOfKind

`Element::subobjectsOfKind(type : Classifier) : Set(T)`

Same as `subobjects` but filters according to the type. The returned Set element type *T* is the type specified as *type*.

8.3.4.9 allSubobjectsOfKind

`Element::allSubobjectsOfKind(type : Classifier) : Set(T)`

Same as `allSubobjects` but filters according to the type. The returned Set element type *T* is the type specified as *type*.

8.3.4.10 clone

`Element::clone() : T`

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. The returned type *T* is the type of the source element as known at compile time. The returned object is added to the root of the model containing the source element.

8.3.4.11 deepclone

`Element::deepclone() : T`

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. The returned type *T* is the type of the source element as known at compile time. The returned object is added to the root of the model containing the source element.

8.3.4.12 markedAs

`Element::markedAs(value:String) : 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 is used to return 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 is used to check whether an instance is “stereotyped.” The definition depends on the metamodel.

8.3.4.15 stereotypedStrictlyBy

`Element::stereotypedStrictlyBy(String) : Boolean`

Same as `stereotypedBy` except that base stereotypes are not taken into account.

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 set of all objects in the model.

8.3.5.2 objectsOfKind

`Model::objectsOfKind(type : Classifier) : Set(T)`

Returns the list of the objects in the model extent that have the type given. The returned Set element type *T* is the type specified as *type*.

8.3.5.3 objectsOfType

`Model::objectsOfType(type : Classifier) : Set(T)`

Returns the list of the objects in the model extent that have the exact type given. The returned Set element type *T* is the type specified as *type*.

8.3.5.4 rootObjects

`Model::rootObjects() : Set(Element)`

Returns the set of all objects in the model that are not contained by other objects in the model.

8.3.5.5 addElement

`Model::addElement (anObject : Element): Void`

The object is first displaced from any usage by setting its container to null, then the object is added to the model's extent so that it provides another root for the model. Any non-containment references to and from the object are unaffected.

8.3.5.6 removeElement

`Model::removeElement (Element): Void`

Removes an object from the model so that it becomes an orphan. All references to or from the object are eliminated. References from collections are removed. References from non-collections are set to null.

8.3.5.7 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.

8.3.5.8 copy

`Model::copy() : Model`

Creates a deep copy of a model and its extent. All objects transitively contained in the source model are copied into the new model. The roots of the new model are the initial members of the new extent. The new extent has no associated external file and so the new contents may be lost unless assigned to another model or passed to another transformation.

8.3.5.9 createEmptyModel

`static Model::createEmptyModel() : Model`

Creates and initializes a model of this model's model type without any content. This operation is useful when creating intermediate models within a transformation.

8.3.6 Operations on Transformations

In this clause we provide the operations that 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.

8.3.7 Operations on Status

The following operations may be used to interrogate the Status object that synchronizes the end of a transformation.

8.3.7.1 raisedException

```
Status::raisedException () : Exception
```

Returns the exception raised by the transformation execution.

8.3.7.2 failed

```
Status::failed() : Boolean
```

Returns true if the transformation failed, false otherwise.

8.3.7.3 succeeded

```
Status::succeeded() : Boolean
```

Returns true if the transformation succeeded, false otherwise.

8.3.8 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.

8.3.8.1 get

```
Dictionary(KeyT,T)::get (k:KeyT) : T
```

Returns the value associated with the given key. The null value is returned if k is not present. Modifying the returned value modifies the value stored in the dictionary.

8.3.8.2 hasKey

```
Dictionary(KeyT,T)::hasKey (k:KeyT) : Boolean
```

Checks whether the dictionary has a value for the given key.

8.3.8.3 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.

8.3.8.4 put

```
Dictionary(KeyT,T)::put (k:KeyT, v:T) : Void
```

Modifies the dictionary by assigning a value to a key. Modifying the value modifies the value stored in the dictionary.

8.3.8.5 clear

`Dictionary(KeyT,T)::clear() : Void`
Modifies the dictionary by removing all values.

8.3.8.6 size

`Dictionary(KeyT,T)::size() : Integer`
Returns the number of values stored in the dictionary.

8.3.8.7 values

`Dictionary(KeyT,T)::values() : List(T)`
Returns a new list of the values in the dictionary. The order is arbitrary. Modifying the returned list does not modify the contents of the dictionary. Modifying the values in the list modifies the values stored in the dictionary.

8.3.8.8 keys

`Dictionary(KeyT,T)::keys() : List(KeyT)`
Returns a new list of keys to the dictionary. Modifying the returned list does not modify the contents of the dictionary. Modifying the values in the list may modify the keys to the dictionary contents giving unpredictable behavior. If mutable types such as List or Dictionary are used as Dictionary keys, applications should take care to create clones where appropriate.

8.3.8.9 isEmpty

`Dictionary(KeyT,T)::isEmpty() : Boolean`
Returns true if the dictionary is empty, false otherwise.

8.3.9 Operations on Lists

The following operations can be invoked on any list. A list type is a parameterized type. The symbol T denotes the type of the values. The operations include all those of the OCL Sequence type.

8.3.9.1 =

`List(T)::=(s : List(T)) : Boolean`
True if self contains the same elements as s in the same order.
post: result = (size() = s->size()) and
Sequence{1..size()}->forall(i | at(i) = s->at(i))

8.3.9.2 <>

`List(T)::<>(c : List(T)) : Boolean`
True if c is not equal to self.
post: result = not (self = c)

8.3.9.3 add

List(T)::add() : Void

Adds a value at the end of the mutable list.

post: size() = size@pre() + 1

post: Sequence{1..size@pre()}->forall(i | at(i) = at@pre(i))

post: at(size()) = object

8.3.9.4 append

List(T)::append(object: T) : List(T)

Returns a new list of elements, consisting of all elements of self, followed by object.

post: result->size() = size() + 1

post: Sequence{1..size()}->forall(i | result->at(i) = at(i))

post: result->at(result->size()) = object

8.3.9.5 asBag

List(T)::asBag() : Bag(T)

Returns a Bag containing all the elements from self, including duplicates. The element order is indeterminate.

post: result->forall(elem | self->count(elem) = result->count(elem))

post: self->forall(elem | self->count(elem) = result->count(elem))

8.3.9.6 asList

List(T)::asList() : List(T)

Returns a new list that is a shallow clone of self.

post: Sequence{1..size()}->forall(i | result->at(i) = self->at(i))

8.3.9.7 asOrderedSet

List(T)::asOrderedSet() : OrderedSet(T)

Returns an OrderedSet that contains all the elements from self, in the same order, with duplicates removed.

post: result->forall(elem | self ->includes(elem))

post: self->forall(elem | result->count(elem) = 1)

post: self->forall(elem1, elem2 | self->indexOf(elem1) < self->indexOf(elem2)
implies result->indexOf(elem1) < result->indexOf(elem2))

8.3.9.8 asSequence

List(T)::asSequence() : Sequence(T)

Returns a Sequence that contains all the elements from self, in the same order.

post: result->size() = size()

post: Sequence{1..size()}->forall(i | result->at(i) = self->at(i))

8.3.9.9 asSet

List(T)::asSet() : Set(T)

Returns a Set containing all the elements from self, with duplicates removed. The element order is indeterminate.

post: result->forAll(elem | self->includes(elem))

post: self->forAll(elem | result->includes(elem))

8.3.9.10 at

List(T)::at(i : Integer) : T

Returns the element of the list at the i-th one-based index.

pre : 1 <= i and i <= size()

8.3.9.11 clone

List(T)::clone() : List(T)

Returns a new list that is a shallow clone of self.

post: Sequence{1..size()}->forAll(i | result->at(i) = self->at(i))

8.3.9.12 count

List(T)::count(object : T) : Integer

Returns the number of occurrences of object in self.

8.3.9.13 deepclone

List(T)::deepclone() : List(T)

Returns a new list that is a deep clone of self; that is a new list in which each element is in turn a deepclone of the corresponding element of self.

8.3.9.14 excludes

List(T)::excludes(object : T) : Boolean

True if *object* is not an element of *self*, false otherwise.

post: result = (self->count(object) = 0)

8.3.9.15 excludesAll

List(T)::excludesAll(c2 : Collection(T)) : Boolean

Does *self* contain none of the elements of *c2* ?

post: result = c2->forAll(elem | self->excludes(elem))

8.3.9.16 excludesAll

List(T)::excludesAll(c2 : List(T)) : Boolean

Does *self* contain none of the elements of *c2* ?

post: result = c2->forAll(elem | self->excludes(elem))

8.3.9.17 excluding

```
List(T)::excluding(object : T) : List(T)
```

Returns a new list containing all elements of *self* apart from all occurrences of *object*. The order of the remaining elements is not changed.

```
post: result->includes(object) = false
post: result->size() = self->size()@pre - self->count(object)@pre
post: result = self->iterate(elem; acc : List(T) = List{} |
    if elem = object then acc else acc->append(elem) endif )
```

8.3.9.18 first

```
List(T)::first() : T
```

Returns the first element in *self*.

```
post: result = at(1)
```

8.3.9.19 flatten

```
List(T)::flatten() : List(T2)
```

Returns a new list containing the recursively flattened contents of the old list. The order of the elements is partial.

```
post: result = self->iterate(c; acc : List(T2) = List{} |
    if c.ocltType().elementType.oclIsKindOf(CollectionType)
    then acc->union(c->flatten()->asList())
    else acc->union(c)
    endif)
```

8.3.9.20 includes

```
List(T)::includes(object : T) : Boolean
```

True if *object* is an element of *self*, false otherwise.

```
post: result = (self->count(object) > 0)
```

8.3.9.21 includesAll

```
List(T)::includesAll(c2 : Collection(T)) : Boolean
```

Does *self* contain all the elements of *c2*?

```
post: result = c2->forAll(elem | self->includes(elem))
```

8.3.9.22 includesAll

```
List(T)::includesAll(c2 : List(T)) : Boolean
```

Does *self* contain all the elements of *c2*?

```
post: result = c2->forAll(elem | self->includes(elem))
```

8.3.9.23 including

```
List(T)::including(object : T) : List(T)
```

Returns a new list containing all elements of *self* plus *object* added as the last element.

```
post: result = append(object)
```


8.3.9.24 indexOf

```
List(T)::indexOf(obj : T) : Integer
```

The one-based index of object obj in the list.

```
pre : includes(obj)
```

```
post : at(result) = obj
```

8.3.9.25 insertAt

```
List(T)::insertAt(index : Integer, object : T) : List(T)
```

Returns a new list consisting of self with object inserted at the one-based position index.

```
pre : 1 <= index and index <= size()
```

```
post: result->size() = size() + 1
```

```
post: Sequence{1..(index - 1)}->forall(i | result->at(i) = at(i))
```

```
post: result->at(index) = object
```

```
post: Sequence{(index + 1)..size()}->forall(i | result->at(i + 1) = at(i))
```

8.3.9.26 insertAt

```
List(T)::insertAt(object : T, index : Integer) : Void
```

The list is modified to consist of self with object inserted at the one-based position index.

```
pre : 1 <= index and index <= size()
```

```
post: size() = size@pre() + 1
```

```
post: Sequence{1..(index - 1)}->forall(i | at(i) = at@pre(i))
```

```
post: at(index) = object
```

```
post: Sequence{(index + 1)..size()}->forall(i | at(i) = at@pre(i - 1))
```

8.3.9.27 isEmpty

```
List(T)::isEmpty() : Boolean
```

Is *self* an empty list?

```
post: result = (self->size() = 0)
```

8.3.9.28 joinfields

```
List(T)::joinfields(sep:String,begin:String,end:String) :String
```

Creates a string separated by sep and delimited with begin and end strings.

```
post: result = begin + Sequence{1..size()}->iterate(i; acc : String = '' |  
  acc + if i = 1 then '' else sep endif + at(i).toString()  
  + end
```

8.3.9.29 last

```
List(T)::last() : T
```

Returns the last element in self.

```
post: result = at(size())
```

8.3.9.30 max

List(T)::max() : T

The element with the maximum value of all elements in self. Elements must be of a type supporting the max operation. The max operation - supported by the elements - must take one parameter of type T. Integer and Real fulfill this condition.

post: result = self->iterate(elem; acc : T = self->any(true) | acc.max(elem))

8.3.9.31 min

List(T)::min() : T

The element with the minimum value of all elements in self. Elements must be of a type supporting the min operation. The min operation - supported by the elements - must take one parameter of type T. Integer and Real fulfill this condition.

post: result = self->iterate(elem; acc : T = self->any(true) | acc.min(elem))

8.3.9.32 notEmpty

List(T)::notEmpty() : Boolean

Is *self* not an empty list?

post: result = (self->size() <> 0)

8.3.9.33 prepend

List(T)::prepend(object : T) : List(T)

Returns a new list consisting of object, followed by all elements in self.

post: result->size = size() + 1

post: result->at(1) = object

post: Sequence{1..size()->forall(i | result->at(i + 1) = at(i))

8.3.9.34 product

List(T)::product(c2: Collection(T2)) : Set(Tuple(first: T, second: T2))

The cartesian product operation of *self* and *c2*.

post: result = self->iterate(e1; acc: Set(Tuple(first: T, second: T2)) = Set{
|
| c2->iterate(e2; acc2: Set(Tuple(first: T, second: T2)) = acc |
| acc2->including(Tuple{first = e1, second = e2}))

8.3.9.35 remove

List(T)::remove(element : T) : Void

Removes .all elements from self equal to element.

post: result = self@pre->reject(e = element)

8.3.9.36 removeAll

```
List(T)::removeAll(elements : Collection(T)) : Void
```

Removes .all elements from self equal to any of elements.

```
post: result = self@pre->reject(e | elements->includes(e))
```

8.3.9.37 removeAll

```
List(T)::removeAll(elements : List(T)) : Void
```

Removes .all elements from self equal to any of elements.

```
post: result = self@pre->reject(e | elements->includes(e))
```

8.3.9.38 removeAt

```
List(T)::removeAt(index : Integer) : T
```

Removes .and returns .the list element at index. Returns invalid for an invalid index.

```
pre: 1 <= index and index <= size()
```

```
post: size() = size@pre() - 1
```

```
post: Sequence{1..index}->forall(i | at(i) = at@pre(i))
```

```
post: Sequence{(index+1)..size()->forall(i | at(i) = at@pre(i+1))
```

```
post: result = at@pre(index)
```

8.3.9.39 removeFirst

```
List(T)::removeFirst() : T
```

Removes .and returns .the first list element. Returns invalid for an empty list.

```
pre: 1 <= size()
```

```
post: size() = size@pre() - 1
```

```
post: Sequence{1..size()->forall(i | at(i) = at@pre(i+1))
```

```
post: result = at@pre(1)
```

8.3.9.40 removeLast

```
List(T)::removeLast() : T
```

Removes .and returns .the last list element. Returns invalid for an empty list.

```
pre: 1 <= size()
```

```
post: size() = size@pre - 1
```

```
post: Sequence{1..size()->forall(i | at(i) = at@pre(i))
```

```
post: result = at@pre(size@pre())
```

8.3.9.41 reverse

```
List(T)::reverse() : List(T)
```

Returns a new list containing the same elements but with the opposite order.

```
post: result->size() = self->size()
```

```
post: Sequence{1..size()->forall(i | result->at(i) = at(size() - (i-1)))
```

8.3.9.42 selectByKind

```
List(T)::selectByKind(type : Classifier) : List(T1)
```

Returns a new list containing the non-null elements of self whose type is *type* or a subtype of *type*. The returned list element type T1 is the type specified as *type*.

```
post: result = self
      ->collect(if oclIsKindOf(type) then oclAsType(type) else null endif)
      ->excluding(null)
```

8.3.9.43 selectByType

```
List(T)::selectByType(type : Classifier) : List(T1)
```

Returns a new list containing the non-null elements of self whose type is *type* but which are not a subtype of *type*. The returned list element type T1 is the type specified as *type*.

```
post: result = self
      ->collect(if oclIsTypeOf(type) then oclAsType(type) else null endif)
      ->excluding(null)
```

8.3.9.44 size

```
List(T)::size() : Integer
```

The number of elements in the collection *self*.

```
post: result = self->iterate(elem; acc : Integer = 0 | acc + 1)
```

8.3.9.45 subSequence

```
List(T)::subSequence(lower : Integer, upper : Integer) : Sequence(T)
```

Returns a new sub-List of self starting at number lower, up to and including element number upper.

```
pre : 1 <= lower and lower <= upper and upper <= size()
```

```
post: result->size() = upper - lower + 1
```

```
post: Sequence{lower..upper}->forall(i | result->at(i - lower + 1) = at(i))
```

8.3.9.46 sum

```
List(T)::sum() : T
```

The addition of all elements in *self*. Elements must be of a type supporting the + operation. The + operation must take one parameter of type T. It does not need to be commutative or associative since the iteration order over a list is well-defined. Integer and Real fulfill this condition.

```
post: result = self->iterate(elem; acc : T = 0 | acc + elem)
```

8.3.9.47 union

```
List(T)::union (s : List(T)) : List(T)
```

Returns a new list consisting of all elements in self, followed by all elements in s.

```
post: result->size() = size() + s->size()
```

```
post: Sequence{1..size()->forall(i | result->at(i) = at(i))
```

```
post: Sequence{1..s->size()->forall(i | result->at(i + size()) = s->at(i))
```

8.3.10 Iterations on Lists

There are no iterations defined for Lists since Lists are mutable and iteration domains are immutable. However the iterations defined for Sequences may be used without explicitly converting the List to a Sequence.

Invocation of one the following list iterations returning non-Lists

```
aList->iteration(...)
```

is therefore shorthand for

```
aList->asSequence()->iteration(...)  
List(T)::any(i : T[?]) : T[?]  
List(T)::collect(i : T[?]) : Collection(T1)  
List(T)::collectNested(i : T[?]) : Collection(T1)  
List(T)::exists(i : T[?]) : Boolean[?]  
List(T)::exists(i : T[?], j : T[?]) : Boolean[?]  
List(T)::forall(i : T[?]) : Boolean[?]  
List(T)::forall(i : T[?], j : T[?]) : Boolean[?]  
List(T)::isUnique(i : T[?]) : Boolean  
List(T)::iterate(i : T[?]; acc : T2[?]) : T2[?]  
List(T)::one(i : T[?]) : Boolean
```

Invocation of one the following list iterations returning Lists

```
aList->iteration(...)
```

is shorthand for

```
aList->asSequence()->iteration(...)->asList()  
List(T)::reject(i : T[?]) : List(T)  
List(T)::select(i : T[?]) : List(T)  
List(T)::sortedBy(i : T[?]) : List(T)
```

8.3.11 Operations on Collections

The following operations are added to the standard OCL collections.

8.3.11.1 asList

```
Collection(T)::asList() : List(T)
```

Returns a new list containing all the elements of a collection. Whether the order is determinate depends on the derived collection type.

```
post: result->size() = size()
```

```
post: self->asSet()->forall(e | result->count(e) = self->count(e))
```

8.3.11.2 clone

```
Collection(T)::clone() : Collection(T)
```

Collections are immutable so a clone returns self.

```
post: result = self
```

8.3.11.3 deepclone

`Collection(T)::deepclone() : Collection(T)`

Returns a collection that is a deep clone of self; that is a collection in which each element is in turn a deepclone of the corresponding element of self. May return self if there is no deep mutable content.

8.3.12 Operations on Bags

The following operations are added to the standard OCL bags.

8.3.12.1 asList

`Bag(T)::asList() : List(T)`

Returns a new list containing all the elements of a bag in an indeterminate order.

8.3.12.2 clone

`Bag(T)::clone() : Bag(T)`

Bags are immutable so a clone returns self.

8.3.12.3 deepclone

`Bag(T)::deepclone() : Bag(T)`

Returns a Bag that is a deep clone of self; that is a Bag in which each element is in turn a deepclone of the corresponding element of self. May return self if there is no deep mutable content.

8.3.13 Operations on OrderedSets

The following operations are added to the standard OCL ordered sets.

8.3.13.1 asList

`OrderedSet(T)::asList() : List(T)`

Returns a new list that contains all the elements an ordered set in the same order.

post: `Sequence{1..size()}->forall(i | result->at(i) = self->at(i))`

8.3.13.2 clone

`OrderedSet(T)::clone() : OrderedSet(T)`

OrderedSets are immutable so a clone returns self.

8.3.13.3 deepclone

`OrderedSet(T)::deepclone() : OrderedSet(T)`

Returns an OrderedSet that is a deep clone of self; that is an OrderedSet in which each element is in turn a deepclone of the corresponding element of self. May return self if there is no deep mutable content.

8.3.14 Operations on Sequences

The following operations are added to the standard OCL sequences.

8.3.14.1 asList

`Sequence(T)::asList() : List(T)`

Returns a new list that contains all the elements a sequence in the same order.

post: `Sequence{1..size()}->forall(i | result->at(i) = self->at(i))`

8.3.14.2 clone

`Sequence(T)::clone() : Sequence(T)`

Sequences are immutable so a clone returns self.

8.3.14.3 deepclone

`Sequence(T)::deepclone() : Sequence(T)`

Returns a Sequence that is a deep clone of self; that is a Sequence in which each element is in turn a deepclone of the corresponding element of self. May return self if there is no deep mutable content.

8.3.15 Operations on Sets

The following operations are added to the standard OCL sets.

8.3.15.1 asList

`Set(T)::asList() : List(T)`

Returns a new list containing all the elements of a set in an indeterminate order.

8.3.15.2 clone

`Set(T)::clone() : Set(T)`

Sets are immutable so a clone returns self.

8.3.15.3 deepclone

`Set(T)::deepclone() : Set(T)`

Returns a Set that is a deep clone of self; that is a Set in which each element is in turn a deepclone of the corresponding element of self. May return self if there is no deep mutable content.

8.3.16 Operations on Strings

All string operations defined in OCL 2.4 are available. An OCL String is immutable and so there are no operations that modify strings. In addition there is:

8.3.16.1 format

`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 `Object::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.

8.3.16.2 length

`String::length () : Integer`

The length operation returns the length of the sequence of characters represented by the object at hand.

This is a synonym of the OCL `String::size()` operation. It is therefore deprecated.

8.3.16.3 substringBefore

`String::substringBefore (match : String) : String`

retrieves the substring that is before the matched string.

8.3.16.4 substringAfter

`String::substringAfter (match : String) : String`

retrieves the substring that is after the matched string.

8.3.16.5 toLower

`String::toLowerCase () : String`

Converts all of the characters in this string to lowercase characters.

This is a synonym of the OCL `String::toLowerCase()` operation. It is therefore deprecated.

8.3.16.6 toUpper

`String::toUpperCase () : String`

Converts all of the characters in this string to uppercase characters.

This is a synonym of the OCL `String::toUpperCase()` operation. It is therefore deprecated.

8.3.16.7 firstToUpper

`String::firstToUpper () : String`

Converts the first character in the string to an uppercase character.

8.3.16.8 lastToUpper

`String::lastToUpper () : String`

Converts the last character in the string to an uppercase character.

8.3.16.9 indexOf

```
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.

8.3.16.10 endsWith

```
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.

8.3.16.11 startsWith

```
String::startsWith (match : String) : Boolean
```

Returns true if the string at hand starts with the substring provided in the parameter.

Parameters

. match: the prefix to be searched for.

8.3.16.12 trim

```
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.

8.3.16.13 normalizeSpace

```
String::normalizeSpace() : String
```

Removes all trailing and leading white space and replaces all internal sequences of white space with a single space.

8.3.16.14 replace

```
String::replace (m1:String, m2:String): String
```

All occurrences of m1 in the context string are replaced by m2. The operation returns a new string.

8.3.16.15 match

```
String::match (matchpattern:String) : Boolean
```

Returns true if the value of the String matches the regular expression, else return false. The syntax of regular expression is the syntax or regular expression in Java language.

8.3.16.16 equalsIgnoreCase

```
String::equalsIgnoreCase (match:String) : Boolean
```

Returns true if the value of String, when ignoring letter casing matches the input String “match,” else returns false.

8.3.16.17 find

```
String::find (match:String) : Integer
```

Returns the position of the substring that starts with ‘match.’

8.3.16.18 rfind

```
String::rfind (match:String) : Integer
```

Returns the position of the substring that starts with ‘match.’ The search starts from the right.

8.3.16.19 isQuoted

```
String::isQuoted (s:String) : Boolean
```

Returns true if the string starts and ends with the “s” string.

8.3.16.20 quotify

```
String::quotify (s:String) : String
```

Adds the “s” string at the beginning and the end of the strings and returns it.

8.3.16.21 unquotify

```
String::unquotify (s:String) : String
```

Removes the ‘s’ string at the beginning and at the end of the string and returns the resulting string. If the string ‘s’ does not appear at the beginning or at the end, the content of source string is returned.

8.3.16.22 matchBoolean

```
String::matchBoolean() : Boolean
```

Returns true if the string is “true,” “false,” “0,” or “1.” The method is not case sensitive.

8.3.16.23 matchInteger

```
String::matchInteger() : Boolean
```

Returns true if the string represents an integer.

8.3.16.24 matchFloat

```
String::matchFloat() : Boolean
```

Returns true if the string represents a real.

This is a synonym of the matchReal() operation. It is therefore deprecated.

8.3.16.25 matchReal

```
String::matchReal() : Boolean
```

Returns true if the string represents a real.

8.3.16.26 matchIdentifier

`String::matchIdentifier() : Boolean`

Returns true if the string represents an alphanumeric word.

8.3.16.27 asBoolean

`String::asBoolean() : Boolean`

Returns a Boolean value if the string can be interpreted as a string. Null otherwise.

8.3.16.28 asInteger

`String::asInteger() : Integer`

Returns a Integer value if the string can be interpreted as as integer. Null otherwise.

8.3.16.29 asFloat

`String::asFloat() : Real`

Returns a Real value if the string can be interpreted as as real. Null otherwise.

This is a synonym of the `asReal()` operation. It is therefore deprecated.

8.3.16.30 asReal

`String::asReal() : Real`

Returns a Real value if the string can be interpreted as as real. Null otherwise.

8.3.16.31 startStrCounter

`String::startStrCounter (s:String) : Void`

Associates a counter to the string. Initializes the counter to zero.

8.3.16.32 getStrCounter

`String::getStrCounter (s:String) : Integer`

Returns the current value of the counter associated with the string.

8.3.16.33 incrStrCounter

`String::incrStrCounter (s:String) : Integer`

Increments the current value of the counter associated with the string.

8.3.16.34 restartAllStrCounter

`String::restartAllStrCounter () : Void`

Restarts all the counters associated with strings.

8.3.16.35 addSuffixNumber

`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.

8.3.17 Operations on numeric types

Integer::range (start:Integer,end:Integer) : List(Element)

Returns a list of integers starting from 'start' position to 'end' position.

8.3.18 Operations on Classifiers

8.3.18.1 Classifier::allInstances() : Set(T)

The OCL definition is: *The operation allInstances() returns all instances of the classifier and the classifiers specializing it. May only be used for classifiers that have a finite number of instances. This is the case, for example, for user defined classes because instances need to be created explicitly, and for enumerations, the standard Boolean type, and other special types such as OclVoid. This is not the case, for example, for data types such as collection types or the standard String, UnlimitedNatural, Integer, and Real types.*

This needs clarification for use in an imperative QVTo context for which OCL's expectation of an unchanging context is only valid within sub-expressions of an *ImperativeExpression*. For QVTo, the prevailing state is used; successive calls to *allInstances()* may return different sets of mutable instances.

Instances are returned from all the model extents for input, inout and output models. Instances of intermediate objects are not returned unless they have been added to an extent. Instances from a metamodel are not returned unless the metamodel is also an input model. The *Model::objectsOfKind()* operation may be used to return selected instances from a particular model extent.

8.3.19 Predefined tags

proxy : Boolean

When present this tag indicates that the instance acts as 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 an alternative name. This is used in the concrete syntax to avoid name clashes. This is a purely syntactic tag. Consequently the representation of this Tag can be skipped after parsing the source file.

topclasses : String

Tag used to mark model types in order to provide the list of classes names, comma separated, 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 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 preserves manual changes.

8.4 Concrete Syntax

8.4.1 Files

An operational transformation specification provided using concrete syntax in a text file may import one or more compilation units. A compilation unit corresponds to another operational transformation or library definition given either using the concrete syntax definition or the abstract representation.

Within an **import** statement a compilation unit is referenced by means of a simple unqualified alphanumeric identifier, with no special characters and blank characters allowed, or is referenced by means of a qualified one. In the latter case, the qualification is given by a list of namespaces separated by the dot character. These namespaces have no representation in the metamodel. It is up to an implementation to make a correspondence between the namespace hierarchy and the hierarchy of the file system.

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 transformation 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 linefile.
2. Line comment delimited by “//” and the end of linefile.
3. Multi-line comments delimited by “/*” and “*/.”

8.4.3 Strings

Literal strings that fit in a single line are delimited either by single quotes or by double quotes. Literal strings that fit in multiple lines can be notated as a list of literal strings.

Example:

```
var s:String = 'This is a long string'  
             'that fits in two lines';
```

All the usual escape characters using backslash can be used including the '\n' return-line character. The list of available escape characters are those defined for the Java language.

EscapeSequence:

```
\ b /* \u0008: backspace BS */
\ t /* \u0009: horizontal tab HT */
\ n /* \u000a: linefeed LF */
\ f /* \u000c: form feed FF */
\ r /* \u000d: carriage return CR */
\ " /* \u0022: double quote" */
\ ' /* \u0027: single quote ' */
\\ /* \u005c: backslash \ */

OctalEscape /* \u0000 to \u00ff: from octal value
```

OctalEscape:

```
\ OctalDigit
\ OctalDigit OctalDigit
\ ZeroToThree OctalDigit OctalDigit
```

OctalDigit: one of

```
0 1 2 3 4 5 6 7
```

ZeroToThree: one of

```
0 1 2 3
```

8.4.4 Shorthands used to invoke specific pre-defined operations

In this clause 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 shorthand for invoking the pre-defined 'format' operation. The potential ambiguity with the modulo operator is solved thanks to the type of the first argument, which is a string in our case.
5. The binary operator "==" can replace the "=" comparison operator.
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.

The alternative '==' and '!=' notations can only be used if the source file pre-declares the intention of using them by means of a directive placed before entering the library of the transformation definitions. The syntax of this directive should be a comment having the following form:

```
-- directive: use-traditional-comparison-syntax, or
// directive: use-traditional-comparison-syntax.
```

This declaration makes illegal the usage of the corresponding OCL-like syntax ('=' and '<>') within the compilation unit.

8.4.5 Other Language Shorthands

The following describes additional shorthand conventions not described in the Notation sub-clause 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.

8.4.6 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 Clause 8.4.4.

A MOF Package is notated either using the **package** keyword or the **metamodel** keyword followed by a name. Both notations are equivalent when translating to the corresponding EMOF metamodel representation. The **metamodel** keyword should preferably be used when referring to a top-level package that represents a complete metamodel.

```
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 { ... }
```

A primitive type is declared using the **primitive** keyword.

```
'primitive' <primitivetypname>.
```

Example:

```
primitive Double;
```

An enumeration type is notated using the **enum** keyword followed by the list of enumeration values.

```
enum ParameterDirectionKind { "in", "inout", "out" }
```

An exception is declared using the **exception** keyword followed by the name of the exception. Inheritance between exceptions is introduced using the **extends** keyword.

Example:

```
exception VeryStrangeException extends UnexpectedException;
```

The properties of a class are notated using a declarator made of: a name, a type, optional property qualifiers, and an optional initialization expression.

The general syntax pattern is:

```
<qualifier>* <proprname> ':' <typespec> ('=' <init>)? '<multiplicity>'? 'ordered'?  
(opposites '~'? <proprname> <multiplicity>)?
```

For properties typed by primitive types or collection of primitive types, the valid property qualifiers are 'derived' and 'readonly' to represent respectively a derived attribute or a readonly attribute. When provided, the initialization value for a derived attribute represents the expression used to compute the value of the derived property.

The syntax of multiplicity specification follows the regular convention from UML, where brackets surrounds the definition (such as [0..1], [1], [*] and [0..*]). When absent the [0..1] multiplicity is assumed.

Examples:

```
isStatic : Boolean = 0;  
values : Sequence(String);  
derived size : Integer = self.ownedElement->size();
```

To indicate that an attribute acts as a qualifying identifier for the class, a stereotype qualifier named 'id' is used. The syntax for stereotyped qualifiers is:

```
'<<' <IDENTIFIER> (' <IDENTIFIER>)? '>>'
```

Example:

```
<<id> uuid:String [1]; // a mandatory qualifying uuid attribute
```

For properties referencing non primitive types, the same syntax is used except that the 'composes' or 'references' qualifier is used to distinguish between composite or non composite association ends. The 'opposites' keyword is used to indicate the opposite property, if any. When present, the '~' annotation indicates that the opposite property is not navigable.

Example:

```
composes ownedElement : Element [*] ordered opposites namespace;  
references usedElement : Element [*];
```

An operation declaration has a name, a list of formal parameters, and a return type.

```
getAllBaseClasses() : Set(Class);
```

8.4.7 EBNF

We provide below a formal definition of the textual syntax in EBNF.

```
// Syntax for module definitions
```



```

// keywords

Bag, Collection, Dict, OrderedSet, Sequence, Set, Tuple, List, abstract,
access, and, any, assert, blackbox, break, case, class, collect,
collectNested, composes, compute, configuration, constructor, continue,
datatype, default, derived, disjuncts, do, elif, else, end, endif,
enum, except, exists, extends, exception, false, forAll, forEach,
forOne, from, helper, if, implies, import, in, inherits, init,
inout, intermediate, invresolve, invresolveIn, invresolveone,
invresolveoneIn, isUnique, iterate, late, let, library, literal,
log, main, map, mapping, merges, metamodel, modeltype, new, not,
null, object, one, or, ordered, out, package, population, primitive,
property, query, raise, readonly, references, refines, reject, resolve,
resolveIn, resolveone, resolveoneIn, return, select, sortBy, static,
switch, tag, then, transformation, true, try, typedef, unlimited, uses,
var, when, where, while, with, xcollect, xcollectOne, xcollectselect,
xcollectselectOne, xmap, xor, xselect, xselectOne

<identifier> ::= <simpleNameCS> // from OCL

<STRING> ::= (#x22 StringChar* #x22) // from OCL StringLiteralExpCS
           | (#x27 StringChar* #x27)

// start rule

<topLevel> ::= <import>* <unit_element>*

<import> ::= 'from' <unit> 'import' (<identifier_list> | '*') ';'
           | 'import' <unit> ';'
<unit> ::= <identifier> ('.' <identifier>)*

<identifier_list> ::= <identifier> (',' <identifier>)*

// definitions in a compilation unit
<unit_element>
  ::= <transformation>
     | <library>
     | <access_decl>
     | <modeltype>
     | <metamodel>
     | <classifier>
     | <property>
     | <helper>
     | <constructor>
     | <entry>
     | <mapping>
     | <tag>
     | <typedef>

// Transformation and library definitions
<transformation> ::= <transformation_decl> | <transformation_def>
<transformation_decl> ::= <transformation_h> ';'
<transformation_def> ::= <transformation_h> '{' <module_element>* '}' ';'

<library> ::= <library_decl> | <library_def>
<library_decl> ::= <library_h> ';'
<library_def> ::= <library_h> '{' <module_element>* '}' ';'

// Transformation header
<transformation_h> ::= <qualifier>* 'transformation' <identifier>
                   <transformation_signature> <transformation_usage_refine>?
<transformation_usage_refine> ::= <module_usage> | <transformation_refine>
<transformation_signature> ::= <simple_signature>

```

```

<transformation_refine> ::= 'refines' <moduleref>
// Library header
<library_h> ::= 'library' <identifier> <library_signature>? <module_usage>?
<library_signature> ::= <simple_signature>
// import of transformation and library
<module_usage> ::= <access_usage> | <extends_usage>
<access_usage> ::= 'access' <module_kind>? <moduleref_list>
<extends_usage> ::= 'extends' <module_kind>? <moduleref_list>
<module_kind> ::= 'transformation' | 'library'
<moduleref_list> ::= <moduleref> (',' <moduleref>)*
<moduleref> ::= <scoped_identifier> <simple_signature>?
<access_decl> ::= <access_usage> ';'
// module definitions
<module_element>
  ::= <classifier>
     | <property>
     | <helper>
     | <constructor>
     | <entry>
     | <mapping>
     | <tag>
     | <typedef>
     | <access_decl>
// general purpose grammar rules
<qualifier> ::= 'blackbox' | 'abstract' | 'static'
<complete_signature> ::= <simple_signature> ':' param_list)?
<simple_signature> ::= '(' <param_list>? ') '
<param_list> ::= <param> (',' <param>)*
<param> ::= <param_direction>? <declarator>
<param_direction> ::= 'in' | 'inout' | 'out'
<simple_declarator> ::= <typespec>
                    | <scoped_identifier> ':' <typespec>
<declarator> ::= <typespec> <init_part>?
               | <scoped_identifier> ':' <typespec> <init_part>?
<simple_declarator_list> ::= <simple_declarator> (',' <simple_declarator>)*
<declarator_list> ::= <declarator> (',' <declarator>)*
<declarator_semi_list> ::= <declarator> (';' <declarator>)*
<init_part> ::= <init_op> <expression>
<init_op> ::= '=' | ':=' | '::='
<typespec> ::= <type_reference> <extent_location>?
<type_reference> ::= <scoped_identifier> | <complex_type>
<extent_location> ::= '@' <identifier>
<complex_type> ::= <complex_type_key>
                 | <collection_key> '(' <typespec> ') '
                 | 'Tuple' '(' <declarator_list> ') '
                 | 'Dict' '(' <typespec> ',' <typespec> ') '
<complex_type_key> ::= <collection_key> | 'Dict' | 'Tuple'
<collection_key> ::= 'Collection' | 'Set' | 'OrderedSet' | 'Sequence' | 'Bag'
                  | 'List'
<scoped_identifier> ::= <identifier> ('::' <identifier>)*
<scoped_identifier_list> ::= <scoped_identifier> (',' <scoped_identifier>)*
<expression_list> ::= <expression_semi_list> ';'?
<expression_semi_list> ::= <expression> (';' <expression>)*
<expression_comma_list> ::= <expression> (',' <expression>)*
<expression_block> ::= '{' <expression_list>? '}'
<expression_statement> ::= <expression> ';' | <expression_block> ';'?
// model types compliance and metamodel declarations
<modeltype> ::= 'modeltype' <identifier> <compliance_kind>?

```

```

    'uses' <packageref_list> <modeltype_where>? ';'
<modeltype_where> ::= 'where' <expression_block>
<packageref_list> ::= <packageref> (',' <packageref>)*
<packageref> ::= (<scoped_identifier> ( '(' <uri> ') ' )? | <uri>)
<compliance_kind> ::= <STRING> // like: "strict" and "effective"
<uri> ::= <STRING>

// Syntax for defining explicitly metamodel contents
<metamodel> ::= <metamodel_decl> | <metamodel_def>
<metamodel_decl> ::= <metamodel_h> ';';
<metamodel_def> ::= <metamodel_h> '{' <metamodel_element>* '}' ';';

<metamodel_h> ::= ('metamodel' | 'package') scoped_identifier
<metamodel_element> ::= <classifier> | <enumeration> | <tag>

<classifier> ::= <classifier_decl> | <classifier_def>
<classifier_decl> ::= <classifier_h> ';';
<classifier_def> ::= <classifier_h> '{' <classifier_feature_list>? '}' ';';

<classifier_h> ::= <classifier_info> <scoped_identifier> <classifier_extension>?
<classifier_info> ::= 'datatype'
    | 'primitive' | 'exception'
    | 'intermediate'? <qualifier>* 'class'
<classifier_extension> ::= 'extends' <scoped_identifier_list>
<classifier_feature_list> ::= <classifier_feature> (';' <classifier_feature>)* ';';
<classifier_feature> ::= <classifier_property> | <classifier_operation> | <tag>

<classifier_property> ::= <feature_qualifier>? <declarator> <multiplicity>?
    <opposite_property>?
<feature_qualifier> ::= <stereotype_qualifier>? <feature_key>*
<feature_key> ::= 'composes' | 'references' | 'readonly' | 'derived' | 'static'
<stereotype_qualifier> ::= '<<' <identifier_list> '>>'
<multiplicity> ::= '[' multiplicity_range ']'
<multiplicity_range> ::= <INTEGER> | '*' | <INTEGER> '...' <INTEGER> | <INTEGER>
    '...' '*'
<classifier_operation> ::= <feature_qualifier>? <declarator> <complete_signature>

<enumeration> ::= <enumeration_h> ';';
    | <enumeration_h> '{' <identifier_list> '}' ';';

<enumeration_h> ::= 'enum' <identifier>
<opposite_property> ::= 'opposites' '~'? <identifier> <multiplicity>?

<tag> ::= 'tag' <tagid> <scoped_identifier> ('=' <tagvalue>)? ';';
<tagid> ::= <STRING>
<tagvalue> ::= <expression>
<typedef> ::= 'typedef' <identifier> '=' <typespec> ';';

// Properties in transformation
<property> ::= 'intermediate'? <property_key>+ <declarator> ';';
<property_key> ::= 'derived' | 'literal' | 'configuration' | 'property'

// Syntax for helper operations
<helper> ::= <helper_decl> | <helper_simple_def> | <helper_compound_def>
<helper_header> ::= <helper_info> <scoped_identifier> <complete_signature>
<helper_info> ::= <qualifier>* <helper_kind>
<helper_kind> ::= 'helper' | 'query'
<helper_decl> ::= <helper_header> ';';
<helper_simple_def> ::= <helper_header> '=' <expression> ';';
<helper_compound_def> ::= <helper_header> <expression_block> ';';

// Syntax for constructors

```

```

<constructor> ::= <constructor_decl> | <constructor_def>
<constructor_header> ::= <qualifier>* 'constructor' <scoped_identifier>
    <simple_signature>
<constructor_decl> ::= <constructor_header> ';'
<constructor_def> ::= <constructor_header> <expression_block> ';'?

// Syntax for entries
<entry> ::= <entry_decl> | <entry_def>
<entry_header> ::= 'main' <simple_signature>
<entry_decl> ::= <entry_header> ';'
<entry_def> ::= <entry_header> <expression_block> ';'?

// syntax for mapping operations
<mapping> ::= <mapping_decl> | <mapping_def>
<mapping_decl> ::= <mapping_full_header> ';'
<mapping_def> ::= <mapping_full_header> '{' <mapping_body> '}' ';'?
<mapping_full_header> ::= <mapping_header> <when>? <where>?
<mapping_header> ::= <qualifier>* 'mapping' <param_direction>?
    <scoped_identifier> <complete_signature> <mapping_extra>*
<mapping_extra> ::= <mapping_extension> | <mapping_refinement>
<mapping_extension> ::= <mapping_extension_key> <scoped_identifier_list>
<mapping_extension_key> ::= 'inherits' | 'merges' | 'disjuncts'
<when> ::= 'when' <expression_block>
<where> ::= 'where' <expression_block>
<mapping_refinement> ::= 'refines' <scoped_identifier>
<mapping_body> ::= <init_section>? <population_section>? <end_section>?
<init_section> ::= 'init' <expression_block>
<population_section> ::= <expression_list>
    | 'population' <expression_block>
<end_section> ::= 'end' <expression_block>

// Expressions
<expression> ::= <assign_exp>
    | <let_exp>
    | <var_init_exp>

<assign_exp> ::= <implies_exp>
    | <unary_exp> <assign_op> <expression> <default_val>?
    | <unary_exp> <assign_op> <expression_block> <default_val>?

<assign_op> ::= ':= ' | '::=' | '+=' | '-='

<default_val> ::= 'default' <assign_exp>

<implies_exp> ::= <or_exp>
    | <implies_exp> 'implies' <or_exp>

<or_exp> ::= <and_exp>
    | <or_exp> <or_op> <and_exp>

<or_op> ::= 'or' | 'xor'

<and_exp> ::= <cmp_exp>
    | <and_exp> 'and' <cmp_exp>

<cmp_exp> ::= <additive_exp>
    | <cmp_exp> <cmp_op> <additive_exp>

<cmp_op> ::= '=' | '==' | '<' | '<' | '>' | '>' | '<=' | '>='

<additive_exp> ::= <mult_exp>

```

```

        | <additive_exp> <add_op> <mult_exp>
<add_op> ::= '+' | '-'
<mult_exp> ::= <unary_exp>
        | <mult_exp> <mult_op> <unary_exp>
<mult_op> ::= '*' | '/' | '%'
<unary_exp> ::= <postfix_exp>
        | <unary_op> <unary_exp>
<unary_op> ::= '-' | 'not' | '#' | '##' | '*'
<postfix_exp> ::= <primary_exp>
        | <postfix_exp> '(' <arg_list>? ')
        | <postfix_exp> '!'? '[' <declarator_vsep>? <expression> ']'
        | <postfix_exp> <access_op>
          (<scoped_identifier> | <iterator_exp> | <block_exp>
           | <control_exp> | <rule_call_exp>
           | <resolve_exp> | <resolve_in_exp>)
<declarator_vsep> ::= <simple_declarator> '|'
<multi_declarator_vsep> ::= <simple_declarator_list> '|'
<resolve_exp> ::= <resolve_key> '(' <resolve_condition>? ')
<resolve_condition> ::= <declarator> '(' <expression> )?
<resolve_key> ::= 'late'? <resolve_kind>
<resolve_kind> ::= 'resolve' | 'resolveone' | 'invresolve' | 'invresolveone'
<resolve_in_exp> ::= <resolve_in_key> '(' <scoped_identifier>
        (',' <resolve_condition>)? ')'
<resolve_in_key> ::= 'late'? <resolve_in_kind>
<resolve_in_kind> ::= 'resolveIn' | 'resolveoneIn' | 'invresolveIn' | 'invresolveoneIn'

<access_op> ::= '.' | '->' | '!->'

<primary_exp> ::= <literal>
        | <scoped_identifier>
        | <if_exp>
        | <block_exp>
        | <control_exp>
        | <rule_call_exp>
        | <quit_exp>
        | <try_exp>
        | <raise_exp>
        | <assert_exp>
        | <log_exp>
        | '(' <expression> ')'

<literal> ::= <literal_simple>
        | <literal_complex>

<literal_simple> ::= <INTEGER> | <FLOAT> | <STRING>
        | 'true' | 'false' | 'unlimited' | 'null'

<literal_complex> ::= <literal_collection>
        | <literal_tuple>

```

```

        | <literal_dict>

<literal_collection> ::= <collection_key> '{' <collection_item_list>? '}'

<literal_tuple> ::= 'Tuple' '{' <tuple_item_list>? '}'

<literal_dict> ::= 'Dict' '{' <dict_item_list>? '}'

<collection_item> ::= <expression> ('..' <expression>)?
<collection_item_list> ::= <collection_item> (',' <collection_item>)*
<tuple_item_list> ::= <declarator_list>
<dict_item_list> ::= <dict_item> (',' <dict_item>)*
<dict_item> ::= <literal_simple> '=' <expression>

<if_exp> ::= 'if' <expression> <then_part>
           <elif_part>* <else_part>? 'endif'
<then_part> ::= 'then' <if_body>
<elif_part> ::= 'elif' <if_body>
<else_part> ::= 'else' <if_body>
<if_body> ::= <expression> | <expression_block>

<iterator_exp> ::= <simple_iterator_op> '(' <declarator_vsep>? <expression> ')'|
                 | <multi_iterator_op> '(' <multi_declarator_vsep>? <expression> ')'|
                 | <iterate_exp>

<simple_iterator_op> ::= 'reject' | 'select' | 'collect' | 'exists'
                    | 'one' | 'any' | 'isUnique' | 'collectNested'
                    | 'sortedBy' | 'xselect' | 'xcollect'
                    | 'xselectOne' | 'xcollectOne'
                    | 'xcollectselect' | 'xcollectselectOne'

<multi_iterator_op> ::= 'forAll'

<iterate_exp> ::= 'iterate' '(' <declarator_list> ';'
                 <declarator> '|' <expression> ')

<iter_declarator> ::= <declarator>
<iter_declarator_list> ::= <declarator_list>

<block_exp> ::= (<object_exp> | <do_exp> | <switch_exp>)

<object_exp> ::= 'object' '(' <iter_declarator> ')'? <object_declarator>
               <expression_block>
<object_declarator> ::= <typespec> | <identifier> ':' <typespec>?

<do_exp> ::= 'do' <expression_block>

<switch_exp> ::= 'switch' '(' <iter_declarator> ')'? <switch_body>
<switch_body> ::= '{' <switch_alt>+ <switch_else>? '}'
<switch_alt> ::= 'case' '(' <expression> ') ' <expression_statement>
<switch_else> ::= 'else' <expression_statement>

<control_exp> ::= (<while_exp> | <compute_exp> | <for_exp>)

<while_exp> ::= 'while' '(' (<declarator> ';')? <expression> ') '
               <expression_block>
<compute_exp> ::= 'compute' '(' <declarator> ') ' <expression_block>
<for_exp> ::= ('forEach' | 'forOne') '(' <iter_declarator_list>
              (';' <declarator>)? ('|' <expression>)? ')' <expression_block>

```

```

<rule_call_exp> ::= ('map' | 'xmap' | 'new' )
  (('(<declarator> '))'? <scoped_identifier>

<let_exp> ::= 'let' <declarator_list> 'in' <expression>

<var_init_exp> ::= 'var' <declarator_list>
  | 'var' (('(<declarator_list> '))'

<quit_exp> ::= 'break' | 'continue' | <return_exp>)

<return_exp> ::= 'return' <expression>?

<try_exp> ::= 'try' <expression_block> <except>+

<except> ::= 'except' ((' [<identifier> ':'] <scoped_identifier>
  [',' <scoped_identifier>]* ')') <expression_block>

<raise_exp> ::= 'raise' <scoped_identifier> (('(<arg_list>? '))'?
  | 'raise' <STRING>

<arg_list> ::= <expression_comma_list>

<assert_exp> ::= 'assert' <identifier>? ((' <expression> ')')
  ( 'with' <log_exp> )?

<log_exp> ::= 'log' (('(<arg_list> '))' ('when' <expression>))?

```

8.4.7.1 Deprecated syntax

The following alternative constructs may be provided by tools offering backward compatibility to earlier versions of the QVT specification.

```

// typedefs
<typedef> ::= 'typedef' <identifier> '=' <typespec> (typedef_condition)? ';'
<typedef_condition> ::= '[' <expression> ']'

```

8.4.8 Scoped Identifiers

The usages of <scoped_identifier> are clarified in this subclause.

8.4.8.1 Type Identifiers

When referring to a type within an operational transformation definition it is possible to either qualify the type name with a model type or a package name, or, alternatively, leave the name of the type unqualified.

```

<package_name> ::= <identifier>
<model_type> ::= <identifier>
<type_name> ::= <identifier>
<type_id> ::= ((<package_name> '::')+ | (<model_type> '::'))? <type_name>

```

In the latter case, a list of rules applies to resolve the symbol into the denoted type. If the resolution rule provides more than one result, the specification is erroneous.

- First, a type definition existing at the level of the current module (a transformation or a library) is searched.
- If not found, all the packages of the model types declared in the module are recursively visited to found a type with the same name.

8.4.8.2 Transformation Identifiers

A transformation identifier consists of the name of the transformation prefixed by the ::-separated names of any containing packages.

```
<transformation_name> ::= <identifier>  
<transformation_id> ::= (<package_name> '::')* <transformation_name>
```

Leading parts of the <package_name> may be omitted provided the omission introduces no ambiguity.

8.4.8.3 Mapping Identifiers

A mapping identifier is used to reference a contribution to a disjuncted, inherited or merged mapping. It is also used to disambiguate a mapping call. A mapping identifier is the :::-separated and ::-separated concatenation of a transformation identifier, context type identifier and the mapping name.

```
<mapping_name> ::= <identifier>  
<mapping_id> ::= (<transformation_id> ':::')? (<type_id> '::')? <mapping_name>
```

Part or all of <transformation_id> and <type_id> may be omitted provided the omission introduces no ambiguity.

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 Clause 10.

As described in Clause 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.

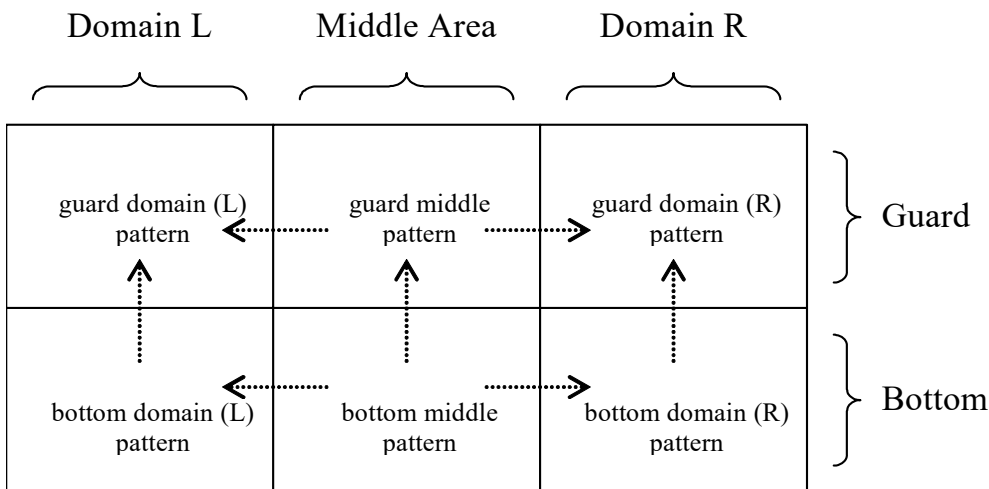


Figure 9.1 - Core Domains and Pattern Dependencies

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 Figure 9.1 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 Clause 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 depend 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 fulfill 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 domain's bottom pattern cannot 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 $b1 \geq b2$ 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:

1. 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.
2. 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

This clause defines 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 Clause 7.11.1 “QVTBase Package” on page 24.

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 packages of the QVT Core formalism are not shaded. Within the class descriptions, metaclasses and meta-properties of the metamodel are rendered in courier font. 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.

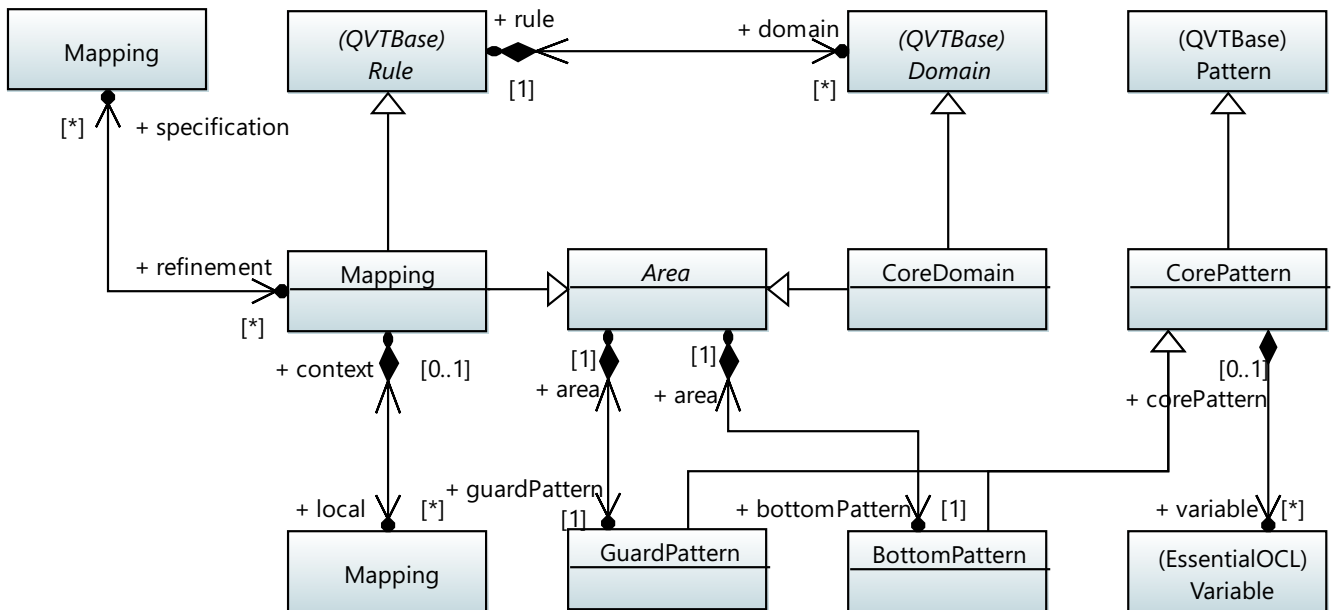


Figure 9.2 - QVTCore Package - Mappings and Patterns

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.

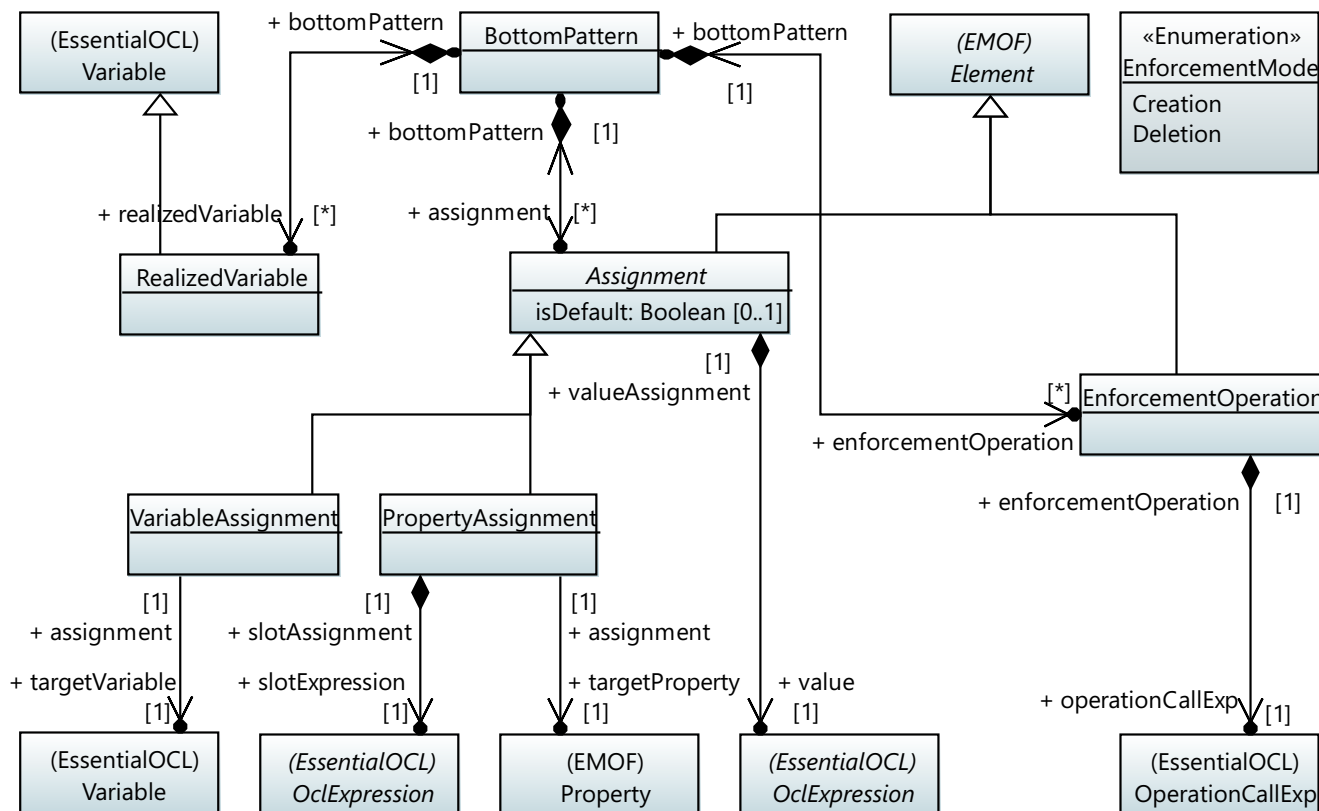


Figure 9.3 - QVT Core Package - Bottom patterns

Superclasses

Pattern

Associations

```
variable: Variable [*] {composes} (opposite end: corePattern?)
```

Unrealized guard pattern variables are the unbound variables of the pattern. Unrealized bottom pattern variables may be used to factor out common expressions.

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 that is evaluated as a guard to the bottom pattern.

```
bottomPattern: BottomPattern [1] {composes}
```

The core pattern that 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 Clause 9.13 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 domain's 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, 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 Clause 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 Clause 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 : CoreDomain [*] {composes} (From QVTBase)`

The domains that 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 Clause 9.15 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 Clause 9.14 for refinement semantics.)

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 [1]`

The bottom pattern in which the realized variable is declared.

9.17.8 Assignment

An assignment sets the property of a target object or sets the value of a variable. See *PropertyAssignment* and *VariableAssignment* definitions for detailed semantic of property assignment and variable assignment.

Attributes

`isDefault : Boolean`

Indicates whether the assignment is default assignment or a non-default assignment.

Associations

value : OclExpression [1] {composes}

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.

bottomPattern : BottomPattern [1]

The bottom pattern that contains the assignment.

9.17.9 PropertyAssignment

A *property 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.

Superclasses

Assignment

Associations

slotExpression: OclExpression [1] {composes}

An OCL expression identifying the object whose property value is to be assigned.

targetProperty: Property [1]

The property whose value is to be assigned. The target property should be defined on the type of the slot expression.

9.17.10 VariableAssignment

A *variable assignment* sets the value of a variable.

Superclasses

Assignment

Associations

targetVariable: Variable [1]

The variable whose value is to be assigned.

9.17.11 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.

9.17.12 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 fulfill the bottom pattern conditions.

Associations

operationCallExp: OperationCallExp [1] {composes}

An OCL Expression identifying an operation invocation and its parameters, which must include the variables that are defined in the owning bottom pattern.

bottomPattern: BottomPattern [1]

The bottom pattern in which the enforcement operation is declared.

9.18 Concrete Syntax

This clause defines the concrete textual syntax for the QVT core language using the EBNF notation.

Production names ending CS are defined by OCL.

```
TopLevel ::= Import* (Transformation | Mapping | Query)*
Import ::= 'import' [simpleNameCS ':' ] <URI> ('::' simpleNameCS)* ['::*'] ';'
<URI> ::= #x27 StringChar* #x27 //from OCL StringLiteralExpCS
Transformation ::=
'transformation' TransformationName
'{' ( Direction';' )* '}'
[ 'extends' TransformationName (',' TransformationName)* ]
Direction ::=
[DirectionName] [ 'imports' PackageName(',' PackageName)* ]
[ 'uses' DirectionName(',' DirectionName)* ]
```

```

Mapping ::=
  ['abstract'] 'map' MappingName ['in' TransformationName] ['refines' MappingName] '{'
  (
    ['check'] ['enforce'] DirectionName ('DomainGuardPattern') '{'
    DomainBottomPattern
    '}' ) *
  'where' '(' MiddleGuardPattern ')' '{'
    MiddleBottomPattern
  '}'
  (
    ComposedMapping ) *
  '}'

ComposedMapping ::= 'map' [MappingName] ['refines' MappingName] '{'
  (['check'] ['enforce'] DirectionName ('DomainGuardPattern') '{'
  DomainBottomPattern
  '}' ) *
  'where' '(' MiddleGuardPattern ')' '{'
  MiddleBottomPattern
  '}'
  (ComposedMapping ) *
  '}'

DomainGuardPattern, MiddleGuardPattern ::= GuardPattern
DomainBottomPattern, MiddleBottomPattern ::= BottomPattern

GuardPattern ::=
  [Variable(',' Variable ) * ]
  ['|' ( Predicate ';' ) +]

BottomPattern ::=
  [ (Variable | RealizedVariable)
  (',' ( Variable | RealizedVariable)) * ]
  ['|' ( (Constraint | EnforcementOperation) ';' ) +]

EnforcementOperation ::=
  ('create' | 'delete') OperationCallExpCS

Variable ::=
  VariableName ':' TypeDeclaration [':' ValueOCLExpr]

RealizedVariable ::=
  'realize' VariableName ':' TypeDeclaration

Constraint ::= Predicate | Assignment

Predicate ::= BooleanOCLExpr

Assignment ::=
  ['default'] SlotOwnerOCLExpr '.' PropertyName ':' ValueOCLExpr

Query ::= 'query' QueryId '(' [ ParamDeclaration (',' ParamDeclaration) * ] ')'
  ':' TypeDeclaration (';' | '{' QueryOCLExpr '}')

QueryId ::= PathNameCS

ParamDeclaration ::= ParameterName ':' TypeDeclaration

DirectionName ::= simpleNameCS
MappingName ::= pathNameCS
PackageName ::= pathNameCS
PropertyName ::= simpleNameCS
QueryName ::= pathNameCS
TransformationName ::= pathNameCS
VariableName ::= simpleNameCS

TypeDeclaration ::= typeCS

BooleanOCLExpr ::= OclExpressionCS
QueryOCLExpr ::= OclExpressionCS
SlotOwnerOCLExpr ::= OclExpressionCS
ValueOCLExpr ::= OclExpressionCS

```

10 Relations to Core Transformation

This clause 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's when and where clauses map to the middle area of a mapping: 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: 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. 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. 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.

Variables of a RelationDomain that occur in the when clause become Variables of the core domain guard.

- 2.1 All other Variables of a RelationDomain become Variables of the core domain bottom pattern.
- 2.2 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.
- 2.3 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.
- 2.4 Predicates of the when clause become predicates of the core mapping guard.
- 2.5 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)

A relation is 'checkonly' if it does not have any enforceable domains.

- 3.1 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.
- 3.2 A property template item in a relation domain becomes an equality predicate in the core domain bottom.
- 3.3 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.
- 3.4 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)

A separate mapping is generated for each enforced domain of the relation.

- 4.1 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).
- 4.2 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 that 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.
- 4.3 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.
- 4.4 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.
- 4.5 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 'Key'; 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)

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.

- 5.1 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.
- 5.2 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)

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.

- 6.1 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.
- 6.2 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 UmIRdbms
{
    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

```

transformation relToCore(relations:QVTMM; core:QVTMM)
{
    key QVTMM::Mapping{name, _transformation};
    key QVTMM::GuardPattern{area};
    key QVTMM::BottomPattern{area};
    key QVTMM::Variable{name, type};
}

```

```

key QVTMM::Type{name};
key QVTMM::Class{name};
key QVTMM::Property{name, class};
key QVTMM::CoreDomain{name, rule};
key QVTMM::TypedModel{name, usedPackage, _transformation};
key QVTMM::Package{name};
key QVTMM::Transformation{name};
key QVTMM::Operation{name};
key QVTMM::Predicate{pattern, conditionExpression};

query getSharedDomainVars(r:QVTMM::Relation):Set(QVTMM::Variable)
{
  r._domain->iterate(d; vars: Set(QVTMM::Variable) = Set{} |
    if (vars->isEmpty())
    then
      vars->union(d.oclasType(QVTMM::RelationDomain).pattern.bindsTo)
    else
      vars->intersection(d.oclasType(QVTMM::RelationDomain).pattern.bindsTo)
    endif
  )
}

query getWhenVars(r:QVTMM::Relation):Set(QVTMM::Variable)
{
  let
    vs:Set(QVTMM::Variable) = Set{}
  in
    r._domain->iterate(d; vars: Set(QVTMM::Variable) = Set{} |
      if (vars->isEmpty())
      then
        vars->union(d.oclasType(QVTMM::RelationDomain).pattern.bindsTo)
      else
        vars->intersection(d.oclasType(QVTMM::RelationDomain).pattern.bindsTo)
      endif
    )
}

-- Get variables occurring in an ocl expression
-- Note: this function is not complete! It needs to be completed for other expressions
query getVarsOfExp(e:QVTMM::OclExpression):Set(QVTMM::Variable)
{
  -- Walk the expr tree of the OclExpression and
  -- collect the variables used in those expressions
  let
    vs:Set(QVTMM::Variable) = Set{}
  in
    if (e.oclasTypeOf(QVTMM::VariableExp))
    then
      vs->including(e.oclasType(QVTMM::VariableExp).referredVariable)
    else
      if (e.oclasTypeOf(QVTMM::OperationCallExp))
      then
        let
          oc:QVTMM::OperationCallExp = e.oclasType(QVTMM::OperationCallExp)
        in
          vs->union(getVarsOfExp(oc.source))->union(
            oc.argument->iterate(a; avs:Set(QVTMM::Variable)=Set{} | avs-
            >union(getVarsOfExp(a)))
          )
        else
          if (e.oclasTypeOf(QVTMM::PropertyCallExp))
          then
            vs->union(getVarsOfExp(e.oclasType(QVTMM::PropertyCallExp).source))
          else
            if (e.oclasTypeOf(QVTMM::RelationCallExp))
            then
              let
                rc:QVTMM::RelationCallExp = e.oclasType(QVTMM::RelationCallExp)

```

```

        in
            vs->union(rc.argument->iterate(a; avs:Set(QVTMM::Variable)=Set{} |
                avs->union(getVarsOfExp(a)))
            )
        else
            vs
        endif
    endif
endif
endif
}

query filterOutPredicatesThatReferToVars(rpSet:Set(QVTMM::Predicate),
    ownrdVars:Set(QVTMM::Variable)) :Set(QVTMM::Predicate)
{
    rpSet->iterate(p:QVTMM::Predicate; fpSet:Set(QVTMM::Predicate) = Set{|
        if (getVarsOfExp(p.conditionExpression)->intersection(ownrdVars)->isEmpty())
        then
            fpSet->including(p)
        else
            fpSet
        endif
    }
)
}

--Check if the given variable is bound to any template other than the one to be skipped
query isVarBoundToSomeOtherTemplate(rootTe:QVTMM::ObjectTemplateExp,
    skipTe:QVTMM::ObjectTemplateExp, v:QVTMM::Variable):Boolean
{
    if (rootTe = skipTe)
    then
        false
    else
        if (rootTe.bindsTo = v)
        then
            true
        else
            rootTe.part.value->select(pe | pe.oclIsKindOf(QVTMM::ObjectTemplateExp))->exists(pet
|
            isVarBoundToSomeOtherTemplate(pet.oclAsType(QVTMM::ObjectTemplateExp), skipTe, v))
        endif
    endif
}

top relation RelationTransformationToMappingTransformation
{
    rtn, tmn:String;

    domain relations rt:RelationTransformation {
        name = rtn,
        modelParameter = rtm:TypedModel {
            name = tmn,
            usedPackage = up:Package{}
        }
    };

    enforce domain core mt:Transformation {
        name = rtn,
        modelParameter = mtm:TypedModel {
            name = tmn,
            usedPackage = up
        }
    };
}

-- 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.
--
top relation RelationToTraceClass
{
  rn, vn:String;

  domain relations r:Relation {
    name = rn,
    _domain = rd:RelationDomain {
      pattern = rdp:DomainPattern {
        templateExpression = 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
{
  vn: String;

  domain relations t:ObjectTemplateExp {
    part = pt:PropertyTemplateItem {
      value = 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
-- PatternVariables 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 'realized'; 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.
--
top relation TopLevelRelationToMappingForChecking
{
  allDomainVars: Set(QVTMM::Variable);
  sharedDomainVars: Set(QVTMM::Variable);
  unsharedWhereVars: Set(QVTMM::Variable);
  whenVars: Set(QVTMM::Variable);
  whereVars: Set(QVTMM::Variable);
  rn: String;
  mbVars:Set(QVTMM::Variable);
  rt: QVTMM::RelationTransformation;
  mt: QVTMM::Transformation;

  domain relations r:Relation {
    _transformation = rt,
    isTopLevel = true,
    name = rn
  } {
    not r._domain->exists(d| d.isEnforceable = true)
  };
  enforce domain core m:Mapping {
    _transformation = mt,
    name = rn,
    guardPattern = mg:GuardPattern {
      area = m
    },
    bottomPattern = mb:BottomPattern {
      bindsTo = vs:Set(Variable) {
        tcv:RealizedVariable {} ++ mbVars
      }
    }
  };
  when {

```

```

    RelationTransformationToMappingTransformation(rt, mt);
}
where {
  allDomainVars = r._domain->iterate(md; acc:Set(QVTMM::RelationDomain)=Set{} |
    acc->including(md.oclAsType(QVTMM::RelationDomain))).pattern.bindsTo;
  whenVars = r._when.bindsTo;
  whereVars = r._where.bindsTo;

  sharedDomainVars = getSharedDomainVars(r);
  unsharedWhereVars =
    (whereVars - whenVars - allDomainVars)->union(sharedDomainVars);

  RelationToTraceClassVar(r, tcv);
  RWhenPatternToMGuardPattern(r, mg);
  if (unsharedWhereVars->isEmpty())
  then
    mbVars = Set{}
  else
    RVarSetToMVarSet(unsharedWhereVars->asSequence(), mbVars)
  endif;
  -- Only non relation invocation predicates are copied from where clause to mapping
  -- bottom.
  RWherePatternToMPattern(r, mb);
  RDomainToMDomainForChecking(r, m);
}
}

relation RWherePatternToMPattern
{
  domain relations r:Relation{
    _where = wherep:Pattern { }
  };
  enforce domain core mp:Pattern {};
  where {
    RSimplePatternToMPattern(wherep, mp);
  }
}

relation UnsharedWhenVarsToMgVars
{
  domain relations unsharedWhenVars:Set(Variable) {_++_};
  enforce domain core mg:GuardPattern {
    bindsTo = mgVars:Set(Variable) {}
  };
  where {
    RVarSetToMVarSet(unsharedWhenVars->asSequence(), mgVars);
  }
}

relation DomainVarsSharedWithWhenToDgVars
{
  domain relations domainVarsSharedWithWhen:Set(Variable) {_++_};
  enforce domain core dg:GuardPattern {
    bindsTo = dgVars:Set(Variable) {}
  };
  where {
    RVarSetToMVarSet(domainVarsSharedWithWhen->asSequence(), dgVars);
  }
}

relation DomainBottomUnSharedVarsToDbVars
{
  domain relations domainBottomUnSharedVars:Set(Variable) {_++_};
  enforce domain core db:BottomPattern {
    bindsTo = dbVars:Set(Variable) {}
  };
  where {
    RVarSetToMVarSet(domainBottomUnSharedVars->asSequence(), dbVars);
  }
}

```

```

}
}

-- 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.
--
top relation TopLevelRelationToMappingForEnforcement
{
  allDomainVars: Set(QVTMM::Variable);
  oppositeDomainVars: Set(QVTMM::Variable);
  sharedDomainVars: Set(QVTMM::Variable);
  predicatesWithVarBindings: Set(QVTMM::Predicate);
  predicatesWithoutVarBindings: Set(QVTMM::Predicate);
  unsharedWhenVars: Set(QVTMM::Variable);
  unsharedWhereVars: Set(QVTMM::Variable);
  domainVarsSharedWithWhen: Set(QVTMM::Variable);
  domainBottomUnSharedVars: Set(QVTMM::Variable);
  rdSeq, rdtSeq, relImplSeq: Sequence(QVTMM::Element);
  rdSet: Set(QVTMM::Element);
  rdVarsSeq: Sequence(Set(QVTMM::Element));
  rdtSet: Set(QVTMM::Element);
  rdtVarsSeq: Sequence(Set(QVTMM::Element));
  rn, dn, tmn: String;
  rOppositeDomains: Set(QVTMM::RelationDomain);
  oppDomainSeq: Sequence(QVTMM::Element);
  whenVars: Set(QVTMM::Variable);
  whereVars: Set(QVTMM::Variable);
  mbVars: Set(QVTMM::Variable);
  rpSet: Set(QVTMM::Predicate);
  rt: QVTMM::RelationTransformation;
  mt: QVTMM::Transformation;

  domain relations r:Relation {
    _transformation = rt,
    isTopLevel = true,
    name = rn,
    _domain = rds:Set(RelationDomain) {
      rd:RelationDomain {
        isEnforceable = true,
        name = dn,
        typedModel = dir:TypedModel {

```



```

        name = tmn,
        usedPackage = up:Package{},
        _transformation = rt
    },
    pattern = dp:DomainPattern {
        bindsTo = domainVars:Set(Variable) {},
        templateExpression = te:ObjectTemplateExp {
            bindsTo = tev:Variable {}
        }
    } ++ rOppositeDomains
};
enforce domain core m:Mapping {
    _transformation = mt,
    name = rn+'_'+dn,
    guardPattern = mg:GuardPattern {
        area = m
    },
    bottomPattern = mb:BottomPattern {
        bindsTo = vs:Set(Variable) {
            tcv:RealizedVariable {} ++ mbVars
        }
    },
    _domain = md:CoreDomain {
        name = dn,
        isEnforceable = true,
        typedModel = mdir:TypedModel {
            name = tmn,
            usedPackage = up,
            _transformation = mt
        },
        guardPattern = dg:GuardPattern {
            area = md
        },
        bottomPattern = db:BottomPattern {
            bindsTo = mtev:Variable {}
        }
    } --TODO: add var only if tev not in whenVars
};
when {
    RelationTransformationToMappingTransformation(rt, mt);
}
where {
    allDomainVars = r._domain->iterate(md; acc:Set(QVTMM::RelationDomain)=Set{} |
        acc->including(md.oclAsType(QVTMM::RelationDomain)).pattern.bindsTo;
    whenVars = r._when.bindsTo;
    whereVars = r._where.bindsTo;

    -- Exclude where clause relation calls.
    -- The predicate corresponding to a where clause relation call is included not in this
    -- mapping but in the one corresponding to the invoked relation (refer to rule 2.6.1)
    rpSet = r._where.predicate->reject(p |
        p.conditionExpression.oclIsTypeOf(QVTMM::RelationCallExp));

    oppositeDomainVars = rOppositeDomains->iterate(d; vars: Set(QVTMM::Variable) = Set{} |
        vars->union(d.oclAsType(QVTMM::RelationDomain).pattern.bindsTo));
    sharedDomainVars = getSharedDomainVars(r);
    domainBottomUnSharedVars = domainVars - whenVars - sharedDomainVars;

    unsharedWhereVars =
        (whereVars - whenVars - allDomainVars)->union(sharedDomainVars);

    predicatesWithVarBindings =
        filterOutPredicatesThatReferToVars(rpSet, domainBottomUnSharedVars);
    predicatesWithoutVarBindings = rpSet - predicatesWithVarBindings;
    unsharedWhenVars = whenVars - allDomainVars;
    domainVarsSharedWithWhen = domainVars->intersection(whenVars);
}

```

```

rdSeq = Sequence{r, rd};
rdSet = Set{r, rd};
rdVarsSeq = Sequence{rdSet, oppositeDomainVars};
rdtSet = Set{r, rd, te};
rdtVarsSeq = Sequence{rdtSet, predicatesWithoutVarBindings, domainBottomUnSharedVars};
oppDomainSeq = Sequence{r, rd};
relImplSeq = Sequence{r, rd};

RelationDomainToTraceClassVar(rdSeq, tcv);
RWhenPatternToMGuardPattern(r, mg);
DomainVarsSharedWithWhenToDgVars(domainVarsSharedWithWhen, dg);
RVarToMRealizedVar(tev, mtev);
if (unsharedWhereVars->isEmpty())
then
  mbVars = Set{}
else
  RVarSetToMVarSet(unsharedWhereVars->asSequence(), mbVars)
endif;
RPredicateSetToMBPredicateSet(predicatesWithVarBindings->asSequence(), mb);
RDomainToMDBottomForEnforcement(rdtVarsSeq, db);
ROppositeDomainVarsToTraceClassProps(rdVarsSeq, mb);
TROppositeDomainsToMappingForEnforcement(oppDomainSeq, m);
RRelImplToMBottomEnforcementOperation(relImplSeq, 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).
--
top relation InvokedRelationToMappingForChecking
{
  allDomainVars: Set(QVTMM::Variable);
  sharedDomainVars: Set(QVTMM::Variable);
  unsharedWhereVars: Set(QVTMM::Variable);
  seqForInvoker: Sequence(QVTMM::Element);
  rn, irn: String;
  mbVars: Set(QVTMM::Variable);
  rt: QVTMM::RelationTransformation;
  mt: QVTMM::Transformation;
  whenVars: Set(QVTMM::Variable);
  whereVars: Set(QVTMM::Variable);

  domain relations r:Relation {
    _transformation = rt,
    isTopLevel = false,
    name = rn,
    relationCallExp = ri:RelationCallExp {
      predicate = p:Predicate {
        pattern = pt:Pattern {
          whereOwner = ir:Relation {name = irn}
        }
      }
    }
  }
} {
  not r._domain->exists(d| d.isEnforceable = true)
};
enforce domain core m:Mapping {
  _transformation = mt,
  name = rn+'_'+irn,
  guardPattern = mg:GuardPattern {

```

```

        area = m
    },
    bottomPattern = mb:BottomPattern {
        bindsTo = vs:Set(Variable) {
            tcv:RealizedVariable {} ++ mbVars
        }
    }
};
when {
    RelationTransformationToMappingTransformation(rt, mt);
}
where {
    allDomainVars = r._domain->iterate(md; acc:Set(QVTMM::RelationDomain)=Set{} |
        acc->including(md.oclAsType(QVTMM::RelationDomain))).pattern.bindsTo;
    whenVars = r._when.bindsTo;
    whereVars = r._where.bindsTo;
    sharedDomainVars = getSharedDomainVars(r);
    unsharedWhereVars =
        (whereVars - whenVars - allDomainVars)->union(sharedDomainVars);
    seqForInvoker = Sequence{ ir, ri, r};

    RelationToTraceClassVar(r, tcv);
    RWhenPatternToMGuardPattern(r, mg);
    RInvokerToMGuard(seqForInvoker, mg);
    if (unsharedWhereVars->isEmpty())
    then
        mbVars = Set{}
    else
        RVarSetToMVarSet(unsharedWhereVars->asSequence(), mbVars)
    endif;
    RWherePatternToMPattern(r, mb);
    RDomainToMDomainForChecking(r, m);
}
}

-- 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).
--
top relation InvokedRelationToMappingForEnforcement
{
    allDomainVars: Set(QVTMM::Variable);
    oppositeDomainVars: Set(QVTMM::Variable);
    sharedDomainVars: Set(QVTMM::Variable);
    predicatesWithVarBindings: Set(QVTMM::Predicate);
    predicatesWithoutVarBindings: Set(QVTMM::Predicate);
    unsharedWhenVars: Set(QVTMM::Variable);
    unsharedWhereVars: Set(QVTMM::Variable);
    domainTopVars: Set(QVTMM::Variable);
    domainBottomUnSharedVars: Set(QVTMM::Variable);
    rdSeq, relImplSeq: Sequence(QVTMM::Element);
    rdSet: Set(QVTMM::Element);
    rdVarsSeq: Sequence(Set(QVTMM::Element));
    rdtSet: Set(QVTMM::Element);
    rdtVarsSeq: Sequence(Set(QVTMM::Element));
    seqForInvoker: Sequence(QVTMM::Element);
    rn, irn, dn, tmn: String;
    rOppositeDomains:Set(QVTMM::RelationDomain);
    oppDomainSeq:Sequence(QVTMM::Element);
    whenVars: Set(QVTMM::Variable);
    whereVars: Set(QVTMM::Variable);
}

```

```

mbVars: Set(QVTMM::Variable);
rpSet: Set(QVTMM::Predicate);
rt: QVTMM::RelationTransformation;
mt: QVTMM::Transformation;

domain relations r:Relation {
  _transformation = rt,
  isTopLevel = false,
  name = rn,
  relationCallExp = ri:RelationCallExp {
    predicate = p:Predicate {
      pattern = pt:Pattern {
        whereOwner = ir:Relation {name = irn}
      }
    }
  },
  _domain = rds:Set(RelationDomain) {
    rd:RelationDomain {
      isEnforceable = true,
      name = dn,
      typedModel = dir:TypedModel {
        name = tmn,
        usedPackage = up:Package{},
        _transformation = rt
      },
      pattern = dp:DomainPattern {
        bindsTo = domainVars:Set(Variable) {},
        templateExpression = te:ObjectTemplateExp {
          bindsTo = tev:Variable {}
        }
      }
    } ++ rOppositeDomains
  }
};

enforce domain core m:Mapping {
  _transformation = mt,
  name = rn+'_'+irn+'_'+dn,
  guardPattern = mg:GuardPattern {
    area = m
  },
  bottomPattern = mb:BottomPattern {
    bindsTo = vs:Set(Variable) {
      tcv:RealizedVariable {} ++ mbVars
    }
  },
  _domain = md:CoreDomain {
    name = dn,
    isEnforceable = true,
    typedModel = mdir:TypedModel {
      name = tmn,
      usedPackage = up,
      _transformation = mt
    },
    guardPattern = dg:GuardPattern {
      bindsTo = dgVars:Set(Variable) {}
    },
    bottomPattern = db:BottomPattern {
      area = md
    }
  }
};

when {
  RelationTransformationToMappingTransformation(rt, mt);
}
where {
  allDomainVars = r._domain->iterate(md; acc:Set(QVTMM::RelationDomain)=Set{} |
    acc->including(md.oclAsType(QVTMM::RelationDomain))).pattern.bindsTo;
  whenVars = r._when.bindsTo;
}

```

```

whereVars = r._where.bindsTo;

-- Exclude where clause relation calls.
-- The predicate corresponding to a where clause relation call is included not in this
-- mapping but in the one corresponding to the invoked relation (refer to rule 2.6.1)
rpSet = r._where.predicate->reject(p |
    p.conditionExpression.oclIsTypeOf(QVTMM::RelationCallExp));

oppositeDomainVars = rOppositeDomains->iterate(d; vars: Set(QVTMM::Variable) = Set{} |
    vars->union(d.pattern.bindsTo));
sharedDomainVars = getSharedDomainVars(r);
domainBottomUnSharedVars =
    (domainVars - whenVars - sharedDomainVars)->excluding(tev);
unsharedWhereVars =
    (whereVars - whenVars - allDomainVars)->union(sharedDomainVars);
predicatesWithVarBindings =
    filterOutPredicatesThatReferToVars(rpSet, domainBottomUnSharedVars);
predicatesWithoutVarBindings = rpSet - predicatesWithVarBindings;
unsharedWhenVars = whenVars - allDomainVars;
domainTopVars = domainVars->intersection(whenVars)->including(tev);
rdSeq = Sequence{r, rd};
rdSet = Set{r, rd};
rdVarsSeq = Sequence{rdSet, oppositeDomainVars};
rdtSet = Set{r, rd, te};
rdtVarsSeq = Sequence{rdtSet, predicatesWithoutVarBindings, domainBottomUnSharedVars};
oppDomainSeq = Sequence{r, ir, rd};
seqForInvoker = Sequence{ir, ri, r};
relImplSeq = Sequence{r, rd};

RelationDomainToTraceClassVar(rdSeq, tcv);
if (unsharedWhereVars->isEmpty())
then
    mbVars = Set{}
else
    RVarSetToMVarSet(unsharedWhereVars->asSequence(), mbVars)
endif;
RPredicateSetToMBPredicateSet(predicatesWithVarBindings->asSequence(), mb);
RWhenPatternToMGuardPattern(r, mg);
RInvokerToMGuard(seqForInvoker, mg);
RVarSetToMVarSet(domainTopVars->asSequence(), dgVars);
RDomainToMDBottomForEnforcement(rdtVarsSeq, db);
ROppositeDomainVarsToTraceClassProps(rdVarsSeq, mb);
IROppositeDomainsToMappingForEnforcement(oppDomainSeq, m);
RRelImplToMDBottomEnforcementOperation(relImplSeq, mb);
}
}

relation RDomainToMDomainForChecking
{
    sharedDomainVars: Set(QVTMM::Variable);
    domainVarsSharedWithWhen: Set(QVTMM::Variable);
    domainBottomUnSharedVars: Set(QVTMM::Variable);
    seqForDomainPtrn: Sequence(QVTMM::Element);
    whenVars: Set(QVTMM::Variable);
    dn, tmn: String;
    rt: QVTMM::RelationTransformation;
    mt: QVTMM::Transformation;

    domain relations r:Relation {
        _domain = rd:RelationDomain {
            name = dn,
            isCheckable = true,
            typedModel = dir:TypedModel {
                name = tmn,
                usedPackage = up:Package{},
                _transformation = rt
            },
        },
        pattern = dp:DomainPattern {

```

```

        bindsTo = domainVars:Set(Variable){},
        templateExpression = te:ObjectTemplateExp {}
    }
}
};
enforce domain core m:Mapping {
    bottomPattern = mb:BottomPattern {
        area = m
    },
    _domain = md:CoreDomain {
        name = dn,
        isCheckable = true,
        typedModel = mdir:TypedModel {
            name = tmn,
            usedPackage = up,
            _transformation = mt
        },
        guardPattern = dg:GuardPattern {
            area = md
        },
        bottomPattern = db:BottomPattern {
            area = md
        }
    }
};
when {
    RelationTransformationToMappingTransformation(rt, mt);
}
where {
    whenVars = r._when.bindsTo;
    sharedDomainVars = getSharedDomainVars(r);
    domainVarsSharedWithWhen = domainVars->intersection(whenVars);
    domainBottomUnSharedVars = domainVars - whenVars - sharedDomainVars;
    seqForDomainPtrn = Sequence{r, te};

    DomainVarsSharedWithWhenToDgVars(domainVarsSharedWithWhen, dg);
    DomainBottomUnSharedVarsToDbVars(domainBottomUnSharedVars, db);
    RDomainPatternToMDBottomPattern(seqForDomainPtrn, db);
    RDomainVarsToTraceClassProps(rd, mb);
}
}

-- opposite domains of a top-level 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 IROppositeDomainsToMappingForEnforcement
{
    sharedDomainVars:Set(QVTMM::Variable);
    domainTopVars: Set(QVTMM::Variable);
    domainBottomUnSharedVars: Set(QVTMM::Variable);
    domainBottomSharedVars: Set(QVTMM::Variable);
    seqForDomainPtrn: Sequence(QVTMM::Element);
    dn, tmn: String;
    c: Boolean;
    mbVars:Set(QVTMM::Variable);
    whenVars:Set(QVTMM::Variable);
    rt: QVTMM::RelationTransformation;
    mt: QVTMM::Transformation;
    up: QVTMM::Package;

    domain relations oppDomainSeq:Sequence(Element) {
        r:Relation {
            _domain = rds:Set(RelationDomain) {
                ord:RelationDomain {          -- opposite domain
                    name = dn,

```

```

        typedModel = dir:TypedModel {
            name = tmn,
            usedPackage = up,
            _transformation = rt
        },
        isCheckable = c,
        pattern = dp:DomainPattern {
            bindsTo = domainVars:Set(Variable) {},
            templateExpression = te:ObjectTemplateExp {
                bindsTo = tev:Variable {}
            }
        } ++ -
    } ++ -
},
ir:Relation{},
rd:RelationDomain{}
++ -
} {
    ord <> rd
};
enforce domain core m:Mapping {
    _domain = cd:CoreDomain {
        name = dn,
        typedModel = mdir:TypedModel {
            name = tmn,
            usedPackage = up,
            _transformation = mt
        },
        isCheckable = c,
        isEnforceable = false,
        guardPattern = dg:GuardPattern {
            bindsTo = dgVars:Set(Variable) {}
        },
        bottomPattern = db:BottomPattern {
            bindsTo = dbVars:Set(Variable) {}
        }
    },
    bottomPattern = mb:BottomPattern {
        area = m
    }
};
when {
    RelationTransformationToMappingTransformation(rt, mt);
}
where {
    whenVars = r._when.bindsTo;
    domainTopVars = domainVars->intersection(whenVars)->including(tev);
    sharedDomainVars = getSharedDomainVars(r);
    domainBottomUnSharedVars = (domainVars - whenVars - sharedDomainVars)->excluding(tev);
    domainBottomSharedVars =
        (domainVars - whenVars)->intersection(sharedDomainVars)->excluding(tev);
    seqForDomainPtrn = Sequence{r, te};

    RVarSetToMVarSet(domainTopVars->asSequence(), dgVars);
    RVarSetToMVarSet(domainBottomUnSharedVars->asSequence(), dbVars);
    RVarSetToMVarSet(domainBottomSharedVars->asSequence(), mb);
    RDomainPatternToMDBottomPattern(seqForDomainPtrn, db);
}
}

-- opposite domains of an invoked relation's enforced domain are mapped as per rules
-- 4.2 and 4.5
--
relation TROppositeDomainsToMappingForEnforcement
{
    sharedDomainVars:Set(QVTMM::Variable);
    domainTopVars: Set(QVTMM::Variable);
}

```

```

domainBottomUnSharedVars: Set(QVTMM::Variable);
domainBottomSharedVars: Set(QVTMM::Variable);
seqForDomainPtrn: Sequence(QVTMM::Element);
dn, tmn: String;
c: Boolean;
mbVars:Set(QVTMM::Variable);
whenVars:Set(QVTMM::Variable);
rt: QVTMM::RelationTransformation;
mt: QVTMM::Transformation;
up: QVTMM::Package;

domain relations oppDomainSeq:Sequence(Element) {
  r:Relation {
    _domain = rds:Set(RelationDomain) {
      ord:RelationDomain {
        name = dn,
        typedModel = dir:TypedModel {
          name = tmn,
          usedPackage = up,
          _transformation = rt
        },
        isCheckable = c,
        pattern = dp:DomainPattern {
          bindsTo = domainVars:Set(Variable) {},
          templateExpression = te:ObjectTemplateExp {}
        }
      } ++ _
    }
  },
  rd:RelationDomain{}
} ++ _
} {
  ord <> rd
};
enforce domain core m:Mapping {
  _domain = cd:CoreDomain {
    name = dn,
    typedModel = mdir:TypedModel {
      name = tmn,
      usedPackage = up,
      _transformation = mt
    },
    isCheckable = c,
    isEnforceable = false,
    guardPattern = dg:GuardPattern {
      area = cd
    },
    bottomPattern = db:BottomPattern {
      bindsTo = dbVars:Set(Variable) {}
    }
  },
  bottomPattern = mb:BottomPattern {
    area = m
  }
};
where {
  whenVars = r._when.bindsTo;
  domainTopVars = domainVars->intersection(whenVars);
  sharedDomainVars = getSharedDomainVars(r);
  domainBottomUnSharedVars = domainVars - whenVars - sharedDomainVars;
  domainBottomSharedVars =
    (domainVars - whenVars)->intersection(sharedDomainVars);
  seqForDomainPtrn = Sequence{r, te};

  RelationTransformationToMappingTransformation(rt, mt);
  RVarSetToMVarSet(domainTopVars->asSequence(), dgVars);
  RVarSetToMVarSet(domainBottomUnSharedVars->asSequence(), dbVars);
  RVarSetToMVarSet(domainBottomSharedVars->asSequence(), mb);
}

```



```

    RDomainPatternToMDBottomPattern(seqForDomainPtrn, db);
  }
}

relation RWhenPatternToMGuardPattern
{
  allDomainVars: Set(QVTMM::Variable);
  unsharedWhenVars: Set(QVTMM::Variable);

  domain relations r:Relation{
    _when = whenp:Pattern {
      bindsTo = whenVars:Set(Variable) {}
    }
  };
  enforce domain core mg:GuardPattern {};
  where {
    allDomainVars = r._domain->iterate(md; acc:Set(QVTMM::RelationDomain)=Set{} |
      acc->including(md.oclAsType(QVTMM::RelationDomain)).pattern.bindsTo;
    unsharedWhenVars = whenVars - allDomainVars;

    RWhenRelCallToMGuard(whenp, mg);
    RSimplePatternToMPattern(whenp, mg);
    UnsharedWhenVarsToMgVars(unsharedWhenVars, mg);
  }
}

relation RVarSetToMVarSet
{
  rvRest: Sequence(QVTMM::Variable);
  mvRest: Set(QVTMM::Variable);

  domain relations rvSeq:Sequence(Variable) {rv:Variable {}++rvRest};
  enforce domain core mvSet:Set(Variable) {mv:Variable {}++mvRest};
  where {
    RVarToMVar(rv, mv);
    if (rvRest->isEmpty())
      then
        mvRest = Set{}
      else
        RVarSetToMVarSet(rvRest, mvRest)
    endif;
  }
}

relation RVarSetToMBVarSet
{
  rvRest: Sequence(QVTMM::Variable);
  mvRest: Set(QVTMM::Variable);

  domain relations rvSeq:Sequence(Variable) {rv:Variable {}++rvRest};
  enforce domain core mb:BottomPattern {
    bindsTo = mv:Variable {}
  };
  where {
    RVarToMVar(rv, mv);
    RVarSetToMBVarSet(rvRest, mb);
  }
}

relation RVarSetToDGVarSet
{
  rvRest: Sequence(QVTMM::Variable);
  mvRest: Set(QVTMM::Variable);

  domain relations rvSeq:Sequence(Variable) {rv:Variable {}++rvRest};
  enforce domain core dg:GuardPattern {
    bindsTo = mv:Variable {}
  };
}

```

```

    where {
      RVarToMVar(rv, mv);
      RVarSetToDGVarSet(rvRest, dg);
    }
  }

relation RVarToMVar
{
  n: String;

  domain relations rv:Variable {name=n, type=t:Type {}};
  enforce domain core mv:Variable {name=n, type=t};
}

relation RVarToMRealizedVar
{
  n: String;

  domain relations rv:Variable {name=n, type=t:Type {}};
  enforce domain core mv:RealizedVariable {name=n, type=t};
}

relation RSimplePatternToMPattern
{
  domain relations rp:Pattern {
    predicate = pd:Predicate {
      conditionExpression = re:OclExpression {}
    }
  }
  {
    not re.oclIsTypeOf(RelationCallExp)
  };
  enforce domain core mp:Pattern {
    predicate = mpd:Predicate{
      conditionExpression = me:OclExpression {}
    }
  };
  where {
    RExpToMExp(re, me);
  }
}

-- Relation invocation in when clause maps to a trace class pattern in mapping guard.
-- Relation call argument position corresponds to the domain position in the invoked relation.
-- Domain's root pattern object var gives us the corresponding trace class prop.
--
relation RWhenRelCallToMGuard
{
  domain relations rp:Pattern {
    predicate = pd:Predicate {
      conditionExpression = e:RelationCallExp {
        referredRelation = r:Relation {
          _domain = dseq:Sequence(RelationDomain) {}
        },
        argument = aseq:Sequence(VariableExp) {}
      }
    }
  };
  enforce domain core mp:GuardPattern {};

  where {
    aseq->forAll( a | RWhenRelCallArgToMGuardPredicate( Sequence{ r, a, dseq->at(aseq-
>indexOf(a)) }, mp) );
  }
}

relation RWhenRelCallArgToMGuardPredicate
{

```

```

tc: QVTMM::Class;
dvn: String;
mv:QVTMM::Variable;

domain relations daSeq:Sequence(Element) {
  r:Relation {},
  ve:VariableExp {
    referredVariable = v:Variable {}
  },
  d:RelationDomain {
    rootVariable = dv:Variable {name = dvn}
  }
  ++ -
};
enforce domain core mp:GuardPattern {
  bindsTo = vd:Variable {
    name = tc.name+'_v',
    type = tc
  },
  predicate = mpd:Predicate {
    conditionExpression = ee:OperationCallExp { -- vd.dvn = mv
      source = pe:PropertyCallExp {
        source = pve:VariableExp{referredVariable = vd},
        referredProperty = pep:Property{name = dvn, class =
vd.type.oclasType(QVTMM::Class)}
      },
      referredOperation = eo:Operation{name = '='},
      argument = ave:VariableExp{referredVariable = mv}
    }
  }
};
when {
  RelationToTraceClass(r, tc);
}
where {
  RVarToMVar(v, mv);
}
}

-- 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 RInvokerToMGuard
{
  domain relations seqForInvoker:Sequence(Element) {
    ir:Relation {}, -- invoking relation
    ri:RelationCallExp {
      argument = aseq:Sequence(VariableExp) {}
    },
    r:Relation { -- invoked relation
      _domain = dseq:Sequence(RelationDomain) {}
    }
  }
  ++ -
};
enforce domain core mg:GuardPattern {};
where {
  aseq->forall( a | RInvokerToMGuardPredicate( Sequence{ ir, a, dseq->at(aseq->indexOf(a))
}, mg) );
}
}

relation RInvokerToMGuardPredicate
{
  vn: String;
  tc: QVTMM::Class;
  mdv: QVTMM::Variable;
}

```

```

domain relations seqForInvoker:Sequence(Element) {
  ir:Relation {}, -- invoking relation
  ve:VariableExp {referredVariable = v:Variable {name=vn}},
  d:RelationDomain { rootVariable = dv:Variable {} }
  ++ -
};
enforce domain core mg:GuardPattern {
  bindsTo = vd:Variable {
    name = tc.name+'_v',
    type = tc
  },
  predicate = pd:Predicate {
    conditionExpression = ee:OperationCallExp { -- vd.vn = mdv
      source = pe:PropertyCallExp {
        source = mve:VariableExp{referredVariable = vd},
        referredProperty = pep:Property{name = vn, class =
vd.type.oclassType(QVTMM::Class)}
      },
      referredOperation = eo:Operation{name = '='},
      argument = ave:VariableExp{referredVariable = mdv}
    }
  }
};
when {
  RelationToTraceClass(ir, tc);
}
where {
  RVarToMVar(dv, mdv);
}
}

relation RDomainPatternToMDBottomPattern
{
  domain relations seqForDomainPtrn:Sequence(Element) {};
  enforce domain core db:BottomPattern {
    area = cd:CoreDomain{
      rule = m:Mapping {
        bottomPattern = mb:BottomPattern{area = m}
      }
    }
  }; -- domain bottom
  where {
    RDomainPatternToMDBottomPatternComposite(seqForDomainPtrn, db);
    RDomainPatternToMDBottomPatternSimpleNonVarExpr(seqForDomainPtrn, db);
    RDomainPatternToMDBottomPatternSimpleUnSharedVarExpr(seqForDomainPtrn, db);
    RDomainPatternToMDBottomPatternSimpleSharedVarExpr(seqForDomainPtrn, mb);
  }
}

relation RDomainToMDBottomForEnforcement
{
  remainingUnBoundDomainVars: Set(QVTMM::Variable);
  predicatesWithVarBindings: Set(QVTMM::Predicate);
  remainingPredicatesWithoutVarBindings: Set(QVTMM::Predicate);
  rdSeq, rtSeq, rtdSeq: Sequence(QVTMM::Element);
  rdtVarsSeqRest: Sequence(Set(QVTMM::Element));
  predicatesWithoutVarBindings: Set(QVTMM::Predicate);
  unboundDomainVars: Set(QVTMM::Variable);
  tcv, mv: QVTMM::Variable;

  domain relations rdtVarsSeq:Sequence(Set(Element)) {
    rdtSet:Set(Element) {
      r:Relation{},
      rd:RelationDomain{},
      te:ObjectTemplateExp {bindsTo = v:Variable {}}
      ++ -
    }
  }
  ++ -
}

```

```

};
enforce domain core db:BottomPattern { -- domain bottom
  area = cd:CoreDomain {
    rule = m:Mapping {
      bottomPattern = mb:BottomPattern {
        area = m
      }
    }
  }
};
where {
  predicatesWithoutVarBindings = rdtVarsSeq->at(2);
  unboundDomainVars = rdtVarsSeq->at(3);

  remainingUnBoundDomainVars = unboundDomainVars - Set{v};
  predicatesWithVarBindings = filterOutPredicatesThatReferToVars(
    predicatesWithoutVarBindings, remainingUnBoundDomainVars);

  remainingPredicatesWithoutVarBindings =
    predicatesWithoutVarBindings - predicatesWithVarBindings;
  rtSeq = Sequence{r, te};
  rtdSeq = Sequence{r, te, rd};
  rdtVarsSeqRest = Sequence{rdtSet, remainingPredicatesWithoutVarBindings,
remainingUnBoundDomainVars};

  RDomainToMDBottomForEnforcementOfIdentityProp(rtSeq, db);
  RDomainVarToMDBottomAssignmnetForEnforcement(rdtVarsSeq, mb);
  --RDomainToMDBottomForEnforcementOfIdentityPropObject(rdtSeq, mb);
  RDomainToMDBottomForEnforcementOfNonIdentityPropPrimitive(rtdSeq, m);
  RDomainToMDBottomForEnforcementOfNonIdentityPropObject(rdtVarsSeqRest, m);
  RDomainToMBottomPredicateForEnforcement(rdtVarsSeq, mb);
}
}

relation RDomainVarToMDBottomAssignmnetForEnforcement
{
  rdSeq : Sequence(QVTMM::Element);
  tcv, mv: QVTMM::Variable;

  domain relations rdtVarsSeq:Sequence(Set(Element)) {
    rdtSet:Set(Element) {
      r:Relation{},
      rd:RelationDomain{},
      te:ObjectTemplateExp {bindsTo = v:Variable {}}
    }
  }
};
enforce domain core mb:BottomPattern { -- domain bottom
  assignment = a:Assignment {
    slotExpression = ve1:VariableExp{referredVariable = tcv},
    targetProperty = tp:Property{name = v.name, class = tcv.type.oclAsType(QVTMM::Class)},
    value = ve2:VariableExp{referredVariable = mv}
  }
};
where {
  rdSeq = Sequence{r, rd};
  RelationDomainToTraceClassVar(rdSeq, tcv);
  RVarToMVar(v, mv);
}
}

relation RDomainToMBottomPredicateForEnforcement
{
  remainingUnBoundDomainVars: Set(QVTMM::Variable);
  predicatesWithVarBindings:Set(QVTMM::Predicate);
  rdSeq: Sequence(QVTMM::Element);
  predicatesWithoutVarBindings:Set(QVTMM::Predicate);
}

```

```

unboundDomainVars:Set(QVTMM::Variable);
tcv, mv: QVTMM::Variable;

domain relations rdtVarsSeq:Sequence(Set(Element)) {
  rdtSet:Set(Element) {
    r:Relation{},
    rd:RelationDomain{},
    te:ObjectTemplateExp {bindsTo = v:Variable {}}
  }
  ++ -
}
};
enforce domain core mb:BottomPattern {
  predicate = pd:Predicate {
    conditionExpression = ee:OperationCallExp { -- tcv.(v.name) = mv
      source = pe:PropertyCallExp {
        --source = tcv,
        source = pve:VariableExp{referredVariable = tcv},
        referredProperty = pep:Property{
          name = v.name,
          class = tcv.type.oclAsType(QVTMM::Class)
        }
      },
      referredOperation = eo:Operation{name = '='},
      argument = ave:VariableExp{referredVariable = mv}
    }
  }
};
where {
  rdSeq = Sequence{}->append(r)->append(rd);
  RelationDomainToTraceClassVar(rdSeq, tcv);
  RVarToMVar(v, mv);

  predicatesWithoutVarBindings = rdtVarsSeq->at(2);
  unboundDomainVars = rdtVarsSeq->at(3);

  remainingUnBoundDomainVars = unboundDomainVars - Set{v};
  predicatesWithVarBindings = filterOutPredicatesThatReferToVars(
    predicatesWithoutVarBindings, remainingUnBoundDomainVars);

  RPredicateSetToMBPredicateSet(predicatesWithVarBindings->asSequence(), mb);
}
}

relation RPredicateSetToMBPredicateSet
{
  rpRest: Sequence(QVTMM::Predicate);

  domain relations predSeq:Sequence(Predicate) {
    rp:Predicate {
      conditionExpression = re:OclExpression {}
    }
  }
  ++ rpRest
};
enforce domain core mb:BottomPattern {
  predicate = mp:Predicate {
    conditionExpression = me:OclExpression {}
  }
};
where {
  RExpToMExp(re, me);
  RPredicateSetToMBPredicateSet(rpRest, mb);
}
}

relation RDomainToMDBottomForEnforcementOfIdentityProp
{
  seqForAssignment: Sequence(QVTMM::Element);
}

```

```

domain relations rtSeq:Sequence(Element) {
  r:Relation{},
  te:ObjectTemplateExp {
    bindsTo = v:Variable {type=c:Class {}},
    part = pt:PropertyTemplateItem {
      referredProperty = pp:Property {},
      value = e:OclExpression {}
    }
  }
  {
    c._key.part->includes(pp)
  }
  ++ -
};
enforce domain core db:BottomPattern {
  area = cd:CoreDomain {
    rule = m:Mapping {
      bottomPattern = mb:BottomPattern{
        area = m
      }
    }
  }
}; -- domain bottom
where {
  seqForAssignment = Sequence{r, v, pp, e};
  RDomainPatternExprToMappingDomainAssignment(seqForAssignment, db);
  RDomainPatternExprToMappingDomainVarAssignment(seqForAssignment, db);
  RDomainPatternExprToMappingDomainTemplateVarAssignment(seqForAssignment, db);
  RDomainPatternExprToMappingBottomVarAssignment(seqForAssignment, mb);
}
}

relation RDomainToMDBottomForEnforcementOfIdentityPropObject
{
  seqForAssignment: Sequence(QVTMM::Element);
  mtv, tcv : QVTMM::Variable;
  rdSeq : Sequence(QVTMM::Element);

  domain relations rtSeq:Sequence(Element) {
    r:Relation{},
    rd:RelationDomain{},
    te:ObjectTemplateExp {
      bindsTo = v:Variable {type=c:Class {}},
      part = pt:PropertyTemplateItem {
        referredProperty = pp:Property {},
        value = e:ObjectTemplateExp {bindsTo = tv:Variable{}}
      }
    }
    {
      c._key.part->includes(pp)
    }
    ++ -
  };
  enforce domain core mb:BottomPattern {
    assignment = a:Assignment {
      slotExpression = ve1:VariableExp{referredVariable = tcv},
      targetProperty = tp:Property{name = tv.name, class =
tcv.type.oclAsType(QVTMM::Class)},
      value = ve2:VariableExp{referredVariable = mtv}
    }
  }; -- domain bottom
  where {
    rdSeq = Sequence{r, rd};
    RelationDomainToTraceClassVar(rdSeq, tcv);
    RVarToMVar(tv, mtv);
  }
}

relation RDomainPatternExprToMappingDomainAssignment

```

```

{
  pn: String;
  mv: QVTMM::Variable;

  domain relations seqForAssignment: Sequence(Element) {
    v:Variable {},
    pp:Property {name = pn},
    e:OclExpression {} {
      not e.oclIsTypeOf(VariableExp) and not e.oclIsTypeOf(ObjectTemplateExp)
    } ++ -
  };
  enforce domain core db:BottomPattern {
    assignment = a:Assignment {
      slotExpression = ve:VariableExp{referredVariable = mv},
      targetProperty = tp:Property{name = pn, class = mv.type.oclAsType(QVTMM::Class)},
      value = me:OclExpression{}
    }
  };
  where {
    RVarToMVar(v, mv);
    RExpToMExp(e, me);
  }
}

relation RDomainPatternExprToMappingDomainVarAssignment
{
  sharedDomainVars: Set(QVTMM::Variable);
  rev, mev : QVTMM::Variable;
  pn: String;

  domain relations seqForAssignment: Sequence(Element) {
    r:Relation {},
    v:Variable {},
    pp:Property {name = pn},
    e:VariableExp {referredVariable = rev}
  } ++ -
  {
    not sharedDomainVars->includes(e.referredVariable)
  };
  enforce domain core db:BottomPattern {
    realizedVariable = mv:RealizedVariable {},
    assignment = a:Assignment {
      slotExpression = ve:VariableExp{referredVariable = mv},
      targetProperty = tp:Property{name = pn, class = mv.type.oclAsType(QVTMM::Class)},
      value = me:VariableExp{referredVariable = mev}
    }
  };
  when {
    sharedDomainVars = getSharedDomainVars(r);
  }
  where {
    RVarToMRealizedVar(v, mv);
    RVarToMVar(rev, mev);
  }
}

relation RDomainPatternExprToMappingDomainTemplateVarAssignment
{
  sharedDomainVars: Set(QVTMM::Variable);
  rev, mev: QVTMM::Variable;
  pn: String;

  domain relations seqForAssignment: Sequence(Element) {
    r:Relation {},
    v:Variable {},
    pp:Property {name = pn},
    e:ObjectTemplateExp {bindsTo = rev}
  }
}

```



```

    ++ -
  } {
    not sharedDomainVars->includes(rev)
  };
enforce domain core db:BottomPattern {
  realizedVariable = mv:RealizedVariable {},
  assignment = a:Assignment {
    slotExpression = ve:VariableExp{referredVariable = mv},
    targetProperty = tp:Property{name = pn, class = mv.type.oclAsType(QVTMM::Class)},
    value = me:VariableExp{referredVariable = mev}
  }
};
when {
  sharedDomainVars = getSharedDomainVars(r);
}
where {
  RVarToMRealizedVar(v, mv);
  RVarToMVar(rev, mev);
}
}

relation RDomainPatternExprToMappingBottomVarAssignment
{
  sharedDomainVars: Set(QVTMM::Variable);
  rev, mev : QVTMM::Variable;
  pn: String;

  domain relations seqForAssignment: Sequence(Element) {
    r:Relation {},
    v:Variable {},
    pp:Property {name = pn},
    e:VariableExp {referredVariable = rev}
  }
  ++ -
  {
    sharedDomainVars->includes(e.referredVariable)
  };
enforce domain core mb:BottomPattern {
  realizedVariable = mv:RealizedVariable {},
  assignment = a:Assignment {
    slotExpression = ve:VariableExp{referredVariable = mv},
    targetProperty = tp:Property{name = pn, class = mv.type.oclAsType(QVTMM::Class)},
    value = me:VariableExp{referredVariable = mev}
  }
};
when {
  sharedDomainVars = getSharedDomainVars(r);
}
where {
  RVarToMRealizedVar(v, mv);
  RVarToMVar(rev, mev);
}
}

relation RDomainToMDBottomForEnforcementOfNonIdentityPropPrimitive
{
  pn: String;
  mv: QVTMM::Variable;
  rtdeSeq: Sequence(QVTMM::Element);

  domain relations rtdSeq:Sequence(Element) {
    r:Relation{
      _transformation = rt:RelationTransformation{}
    },
    te:ObjectTemplateExp {
      bindsTo = v:Variable {type = c:Class {}},
      part = pt:PropertyTemplateItem {
        referredProperty = pp:Property {name = pn},
        value = e:OclExpression {}
      }
    }
  }
}

```

```

    } {
    } {
      (not c._key.part->includes(pp)) and (not e.oclIsKindOf(TemplateExp))
    },
  rd:RelationDomain {
    pattern = rdp:DomainPattern {
      templateExpression = rdt:ObjectTemplateExp {}
    }
  }
  ++ -
};
enforce domain core m:Mapping {
  local = cm:Mapping {
    name = m.name+'_forNonIdentityProp',
    _transformation = mt:Transformation{},
    bottomPattern = bp:BottomPattern {
      assignment = a:Assignment {
        slotExpression = ve:VariableExp{referredVariable = mv},
        targetProperty = tp:Property{name = pn, class =
mv.type.oclAsType(QVTMM::Class)},
        value = me:OclExpression{}
      }
    }
  }
};
when {
  RelationTransformationToMappingTransformation(rt, mt);
}
where {
  RVarToMVar(v, mv);
  RExpToMExp(e, me);

  rtdeSeq = Sequence{r, te, rd, e};

  RDomainToMComposedMappingGuard(rtdeSeq, cm);
}
}

relation RDomainToMComposedMappingGuard
{
  pn, dn, tmn: String;
  tcv, mv: QVTMM::Variable;
  rdSeq: Sequence(QVTMM::Element);
  mt: QVTMM::Transformation;

  domain relations rtdSeq:Sequence(Element) {
    r:Relation{
      _transformation = rt:RelationTransformation{}
    },
    te:ObjectTemplateExp {},
    rd:RelationDomain {
      name = dn,
      typedModel = dir:TypedModel {
        name = tmn,
        usedPackage = up:Package{},
        _transformation = rt
      },
      pattern = rdp:DomainPattern {
        templateExpression = rdt:ObjectTemplateExp {}
      }
    },
    ve:VariableExp {referredVariable = v:Variable {}}
  }
  ++ -
} {
  isVarBoundToSomeOtherTemplate(rdt, te, v)
};
enforce domain core cm:Mapping {
  guardPattern = mg:GuardPattern {

```

```

    predicate = pd:Predicate {
      conditionExpression = ee:OperationCallExp { -- vd.vn = mdv
        source = pe:PropertyCallExp {
          source = vel:VariableExp{referredVariable = tcv},
          referredProperty = tp:Property {
            name = mv.name,
            class = mv.type.oclAsType(QVTMM::Class)
          }
        },
        referredOperation = eo:Operation{name = '='},
        argument = ve2:VariableExp{referredVariable = mv}
      }
    },
  },
  _domain = cd:CoreDomain {
    name = dn,
    typedModel = mdir:TypedModel {
      name = tmn,
      usedPackage = up,
      _transformation = mt
    },
    guardPattern = cmdg:GuardPattern {
      bindsTo = mv
    }
  }
};
when {
  RelationTransformationToMappingTransformation(rt, mt);
}
where {
  rdSeq = Sequence{r, rd};
  RelationDomainToTraceClassVar(rdSeq, tcv);
  RVarToMVar(v, mv);
}
}

relation RDomainToMDBottomForEnforcementOfNonIdentityPropObject
{
  rdtSetNext: Set(QVTMM::Element);
  rdtVarsSeqRest: Sequence(Set(QVTMM::Element));
  predicatesWithoutVarBindings:Set(QVTMM::Predicate);
  unboundDomainVars:Set(QVTMM::Variable);
  dn, pn, tmn: String;
  mv: QVTMM::Variable;

  domain relations rdtVarsSeq:Sequence(Set(Element)) {
    rdtSet:Set(Element) {
      r:Relation{
        _transformation = rt:RelationTransformation{}
      },
      rd:RelationDomain {
        name = dn,
        typedModel = dir:TypedModel {
          name = tmn,
          usedPackage = up:Package{},
          _transformation = rt
        }
      },
      te:ObjectTemplateExp {
        bindsTo = v:Variable {type = c:Class {}},
        part = pt:PropertyTemplateItem {
          referredProperty = pp:Property {name = pn},
          value = pte:ObjectTemplateExp {bindsTo = pv:Variable {}}
        }
      } {
        not c._key.part->includes(pp)
      }
    }
  }
}
++ _

```

```

    }
    ++ -
};
enforce domain core m:Mapping {
  local = cm:Mapping {
    name = m.name+'_for_'+pv.name,
    _transformation = mt:Transformation{},
    _domain = cd:CoreDomain {
      name = dn,
      isEnforceable = true,
      typedModel = mdir:TypedModel {
        name = tmn,
        usedPackage = up,
        _transformation = mt
      },
      bottomPattern = cmdb:BottomPattern {
        realizedVariable = mpv:RealizedVariable {},
        assignment = a:Assignment {
          slotExpression = vel:VariableExp{referredVariable = mv},
          targetProperty = tp:Property{name = pn, class =
mv.type.oclAsType(QVTMM::Class)},
          value = ve2:VariableExp{referredVariable = mpv}
        }
      },
      bottomPattern = mb:BottomPattern {
        area = cm
      }
    }
  };
  when {
    RelationTransformationToMappingTransformation(rt, mt);
  }
  where {
    RVarToMVar(v, mv);
    RVarToMRealizedVar(pv, mpv);

    predicatesWithoutVarBindings = rdtVarsSeq->at(2);
    unboundDomainVars = rdtVarsSeq->at(3);

    rdtSetNext = Set{r, rd, pte};
    rdtVarsSeqRest = Sequence{rdtSetNext, predicatesWithoutVarBindings, unboundDomainVars};
    RDomainToMDBottomForEnforcement(rdtVarsSeqRest, cmdb);
  }
}

relation RDomainPatternToMDBottomPatternComposite
{
  nextSeqForDomainPtrn: Sequence(QVTMM::Element);
  sharedDomainVars:Set(QVTMM::Variable);
  pn: String;
  mvte, mvpte: QVTMM::Variable;

  domain relations seqForDomainPtrn:Sequence(Element) {
    r:Relation{},
    te:ObjectTemplateExp {
      bindsTo = vte:Variable {},
      part = pt:PropertyTemplateItem {
        referredProperty = pp:Property {name = pn},
        value = pte:ObjectTemplateExp {bindsTo = vpte:Variable {}}
      }
    }
  }
  ++ -
};
enforce domain core db:BottomPattern {
  assignment = a:Assignment {
    slotExpression = vel:VariableExp{referredVariable = mvte},
    targetProperty = tp:Property{name = pn, class = mvte.type.oclAsType(QVTMM::Class)},

```

```

        value = ve2:VariableExp{referredVariable = mvpte}
    }
};
where {
    RVarToMVar(vte, mvte);
    RVarToMVar(vppte, mvpppte);
    nextSeqForDomainPtrn = Sequence{r, pte};
    RDomainPatternToMDBottomPattern(nextSeqForDomainPtrn, db);
}
}

relation RDomainPatternToMDBottomPatternSimpleUnSharedVarExpr
{
    sharedDomainVars: Set(QVTMM::Variable);
    pn: String;
    mvte, mvpppte: QVTMM::Variable;

    domain relations seqForDomainPtrn:Sequence(Element) {
        r:Relation{},
        te:ObjectTemplateExp {
            bindsTo = vte:Variable {},
            part = pt:PropertyTemplateItem {
                referredProperty = pp:Property {name = pn},
                value = e:VariableExp {referredVariable = vppte:Variable {}}
            }
        }
    } ++ -
} {
    not sharedDomainVars->includes(vppte)
};
enforce domain core db:BottomPattern {
    assignment = a:Assignment {
        slotExpression = ve1:VariableExp{referredVariable = mvte},
        targetProperty = tp:Property{name = pn, class = mvte.type.oclAsType(QVTMM::Class)},
        value = ve2:VariableExp{referredVariable = mvpppte}
    }
};
when {
    sharedDomainVars = getSharedDomainVars(r);
}
where {
    RVarToMVar(vte, mvte);
    RVarToMVar(vppte, mvpppte);
}
}

relation RDomainPatternToMDBottomPatternSimpleSharedVarExpr
{
    sharedDomainVars: Set(QVTMM::Variable);
    pn: String;
    mvte, mvpppte: QVTMM::Variable;

    domain relations seqForDomainPtrn:Sequence(Element) {
        r:Relation{},
        te:ObjectTemplateExp {
            bindsTo = vte:Variable {},
            part = pt:PropertyTemplateItem {
                referredProperty = pp:Property {name = pn},
                value = e:VariableExp {referredVariable=vppte:Variable {}}
            }
        }
    } ++ -
} {
    sharedDomainVars->includes(vppte)
};
enforce domain core mb:BottomPattern {
    assignment = a:Assignment {
        slotExpression = ve1:VariableExp{referredVariable = mvte},

```

```

        targetProperty = tp:Property{name = pn, class = mvte.type.oclAsType(QVTMM::Class)},
        value = ve2:VariableExp{referredVariable = mvpte}
    }
};
when {
    sharedDomainVars = getSharedDomainVars(r);
}
where {
    RVarToMVar(vte, mvte);
    RVarToMVar(vppte, mvpte);
}
}

relation RDomainPatternToMDBottomPatternSimpleNonVarExpr
{
    pn: String;
    mvte: QVTMM::Variable;

    domain relations seqForDomainPtrn:Sequence(Element) {
        -'
        te:ObjectTemplateExp {
            bindsTo = vte:Variable {},
            part = pt:PropertyTemplateItem {
                referredProperty = pp:Property {name = pn},
                value = e:OclExpression {}
            }
        } ++ -
    } {
        not e.oclIsKindOf(TemplateExp) and not e.oclIsTypeOf(VariableExp)
    };
    enforce domain core db:BottomPattern {
        assignment = a:Assignment {
            slotExpression = ve:VariableExp{referredVariable = mvte},
            targetProperty = tp:Property{name = pn, class = mvte.type.oclAsType(QVTMM::Class)},
            value = me:OclExpression{}
        }
    };
    where {
        RVarToMVar(vte, mvte);
        RExpToMExp(e, me);
    }
}

relation RDomainVarsToTraceClassProps
{
    tcv, mdv: QVTMM::Variable;

    domain relations rd:RelationDomain {
        rule = r:Relation{},
        pattern = dp:DomainPattern {
            bindsTo = domainVars:Set(Variable) {dv:Variable {templateExp = te: TemplateExp{}}++_}
        }
    };
    enforce domain core mb:BottomPattern {
        assignment = a:Assignment {
            slotExpression = ve1:VariableExp{referredVariable = tcv},
            targetProperty = tp:Property{name = dv.name, class =
            tcv.type.oclAsType(QVTMM::Class)},
            value = ve2:VariableExp{referredVariable = mdv}
        }
    };
    where {
        RelationToTraceClassVar(r, tcv);
        RVarToMVar(dv, mdv);
    }
}

relation ROppositeDomainVarsToTraceClassProps

```

```

{
  rdSeq: Sequence(QVTMM::Element);
  tcv, mdv: QVTMM::Variable;

  domain relations rdVarsSeq:Sequence(Set(Element)) {
    rdSet: Set(Element) {
      r:Relation {},
      rd:RelationDomain {} ++ _
    },
    domainVars:Set(Variable) {dv:Variable{templateExp = te:TemplateExp{}} ++ _}
    ++ _
  };
  enforce domain core mb:BottomPattern {
    assignment = a:Assignment {
      slotExpression = ve1:VariableExp{referredVariable = tcv},
      targetProperty = tp:Property{name = dv.name, class =
tcv.type.oclAsType(QVTMM::Class)},
      value = ve2:VariableExp{referredVariable = mdv}
    }
  };
  where {
    rdSeq = Sequence{}->append(r)->append(rd);
    RelationDomainToTraceClassVar(rdSeq, tcv);
    RVarToMVar(dv, mdv);
  }
}

relation RRelImplToMBottomEnforcementOperation
{
  emptySet:Set(QVTMM::EnforcementOperation);

  domain relations repImplSeq:Sequence(Element) {
    r:Relation {
      operationalImpl = ri:RelationImplementation {
        inDirectionOf = tm:TypedModel{},
        impl = op:Operation{}
      }
    },
    rd:RelationDomain {typedModel = tm:TypedModel{}}
    ++ _
  };
  enforce domain core mb:BottomPattern {
    enforcementOperation = eoSet:Set(EnforcementOperation) {
      eoc:EnforcementOperation {
        enforcementMode = 'Creation',
        operationCallExp = oce:OperationCallExp {
          referredOperation = op
        }
      },
      eod:EnforcementOperation {
        enforcementMode = 'Deletion',
        operationCallExp = oce
      }
    }
    ++ emptySet
  }
}
default_values
{
  emptySet = Set{};
};
where {
  RRelDomainsToMOpCallArg(r, oce);
}
}

relation RRelDomainsToMOpCallArg
{
  domain relations r:Relation {

```

```

    _domain = rd:RelationDomain {
      pattern = p:DomainPattern{bindsTo = rv:Variable{}}
    }
  };
  enforce domain core oce:OperationCallExp {
    argument = ar:VariableExp {
      referredVariable = mv:Variable {}
    }
  };
  where {
    RVarToMVar(rv, mv);
  }
}

relation RelationToTraceClassVar
{
  rn: String;
  tc: QVTMM::Class;

  domain relations r:Relation {name = rn};
  enforce domain core tcv:RealizedVariable {
    name = rn+'_v',
    type = tc
  };
  when {
    RelationToTraceClass(r, tc);
  }
}

relation RelationDomainToTraceClassVar
{
  rn, dn: String;
  tc: QVTMM::Class;

  domain relations rdSeq:Sequence(Element) {
    r:Relation {name = rn},
    d:RelationDomain{name = dn}
    ++ -
  };
  enforce domain core tcv:RealizedVariable {
    name = rn+'_'+dn+'_v',
    type = tc
  };
  when {
    RelationToTraceClass(r, tc);
  }
}

-- copy an ocl expression
-- For space reasons this relation is not expanded out here
relation RExpToMExp
{
  domain relations re:OclExpression{};
  enforce domain core me:OclExpression{} implementedby CopyOclExpression(re, me);
}
}

```


11 QVT For CMOF

For the sake of simplicity all previous clauses assume QVT used in the context of EMOF conformant metamodels. However this specification is also applicable to CMOF metamodels with a few restrictions.

11.1 The QVT Metamodel for CMOF

The QVT metamodel for CMOF is a CMOF metamodel that is obtained by executing the following steps:

- The EMOF package is replaced by the CMOF Package.
- All other packages - including the EssentialOCL - are cloned, with the exception that all references to the original EMOF metaclasses are replaced by references to the corresponding CMOF metaclass.

11.2 Semantics Specificities

The semantics of CMOF concerning the access and the modification of properties replaces the semantics of EMOF. For instance, in CMOF, setting a property that is specified as an association end implies that the corresponding association link instance is created and that any related sub-setted association is updated accordingly.

There are some limitations when using QVT on CMOF metamodels which comes from the fact that we are cloning EssentialOCL - at the time being, the OCL specification does not define an “OCL for CMOF metamodels.”

It is not possible to refer directly to an association; instead an association has to be accessed as a property from one of the owning classes. However, this does not address the case where both the ends of an association are owned by the association itself.

Annex A: Additional Examples

(normative)

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.

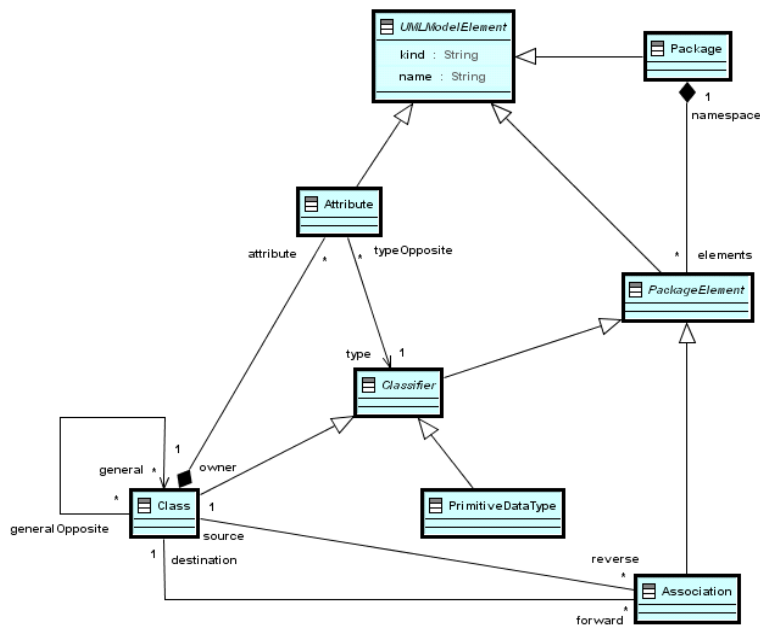


Figure A.1 - Simple UML Metamodel

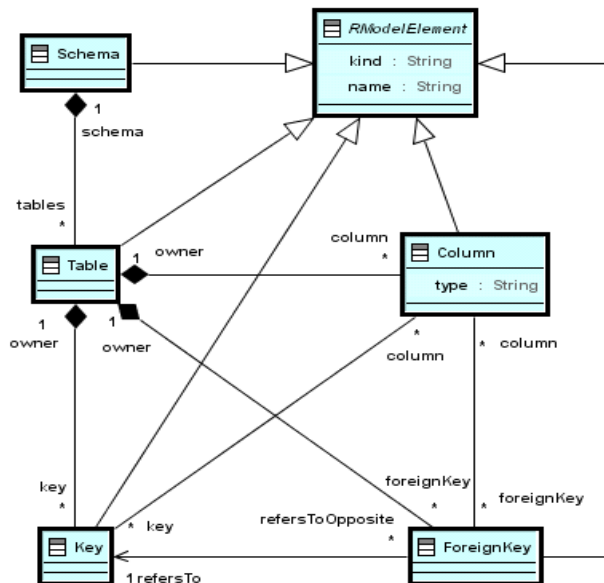


Figure A.2 - Simple RDBMS Metamodel

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 'Key';
                                -- 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, kind='primary'}};
    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 AssocToFKKey
{
  srcTbl, destTbl: SimpleRDBMS::Table;
  pKey: SimpleRDBMS::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'->select(kind='primary');
}
where {
    fkn=scn+'_'+an+'_'+dcn;
    fcn=fkn+'_tid';
}
}

query 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

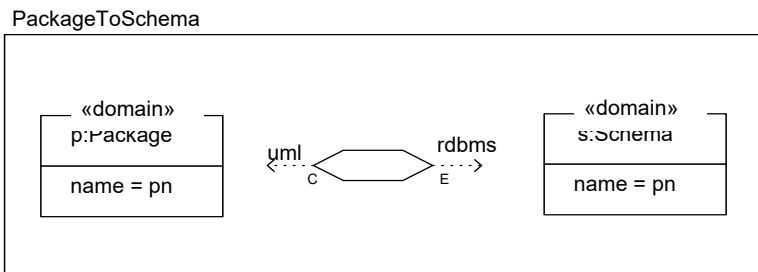


Figure A.3 - PackageToSchema relation

ClassToTable

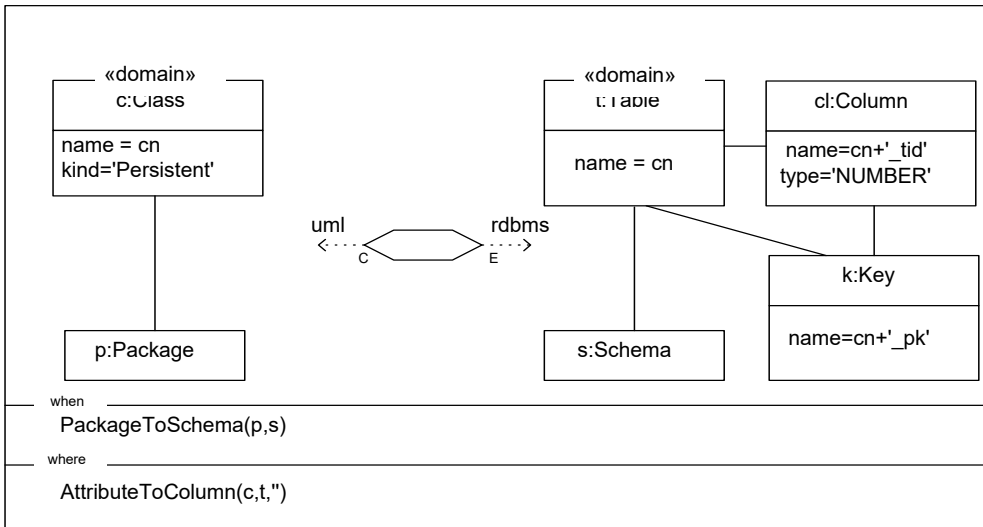


Figure A.4 - ClassToTable relation

PrimitiveAttributeToColumn

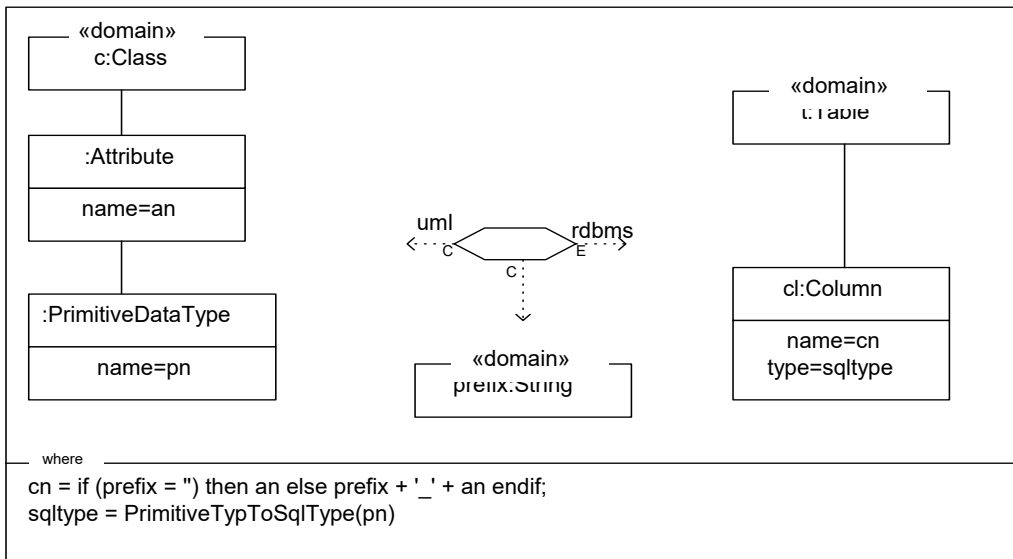


Figure A.5 - PrimitiveAttributeToColumn relation

ComplexAttributeToColumn

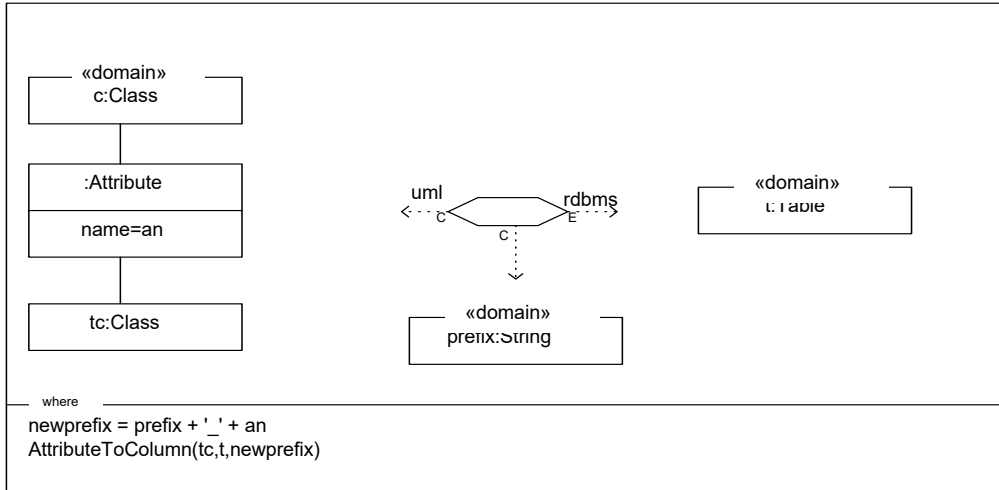


Figure A.6 - ComplexAttributeToColumn relation

SuperAttributeToColumn

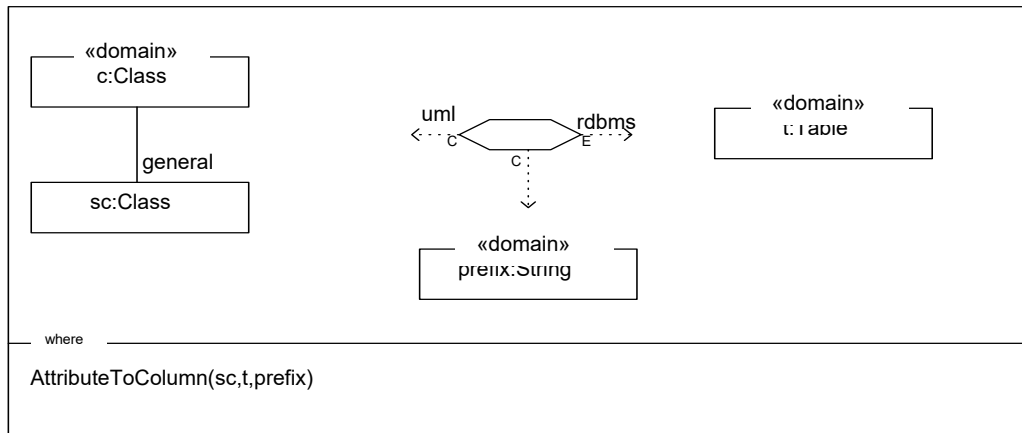


Figure A.7 - SuperAttributeToColumn relation

AssocToFKey

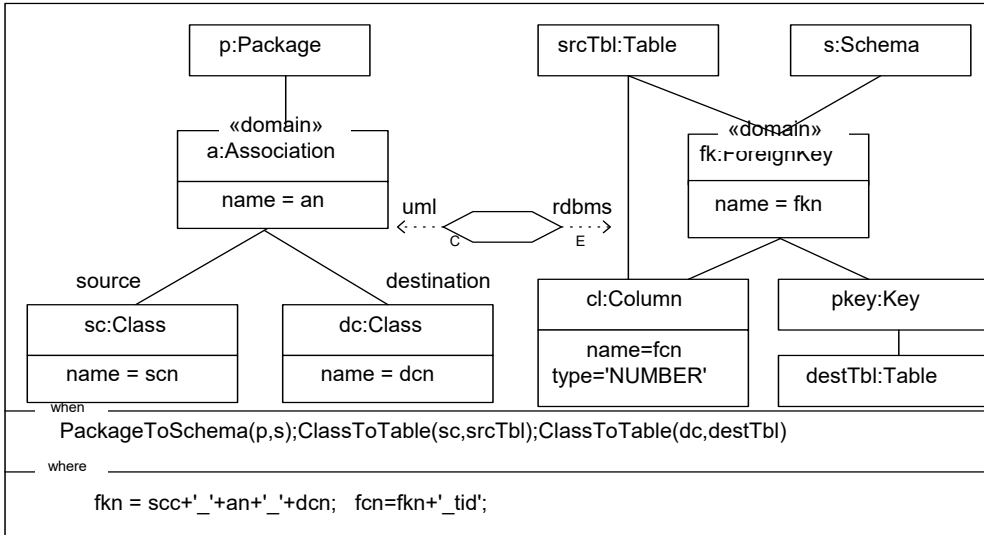


Figure A.8 - AssocToFKey relation

A.2 Operational Mapping Examples

A.2.1 Book To Publication example

```

metamodel BOOK {
  class Book {title: String; composes 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->objectsOfType(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);
  
```

```

// Indicating that UML1.4 Name type is to be treated as a String
tag "TypeEquivalence" UML::Name = "String";

-- 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.feature[Attribute];}
  operation := { -- assignment with additive semantics
    attrs->object(a) Operation {
      name := "get_" + self.name.firstToUpper();
      visibility := "public";
      type := a.type;
    };
    attrs->object(a) Operation {
      name := "set_" + self.name.firstToUpper();
      visibility := "public";
      parameter := object Parameter {
        name := 'a_' + self.name.firstToUpper();
        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

The metamodels used here are the same metamodels used for the relational version given in Appendix A.1.1. We provide below their definition using the concrete syntax for metamodels. Note we are assuming that all multi-valued associations are ordered.

```

metamodel SimpleUml {

  abstract class UMLModelElement {
    kind : String;
    name : String;
  }

  class Package extends UMLModelElement {
    composes elements : PackageElement [*] ordered opposites namespace [1];
  }

  abstract class PackageElement extends UMLModelElement {

```

```

}

class Classifier extends PackageElement {
}

class Attribute extends UMLModelElement {
  references type : Classifier [1];
}

class Class extends Classifier {
  composes attribute : Attribute [*] ordered opposites owner [1];
  references general : Classifier [*] ordered;
}

class Association extends PackageElement {
  source : Class [1] opposites reverse [*];
  destination : Class [1] opposites forward [*];
}

class PrimitiveDataType extends Classifier {
}
}

metamodel SimpleRdbms {
  abstract class RModelElement {
    kind : String;
    name : String;
  }

  class Schema extends RModelElement {
    composes tables : Table [*] ordered opposites schema [1];
  }

  class Table extends RModelElement {
    composes column : Column [*] ordered opposites owner[1];
    composes _key : Key [*] ordered opposites owner[1];
    // '_key' is an automatic alias for 'key'
    composes foreignKey : ForeignKey [*] ordered opposites owner[1];
  }

  class Column extends RModelElement {
    type : String;
  }

  class Key extends RModelElement {
    references column : Column [*] ordered opposites _key [*];
  }

  class ForeignKey extends RModelElement {
    references refersTo : Key [1];
    references column : Column [*] ordered opposites foreignKey [*];
  }
}

Below the transformation definition
transformation Uml2Rdb(in srcModel:UML,out dest:RDBMS);

-- Aliases to avoid name conflicts with keywords
tag "alias" RDBMS::Table::key_ = "key";
-- defining intermediate data to reference leaf attributes that may
-- appear when struct data types are used
intermediate class LeafAttribute {
  name:String;
  kind:String;
  attr:UML::Attribute;
};

```

```

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

-- defining specific helpers

query UML::Association::isPersistent() : Boolean {
    result = (self.source.kind='persistent' and self.destination.kind='persistent');
}

-- 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 initialization
            self.leafAttributes := self.attribute
                ->map attr2LeafAttrs("", ""); // ->flatten();
        }
        -- 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 := result.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.oclIsKindOf(PrimitiveDataType)
            then -- creates a sequence with a LeafAttribute instance
                Sequence {
                    object LeafAttribute {attr:=self;name:=prefix+self.name;kind:=k;}
                }
            else self.type.oclAsType(Class).attribute
                ->map attr2LeafAttrs(self.name+"_", k)->asSequence()
            endif;
    }
}

-- Mapping that creates an ordinary column 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() : ForeignKey {
  name := 'f_' + self.name;
  refersTo := self.source.resolveone(Table).key_;
  column := self.source.leafAttributes[kind='primary']
    ->map leafAttr2ForeignColumn(self.source.name+'_');
}

-- Mapping to create a Foreign key from a leaf 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 between 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 Annex 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 stringWithVarchar 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;
}

abstract 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;
    }
  }
}

abstract map classAttributes in umlRdbms refines attributes {
  where (fao:ClassToTable| fao.umlClass=c) {}
  map {
    where {
      fa.name := a.name;
      a.name := fa.name;
    }
  }
}

abstract 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};
    }
}

abstract 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;
    }
  }
}

abstract 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,
primitiveAttribute {}
map complexAttributeComplexAttributes in umlRdbms refines complexAttributeAttributes,
complexAttribute {}

-- 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: Semantics of Relations

(normative)

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 Clause 7.10.3 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)
```

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 clause 'Restrictions on
      // expressions used in the relational 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 “one” clashes if the object being assigned is different from the one that already exists.

createOrUpdate(objectTemplate, unboundObjectVar, boundVariableContext): 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_l_pattern> and <domain_l_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

```


INDEX

-- comment 38
_ variable 31

Symbols

::= operator 128
:= operator 128, 130
! operator 127
!= operator 162
[...] shorthand 126
@ operator 108, 116, 135
* stereotype operator 162
/* comment 38, 161
operator 162
operator 162
% operator 162
+ operator 163
+= operator 130
+= shorthand 163
<> operator 163
= operator 163
== operator 162
> operator 126
~ operator 164

A

abstract 28
access 89, 94, 99, 100
alias 13, 85, 160
allInstances 12, 160
Alternative expression 127
AltExp 127
Any 139
Area 4, 184
asList 153
assert 134
Assert expression 134
AssertExp 139
assertion 86, 105
AssertionFailed 139
AssignExp 129
Assignment 187
assignment expression 129
Assignments 180
asType 139
atomic update 19

B

Bag 154
Bindings 175
Black box implementations 10
Black-box MOF Operation 10
Black-box operations with relations 19
Block expression 121
BlockExp 121
Bottom Pattern 4
Bottom pattern 185
Bottom patterns 176
BottomPattern 185

break 133
Break expression 133
BreakExp 133

C

candidate mapping 74, 81, 106
Catch expression 131
CatchExp 131
Change propagation 19
Check 15
Checking 176
Checking semantics 21
checkonly 15
Checkonly mode 20
Class 76, 102, 106
class 163
class2table 71
clone 153
CMOF-compliant 2
Collection 153
Collection Patterns 23
CollectionTemplateExp 31
Comments 38
Compilation unit 38
complex assignment 129
composes 164
Composing Transformations 81
Composition 182
compute 121
Compute expression 121
ComputeExp 121
configuration 94
Conformance 1
Constructor 103
constructor 104, 109
Constructor body 109
containment 67, 131
Contextual property 104
ContextualProperty 104
continue 133
Continue expression 133
Core 9
Core Domain 4
Core domain 185
Core metamodel 9
Core pattern 183
Core Transformation 4
CoreDomain 185
CorePattern 183
Creation 189

D

DataType 76, 102, 106
Declarative architecture 9
deepclone 154
default 130
deferred assignment 114, 129
Deletion 189
derived 164
Dictionary 144
Dictionary literal expression 137

- Dictionary type 136
- dictionary types 85
- direction 15
- Direction kind 100
- DirectionKind 100
- disjuncting mapping 74, 82
- disjuncts 81
- do 121
- document URI 13
- Domain 4, 27
- DomainPattern 36
- Domains 14
- Dynamic definition 87

E

- EBNF 164
- Element 140
- elif 127
- else 127
- EMOF 12
- EMOF-compliant 2
- end 110
- end section 70, 105
- endif 127
- Enforce 15
- enforced 15
- Enforcement 178, 181
- Enforcement mode 189
- Enforcement operation 189
- Enforcement semantics 21
- EnforcementMode 189
- EnforcementOperation 189
- Entry operation 101
- EntryOperation 101
- enum 163
- error 134
- EssentialOCL 12
- except 131, 132
- exceptClause 131
- Exception 139
- exception 164
- exceptionVariable 131
- executed-mapping 74
- explicit disjuncts 82
- Expressions 18
- Expressions syntax 39
- extend 89
- extends 94, 99, 163
- extension 100
- Extent 66

F

- fatal 134
- For expression 122
- forEach 122, 123
- forEach shorthand 135
- ForExp 122
- forOne 122, 123
- from 161
- Function 28
- FunctionParameter 29

- Functions 182

G

- Graphical syntax 40
- Guard Pattern 4
- Guard pattern 185
- GuardPattern 185
- Guards 176

H

- Helper 73, 102
- helper 102, 103

I

- id 164
- Identifying Property 4
- if 127
- Imperative call expression 111
- Import kind 100
- imperative collect shorthand 112, 127
- Imperative expression 117
- Imperative iterate expression 124
- Imperative loop expression 122
- Imperative operation 101
- ImperativeCallExp 111
- ImperativeExpression 117
- ImperativeIterateExp 124
- ImperativeLoopExp 122
- ImperativeOCL package 117
- ImperativeOperation 101
- implicit disjunct 82
- implicit disjunction 105
- import 161
- import statement 13
- ImportKind 100
- in 66, 97, 100, 102, 106
- Incremental Update 4
- inherit 84
- inhibition section 70, 75
- init 110
- init section 70
- initialization section 75, 105
- inout 66, 97, 100, 102, 106
- In-place transformations 19
- Instantiation expression 135
- instantiation section 70, 74, 75
- Integer 160
- intermediate 104
- Intermediate Data 73
- interoperability dimension 1, 2
- invalid 86
- invoked-mapping 74
- invresolve 77, 79
- invresolveIn 115
- invresolveone 78
- invresolveoneIn 78, 115
- isKindOf 139
- issues/problems xx
- isTypeOf 139

J

- Java Native Interface (JNI) 10

K

Key 4, 17, 36
key 33
Keys 17
Keywords 38

L

language dimension 1, 2
late 78
late resolve 80
Libraries 68
Library 95
library 95
List type 136
Lists 145
log 134
Log expression 133
LogExp 133

M

main 102
manuallyChanged 161
map 112
Mapping 186
mapping 107, 110
Mapping (Core) 4
Mapping body 109
Mapping call expression 111
mapping identifier 172
mapping inheritance 83
mapping merge 83
Mapping Operation 4
Mapping operation 104
Mapping Operations 10
Mapping Overloading 81
Mapping parameter 107
Mapping rules 192
MappingBody 109
MappingCallExp 111
MappingOperation 104
MappingParameter 107
Mappings 173
merge 84
metamodel 163
middle model 27
Model 67, 138, 142
Model parameter 97
Model Type 5
Model type 98
Model types 66, 173
ModelParameter 97
ModelParameter. 66
ModelType 98
modeltype 99
Module 95
Module import 97
ModuleImport 97
MOF Metamodels 11
multiplicity 164
mutable lists 85

N

namespace URI 13
nested exception 131
new 135
Notation for metamodels 163
null 86

O

Object 140
object 116
Object expression 116
Object template expressions 16
ObjectExp 116
ObjectTemplateExp 30
OCL standard library 138
OclAny 139
oclAsType 139
oclIsInvalid 87
oclIsKindOf 139
oclIsTypeOf 139
OclVoid 139
Operation body 108
Operational mapping language 10
Operational Mappings 10
Operational Transformation 5
Operational transformation 10, 89
OperationalTransformation 89
OperationBody 108
opposites 164
ordered 164
OrderedSet 154
orphan 68, 143
out 66, 97, 100, 102, 106

P

package 163
PackageToSchema 69
Parallelism 87
Pattern 17, 29
Pattern matching 16
Pattern matching semantics 21
Patterns 174
population 110
population section 70, 105
post-condition 105
pre-condition 105
Predefined tags 160
Predicate 29
primitive 163
Primitive Domain 16
property 104
Property assignment 188
PropertyAssignment 188
PropertyTemplateItem 31
proxy 160

Q

query 28, 102, 103
QVT compliance 3
QVT metamodel for CMOF 229
QVT Operational Mapping language 65
QVT Operational Mappings Library 138

- QVTBase package 24
- QVTc 9
- QVT-Core-SyntaxExportable 1
- QVTo 10
- QVTOperational package 88
- QVT-Operational-SyntaxExecutable 1
- QVT-Operational-XMIExportable 1
- QVTr 9
- QVTRelation package 33
- QVT-Relations-SyntaxExecutable 1
- QVT-Relations-XMIExportable 1
- QVT-SyntaxExecutable 1
- QVT-SyntaxExportable 1
- QVTTemplate Package 29
- QVT-XMIExecutable 1
- QVT-XMIExportable 1

R

- raise 132
- Raise expression 132
- RaiseExp 132, 139
- RDBMS 66
- readonly 164
- Realized variable 180, 187
- RealizedVariable 187
- references 164
- reference-to-variable 102, 106
- Refinement 181
- refines 94
- Relation 5, 33
- Relation Domain 5
- Relational Transformation 5
- RelationalTransformation 33
- RelationCallExp 37
- RelationDomain 35
- RelationDomainAssignment 37
- RelationImplementation 37
- Relations 9, 14
- Relations language 13
- Relations metamodel 9
- rememberChanges 160
- repr 133
- residence 67, 131
- resolve 76, 80
- Resolve expression 113
- Resolve in expression 115
- ResolveExp 113
- resolveIn 77, 79, 80, 115
- ResolveInExp 115
- resolveone 78, 79
- resolveoneIn 78, 80, 81, 115
- Restrictions on Expressions 18
- result 102, 106
- Result variable 86
- return 133
- Return expression 132
- ReturnExp 132
- Rule 28
- Rules 192

S

- self 103
- Self variable 86
- Semantics of Relations 251
- Sequence 155
- Set 155
- Severity kind 134
- SeverityKind 134
- side effect 118
- side-effect 28
- SimpleUml metamodel 66
- standard semantics 105, 111
- Status 139, 144
- Stdlib 138
- strict semantics 105, 111
- StringException 139
- Strings 155, 161
- Switch expression 127
- SwitchExp 127

T

- Template parameter type 137
- Template Pattern 5
- TemplateExp 29
- this variable 86
- topclasses 160
- Trace Class 5
- Trace class generation 192
- Trace Instance 6
- trace-data 74
- trace-record 74
- Transformation 13, 25, 143
- transformation 65, 94
- transformation identifier 172
- Transformations 173
- try 131
- Try expression 131
- TryExp 131
- tuple 86, 102
- type identifier 171
- Typedef 135
- typedef 85
- TypedModel 27

U

- UML 66
- unlink 131
- UnlinkExp 130
- URI 13
- uses 99

V

- Variable assignment 188
- Variable initialization expression 128
- Variable parameter 100
- VariableAssignment 188
- VariableInitExp 128
- Variables 86
- VarParameter 100
- Virtual machine 10
- Void 139

W

warning 134
when 105
When clause 14
where 99, 105
Where clause 14
while 122
While expression 121
WhileExp 121
wildcard 13
with 134

X

xcollect 126
xcollectselectOne 127
xmap 112
xselect 126
xselectOne 127

