Response to the UML 2.0 OCL RfP (ad/2000-09-03)

Revised Submission, Version 1.5
June 3, 2002

OMG Document ad/2002-05-09

Submitters

Boldsoft
Rational Software Corporation
IONA
Adaptive Ltd.

Supporters

Klasse Objecten
Kings College
University of Bremen
Dresden University of Technology
Kabira Technologies, Inc.
International Business Machines Corp.
Telelogic
University of Kent
Project Technology, Inc.
University of York
Compuware
Syntropy Ltd.
Oracle
Softeam

Contents

List of figures

List of tables

Foreword

COPYRIGHT WAIVER

SUBMISSION CONTACT PERSON

Feedback to this submission should preferrably be directed to Jos Warmer (J.Warmer@klasse.nl), but may also be directed any of the other authors. Their email addresses are given in 1.2 (“The Submission Team”).

SUBMITTER CONTACTS

Anders Ivner (anders.ivner@boldsoft.com)
Jonas Högström (jonas.hogstrom@boldsoft.com)
BoldSoft
Drakens Gränd 8
111 30 Stockholm
Sweden

Simon Johnston (sjohnsto@rational.com)
Rational Software Corporation
8383 158th Ave NE
Redmond, WA 98052
USA

David S. Frankel (david.frankel@iona.com)
IONA
Total Business Integration (TM)
741 Santiago Court
Chico, CA 95973-8781
USA

Pete Rivett (pete.rivett@adaptive.com)
Adaptive Ltd
Dean Park House,
8-10 Dean Park Crescent,
Bournemouth, BH1 1HL, UK

SUPPORTER CONTACTS

Jos Warmer (J.Warmer@klasse.nl)
Anneke Kleppe (A.Kleppe@klasse.nl)
Klasse Objecten
Postbus 3082
NL-3760 DB Soest
The Netherlands

Tony Clark (anclark@dcs.kcl.ac.uk)
Kings College, London
Software Engineering Research Group,
Department of Computer Science,
King's College Strand, London,
UK, WC2R 2LS

Martin Gogolla (gogolla@Informatik.Uni-Bremen.DE)
Mark Richters (mr@Informatik.Uni-Bremen.DE)
University of Bremen
FB3 Mathematics & Computer Science
AG Datenbanksysteme
P.O. Box 330440
D-28334 Bremen
GERMANY

Heinrich Hussmann (Heinrich.Hussmann@inf.tu-dresden.de)
Steffen Zschaler (Steffen.Zschaler@inf.tu-dresden.de)
Dresden University of Technology
Department of Computer Science
01062 Dresden, Germany

Conrad Bock (conrad.bock@kabira.com)
Kabira Technologies, Inc
One McInnis Parkway
Suite 200
San Rafael, CA 94903-2764

Steve Cook (sj_cook@uk.ibm.com)
International Business Machines

Cris Kobryn (cris.kobryn@telelogic.com)
Telelogic

Stuart Kent (sjhk@uck.ac.uk)
University of Kent, UK.

Steve Mellor (steve@projtech.com)
Project Technology, Inc.

Andy Evans (andye@cs.york.ac.uk)
University of York, UK

Wim Bast (Wim.Bast@nl.compuware.com)
Compuware

John Daniels (jd@syntropy.co.uk)
Syntropy Ltd, UK

Guus Ramackers (Guus.Ramackers@oracle.com)
Oracle

Philippe Desfray (philippe.desfray@softeam.fr)
Softeam

1

Overview

1.1 INTRODUCTION

This document contains a response to the Object Management Group's UML 2.0 OCL RFP (document reference ad/2000-09-03) for an updated specification of the Object Constraint Language, version 2.0. This version is based on the OCL definition as described in the UML 1.4 specification.

In August 2001 an initial submission was published and submitted to the OMG. This document is a revised and extended version of this submission.

1.2 THE SUBMISSION TEAM

This submission is written by the following team.

Name	**Email**	**Organisation**
Jos Warmer	J.Warmer@klasse.nl	Klasse Objecten
Anneke Kleppe	A.Kleppe@klasse.nl	Klasse Objecten
Tony Clark	anclark@dcs.kcl.ac.uk	Kings College, London
Anders Ivner	anders.ivner@boldsoft.com	Boldsoft
Jonas Högström	jonas.hogstrom@boldsoft.com	Boldsoft
Martin Gogolla	gogolla@Informatik.Uni-Bremen.DE	University of Bremen
Mark Richters	mr@Informatik.Uni-Bremen.DE	University of Bremen
Heinrich Hussmann	Heinrich.Hussmann@inf.tu-dresden.de	Technical university of Dresden
Steffen Zschaler	Steffen.Zschaler@inf.tu-dresden.de	Technical university of Dresden
Simon Johnston	sjohnsto@rational.com	Rational

1.3 ACKNOWLEDGEMENTS

The authoring team would like to thank the many people that have send us their ideas, or reviewed earlier versions of the specification and who participated in the various OCL Workshops that have been held over the last years.

We are not able to name all the individuals that have helped us, but we would specifically like to thank the following people: Steven Forgey, Dave Akehurst, Behzad Bordbar, Andy Schuerr, Thomas Baar, Stephan Flake,

Steve Cook, Perdita Stevens, B. Wolff, A. Brucker, Shane Sendall, Frédéric Fondement, Sune Vester Lindhe, Juliana Kuester-Filipe.

1.4 GOALS OF THE SUBMISSION

Obviously, the major goals of this submission are to meet the requirements outlined in the RFP mentioned above. However no such undertaking is done without some additional goals in the area of improvement either in functionality, clarity of definition or ease of use. This section attempts to capture all of these goals as the team itself defined them.

1.4.1 Relationship with existing OCL specification in UML 1.4

This submission supercedes chapter 6 from the OMG adopted UML 1.4 specification.

1.4.2 OCL 2.0 Metamodel

Today, OCL (up to, and including UML 1.4) has no metamodel, which makes it difficult to formally define the integration with the UML metamodel. As a response to a direct RFP requirement the OCL 2.0 submission will focus on the following.

1. Define a MOF 2.0-compliant metamodel for OCL. This metamodel should define the concepts and semantics of OCL and act as an abstract syntax for the language.
2. (Re)define the OCL 1.4 syntactical definition, that is done by means of a grammar, as a concrete syntax expressing the abstract syntax defined above.
3. To allow for alternative concrete syntaxes (e.g. Java-like syntax or visual constraint diagrams), by defining a strict separation between the metamodel and the concrete syntax.

1.4.3 OCL Expressibility and Usability

OCL 1.4 lacks expressibility in several areas. In the issues list of the UML 1.4 RTF a number of these issues have been delayed until UML 2.0. The OCL 2.0 submission will review these issues and define a solution when appropriate. The following is the lists of issues resolved by the OCL 2.0.

- OCL is currently defined as a language for describing constraints. OCL 2.0 specifies the Object Constraint Languages as a general object query language that can be used wherever expressions over UML models are required. This ibcludes the poossibility to define expressions over MOF models.
- Additional concepts to express messages sent by components, classes or other constructs that may have behavior have been added to OCL to allow the specification of behavioral constraints. Specifically the OclMessage concept is added for this purpose.
- All concepts defined in OCL, whether they are already in UML 1.4 or newly added to OCL 2.0 will be consistent with the concepts defined in the other two UML 2.0 submissions. This ensures that the three parts of UML 2.0 will seamlessly fit together.
- Because the UML and MOF core are identical in UML and MOF 2.0, OCL is consistent with both the MOF and the UML. The use of OCL to specify constraints in the UML metamodel is formalized, because the OCL definition can be used with the MOF 2.0.
- The simplicity and usability requirements that the OCL 1.4 definition is built upon remain the major guideline for the OCL 2.0.

1.4.4 OCL Semantics

Precise semantics of OCL should be defined as much as feasible.

- The submission includes a normative semantics for the abstract syntax, expressed using UML itself. In the appendix a non-normative mathematical based description of this semantics is expressed.

The semantics of OCL not only defines semantics for boolean expressions, as required for using OCL as constraint language. It also defines semantics for the use of OCL as a general UML expression and query language.

1.5 DESIGN RATIONALE

This section describes design decisions that have been made during the development of the OCL 2.0 specification according to the goals outlined above. These decisions usually reflect a change or major clarification with respect to OCL 1.4. Therefore they are given here to guide the OCL user through the major differences between OCL 1.4 and 2.0.

1.5.1 Abstract syntax

1. Collections can be nested. This is different from OCL 1.4 where collections were always implicitly flattened. In OCl 1.4 flattening was only applied to the collect and iterate operations. For handling nested collections and the collect operation OCL 2.0 now distinguises three different operations:
   - collectNested() which is identical to the OCL 1.4 collect, but without flattening applied,
   - flatten(), which flattens a nested collection,
   - collect(), which is identical to collectNested()->flatten().

   The current syntax retains the meaning of automatic flattening, because this needs to be backwards compatible. The flatten operation is a deep flatten, it completely flattens a nested collection of any depth.
2. The type OclExpression is removed. This is not a real type and doesn't fit well in the type system.
3. OclType (OclMetaType in the abstract syntax) is retained, but only as an enumeration kind. No access to the metalevel and UML metamodel is provided. If the UML 2.0 Infrastructure will include a reflection mechanism, OCL 2.0 will borrow the same mechanism to get access to the metalevel.
4. Although proposed, function types are not added to OCL. It will make the language more general, but this is (at least) one bridge too far for OCL 2.0.
5. The OCL type model follows the UML kernel type model as close as possible. Therefore the base type for OCL types has become Classifier from the UML core.
6. OCL should be extendible. We plan to (re)use any extension mechanism as described in the UML infrastructure submission. We have chosen not to define a separate mechanism for OCL.
7. We want to enable full use of OCL as a query language. Therefore the concept of Tuple is added to OCL. This gives OCL at least the same expressive power as SQL.
8. Being a full query language, OCL also becomes suitable to specify any type of relationship between different models at the M2 (MOF) level. It allows the use of OCL as part of the specifying of mappings to enable MDA tranformations.

1.5.2 Concrete syntax

1. The concrete syntax of OCL 2.0 is backwards compatible with OCL 1.4. This means at least that any OCL 1.4 expression that is also valid in OCL 2.0 will have the same meaning.

2. The Abstract Syntax does not depend on Concrete Syntax. There is a strict separation.
3. The OCL 2.0 grammar uses a different formalism as the OCL grammar in UML 1.4, but describes essentially the same concrete syntax. The grammar has been derived directly from the abstract syntax, which simplifies the mapping from the concrete syntax to the abstract syntax. In the concrete syntax section the approach is explained.
4. There is a complete and explicit mapping from the concrete syntax to the abstract syntax.

1.5.3 Semantics

1. The semantic description is based on a description in UML. In the appendix an equivalent mathematical description is given.
2. A description of the semantics in UML is be used to clarify the mathematical semantics for readers familiar with UML, but not with the mathematical formalism. It is placed in appendix A. This description includes the semantics for OclMessage.

The equivalance of the two semantic descriptions has not been formally established. Wherever they are conflicting, the description in section A (“Semantics”) is normative.

1.5.4 OCL Standard Library

The so-called predefined types and operations in UML 1.4 are now defined as the OCL Standard Library. This includes all the standard instances of the metaclass IteratorExp. Note that the iterator operations are not normal operations in the abstract syntax, but a specialized construct.

1.6 COMPLIANCE TO THE RFP REQUIREMENTS

This section outlines the items in the RFP to be addressed and should act as a guide to the reader in understanding how this submission meets the requirements of the RFP.

1.6.1 General Requirements

5.1.2; This specification of OCL includes detailed semantics and a normative formalism defining its operational behavior, sequencing and side-effects.

5.1.3; The inclusion of a normative formalism and the separation of abstract from concrete syntax does provide both a complete and precise specification.

5.1.4; Although this specification does not provide any interfaces, the implementation of the abstract syntax is mandatory and that the support for the canonical concrete syntax is mandatory in the absence of any alternative substitutable implementation.

5.1.10, 5.1.11; the separation of abstract and concrete syntax allows the flexibility for independent implementations of the concrete syntax to be substitutable for the canonical concrete syntax defined herein.

The following requirements are deemed irrelevant to the activity of defining the UML 2.0 OCL; 5.1.1, 5.1.5, 5.1.7, 5.1.8, 5.1.9, 5.1.12, 5.1.13, 5.1.14.

The considerations outlined in section 5.2 (except 5.2.5) where not explicitly accounted for in the development of this specification.

1.6.2 Specific Requirements - Mandatory

6.5.1; This submission clearly separates the abstract from concrete syntax and defines a metamodel and formalism for the abstract syntax. This specification also provides a mapping from the concrete syntax to the abstract syntax.

This submission attempts to provide backwards compatibility to the OCL defined in the UML 1.x family; however as there was no metamodel for OCL defined in those specifications this is only accomplished at the concrete syntax level.

This submission does retire a minor language feature from the UML 1.x OCL specifications.

- The type ExpressionType has been removed.

An XMI DTD for the OCL metamodel is provided as a normative appendix.

6.5.2; This submission, at its very heart, provides a complete and formal metamodel (the abstract syntax) for the OCL language.

1.6.3 Specific Requirements - Optional

6.6.2; This submission does provide a mathematically based, formalism for the abstract syntax.

This submission does provide certain additional features to the OCL language to improve its expressive power, these are clearly defined in the body of the document.

1.6.4 Issues to be Discussed

6.7; The exchange of existing, and future, models that are annotated with constraints represented as strings is not affected by this specification.

In the area of compliance, this submission does not cover the requirements for, or the approach to compliance testing of OCL implementations.

1.7 STRUCTURE OF THIS SUBMISSION

The document is divided into several sections.

Section 2 (“OCL Language Description”) gives an informal description of OCL in the style that has been used in the UML 1.1 through 1.4. This section is not normative, but meant to be explanatory.

MOF 2.0 Section 3 (“Abstract Syntax”) describes the abstract syntax of OCL using a MOF 2.0 compliant metamodel. This is the same approach as used in the UML 1.4 and other UML 2.0 submissions. The metamodel is MOF 2.0 compliant in the sense that it only uses constructs that are defined in the MOF 2.0.

Section 4 (“Concrete Syntax”) describes the canonical concrete syntax using an attributed EBNF grammar. This syntax is mapped onto the abstract syntax, achieving a complete separation between concrete and abstract syntax.

Section 5 (“Semantics Described using UML”) describes the semantics for OCL using UML.

In section 6 (“The OCL Standard Library”) the OCL Standard Library is described. This defines type like Integer, Boolean, etc. and all the collection types. OCL is not a stand-alone language, but an integral part of the UML. An OCL expression needs to be placed within the context of a UML model.

Section 7 (“The Use of Ocl Expressions in UML Models”) describes a number of places within the UML where OCL expressions can be used.

Appendix A ("Semantics") describes the underlying semantics of OCL using a mathematical formalism. This appendix, however is not normative, but ment for the readers that need a mathematical description for the semantics of OCL.

Appendix B ("Interchange Format") is currently a place holder for an interchange format, which can be defined along the same lines as XMI.

1.8 ALIGNMENT ISSUES WITH RESPECT TO UML 2.0 INFRASTRUCTURE AND MOF 2.0 CORE

This section describes some of the issues we expect to be imprtant in aligning with the UML 2.0 Infrastructure. We also expect to get feedback from the OMG Analysis and Design task force (ADTF) which will be taken into account. Furthermore we are open to additional features, when we have the opportunity to add them properly.

Alignment with UML 2.0 Infrastructure

The specification in this document is fully based on the UML 1.4 definition. As such, this specification could replace the OCL definition in UML 1.4. The integration with the UML metamodel takes place through a set of metaclasses from the UML 1.4 that are referenced in the OCL abstract syntax metamodel.

In the alignemnt work the references to UML 1.4 metaclasses should all be changed into references to UML 2.0 metaclasses.

Pairs of Pre and Postconditions

Presently, the proposed use of OCL in UML considers pre- and post-conditions separately, while the OCL semantics definition uses operation specifications (i.e., *pairs* of pre- and post-conditions). This needs to be aligned, with the UML 2.0 including a clarification of how multiple pre- and/or post-conditions are merged into one operation specification.

Frame Conditions

A syntax and/or semantics might be defined to allow users of OCL to specify what is sometimes called "expression closure" or "frame condition". This means that it should be possible to state that a specification of an operation is complete and everything which is not explicitly mentioned by the postconditions has to stay unchanged. Without such a mechanism, it is difficult to exclude unexpected side effects of operations. As this is not a property of an OCL expression, but of the context where it is being used, this should be part of the (postcondition) context in section 7 ("The Use of Ocl Expressions in UML Models") and of the UML 2.0.

2

OCL Language Description

This chapter introduces the Object Constraint Language (OCL), a formal language used to describe expressions on UML models. These expressions typically specify invariant conditions that must hold for the system being modeled or queries over objects described in a model. Note that when the OCL expressions are evaluated, they do not have side effects; i.e. their evaluation cannot alter the state of the corresponding executing system.

OCL expressions can be used to specify operations / actions that, when executed, do alter the state of the system. UML modelers can use OCL to specify application-specific constraints in their models. UML modelers can also use OCL to specify queries on the UML model, which are completely programming language independent.

2.1 WHY OCL?

A UML diagram, such as a class diagram, is typically not refined enough to provide all the relevant aspects of a specification. There is, among other things, a need to describe additional constraints about the objects in the model. Such constraints are often described in natural language. Practice has shown that this will always result in ambiguities. In order to write unambiguous constraints, so-called formal languages have been developed. The disadvantage of traditional formal languages is that they are usable to persons with a strong mathematical background, but difficult for the average business or system modeler to use.

OCL has been developed to fill this gap. It is a formal language that remains easy to read and write. It has been developed as a business modeling language within the IBM Insurance division, and has its roots in the Syntropy method.

OCL is a pure specification language; therefore, an OCL expression is guaranteed to be without side effect. When an OCL expression is evaluated, it simply returns a value. It cannot change anything in the model. This means that the state of the system will never change because of the evaluation of an OCL expression, even though an OCL expression can be used to *specify* a state change (e.g., in a post-condition).

OCL is not a programming language; therefore, it is not possible to write program logic or flow control in OCL. You cannot invoke processes or activate non-query operations within OCL. Because OCL is a modeling language in the first place, OCl expressions are not by definition directly executable.

OCL is a typed language, so that each OCL expression has a type. To be well formed, an OCL expression must conform to the type conformance rules of the language. For example, you cannot compare an Integer with a String. Each Classifier defined within a UML model represents a distinct OCL type. In addition, OCL includes a set of supplementary predefined types. These are described in section 6 (“The OCL Standard Library”).

As a specification language, all implementation issues are out of scope and cannot be expressed in OCL.

The evaluation of an OCL expression is instantaneous. This means that the states of objects in a model cannot change during evaluation.

2.1.1 Where to Use OCL

OCL can be used for a number of different purposes:

- As a query language
- To specify invariants on classes and types in the class model
- To specify type invariant for Stereotypes
- To describe pre- and post conditions on Operations and Methods
- To describe Guards
- To specify target (sets) for messages and actions
- To specify constraints on operations
- To specify derivation rules for attributes.
- for any expression over a UML model

2.2 INTRODUCTION

2.2.1 Legend

Text written in the Letter Gothic typeface as shown below is an OCL expression.

```
'This is an OCL expression'
```

The *context* keyword introduces the context for the expression. The keyword *inv*, *pre* and *post* denote the stereotypes, respectively «invariant», «precondition», and «postcondition», of the constraint. The actual OCL expression comes after the colon.

```
context TypeName inv:
'this is an OCL expression with stereotype <<invariant>> in the
context of TypeName' = 'another string'
```

In the examples the keywords of OCL are written in boldface in this document. The boldface has no formal meaning, but is used to make the expressions more readable in this document. OCL expressions are written using ASCII characters only.

Words in *Italics* within the main text of the paragraphs refer to parts of OCL expressions.

2.2.2 Example Class Diagram

The diagram below is used in the examples in this chapter.

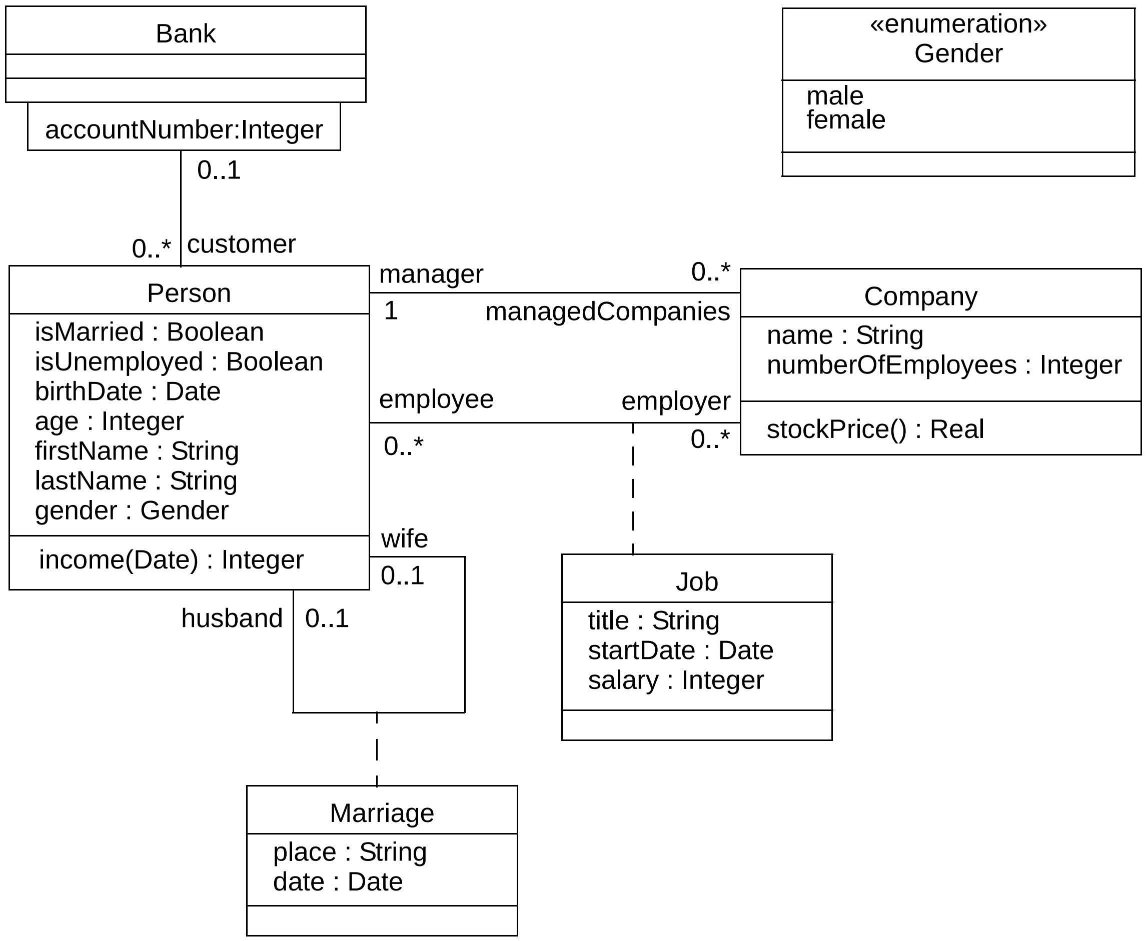


Figure 2-1 *Class Diagram Example*

2.3 RELATION TO THE UML METAMODEL

2.3.1 Self

Each OCL expression is written in the context of an instance of a specific type. In an OCL expression, the reserved word *self* is used to refer to the contextual instance. For instance, if the context is Company, then *self* refers to an instance of Company.

2.3.2 Specifying the UML context

The context of an OCL expression within a UML model can be specified through a so-called context declaration at the beginning of an OCL expression. The context declaration of the constraints in the following sections is shown.

If the constraint is shown in a diagram, with the proper stereotype and the dashed lines to connect it to its contextual element, there is no need for an explicit context declaration in the test of the constraint. The context declaration is optional.

2.3.3 Invariants

The OCL expression can be part of an Invariant which is a Constraint stereotyped as an «invariant». When the invariant is associated with a Classifier, the latter is referred to as a "type" in this chapter. An OCL expression is an invariant of the type and must be true for all instances of that type at any time. (Note that all OCL expressions that express invariants are of the type Boolean.)

For example, if in the context of the Company type in figure 2-1 on page 2-3, the following expression would specify an invariant that the number of employees must always exceed 50:

```
self.numberOfEmployees > 50
```

where *self* is an instance of type Company. (We can view *self* as the object from where we start evaluating the expression.) This invariant holds for every instance of the Company type.

The type of the contextual instance of an OCL expression, which is part of an invariant, is written with the *context* keyword, followed by the name of the type as follows. The label *inv:* declares the constraint to be an «invariant» constraint.

```
context Company inv:
self.numberOfEmployees > 50
```

In most cases, the keyword *self* can be dropped because the context is clear, as in the above examples. As an alternative for self, a different name can be defined playing the part of self:

```
context c : Company inv:
c.numberOfEmployees > 50
```

This invariant is equivalent to the previous one.

Optionally, the name of the constraint may be written after the *inv* keyword, allowing the constraint to be referenced by name. In the following example the name of the constraint is *enoughEmployees*. In the UML 1.4 metamodel, this name is a (meta-)attribute of the metaclass Constraint that is inherited from ModelElement.

```
context c : Company inv enoughEmployees:
c.numberOfEmployees > 50
```

2.3.4 Pre- and Postconditions

The OCL expression can be part of a Precondition or Postcondition, corresponding to «precondition» and «postcondition» stereotypes of Constraint associated with an Operation or other behavioral feature. The contextual instance *self* then is an instance of the type which owns the operation or method as a feature. The context declaration in OCL uses the *context* keyword, followed by the type and operation declaration. The stereotype of constraint is shown by putting the labels 'pre:' and 'post:' before the actual Preconditions and Postconditions

```
context Typename::operationName(param1 : Type1, ... ): ReturnType
pre :  param1 > ...
post:  result = ...
```

The name *self* can be used in the expression referring to the object on which the operation was called. The reserved word *result* denotes the result of the operation, if there is one. The names of the parameters (*param1*) can also be used in the OCL expression. In the example diagram, we can write:

```
context Person::income(d : Date) : Integer
post:  result = 5000
```

Optionally, the name of the precondition or postcondition may be written after the *pre or post* keyword, allowing the constraint to be referenced by name. In the following example the name of the precondition is *parameterOk* and the name of the postcondition is *resultOk*. In the UML metamodel, these names are the values of the attribute *name* of the metaclass Constraint that is inherited from ModelElement.

```
context Typename::operationName(param1 : Type1, ... ): ReturnType
pre  parameterOk:  param1 > ...
post resultOk   :  result = ...
```

2.3.5 Package context

The above context declaration is precise enough when the package in which the Classifier belongs is clear from the environment. To specify explicitly in which package invariant, pre or postcondition Constraints belong, these constraints can be enclosed between 'package' and 'endpackage' statements. The package statements have the syntax:

```
package Package::SubPackage

context X inv:
... some invariant ...
context X::operationName(..)
pre: ... some precondition ...

endpackage
```

An OCL file (or stream) may contain any number package statements, thus allowing all invariant, preconditions and postconditions to be written and stored in one file. This file may co-exist with a UML model as a separate entity.

2.3.6 Other Types of Expressions

Any OCL expression can be used as the value for an attribute of the UML metaclass Expression or one of its subtypes. In that case, the semantics section describes the meaning of the expression. A special subclass of Expression, called ExpressionInOcl is used for this purpose. See section 7.1 (“Introduction”) for a definition.

2.4 BASIC VALUES AND TYPES

In OCL, a number of basic types are predefined and available to the modeler at all time. These predefined value types are independent of any object model and part of the definition of OCL.

The most basic value in OCL is a value of one of the basic types. The basic types of OCL, with corresponding examples of their values, are shown in Table 1.

type	values
Boolean	true, false
Integer	1, -5, 2, 34, 26524, ...
Real	1.5, 3.14, ...
String	'To be or not to be...'

Table 1. *Basic Types*

OCL defines a number of operations on the predefined types. Table 2. gives some examples of the operations on the predefined types. See 6.4 (“Primitive Types”) for a complete list of all operations.

Collection, Set, Bag, Sequence and Tuple are basic types as well. Their specifics will be described in the upcoming sections.

type	operations
Integer	*, +, -, /, abs()
Real	*, +, -, /, floor()
Boolean	and, or, xor, not, implies, if-then-else
String	concat(), size(), substring()

Table 2. *Operations on predefined types*

2.4.1 Types from the UML Model

Each OCL expression is written in the context of a UML model, a number of classifiers (types/classes, ...), their features and associations, and their generalizations. All classifiers from the UML model are types in the OCL expressions that are attached to the model.

2.4.2 Enumeration Types

Enumerations are Datatypes in UML and have a name, just like any other Classifier. An enumeration defines a number of enumeration literals, that are the possible values of the enumeration. Within OCL one can refer to the value of an enumeration. When we have Datatype named Gender in the example model with values 'female' or 'male' they can be used as follows:

```
context Person inv: gender = Gender::male
```

2.4.3 Let Expressions

Sometimes a sub-expression is used more than once in a constraint. The *let* expression allows one to define a variable which can be used in the constraint.

```
context Person inv:
let income : Integer = self.job.salary->sum() in
if isUnemployed then
  income < 100
else
  income >= 100
endif
```

A let expression may be included in any kind of OCL expression. It is only known within this specific expression.

2.4.4 Additional operations/attributes through «definition» Constraints

The Let expression allows a variable to be used in one Ocl expression. To enable reuse of variables/operations over multiple OCl expressions one can use a Constraint with the stereotype «definition», in which helper variables/operations are defined. This «definition» Constraint must be attached to a Classifier and may only contain variable and/or operation definitions, nothing else. All variables and operations defined in the «definition» constraint are known in the same context as where any property of the Classifier can be used. Such variables and operations are attributes and operations with stereotype «OclHelper» of the classifier. They are used in an OCL expression in exactly the same way as normal attributes or operations are used. The textual notation for a «definition» Constraint uses the keyword ‘def’ as shown below. The syntax of the attribute or operation definitions os similar to the Let expression, but the keywords *attr* and *oper* are used for respectively attribute and operation definitions.

```
context Person def:
attr income : Integer = self.job.salary->sum() ,
     nickname : String = 'Little Red Rooster'
oper hasTitle(t : String) : Boolean = self.job->exists(title = t)
```

The names of the attributes / operations in a let expression may not conflict with the names of respective attributes/associationEnds and operations of the Classifier.

Using this definition syntax is identical to defining an attribute/operation in the UML with stereotype «OclHelper» with an attached OCl constraint for its derivation.

2.4.5 Type Conformance

OCL is a typed language and the basic value types are organized in a type hierarchy. This hierarchy determines conformance of the different types to each other. You cannot, for example, compare an Integer with a Boolean or a String.

An OCL expression in which all the types conform is a valid expression. An OCL expression in which the types don't conform is an invalid expression. It contains a t*ype conformance error*. A type *type1* conforms to a type *type2* when an instance of *type1* can be substituted at each place where an instance of *type2* is expected. The type conformance rules for types in the class diagrams are simple.

- Each type conforms to each of its supertypes.
- Type conformance is transitive: if *type1* conforms to *type2*, and *type2* conforms to *type3*, then *type1* conforms to *type3*.

The effect of this is that a type conforms to its supertype, and all the supertypes above. The type conformance rules for the types from the OCL Standard Library are listed in table 3.

Type	**Conforms to/Is a subtype of**	**Condition**
`Set(T1)`	`Collection(T2)`	`if T1 conforms to T1`
`Sequence(T1)`	`Collection(T2)`	`if T1 conforms to T1`
`Bag(T1)`	`Collection(T2)`	`if T1 conforms to T1`
`Integer`	`Real`	

Table 3. *Type conformance rules*

The conformance relation between the collection types only holds if they are collections of element types that conform to each other. See "Collection Type Hierarchy and Type Conformance Rules" on page -17 for the complete conformance rules for collections.

Table 4 provides examples of valid and invalid expressions.

OCL expression	**valid**	**explanation**
`1 + 2 * 34`	yes	
`1 + 'motorcycle'`	no	type String does not conform to type Integer
`23 * false`	no	type Boolean does not conform to Integer
`12 + 13.5`	yes	

Table 4. *Valid expressions*

2.4.6 Re-typing or Casting

In some circumstances, it is desirable to use a property of an object that is defined on a subtype of the current known type of the object. Because the property is not defined on the current known type, this results in a type conformance error.

When it is certain that the actual type of the object is the subtype, the object can be re-typed using the operation *oclAsType(OclType)*. This operation results in the same object, but the known type is the argument *OclType*. When there is an object *object* of type *Type1* and *Type2* is another type, it is allowed to write:

```
object.oclAsType(Type2) --- evaluates to object with type Type2
```

An object can only be re-typed to one of its subtypes; therefore, in the example, *Type2* must be a subtype of *Type1*.

If the actual type of the object is not a subtype of the type to which it is re-typed, the expression is undefined (see 2.4.11 (“Undefined Values”)).

2.4.7 Precedence Rules

The precedence order for the operations, starting with highest precedence, in OCL is:

- @pre
- dot and arrow operations: ‘.’ and ‘->’
- unary ‘not’ and unary minus ‘-’
- ‘*’ and ‘/’
- ‘+’ and binary ‘-’
- ‘if-then-else-endif’
- ‘<’, ‘>’, ‘<=’, ‘>=’
- ‘=’, ‘<>’
- ‘and’, ‘or’ and ‘xor’
- ‘implies’

Parentheses ‘(’ and ‘)’ can be used to change precedence.

2.4.8 Use of Infix Operators

The use of infix operators is allowed in OCL. The operators ‘+’, ‘-’, ‘*’. ‘/’, ‘<‘, ‘>’, ‘<>’ ‘<=’ ‘>=’ are used as infix operators. If a type defines one of those operators with the correct signature, they will be used as infix operators. The expression:

```
a + b
```

is conceptually equal to the expression:

```
a.+(b)
```

that is, invoking the ‘+’ operation on `a` with `b` as the parameter to the operation.

The infix operators defined for a type must have exactly one parameter. For the infix operators ‘<‘, ‘>’, ‘<=’, ‘>=’, ‘<>’, ‘and’, ‘or’, and ‘xor’ the return type must be Boolean.

2.4.9 Keywords

Keywords in OCL are reserved words. That means that the keywords cannot occur anywhere in an OCL expression as the name of a package, a type or a property. The list of keywords is shown below:

```
and
attr
context
def
else
endif
endpackage
if
implies
in
inv
```

```
let
not
oper
or
package
post
pre
then
xor
```

2.4.10 Comment

Comments in OCL are written following two successive dashes (minus signs). Everything immediately following the two dashes up to and including the end of line is part of the comment. For example:

```
-- this is a comment
```

2.4.11 Undefined Values

Some expressions will, when evaluated, have an undefined value. For instance, typecasting with oclAsType() to a type that the object does not support or getting the ->first() element of an empty collection will result in undefined. In general, an expression where one of the parts is undefined will itself be undefined. There are some important exceptions to this rule, however. First, there are the logical operators:

- True OR-ed with anything is True
- False AND-ed with anything is False
- False IMPLIES anything is True

The rules for OR and AND are valid irrespective of the order of the arguments and they are valid whether the value of the other sub-expression is known or not.

The IF-expression is another exception. It will be valid as long as the chosen branch is valid, irrespective of the value of the other branch.

Finally, there is an explicit operation for testing if the value of an expression is undefined. oclIsUndefined() is an operation on OclAny that results in True if its argument is undefined and False otherwise.

2.5 OBJECTS AND PROPERTIES

OCL expressions can refer to Classifiers, e.g. types, classes, interfaces, associations (acting as types) and datatypes. Also all attributes, association-ends, methods, and operations without side-effects that are defined on these types, etc. can be used. In a class model, an operation or method is defined to be side-effect-free if the isQuery attribute of the operations is true. For the purpose of this document, we will refer to attributes, association-ends, and side-effect-free methods and operations as being *properties*. A property is one of:

- an Attribute
- an AssociationEnd
- an Operation with *isQuery* being true
- a Method with *isQuery* being true

The value of a property on an object that is defined in a class diagram is specified in an OCL expression by a dot followed by the name of the property.

```
context Person inv:
self.isMarried
```

If *self* is a reference to an object, then *self.property* is the value of the *property* property on *self*.

2.5.1 Properties: Attributes

For example, the age of a Person is written as *self.age*:

```
context Person inv:
self.age > 0
```

The value of the subexpression *self.age* is the value of the *age* attribute on the particular instance of Person identified by *self*. The type of this subexpression is the type of the attribute *age*, which is the standard type Integer.

Using attributes, and operations defined on the basic value types, we can express calculations etc. over the class model. For example, a business rule might be “the age of a Person is always greater than zero.” This can be stated by the invariant above.

Attributes may have multiplicities in a UML model. Wheneven the multiplicity of an attribute is greater than 1, the result type is collection of values. Collections in OCL are described later in this chapter.

2.5.2 Properties: Operations

Operations may have parameters. For example, as shown earlier, a Person object has an income expressed as a function of the date. This operation would be accessed as follows, for a Person *aPerson* and a date *aDate*:

```
aPerson.income(aDate)
```

The result of this operation call is a value of the return type of the operation, which is Integer in this example. If the operation has out or in/out parameters, the result of this operation is a tuple containing all out, in/out parameters and the return value. For example, if the income operation would have an out parameter *bonus*, the result of the above operation call is of type *Tuple(bonus: Integer, result: Integer)*. You can access these values using the names of the out parameters, and the keyword *result*, for example:

```
aPerson.income(aDate).bonus = 300 and
aPerson.income(aDate).result = 5000
```

Note that the out parameters need not be included in the operation call. Values for all in or in/out parameters are neccessary.

Defining operations

The operation itself could be defined by a postcondition constraint. This is a constraint that is stereotyped as «postcondition». The object that is returned by the operation can be referred to by *result*. It takes the following form:

```
context Person::income (d: Date) : Integer
post: result = age * 1000
```

The right-hand-side of this definition may refer to the operation being defined (i.e., the definition may be recursive) as long as the recursion is not infinite. Inside a pre- or postcondition one can also use the parameters of the operation. The type of *result*, when the operation has no out or in/out parameters, is the return type of the operation, which is Integer in the above example. When the operation does have out or in/out parameters, the return type is a Tuple as explained above. The postcondition for the income operation with out parameter bonus may take the following form:

```
context Person::income (d: Date, bonus: Integer) : Integer
post: result = Tuple { bonus = ...,
                       result = .... }
```

To refer to an operation or a method that doesn’t take a parameter, parentheses with an empty argument list are mandatory:

```
context Company inv:
self.stockPrice() > 0
```

2.5.3 Properties: AssociationEnds and Navigation

Starting from a specific object, we can navigate an association on the class diagram to refer to other objects and their properties. To do so, we navigate the association by using the opposite association-end:

```
object.associationEndName
```

The value of this expression is the set of objects on the other side of the *associationEndName* association. If the multiplicity of the association-end has a maximum of one (“0..1” or “1”), then the value of this expression is an object. In the example class diagram, when we start in the context of a Company (i.e., *self* is an instance of Company), we can write:

```
context Company
inv: self.manager.isUnemployed = false
inv: self.employee->notEmpty()
```

In the first invariant *self.manager* is a Person, because the multiplicity of the association is one. In the second invariant *self.employee* will evaluate in a Set of Persons. By default, navigation will result in a Set. When the association on the Class Diagram is adorned with {ordered}, the navigation results in a Sequence.

Collections, like Sets, Bags, and Sequences are predefined types in OCL. They have a large number of predefined operations on them. A property of the collection itself is accessed by using an arrow ‘->’ followed by the name of the property. The following example is in the context of a person:

```
context Person inv:
self.employer->size() < 3
```

This applies the *size* property on the Set *self.employer*, which results in the number of employers of the Person *self*.

```
context Person inv:
self.employer->isEmpty()
```

This applies the *isEmpty* property on the Set *self.employer*. This evaluates to true if the set of employers is empty and false otherwise.

Missing AssociationEnd names

When the name of an association-end is missing at one of the ends of an association, the name of the type at the association end is used as the rolename. If this results in an ambiguity, the rolename is mandatory. This is e.g. the case with unnamed rolenames in reflexive associations. If the rolename is ambiguous, then it cannot be used in OCL.

Navigation over Associations with Multiplicity Zero or One

Because the multiplicity of the role manager is one, *self.manager* is an object of type Person. Such a single object can be used as a Set as well. It then behaves as if it is a Set containing the single object. The usage as a set is done through the arrow followed by a property of Set. This is shown in the following example:

```
context Company inv:
self.manager->size() = 1
```

The sub-expression *self.manager* is used as a Set, because the arrow is used to access the *size* property on Set. This expression evaluates to true.

```
context Company inv:
self.manager->foo
```

The sub-expression *self.manager* is used as Set, because the arrow is used to access the *foo* property on the Set. This expression is incorrect, because *foo* is not a defined property of Set.

```
context Company inv:
self.manager.age > 40
```

The sub-expression *self.manager* is used as a Person, because the dot is used to access the *age* property of Person.

In the case of an optional (0..1 multiplicity) association, this is especially useful to check whether there is an object or not when navigating the association. In the example we can write:

```
context Person inv:
self.wife->notEmpty() implies self.wife.gender = Gender::female
```

Combining Properties

Properties can be combined to make more complicated expressions. An important rule is that an OCL expression always evaluates to a specific object of a specific type. After obtaining a result, one can always apply another property to the result to get a new result value. Therefore, each OCL expression can be read and evaluated left-to-right.

Following are some invariants that use combined properties on the example class diagram:

[1] Married people are of age >= 18

```
context Person inv:
self.wife->notEmpty() implies self.wife.age >= 18 and
self.husband->notEmpty() implies self.husband.age >= 18
```

[2] a company has at most 50 employees

```
context Company inv:
self.employee->size() <= 50
```

2.5.4 Navigation to Association Classes

To specify navigation to association classes (Job and Marriage in the example), OCL uses a dot and the name of the association class starting with a lowercase character:

```
context Person inv:
self.job
```

The sub-expression *self.job* evaluates to a Set of all the jobs a person has with the companies that are his/her employer. In the case of an association class, there is no explicit rolename in the class diagram. The name *job* used in this navigation is the name of the association class starting with a lowercase character, similar to the way described in the section “Missing Rolenames” above.

In case of a recursive association, that is an association of a class with itself, the name of the association class alone is not enough. We need to distinguish the direction in which the association is navigated as well as the name of the association class. Take the following model as an example.

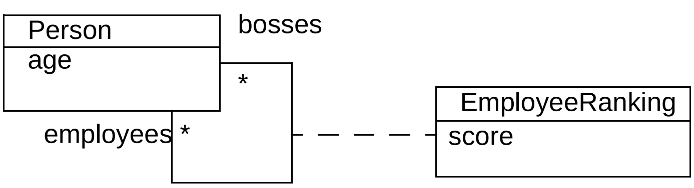


Figure 2-2 *Navigating recursive association classes*

When navigating to an association class such as *employeeRanking* there are two possibilities depending on the direction. For instance, in the above example, we may navigate towards the *employees* end, or the *bosses* end. By using the name of the association class alone, these two options cannot be distinguished. To make the distinction, the rolename of the direction in which we want to navigate is added to the association class name, enclosed in square brackets. In the expression

```
context Person inv:
self.employeeRanking[bosses]->sum() > 0
```

the *self.employeeRanking[bosses]* evaluates to the set of *EmployeeRankings* belonging to the collection of *bosses*. And in the expression

```
context Person inv:
self.employeeRanking[employees]->sum() > 0
```

the *self.employeeRanking[employees]* evaluates to the set of *EmployeeRankings* belonging to the collection of *employees*. The unqualified use of the association class name is not allowed in such a recursive situation. Thus, the following example is invalid:

```
context Person inv:
self.employeeRanking->sum() > 0 -- INVALID!
```

In a non-recursive situation, the association class name alone is enough, although the qualified version is allowed as well. Therefore, the examples at the start of this section could also be written as:

```
context Person inv:
self.job[employer]
```

2.5.5 Navigation from Association Classes

We can navigate from the association class itself to the objects that participate in the association. This is done using the dot-notation and the role-names at the association-ends.

```
context Job
inv: self.employer.numberOfEmployees >= 1
inv: self.employee.age > 21
```

Navigation from an association class to one of the objects on the association will always deliver exactly one object. This is a result of the definition of AssociationClass. Therefore, the result of this navigation is exactly one object, although it can be used as a Set using the arrow (->).

2.5.6 Navigation through Qualified Associations

Qualified associations use one or more qualifier attributes to select the objects at the other end of the association. To navigate them, we can add the values for the qualifiers to the navigation. This is done using square brackets, following the role-name. It is permissible to leave out the qualifier values, in which case the result will be all objects at the other end of the association. The following example results in a Set(Person) containing all customers of the Bank.

```
context Bank inv:
self.customer
```

The next example results in one Person, having accountnumber 8764423.

```
context Bank inv:
self.customer[8764423]
```

If there is more than one qualifier attribute, the values are separated by commas, in the order which is specified in the UML class model. It is not permissible to partially specify the qualifier attribute values.

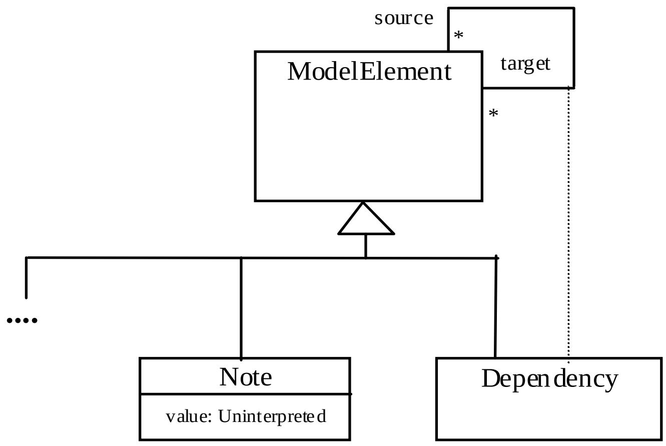


Figure 2-3 *Accessing Overridden Properties Example*

2.5.7 Using Pathnames for Packages

Within UML, types are organized in packages. OCL provides a way of explicitly referring to types in other packages by using a package-pathname prefix. The syntax is a package name, followed by a double colon:

```
Packagename::Typename
```

This usage of pathnames is transitive and can also be used for packages within packages:

```
Packagename1::Packagename2::Typename
```

2.5.8 Accessing overridden properties of supertypes

Whenever properties are redefined within a type, the property of the supertypes can be accessed using the *oclAsType()* operation. Whenever we have a class B as a subtype of class A, and a property p1 of both A and B, we can write:

```
context B inv:
self.oclAsType(A).p1  -- accesses the p1 property defined in A
self.p1         -- accesses the p1 property defined in B
```

Figure 2-3 shows an example where such a construct is needed. In this model fragment there is an ambiguity with the OCL expression on Dependency:

```
context Dependency inv:
self.source <> self
```

This can either mean normal association navigation, which is inherited from ModelElement, or it might also mean navigation through the dotted line as an association class. Both possible navigations use the same rolename, so this is always ambiguous. Using *oclAsType()* we can distinguish between them with:

```
context Dependency
inv: self.oclAsType(Dependency).source->isEmpty()
inv: self.oclAsType(ModelElement).source->isEmpty()
```

2.5.9 Predefined properties on All Objects

There are several properties that apply to all objects, and are predefined in OCL. These are:

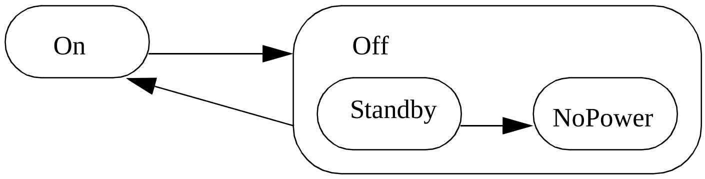
```
oclIsTypeOf (t : OclType)  : Boolean
oclIsKindOf (t : OclType)  : Boolean
oclInState (s : OclState)  : Boolean
oclIsNew   ()             : Boolean
oclAsType (t : OclType) : instance of OclType
```

The operation is *oclIsTypeOf* results in true if the *type* of self and *t* are the same. For example:

```
context Person
inv: self.oclIsTypeOf( Person )      -- is true
inv: self.oclIsTypeOf( Company)      -- is false
```

The above property deals with the direct type of an object. The *oclIsKindOf* property determines whether *t* is either the direct type or one of the supertypes of an object.

The operation *oclInState(s)* results in true if the object is in the state *s*. Values for *s* are the names of the states in the statemachine(s) attached to the Classifier of *object*. For nested states the statenames can be combined using the double colon '::' .



In the example statemachine above, values for *s* can be *On*, *Off*, *Off::Standby*, *Off::NoPower*. If the classifier of *object* has the above associated statemachine valid OCL expressions are:

```
object.oclInState(On)
object.oclInState(Off)
object.oclInstate(Off::Standby)
object.oclInState(Off::NoPower)
```

If there are multiple statemachines attached to the object's classifier, then the statename can be prefixed with the name of the statemachine containing the state and the double colon '::', as with nested states.

The operation *oclIsNew* evaluates to true if, used in a postcondition, the object is created during performing the operation. i.e., it didn't exist at precondition time.

2.5.10 Features on Classes Themselves

All properties discussed until now in OCL are properties on instances of classes. The types are either predefined in OCL or defined in the class model. In OCL, it is also possible to use features defined on the types/classes themselves. These are, for example, the *class*-scoped features defined in the class model. Furthermore, several features are predefined on each type.

A predefined feature on classes, interfaces and enumerations is *allInstances*, which results in the Set of all instances of the type in existence at the specific time when the expression is evaluated. If we want to make sure that all instances of Person have unique names, we can write:

```
context Person inv:
Person.allInstances()->forAll(p1, p2 |
                              p1 <> p2 implies p1.name <> p2.name)
```

The *Person.allInstances()* is the set of all persons and is of type Set(Person). It is the set of all persons that exist in the system at the time that the expression is evaluated.

2.5.11 Collections

Single navigation of an association results in a Set, combined navigations in a Bag, and navigation over associations adorned with {ordered} results in a Sequence. Therefore, the collection types define in the OCL Standard Library play an important role in OCL expressions.

The type Collection is predefined in OCL. The Collection type defines a large number of predefined operations to enable the OCL expression author (the modeler) to manipulate collections. Consistent with the definition of OCL as an expression language, collection operations never change collections; *isQuery* is always true. They may result in a collection, but rather than changing the original collection they project the result into a new one.

Collection is an abstract type, with the concrete collection types as its subtypes. OCL distinguishes three different collection types: Set, Sequence, and Bag. A Set is the mathematical set. It does not contain duplicate elements. A Bag is like a set, which may contain duplicates (i.e., the same element may be in a bag twice or more). A Sequence is like a Bag in which the elements are ordered. Both Bags and Sets have no order defined on them.

Collection Literals

Sets, Sequences, and Bags can be specified by a literal in OCL. Curly brackets surround the elements of the collection, elements in the collection are written within, separated by commas. The type of the collection is written before the curly brackets:

```
Set { 1 , 2 , 5 , 88 }
Set { 'apple' , 'orange', 'strawberry' }
```

A Sequence:

```
Sequence { 1, 3, 45, 2, 3 }
Sequence { 'ape', 'nut' }
```

A bag:

```
Bag {1 , 3 , 4, 3, 5 }
```

Because of the usefulness of a Sequence of consecutive Integers, there is a separate literal to create them. The elements inside the curly brackets can be replaced by an interval specification, which consists of two expressions of type Integer, *Int-expr1* and *Int-expr2*, separated by '..'. This denotes all the Integers between the values of *Int-expr1* and *Int-expr2*, including the values of *Int-expr1* and *Int-expr2* themselves:

```
Sequence{ 1..(6 + 4) }
Sequence{ 1..10 }
-- are both identical to
Sequence{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
```

The complete list of Collection operations is described in chapter 6 ("The OCL Standard Library").

Collections can be specified by a literal, as described above. The only other way to get a collection is by navigation. To be more precise, the only way to get a Set, Sequence, or Bag is:

1. a literal, this will result in a Set, Sequence, or Bag:

```
Set      {1 , 2, 3 , 5 , 7 , 11, 13, 17 }
Sequence {1 , 2, 3 , 5 , 7 , 11, 13, 17 }
Bag      {1, 2, 3, 2, 1}
```

2. a navigation starting from a single object can result in a collection:

```
context Company inv:
   self.employee
```

3. operations on collections may result in new collections:

```
collection1->union(collection2)
```

2.5.12 Collections of Collections

In UML 1.4 a collection in OCL was always flattened, i.e. a collection could never contain other collections as elements. This restriction is relieved in UML 2.0. OCL allows elements of collections to be collections themselves. The OCL Standard Library includes specific flatten operations for collections. These can be used to flatten collections of collections explicitly.

2.5.13 Collection Type Hierarchy and Type Conformance Rules

In addition to the type conformance rules in 2.4.5 ("Type Conformance"), the following rules hold for all types, including the collection types:

- The types Set (X), Bag (X) and Sequence (X) are all subtypes of Collection (X).

Type conformance rules are as follows for the collection types:

- *Type1* conforms to *Type2* when they are identical (standard rule for all types).
- *Type1* conforms to *Type2* when it is a subtype of Type2 (standard rule for all types).
- *Collection(Type1)* conforms to *Collection(Type2)*, when *Type1* conforms to *Type2*. This is also true for *Set(Type1)/Set(Type2), Sequence(Type1)/Sequence(Type2), Bag(Type1)/Bag(Type2)*
- Type conformance is transitive: if *Type1* conforms to *Type2*, and *Type2* conforms to *Type3*, then *Type1* conforms to *Type3* (standard rule for all types).

For example, if *Bicycle* and *Car* are two separate subtypes of *Transport:*

```
Set(Bicycle)  conforms to  Set(Transport)
Set(Bicycle)  conforms to  Collection(Bicycle)
Set(Bicycle)  conforms to  Collection(Transport)
```

Note that Set(Bicycle) does not conform to Bag(Bicycle), nor the other way around. They are both subtypes of Collection(Bicycle) at the same level in the hierarchy.

2.5.14 Previous Values in Postconditions

As stated in 2.3.4 ("Pre- and Postconditions"), OCL can be used to specify pre- and post-conditions on operations and methods in UML. In a postcondition, the expression can refer to values for each property of an object at two moments in time:

- the value of a property at the start of the operation or method
- the value of a property upon completion of the operation or method

The value of a property in a postcondition is the value upon completion of the operation. To refer to the value of a property at the start of the operation, one has to postfix the property name with the keyword '*@pre*':

```
context Person::birthdayHappens()
post: age = age@pre + 1
```

The property *age* refers to the property of the instance of Person which executes the operation. The property *age@pre* refers to the value of the property *age* of the Person that executes the operation, at the start of the operation.

If the property has parameters, the '@pre' is postfixed to the propertyname, before the parameters.

```
context Company::hireEmployee(p : Person)
post: employees = employees@pre->including(p) and
      stockprice() = stockprice@pre() + 10
```

When the pre-value of a property evaluates to an object, all further properties that are accessed of this object are the new values (upon completion of the operation) of this object. So:

```
a.b@pre.c     -- takes the old value of property b of a, say x
              -- and then the new value of c of x.
a.b@pre.c@pre-- takes the old value of property b of a, say x
              -- and then the old value of c of x.
```

The ‘@pre’ postfix is allowed only in OCL expressions that are part of a Postcondition. Asking for a current property of an object that has been destroyed during execution of the operation results in OclUndefined. Also, referring to the previous value of an object that has been created during execution of the operation results in OclUndefined.

2.5.15 Tuples

It is possible to compose several values into a *tuple*. A tuple consists of named parts, each of which can have a distinct type. Some examples of tuples are:

```
Tuple {name: String = ‘John’,  age: Integer = 10}
Tuple {a: Collection(Integer) = Set{1, 3, 4}, b: String = ‘foo’, c: String = ‘bar’}
```

This is also the way to write tuple literals in OCL; they are enclosed in curly brackets, and the parts are separated by commas. The type names are optional, and the order of the parts is unimportant. Thus:

```
Tuple {name: String = ‘John’, age: Integer = 10} is equivalent to
Tuple {name = ‘John’, age = 10} and to
Tuple {age = 10, name = ‘John’}
```

Also, note that the values of the parts may be given by arbitrary OCL expressions, so for example we may write:

```
context Person def:
attr statistics : Set(TupleType(company: Company, numEmployees: Integer,
                         wellpaidEmployees: Set(Person), totalSalary: Integer)) =
      managedCompanies->collect(c |
       Tuple { company: Company = c,
           numEmployees: Integer = c.employee->size(),
          wellpaidEmployees: Set(Person) = c.job->select(salary>10000).employee->asSet(),
           totalSalary: Integer = c.job.salary->sum()
          }
      )
```

This results in a bag of tuples summarizing the company, number of employees, the best paid employees and total salary costs of each company a person manages.

The parts of a tuple are accessed by their names, using the same dot notation that is used for accessing attributes. Thus:

```
Tuple {x: Integer = 5, y: String = ‘hi’}.x = 5
```

is a true, if somewhat pointless, expression. Using the definition of statistics above, we can write:

```
context Person inv:
statistics->sortedBy(totalSalary)->last().wellpaidEmployees->includes(self)
```

This asserts that a person is one of the best-paid employees of the company with the highest total salary that he manages. In this expression, both ‘totalSalary’ and ‘wellpaidEmployees’ are accessing tuple parts.

2.6 COLLECTION OPERATIONS

OCL defines many operations on the collection types. These operations are specifically meant to enable a flexible and powerful way of projecting new collections from existing ones. The different constructs are described in the following sections.

2.6.1 Select and Reject Operations

Sometimes an expression using operations and navigations results in a collection, while we are interested only in a special subset of the collection. OCL has special constructs to specify a selection from a specific collection. These are the *select* and *reject* operations. The select specifies a subset of a collection. A select is an operation on a collection and is specified using the arrow-syntax:

```
collection->select( ... )
```

The parameter of select has a special syntax that enables one to specify which elements of the collection we want to select. There are three different forms, of which the simplest one is:

```
collection->select( boolean-expression )
```

This results in a collection that contains all the elements from *collection* for which the *boolean-expression* evaluates to true. To find the result of this expression, for each element in *collection* the expression *boolean-expression* is evaluated. If this evaluates to true, the element is included in the result collection, otherwise not. As an example, the following OCL expression specifies that the collection of all the employees older than 50 years is not empty:

```
context Company inv:
self.employee->select(age > 50)->notEmpty()
```

The *self.employee* is of type Set(Person). The *select* takes each person from *self.employee* and evaluates *age > 50* for this person. If this results in *true*, then the person is in the result Set.

As shown in the previous example, the context for the expression in the select argument is the element of the collection on which the select is invoked. Thus the *age* property is taken in the context of a person.

In the above example, it is impossible to refer explicitly to the persons themselves; you can only refer to properties of them. To enable to refer to the persons themselves, there is a more general syntax for the select expression:

```
collection->select( v | boolean-expression-with-v )
```

The variable *v* is called the iterator. When the select is evaluated, *v* iterates over the *collection* and the *boolean-expression-with-v* is evaluated for each *v*. The *v* is a reference to the object from the collection and can be used to refer to the objects themselves from the *collection*. The two examples below are identical:

```
context Company inv:
self.employee->select(age > 50)->notEmpty()

context Company inv:
self.employee->select(p | p.age > 50)->notEmpty()
```

The result of the complete select is the collection of persons *p* for which the *p.age > 50* evaluates to True. This amounts to a subset of *self.employee.*

As a final extension to the select syntax, the expected type of the variable v can be given. The select now is written as:

```
collection->select( v : Type | boolean-expression-with-v )
```

The meaning of this is that the objects in *collection* must be of type *Type*. The next example is identical to the previous examples:

```
context Company inv:
self.employee.select(p : Person | p.age > 50)->notEmpty()
```

The compete select syntax now looks like one of:

```
collection->select( v : Type | boolean-expression-with-v )
collection->select( v | boolean-expression-with-v )
collection->select( boolean-expression )
```

The *reject* operation is identical to the select operation, but with reject we get the subset of all the elements of the collection for which the expression evaluates to False. The reject syntax is identical to the select syntax:

```
collection->reject( v : Type | boolean-expression-with-v )
collection->reject( v | boolean-expression-with-v )
collection->reject( boolean-expression )
```

As an example, specify that the collection of all the employees who are **not** married is empty:

```
context Company inv:
self.employee->reject( isMarried )->isEmpty()
```

The reject operation is available in OCL for convenience, because each reject can be restated as a select with the negated expression. Therefore, the following two expressions are identical:

```
collection->reject( v : Type | boolean-expression-with-v )
collection->select( v : Type  | not (boolean-expression-with-v) )
```

2.6.2 Collect Operation

As shown in the previous section, the select and reject operations always result in a sub-collection of the original collection. When we want to specify a collection which is derived from some other collection, but which contains different objects from the original collection (i.e., it is not a sub-collection), we can use a *collect* operation. The collect operation uses the same syntax as the select and reject and is written as one of:

```
collection->collect( v : Type | expression-with-v )
collection->collect( v | expression-with-v )
collection->collect( expression )
```

The value of the reject operation is the collection of the results of all the evaluations of *expression-with-v.*

An example: specify the collection of *birthDates* for all employees in the context of a company. This can be written in the context of a Company object as one of:

```
self.employee->collect( birthDate )
self.employee->collect( person | person.birthDate )
self.employee->collect( person : Person | person.birthDate )
```

An important issue here is that the resulting collection is not a Set, but a Bag. When more than one employee has the same value for *birthDate*, this value will be an element of the resulting Bag more than once. The Bag resulting from the *collect* operation always has the same size as the original collection.

It is possible to make a Set from the Bag, by using the asSet property on the Bag. The following expression results in the Set of different *birthDates* from all employees of a Company:

```
self.employee->collect( birthDate )->asSet()
```

Shorthand for Collect

Because navigation through many objects is very common, there is a shorthand notation for the collect that makes the OCL expressions more readable. Instead of

```
self.employee->collect(birthdate)
```

we can also write:

```
self.employee.birthdate
```

In general, when we apply a property to a collection of Objects, then it will automatically be interpreted as a *collect* over the members of the collection with the specified property.

For any *propertyname* that is defined as a property on the objects in a collection, the following two expressions are identical:

```
collection.propertyname
collection->collect(propertyname)
```

and so are these if the property is parameterized:

```
collection.propertyname (par1, par2, ...)
collection->collect (propertyname(par1, par2, ...))
```

2.6.3 ForAll Operation

Many times a constraint is needed on all elements of a collection. The forAll operation in OCL allows specifying a Boolean expression, which must hold for all objects in a collection:

```
collection->forAll( v : Type | boolean-expression-with-v )
collection->forAll( v | boolean-expression-with-v )
collection->forAll( boolean-expression )
```

This *forAll* expression results in a Boolean. The result is true if the *boolean-expression-with-v* is true for all elements of *collection*. If the *boolean-expression-with-v* is false for one or more *v* in *collection*, then the complete expression evaluates to false. For example, in the context of a company:

```
context Company
inv:  self.employee->forAll( age <= 65 )
inv:  self.employee->forAll( p | p.age <= 65 )
inv:  self.employee->forAll( p : Person | p.age <= 65 )
```

These invariants evaluate to true if the *age* property of each employee is less or equal to 65.

The *forAll* operation has an extended variant in which more then one iterator is used. Both iterators will iterate over the complete collection. Effectively this is a *forAll* on the Cartesian product of the collection with itself.

```
context Company inv:
self.employee->forAll( e1, e2 : Person |
            e1 <> e2 implies e1.forename <> e2.forename)
```

This expression evaluates to true if the forenames of all employees are different. It is semantically equivalent to:

```
context Company inv:
self.employee->forAll (e1 | self.employee->forAll (e2 |
                      e1 <> e2 implies e1.forename <> e2.forename))
```

2.6.4 Exists Operation

Many times one needs to know whether there is at least one element in a collection for which a constraint holds. The *exists* operation in OCL allows you to specify a Boolean expression which must hold for at least one object in a collection:

```
collection->exists( v : Type | boolean-expression-with-v )
collection->exists( v | boolean-expression-with-v )
collection->exists( boolean-expression )
```

This exists operation results in a Boolean. The result is true if the *boolean-expression-with-v* is true for at least one element of *collection*. If the *boolean-expression-with-v* is false for all *v* in *collection*, then the complete expression evaluates to false. For example, in the context of a company:

```
context Company inv:
self.employee->exists( forename = 'Jack' )
```

```
context Company inv:
self.employee->exists( p | p.forename = 'Jack' )

context Company inv:
self.employee->exists( p : Person | p.forename = 'Jack' )
```

These expressions evaluate to true if the *forename* property of at least one employee is equal to ‘Jack.’

2.6.5 Iterate Operation

The *iterate* operation is slightly more complicated, but is very generic. The operations *reject, select, forAll, exists, collect*, can all be described in terms of *iterate*. An accumulation builds one value by iterating over a collection.

```
collection->iterate( elem : Type; acc : Type = <expression> |
                expression-with-elem-and-acc )
```

The variable *elem* is the iterator, as in the definition of *select, forAll,* etc. The variable *acc* is the accumulator. The accumulator gets an initial value *<expression>*. When the iterate is evaluated, *elem* iterates over the *collection* and the *expression-with-elem-and-acc* is evaluated for each *elem*. After each evaluation of *expression-with-elem-and-acc*, its value is assigned to *acc*. In this way, the value of *acc* is built up during the iteration of the collection. The collect operation described in terms of iterate will look like:

```
collection->collect(x : T | x.property)
-- is identical to:
collection->iterate(x : T; acc : T2 = Bag{} |
                    acc->including(x.property))
```

Or written in Java-like pseudocode the result of the iterate can be calculated as:

```
iterate(elem : T; acc : T2 = value)
{
   acc = value;
   for(Enumeration e = collection.elements() ; e.hasMoreElements(); ){
       elem = e.nextElement();
       acc  = <expression-with-elem-and-acc>
   }
   return acc;
}
```

Although the Java pseudo code uses a ‘next element’, the *iterate* operation is defined not only for Sequqnce, but for each collection type. The order of the iteration through the elements in the collection is not defined for Set and Bag. For a Sequence the order is the order of the elements in the sequence.

2.7 MESSAGES IN OCL

This section contains some examples of the concrete syntax and explains the finer details of the message expression. In earlier versions the phrase "actions in OCL" was used, but message was found to capture the meaning more precisely.

2.7.1 Calling operations and sending signals

To specify that communication has taken place, the hasSent (‘^’) operator is used:

```
context Subject::hasChanged()
post:  observer^update(12, 14)
```

The *observer^update(12, 14)* results in true if an update message with arguments 12 and 14 was sent to *observer* during the execution of the operation. *Update()* is either an *Operation* that is defined in the class of *observer*, or it is a *Signal* specified in the UML model. The argument(s) of the message expression (12 and 14 in this example) must conform to the parameters of the operation/signal definition.

If the actual arguments of the operation/signal are not known, or not restricted in any way, it can be left unspecified. This is shown by using a question mark. Following the question mark is an optional type, which may be needed to find the correct operation when the same operation exists with different parameter types.

```
context Subject::hasChanged()
post:  observer^update(? : Integer, ? : Integer)
```

This example states that the message update has been sent to *observer*, but that the values of the parameters are not known.

OCL also defines a special *OclMessage* type. One can get the actual OclMessages through the message operator: ^^.

```
context Subject::hasChanged()
post:  observer^^update(12, 14)
```

This results in the Sequence of messages sent. Each element of the collection is sn instance of *OclMessage*. In the remainder of the constraint one can refer to the parameters of the operation using their formal parameter name from the operation definition. If the operation update has been defined with formal parameters named *i* and *j*, then we can write:

```
context Subject::hasChanged()
post: let messages : Sequence(OclMessage) = observer^^update(? : Integer, ? : Integer) in
      messages->notEmpty() and
      messages->exists( m | m.i > 0 and m.j >= m.i )
```

The value of the parameter *i* is not known, but it must be greater than zero and the value of parameter j must be larger or equal to i..

Because the ^^ operator results in an instance of *OclMessage*, the message expression can also be used to specify collections of messages sent to different targets. For an observer pattern we can write:

```
context Subject::hasChanged()
post:  let messages : Sequence(OclMessage) =
                   observers->collect(o | o^^update(? : Integer, ? : Integer) ) in
       messages->forAll(m | m.i <= m.j )
```

Messages is now a set of *OclMessage* instances, where every *OclMessage* instance has one of the *observers* as a *target*.

2.7.2 Accessing result values

A signal sent message is by definition asynchronous, so there never is a return value. If there is a logical return value it must be modeled as a separate signal message. Yet, for an operation call there is a potential return value. This is only available if the operation has already returned (not neccesary if the operation call is aynchronous), and it specifies a return type in its definition. The standard operation *result()* of OclMessage contains the return value of the called operation. If getMoney(...) is an operation on Company that returns a boolean, as in *Company::getMoney(amount : Integer) : Boolean*, we can write:

```
context Person::giveSalary(amount : Integer)
post: let message : OclMessage = company^getMoney(amount) in
      message.hasReturned()                   -- getMoney was sent and returned
      and
      message.result() = true                 -- the getMoney call returned true
```

As with the previous example we can also access a collection of return values from a collection of OclMessages. If *message.hasReturned()* is false, then *message.result()* will be undefined.

2.7.3 An example

This section shows an example of using the OCL message expression.

The Example and Problem

Suppose we have build a component, which takes any form of input and transforms it into garbage (aka encrypts it). The component *GarbageCan* uses an interface *UsefulInformationProvider* which must be implemented by users of the component to provide the input. The operation *getNextPieceOfGarbage* of *GarbageCan* can then be used to retrieve the garbled data. Figure 2-4 shows the component's class diagram. Note that none of the operations are marked as queries.

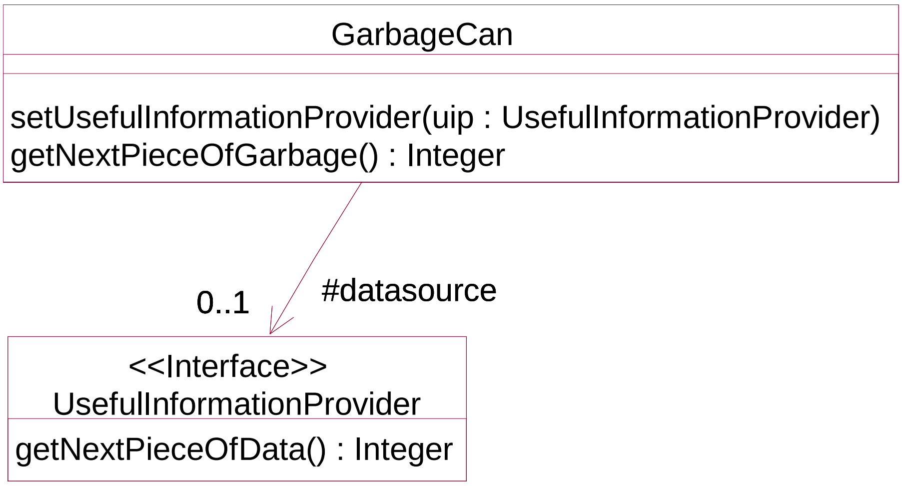


Figure 2-4 *OclMessage Example*

When selling the component, we do not want to give the source code to our customers. However, we want to specify the component's behavior as precisely as possible. So, for example, we want to specify, what *getNextPieceOfGarbage* does. Note that we cannot write:

```
context GarbageCan::getNextPieceOfGarbage() : Integer
post: result = (datasource.getNextPieceOfData() * .7683425 + 10000) / 20 + 3
```

because *UsefulInformationProvider::getNextPieceOfData()* is not a query (e.g., it may increase some internal pointer so that it can return the next piece of data at the next call). Still we would like to say something about how the garbage is derived from the original data.

The solution

To solve this problem, we can use an OclMessage to represent the call to *getNextPieceOfData*. This allows us to check for the result. Note that we need to demand that the call has returned before accessing the result:

```
context GarbageCan::getNextPieceOfGarbage() : Integer
post: let message : OclMessage = datasource^^getNextPieceOfData()->first() in
      message.hasReturned()
      and
      result = message.result() * .7683425 + 10000) / 20 + 3
```

2.8 RESOLVING PROPERTIES

For any property (attribute, operation, or navigation), the full notation includes the object of which the property is taken. As seen in Section 2.3.3, *self* can be left implicit, and so can the iterator variables in collection operations.

At any place in an expression, when an iterator is left out, an implicit iterator-variable is introduced. For example in:

```
context Person inv:
employer->forAll( employee->exists( lastName = name) )
```

three implicit variables are introduced. The first is *self*, which is always the instance from which the constraint starts. Secondly an implicit iterator is introduced by the *forAll* and third by the *exists*. The implicit iterator variables are unnamed. The properties *employer*, *employee*, *lastName* and *name* all have the object on which they are applied left out. Resolving these goes as follows:

- at the place of *employer* there is one implicit variable: *self : Person*. Therefore *employer* must be a property of *self*.
- at the place of *employee* there are two implicit variables: *self : Person* and *iter1 : Company*. Therefore *employer* must be a property of either *self* or *iter1*. If *employee* is a property of both *self* and *iter1* then it is defined to belong to the variable in the most inner scope, which is *iter1*.
- at the place of *lastName* and *name* there are three implicit variables: *self : Person* , *iter1 : Company* and *iter2 : Person*. Therefore *lastName* and *name* must both be a property of either *self* or *iter1* or *iter2*. In the UML model property *name* is a property of *iter1*. However, *lastName* is a property of both *self* and *iter2*. This is ambiguous and therefore the *lastName* refers to the variable in the most inner scope, which is *iter2*.

Both of the following invariant constraint are correct, but have a different meaning:

```
context Person
inv: employer->forAll( employee->exists( p | p.lastName = name) )
inv: employer->forAll( employee->exists( self.lastName = name) )
```

3

Abstract Syntax

This section describes the abstract syntax of the OCL. In this abstract syntax a number of metaclasse from the UML metamodel are imported. These metaclasses are shown in the models with the annotation '(from <UML package>)' and shown with a transparant fill color. All metaclasses defined as part of the OCL abstract syntax are shown with a light gray background.

3.1 INTRODUCTION

The abstract syntax as described below defines the concepts that are part of the OCL using a MOF compliant metamodel. The abstract syntax is divided into several packages.

- The *Types* package describes the concepts that define the type system of OCL. It shows which types are predefined in OCL and which types are deduced from the UML models.
- The *Expressions* package describes the structure of OCL expressions.

3.2 THE TYPES PACKAGE

OCL is a typed language. Each expression has a type which is either explicitly declared or can be statically derived. Evaluation of the expression yields a value of this type. Therefore, before we can define expressions, we have to provide a model for the concept of type. A metamodel for OCL types is shown in this section. Note that instances of the classes in the metamodel are the types themselves (e.g. Integer) not instances of the domain they represent (e.g. -15, 0, 2, 3).

The model in figure 3-1 shows the OCL types. The basic type is the UML Classifier, which includes all subtypes of Classifier from the UML infrastructure.

In the model the CollectionType and its subclasses and the TupleType are special. One can never instantiate all collection types, because there is an infinite number, especially when nested collections are taken in account. Users will never instantiate these types explicitly. Conceptually all these types do exist, but such a type should be (lazily) instantiated by a tool, whenever it is needed in an expression.

In comparison with UML 1.4 the type OclType has been removed from the type hierarchy. This means that a Classifier is not a valid OCL expression any more.

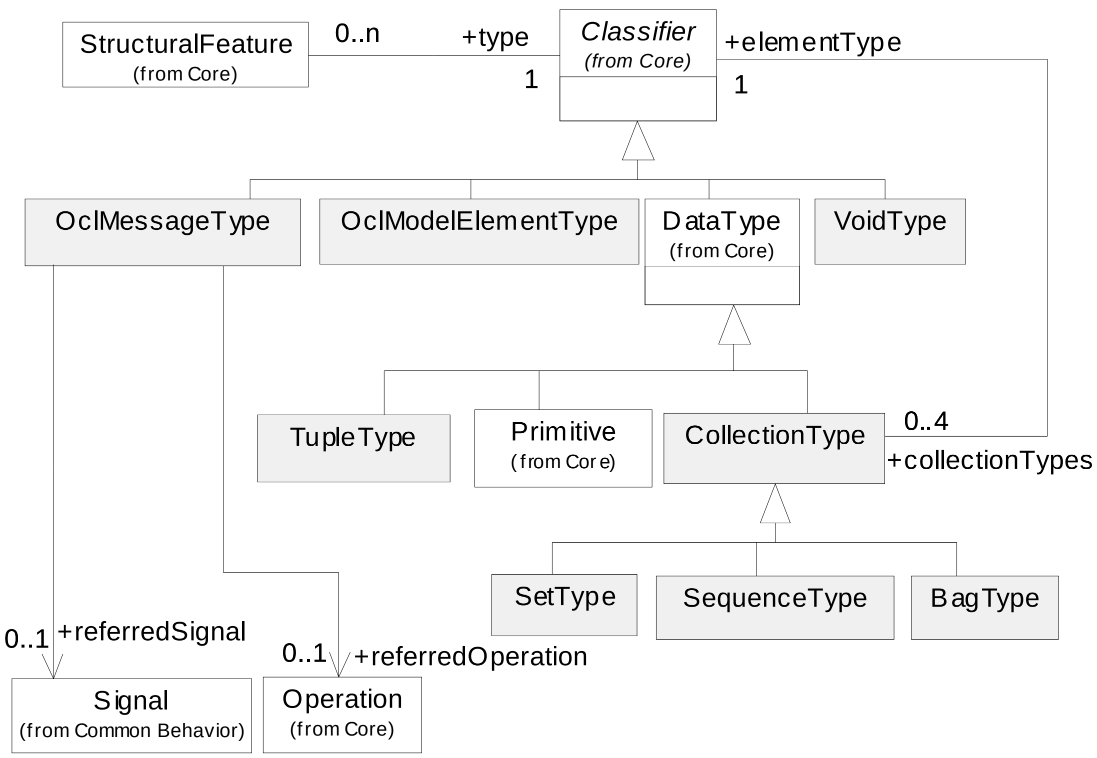


Figure 3-1 *Abstract syntax kernel metamodel for OCL Types*

BagType

BagType is a collection type which describes a multiset of elements where each element may occur multiple times in the bag. The elements are unordered. Part of a *BagType* is the declaration of the type of its elements.

CollectionType

CollectionType describes a list of elements of a particular given type. *CollectionType* is an abstract class. Its concrete subclasses are *SetType*, *SequenceType* and *BagType* types. Part of every collection type is the declaration of the type of its elements, i.e. a collection type is *parameterized* with an element type. In the metamodel, this is shown as an association from *CollectionType* to *Classifier*. Note that there is no restriction on the element type of a collection type. This means in particular that a collection type may be parameterized with other collection types allowing collections to be nested arbitrarily deep.

Associations

elementType — The type of the elements in a collection. All elements in a collection must conform to this type.

OclMessageType

OclMessageType describe ocl messages. Like to the collection types, *OclMessageType* describes a set of types in the standard library. Part of every *OclMessageType* is a reference to the declaration of the type of its operation or signal, i.e. an ocl message type is *parameterized* with an operation or signal. In the metamodel, this is shown as an association from *OclMessageType* to *Operation* and to *Signal*. *OclMessageType* is part of the abstract syntax of OCL, residing on M2 level. Its instances, called *OclMessage,* and subtypes of *OclMessage,* reside on M1 level.

Associations

referredSignal	The *Signal* that is sent by the message.
referredOperation	The *Operation* that is called by the message.

OclModelElementType

OclModelElementType represents the types of elements that are *ModelElements* in the UML metamodel. It is used to be able to refer to states and classifiers in e.g. *oclInState*(...) and *oclIsKindOf*(...)

SequenceType

SequenceType is a collection type which describes a list of elements where each element may occur multiple times in the sequence. The elements are ordered by their position in the sequence. Part of a *SequenceType* is the declaration of the type of its elements.

SetType

SetType is a collection type which describes a set of elements where each distinct element occurs only once in the set. The elements are not ordered. Part of a *SetType* is the declaration of the type of its elements.

TupleType

TupleType (informaly known as record type or struct) combines different types into a single aggregate type. The parts of a *TupleType* are described by its attributes, each having a name and a type. There is no restriction on the kind of types that can be used as part of a tuple. In particular, a *TupleType* may contain other tuple types and collection types. Each attribute of a *TupleType* represents a single feature of a *TupleType*. Each part is to uniquely identified by its name.

VoidType

VoidType represents a type that conforms to all types. The only instance of *VoidType* is *OclVoid*, which is further defined in the standard library. Furthermore *OclVoid* has exactly one instance called *OclUndefined*.

3.2.1 Type Conformance

The type conformance rules are formally underpinned in the Semantics section of the specification. To ensure that the rules are accessible to UML modellers they are specified in this section using OCL. For this, the additional operation *conformsTo(c : Classifier) : Boolean* is defined on *Classifier*. It evaluates to true, if the self *Classifier* conforms to the argument *c. The following OCL statements define type conformance for individual types.*

BagType

[1] Different bag types conform to each other if their element types conform to each other.

```
context BagType
inv: BagType.allInstances()->forAll(b |
        self.elementType.conformsTo(b.elementType) implies self.conformsTo(b))
```

Classifier

[1] Conformance is a transitive relationship.

```
context Classifier
inv Transitivity: Classifier.allInstances()->forAll(x|Classifier.allInstances()
                    ->forAll(y|
                       (self.conformsTo(x) and x.conformsTo(y)) implies self.conformsTo(y)))
```

[2] All classifiers except collections conform to OclAny.

```
context Classifier
inv: (not self.oclIsKindOf (CollectionType)) implies
      Primitive.allInstances()->forAll(p | (p.name = 'OclAny') implies self.conformsTo(p))
```

[3] Classes conform to superclasses and interfaces that they realize.

```
context Class
inv : self.generalization.parent->forAll (p |
          (p.oclIsKindOf(Class) or p.oclIsKindOf(Interface)) implies
                                            self.conformsTo(p.oclAsType(Classifier)))
```

[4] Interfaces conforms to super interfaces.

```
context Interface
inv : self.generalization.parent->forAll (p |
              p.oclIsKindOf(Interface) implies self.conformsTo(p.oclAsType(Interface)))
```

[5] The Conforms operation between Types is reflexive, a Classifier always conform to itself.

```
context Classifier
inv: self.conformsTo(self)
```

[6] The Conforms operation between Types is anti-symmetric.

```
context Classifier
inv: Classifier.allInstances()->forAll(t1, t2 |
           (t1.conformsTo(t2) and t2.conformsTo(t1)) implies t1 = t2)
```

CollectionType

[1] Specific collection types conform to collection type.

```
context CollectionType
inv: -- all instances of SetType, SequenceType, BagType conform to a
     -- CollectionType if the elementTypes conform
        CollectionType.allInstances()->forAll (c |
                c.oclIsTypeOf(CollectionType) and
                self.elementType.conformsTo(c.elementType) implies
                          self.conformsTo(c))
```

[2] Collections do not conform to any primitive type.

```
context CollectionType
inv: Primitive.allInstances()->forAll (p | not self.conformsTo(p))
```

[3] Collections of non-conforming types do not conform.

```
context CollectionType
inv: CollectionType.allInstances()->forAll (c |
    (not self.elementType.conformsTo (c.elementType)) implies (not self.conformsTo (c)))
```

Primitive

[1] Integer conforms to real.

```
context Primitive
inv: (self.name = 'Integer') implies
         Primitive.allInstances()->forAll (p | (p.name = 'Real') implies
                                                (self.conformsTo(p))))
```

SequenceType

[1] Different sequence types conform to each other if their element types conform to each other.

```
context SequenceType
inv: SequenceType.allInstances()->forAll(s |
               self.elementType.conformsTo(s.elementType) implies self.conformsTo(s))
```

SetType

[1] Different set types conform to each other if their element types conform to each other.

```
context SetType
inv: SetType.allInstances()->forAll(s |
                  self.elementType.conformsTo(s.elementType) implies self.conformsTo(s))
```

TupleType

[1] Tuple types conform to each other when their names and types conform to each other. Note that *allAttributes* is an additional operation in the UML 1.4.

```
context TupleType
inv: TupleType.allInstances()->forAll (t |
       ( t.allAttributes()->forAll (tp |
          -- make sure at least one tuplepart has the same name
          -- (uniqueness of tuplepart names will ensure that not two
          -- tupleparts have the same name within one tuple)
          self.allAttributes()->exists(stp|stp.name = tp.name) and
          -- make sure that all tupleparts with the same name conforms.
          self.allAttributes()->forAll(stp | (stp.name = tp.name) and
                                                  stp.type.conformsTo(tp.type))
       )
       implies
            self.conformsTo(t)
     ) )
```

VoidType

[1] Void conforms to all other types.

```
context VoidType
inv: Classifier.allInstances()->forAll (c | self.conformsTo (c))
```

3.2.2 Well-formedness Rules for the Types Package

BagType

[1] The name of a bag type is “Bag” followed by the element type’s name in parentheses.

```
context BagType
inv: self.name = ’Bag(’ + self.elementType.name + ’)’
```

CollectionType

[1] The name of a collection type is “Collection” followed by the element type’s name in parentheses.

```
context CollectionType
inv: self.name = 'Collection(' + self.elementType.name + ')'
```

Classifier

[1] For each classifier at most one of each of the different collection types exist.

```
context Classifier
inv: collectionTypes->select(oclIsTypeOf(CollectionType))->size() <= 1
inv: collectionTypes->select(oclIsTypeOf(BagType       ))->size() <= 1
inv: collectionTypes->select(oclIsTypeOf(SequenceType  ))->size() <= 1
inv: collectionTypes->select(oclIsTypeOf(SetType       ))->size() <= 1
```

OclMessageType

[1] OclMessageType has either a link with a Signal or with an operation, but not both

```
context OclMessageType
inv: referredOperation->size() + referredSignal->size() = 1
```

[2] The parameters of the referredOperation become attributes of the instance of OclMessageType

```
context OclMessageType
inv: referredOperation->size() = 1 implies
         self.feature = referredOperation.parameter.asAttribute()
```

[3] The attributes of the referredSignal become attributes of the instance of OclMessageType

```
context OclMessageType
inv: referredSignal->size() = 1 implies
         self.feature = referredSignal.feature
```

SequenceType

[1] The name of a sequence type is “Sequence” followed by the element type’s name in parentheses.

```
context SequenceType
inv: self.name = 'Sequence(' + self.elementType.name + ')'
```

SetType

[1] The name of a set type is “Set” followed by the element type’s name in parentheses.

```
context SetType
inv: self.name = 'Set(' + self.elementType.name + ')'
```

TupleType

[1] The name of a tuple type includes the names of the individual parts and the types of those parts.

```
context TupleType
inv: name =
  'Tuple('.concat (
    Sequence{1..allAttributes()->size()}->iterate (pn; s: String = '' |
      let p: Attribute = allAttributes()->at (pn) in (
        s.concat (
          (if (pn>1) then ',' else '' endif)
          .concat (p.name).concat (':')
```

```
            .concat (p.type.name)
          )
        )
      )
    ).concat (’)’)
```

[2] All parts belonging to a tuple type have unique names.

```
context TupleType
inv: -- always true, because attributes must have unique names.
```

[3] A TupleType instance has only features that are Attributes (tuple parts).

```
context TupleType
inv: feature->forAll (f | f.oclIsTypeOf(Attribute))
```

3.3 THE EXPRESSIONS PACKAGE

This section defines the abstract syntax of the expressions package. This package defines the structure that OCL expressions can have. An overview of the inheritance relationships between all classes defined in this package is shown in figure 3-8 on page 3-26.

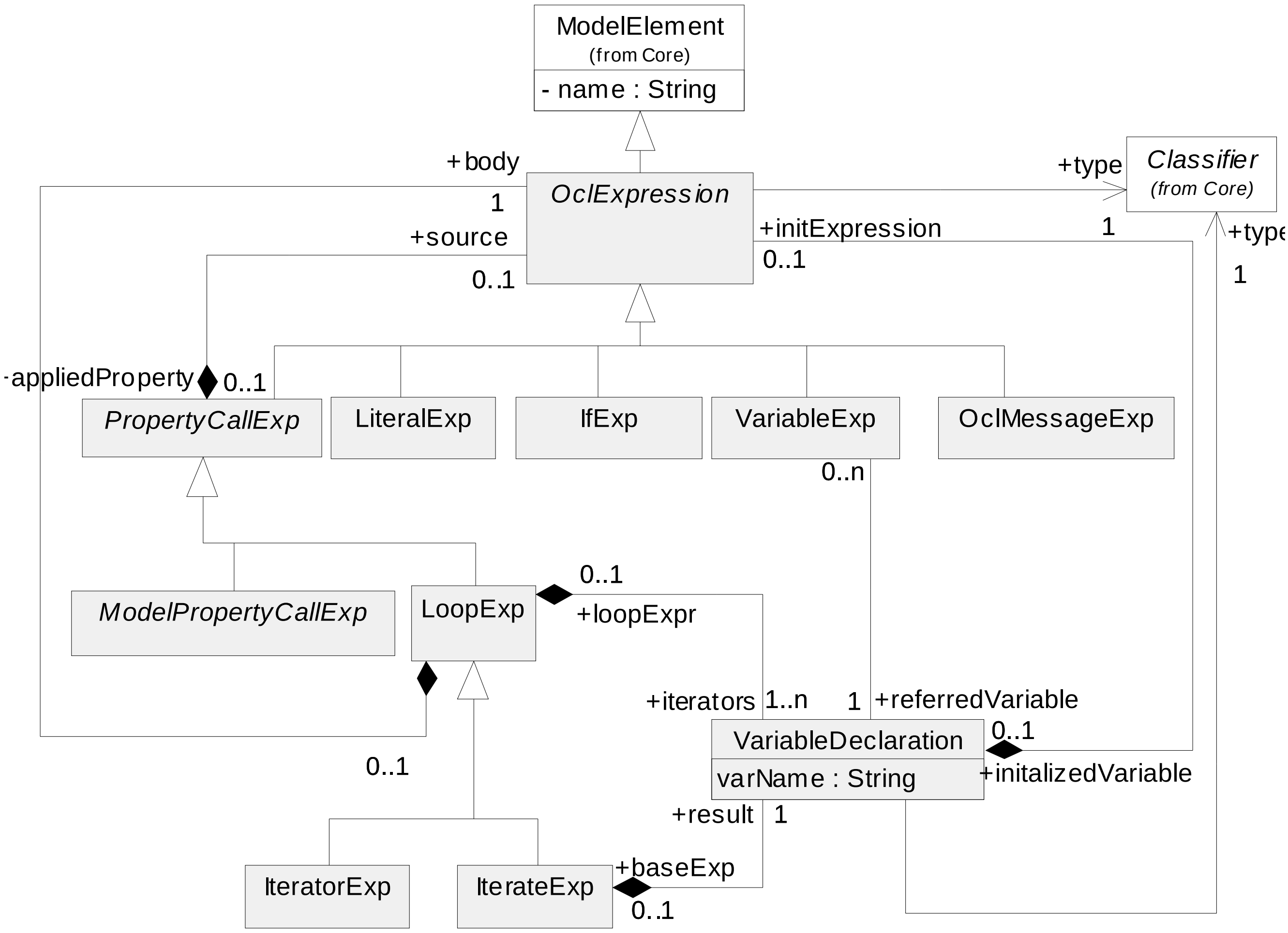


Figure 3-2 *The basic structure of the abstract syntax kernel metamodel for Expressions*

3.3.1 Expressions Core

Figure 3-2 on page 3-8 shows the core part of the Expressions package. The basic structure in the package consists of the classes *OclExpression*, *PropertyCallExp* and *VariableExp*. An *OclExpression* always has a type, which is usually not explicitly modeled, but derived. Each *PropertyCallExp* has exactly one source, identified by an *OclExpression*. In this section we use the term ’property’, which is a generalization of *Feature*, *AssociationEnd* and predefined iterating OCL collection operations.

A *ModelPropertyCallExp* generalizes all propertycalls that refer to *Features* or *associations* or *AssociationEnds* in the UML metamodel. In figure 3-3 on page 3-11 the various subtypes of *ModelPropertyCallExp* are defined.

Most of the remainder of the expressions package consists of a specification of the different subclasses of *PropertyCallExp* and their specific structure. From the metamodel it can be deduced that an OCL expression always starts with a variable or literal, on which a property is recusively applied.

IfExp

An *IfExp* is defined in section 3.3.3 (“If Expressions”), but included in this diagram for completeness.

IterateExp

An *IterateExp* is an expression which evaluates its *body* expression for each element of a collection. It acts as a loop construct that iterates over the elements of its *source* collection and results in a value. An iterate expression evaluates its *body* expression for each element of its *source* collection. The evaluated value of the *body* expression in each iteration-step becomes the new value for the *result* variable for the succeding iteration-step. The result can be of any type and is defined by the *result* association. The *IterateExp* is the most fundamental collection expression defined in the OCL Expressions package.

Associations

result	The *VariableDeclaration* that represents the result variable.

IteratorExp

An *IteratorExp* is an expression which evaluates its *body* expression for each element of a collection. It acts as a loop construct that iterates over the elements of its *source* collection and results in a value. The type of the iterator expression depends on the name of the expression, and sometimes on the type of the associated *source* expression. The *IteratorExp* represents all other predefined collection operations that use an iterator. This includes select, collect, reject, forAll, exists, etc. The OCL Standard Library defines a number of predefined iterator expressions. Their semantics is defined in terms of the iterate expression in , see 6.6 (“Predefined Iterator Expressions”).

LiteralExp

A *LiteralExp* is an expression with no arguments producing a value. In general the result value is identical with the expression symbol. This includes things like the integer 1 or literal strings like ’this is a LiteralExp’.

LoopExp

A *LoopExp* is an expression that respresent a loop construct over a collection. It has an iterator variable that represents the elements of the collection during iteration. The body expression is evaluated for each element in the collection. The result of a loop expression depends on the specific kind and its name.

Associations

iterators	The *VariableDeclarations* that represents the iterator variables. These variables are, each in its turn, bound to every element value of the *source* collection while evaluating the *body* expression.
body	The *OclExpression* that is evaluated for each element in the source collection.

ModelPropertyCallExp

A *ModelPropertyCall* expression is an expression that refers to a property that is defined for a *Classifier* in the UML model to which this expression is attached. Its result value is the evaluation of the corresponding property. In section 3.3.2 (“Model PropertyCall Expressions”) the various subclasses of *ModelPropertyCallExp* are defined.

OclExpression

An *OclExpression* is an expression that can be evaluated in a given environment. *OclExpression* is the abstract superclass of all other expressions in the metamodel. It is the top-level element of the OCL Expressions package. Every *OclExpression* has a type that can be statically determined by analyzing the expression and its context. Evaluation of an expression results in a value. Expressions with boolean result can be used as constraints, e.g. to specify an invariant of a class. Expressions of any type can be used to specify queries, initial attribute values, target sets, etc..

The environment of an *OclExpression* defines what model elements are visible and can be referred to in an expression. At the topmost level the environment will be defined by the *ModelElement* to which the OCL expression is attached, for example by a *Classifier* if the OCL expression is used as an invariant. On a lower level, each iterator expression can also introduce one or more iterator variables into the environment. the environment is not modeled as a separate metaclass, because it can be completely derived using derivation rules. The complete derivation rules can be found in chapter 4 (“Concrete Syntax”).

Associations

appliedProperty	The property that is applied to the instance that results from evaluating this *OclExpression*.
type	The type of the value that is the result of evaluating the *OclExpression*.
parentOperation	The *OperationCallExp* where this *OclExpression* is an argument of. See figure 3-3 on page 3-11.
initializedVariable	The variable of which the result of this expression is the initial value.

OclMessageExp

OclMessageExp is defined in section 3.3.4 (“Message Expressions”), but included in this diagram for completeness.

PropertyCallExp

A *PropertyCallExp* is an expression that refers to a property (operation, attribute, association end, predefined iterator for collections). Its result value is the evaluation of the corresponding property. This is an abstract metaclass.

Associations

source	The result value of the source expression is the instance that performs the property call.

VariableDeclaration

A *VariableDeclaration* declares a variable name and binds it to a type. The variable can be used in expressions where the variable is in scope. This metaclass represents amongst others the variables *self* and *result* and the variables defined using the Let expression.

Associations

initExpression	The *OclExpression* that represents the initial value of the variable. Depending on the role that a variable declaration plays, the init expression might be mandatory.
type	The *Classifier* which represents the type of the variable.

Attributes

varName	The *String* that is the name of the variable.

VariableExp

A *VariableExp* is an expression which consists of a reference to a variable. References to the variables *self* and *result* or to variables defined by Let espressions are examples of such variable expressions.

Associations

referredVariable	The *VariableDeclaration* to which this variable expression refers. In the case of a self expression the variable declaration is the definition of the self variable.

3.3.2 Model PropertyCall Expressions

A *ModelPropertyCallExp* can refer to any of the subtypes of *Feature* as defined in the UML kernel. This is shown in figure 3-3 by the three different subtypes , each of which is associated with its own type of *ModelElement*.

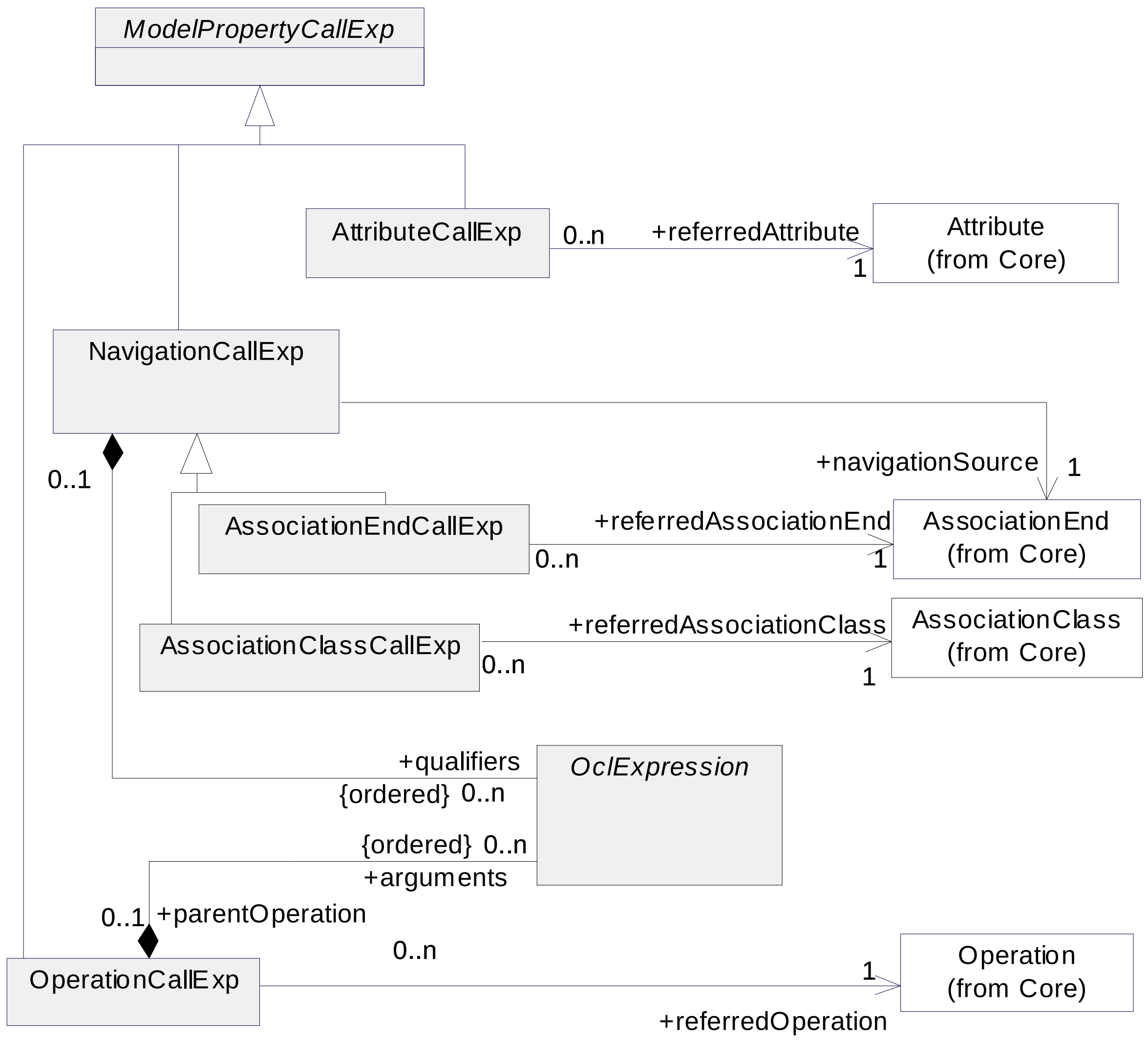


Figure 3-3 *Abstract syntax metamodel for ModelPropertyCallExp in the Expressions package*

AssociationEndCallExp

An *AssociationEndCallExp* is a reference to an *AssociationEnd* defined in a UML model. It is used to determine objects linked to a target object by an association. The expression refers to these target objects by the role name of the association end connected to the target class.

Associations

referredAssociationEnd	The *AssociationEnd* to which this *AssociationEndCallExp* is a reference. This refers to an *AssociationEnd* of an *Association* that is defined in the UML model.

AssociationClassCallExp

An *AssociationClassCallExp* is a reference to an *AssociationClass* defined in a UML model. It is used to determine objects linked to a target object by an association class. The expression refers to these target objects by the name of the target associationclass.

Associations

referredAssociationClass	The *AssociationClass* to which this *AssociationClassCallExp* is a reference. This refers to an *AssociationClass* that is defined in the UML model.

AttributeCallExp

An *AttributeCallExpression* is a reference to an *Attribute* of a *Classifier* defined in a UML model. It evaluates to the value of the attribute.

Associations

referredAttribute	The *Attribute* to which this *AttributeCallExp* is a reference.

NavigationCallExp

A *NavigationCallExp* is a reference to an *AssociationEnd* or an *AssociationClass* defined in a UML model. It is used to determine objects linked to a target object by an association. If there is a qualifier attached to the source end of the association then additional qualifiers expressions may be used to specify the values of the qualifying attributes.

Associations

qualifiers	The values for the qualifier attributes if applicable.
navigationSource	The source denotes the AssociationEnd at the end of the object itself. This is used to resolve ambiguities when the same Classifier participates in more than one AssociationEnd in the same association. In other cases it can be derived.

OperationCallExp

A OperationCallExp refers to an operation defined in a Classifier. The expression may contain a list of argument expressions if the operation is defined to have parameters. In this case, the number and types of the arguments must match the parameters.

Associations

arguments	The arguments denote the arguments to the operation call. This is only useful when the operation call is related to an *Operation* that takes parameters.
referredOperation	The *Operation* to which this *OperationCallExp* is a reference. This is an *Operation* of a *Classifier* that is defined in the UML model.

3.3.3 If Expressions

This section describes the if expression in detail. Figure 3-4 shows the structure of the if expression.

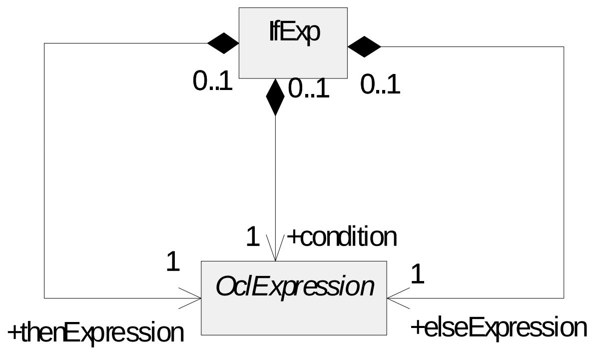


Figure 3-4 *Abstract syntax metamodel for if expression*

IfExp

An *IfExp* results in one of two alternative expressions depending on the evaluated value of a *condition*. Note that both the *thenExpression* and the *elseExpression* are mandatory. The reason behind this is that an if expression should always result in a value, which cannot be guaranteed if the else part is left out.

Associations

condition	The *OclExpression* that represents the boolean condition. If this condition evaluates to true, the result of the if expression is identical to the result of the *thenExpression*. If this condition evaluates to false, the result of the if expression is identical to the result of the *elseExpression*
thenExpression	The *OclExpression* that represents the then part of the if expression.
elseExpression	The *OclExpression* that represents the else part of the if expression.

3.3.4 Message Expressions

In the specification of communication between instances we unify the notions of asynchronous and synchronous communication. The structure of the message expressions is shown in figure 3-5.

OclMessageExp

An *OclMessageExp* is an expression that results in an collection of *OclMessage* value. An *OclMessage* is the unification of a signal sent, and an operation call. The target of the operation call or signal sent is specified by the *target* OclExpression. Arguments can be *OclExpressions*, but may also be unspecified value expressions for arguments whose value is not specified. It covers both synchronous and asynchronous actions. See [Kleppe2000] for a complete description and motivation of this type of expression, also called "action clause".

Associations

target	The *OclExpression* that represents the target instance to which the signal is sent.
arguments	The *SignalArg*s that represents the parameters to the *Operation* or *Signal*. The number and type of arguments should conform to those defined in the *Operation* or *Signal*. The order of the arguments is the same as the order of the parameters of the *Operation* or the attributes of a *Signal*.
calledOperation	If this is a message to request an operation call, this is the requested *CallAction*.
sentSignal	If this is a UML signal sent, this is the *SendAction*.

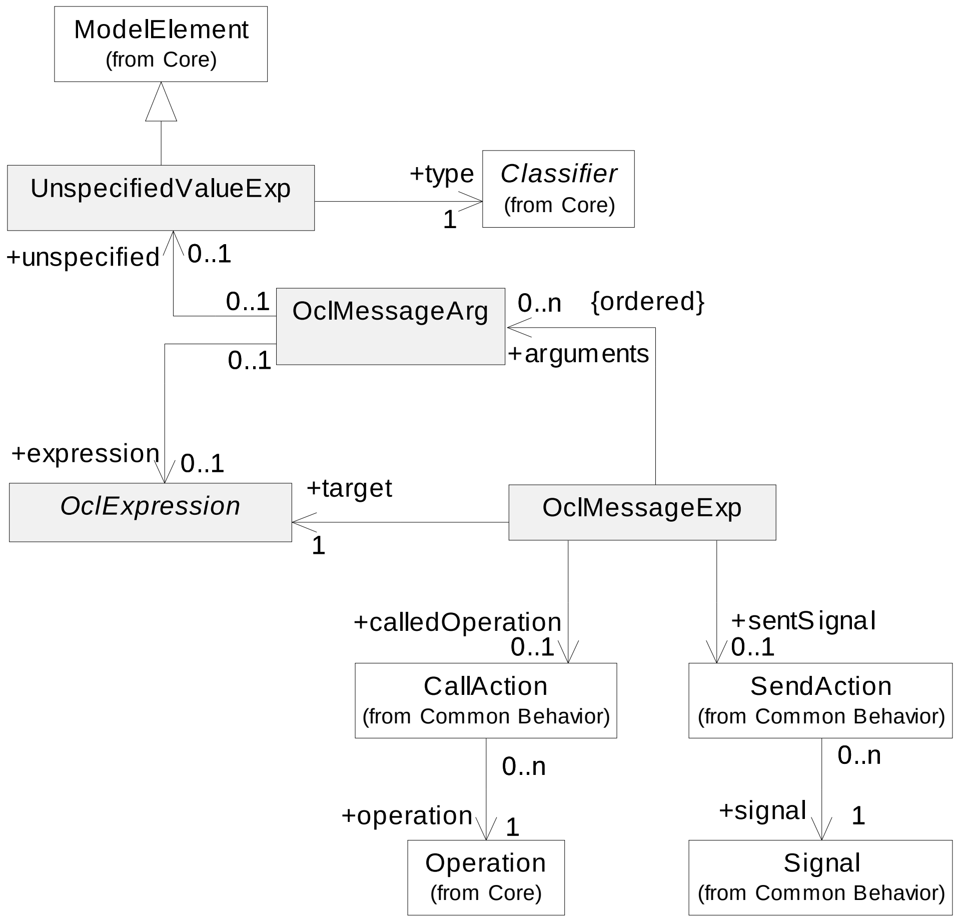


Figure 3-5 *The abstract syntax of Ocl messages*

OclMessageArg

An *OclMessageArg* is an argument of an *OclMessageExp*. It is either an *OclExpression*, or an *UnspecifiedValueExp*. An *OclExpression* is used to specify the exact value of the parameter. An *UnspecifiedValueExp* is used when one does not want, or is not able to specify the exact value of the parameter at the time of sending of the message. An *OclMessageArg* has either a specified or an unspecified value.

Associations

expression — The *OclExpression* that represents an actual parameters to the *Operation* or *Signal*.

unspecified — The *UnspecifiedValueExp* that represents a random value that conforms to the type of this expression.

UnspecifiedValueExp

An *UnpecifiedValueExp* is an expression whose value is unspecified in an OCL expression. It is used within OCL messages to leave parameters of messages unspecified.

3.3.5 Literal Expressions

This section defines the different types of literal expressions of OCL. It also refers to enumeration types and enumeration literals. Figure 3-6 shows all types of literal expressions.

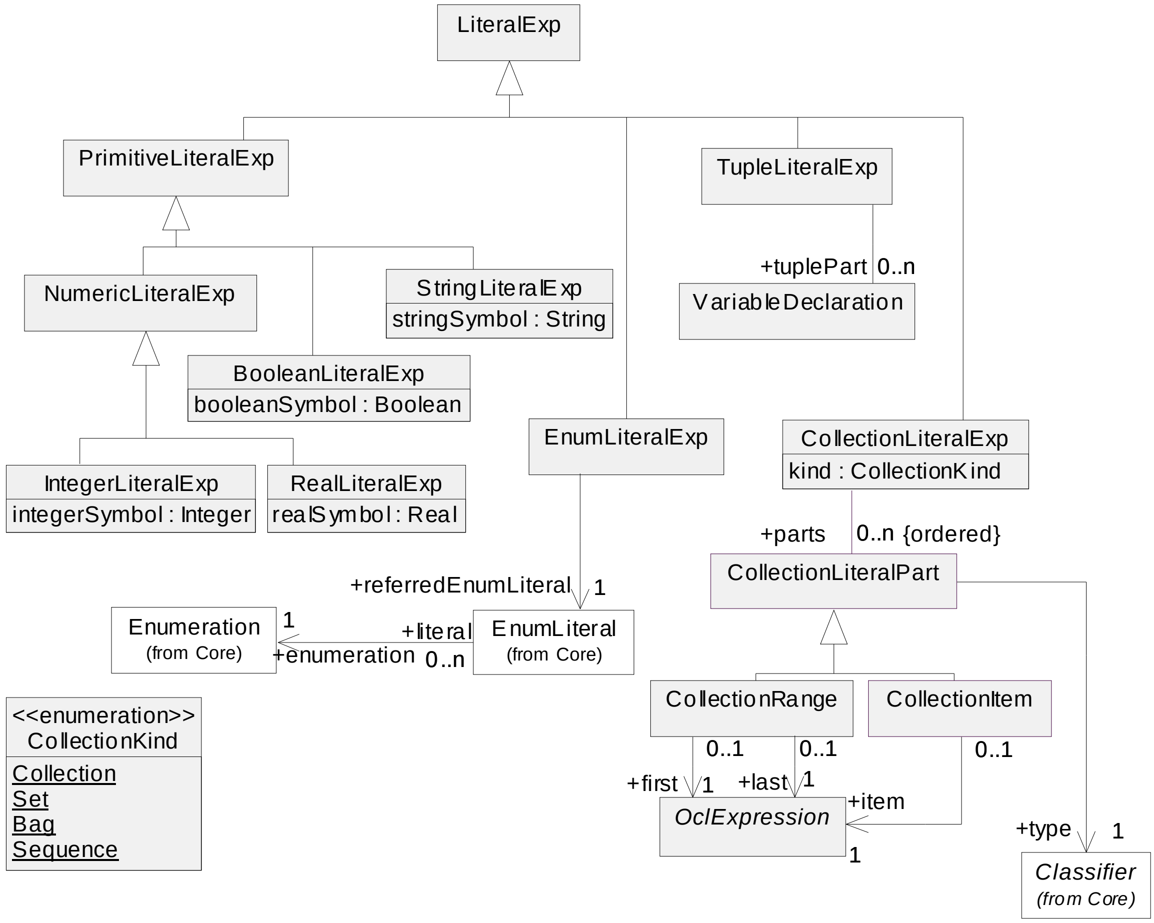


Figure 3-6 *Abstract syntax metamodel for Literal expression*

BooleanLiteralExp

A *BooleanLiteralExp* represents the value true or false of the predefined type Boolean.

Attributes

booleanSymbol The *Boolean* that represents the value of the literal.

CollectionItem

A *CollectionItem* represents an individual element of a collection.

CollectionKind

A *CollectionKind* is an enumeration of kinds of collections.

CollectionLiteralExp

A *CollectionLiteralExp* represents a reference to collection literal.

Attributes

kind	The kind of collection literal that is specified by this *CollectionLiteralExp*.

CollectionLiteralPart

A *CollectionLiteralPart* is a member of the collection literal.

Associations

type	The type of the collection literal.

CollectionRange

A *CollectionRange* represents a range of integers.

EnumLiteralExp

An *EnumLiteralExp* represents a reference to an enumeration literal.

Associations

referredEnumLiteral	The *EnumLiteral* to which the enum expression refers.

IntegerLiteralExp

A *IntegerLiteralExp* denotes a value of the predefined type Integer.

Attributes

integerSymbol	The *Integer* that represents the value of the literal.

NumericLiteralExp

A *NumericLiteralExp* denotes a value of either the type Integer or the type Real.

PrimitiveLiteralExp

A *PrimitiveLiteralExp* literal denotes a value of a primitive type.

Attributes

symbol	The *String* that represents the value of the literal.

RealLiteralExp

A *RealLiteralExp* denotes a value of the predefined type Real.

Attributes

realSymbol	The *Real* that represents the value of the literal.

StringLiteralExp

A *StringLiteralExp* denotes a value of the predefined type String.

Attributes

stringSymbol	The *String* that represents the value of the literal.

TupleLiteralExp

A *TupleLiteralExp* denotes a tuple value. It contains a name and a value for each part of the tuple type.

3.3.6 Let expressions

This section defines the abstract syntax metamodel for Let expressions. The only addition to the abstract syntax is the metaclass *LetExp* as shown in figure 3-7. The other metaclasses are re-used from the previous diagrams.

Note that Let expressions that take arguments are no longer allowed in OCL 2.0. This feature is redundant. Instead, a modeler can define an additional operation in the UML Classifier, potentially with a special stereotype to denote that this operation is only ment to be used as a helper operation in OCL expressions. The postcondition of such an additional operation can then define its result value. Removal of Let functions will therefore not affect the expressibility of the modeler. Another way to define such helper operations is through the <<definition>> constraint, which reuses some of the concrete syntax defined for Let expressions (see section 7.3.1), but is nothing more than an OCL-based syntax for defining helper attributes and operations.

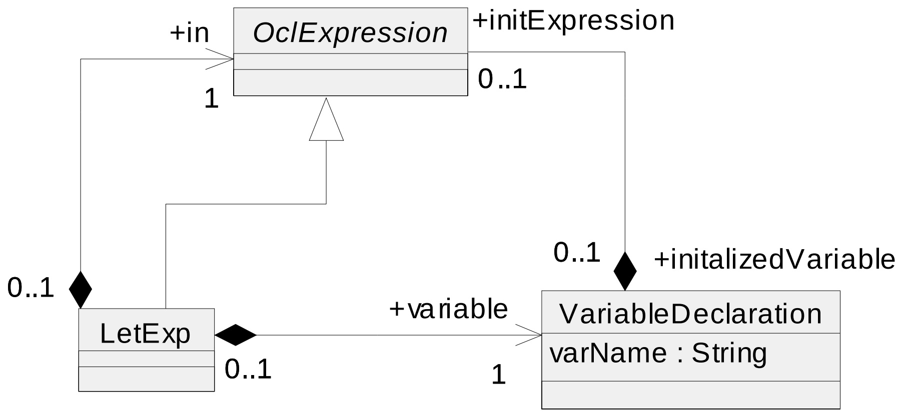


Figure 3-7 *Abstract syntax metamodel for let expression*

LetExp

A *LetExp* is a special expression that defined a new variable with an initial value. A variable defined by a *LetExp* cannot change its value. The value is always the evaluated value of the initial expression. The variable is visible in the *in* expression.

Associations

variable	The *VariableDeclaration* that defined the variable.
in	The *OclExpression* in whose environment the defined variable is visible.

3.3.7 Well-formedness Rules of the Expressions package

The metaclasses defined in the abstract syntax have the following well-formednes rules:

AttributeCallExp

[1] The type of the Attribute call expression is the type of the referred attribute.

```
context AttrubuteCallExp
inv: type = referredAttribute.type
```

BooleanLiteralExp

[1] The type of a boolean Literal expression is the type Boolean.

```
context BooleanLiteralExp
inv: self.type.name = 'Boolean'
```

CollectionLiteralExp

[1] 'Collection' is an abstract class on the M1 level and has no M0 instances.

```
context CollectionLiteralExp
inv: kind <> CollectionKind::Collection
```

[2] The type of a collection literal expression is determined by the collection kind selection and the common supertype of all elements. Note that the definition below implicitly states that empty collections have *OclVoid* as their *elementType*.

```
context CollectionLiteralExp
inv: kind = CollectionKind::Set       implies type.oclIsKindOf (SetType     )
inv: kind = CollectionKind::Sequence  implies type.oclIsKindOf (SequenceType)
inv: kind = CollectionKind::Bag       implies type.oclIsKindOf (BagType     )
inv: type.oclAsType (CollectionType).elementType = parts->iterate (p; c : Classifier =
OclVoid | c.commonSuperType (p.type))
```

CollectionLiteralPart

No additional well-formedness rules.

CollectionItem

[1] The type of a CollectionItem is the type of the item expression.

```
context CollectionItem
inv: type = item.type
```

CollectionRange

[1] The type of a CollectionRange is the common supertype of the expressions taking part in the range.

```
context CollectionRange
inv: type = first.type.commonSuperType (last.type)
```

EnumLiteralExp

[1] The type of an enum Literal expression is the type of the referred literal.

```
context EnumLiteralExp
inv: self.type = referredEnumLiteral.enumeration
```

IfExp

[1] The type of the condition of an if expression must be Boolean.

```
context IfExp
inv: self.condition.type.oclIsKindOf(Primitive) and self.condition.type.name = 'Boolean'
```

[2] The type of the if expression is the most common supertype of the else and then expressions.

```
context IfExp
inv: self.type = thenExpression.type.commonSuperType(elseExpression.type)
```

IntegerLiteralExp

[1] The type of an integer Literal expression is the type Integer.

```
context IntegerLiteralExp
inv: self.type.name = 'Integer'
```

IteratorExp

[1] If the iterator is 'forAll', 'isUnique', or 'exists' the type of the iterator must be Boolean.

```
context IteratorExp
inv: name = 'exists' or name = 'forAll' or name = 'isUnique'
     implies type.oclIsKindOf(Primitive) and type.name = 'Boolean'
```

[2] The result type of the collect operation on a sequence type is a sequence, the result type of 'collect' on any other collection type is a Bag. The type of the body is always the type of the elements in the return collection.

```
context IteratorExp
inv: name = 'collect' implies
     if source.type.oclIsKindOf(SequenceType) then
       type = expression.type.collectionType->select(oclIsTypeOf(SequenceType))->first()
     else
       type = expression.type.collectionType->select(oclIsTypeOf(BagType))->first()
     endif
```

[3] The 'select'and 'reject' iterators have the same type as its source.

```
context IteratorExp
inv: name = 'select' or name = 'reject' implies type = source.type
```

[4] The type of the body of the select, reject,exists and forAll must be boolean.

```
context IteratorExp
inv: name = 'exists' or name = 'forAll' or name = 'select' or name = 'reject'
     implies body.type.name = 'Boolean'
```

IterateExp

[1] The type of the iterate is the type of the result variable.

```
context IterateExp
inv: type = result.type
```

[2] The type of the body expression must conform to the declared type of the result variable.

```
context IterateExp
body.type.conformsTo(result.type)
```

[3] A result variable must have an init expression.

```
context IterateExp
inv: self.result.initExpression->size() = 1
```

LetExp

[1] The type of a Let expression is the type of the in expression.

```
context LetExp
inv: type = in.type
```

LiteralExp

No additional well-formedness rules.

LoopExp

[1] The type of the source expression must be a collection.

```
context LoopExp
inv: source.type.oclIsKindOf (CollectionType)
```

[2] The loop variable of an iterator expression has no init expression.

```
context LoopExp
inv: self.iterators->forAll(initExpression->isEmpty())
```

[3] The type of each iterator variable must be the type of the elements of the source collection.

```
context IteratorExp
inv: self.iterators->forAll(type = source.type.oclAsType (CollectionType).elementType)
```

ModelPropertyCallExp

No additional well-formedness rules.

NumericLiteralExp

No additional well-formedness rules.

OclExpression

No additional well-formedness rules.

OclMessageArg

[1] There is either an expression or an unspecified value.

```
context OclMessageArg
inv: expression->size() + unspecified->size() = 1
```

OclMessageExp

[1] If the message is a call action, the arguments must conform to the parameters of the operation.

```
context OclMessageExp
inv: calledOperation->notEmpty() implies
        arguments->forall (a | a.getType().conformsTo
              (self.calledOperation.operation.parameter->
                     select( kind = ParameterDirectionKind::in )
                                  ->at (arguments->indexOf (a)).type))
```

[2] If the message is a send action, the arguments must conform to the attributes of the signal.

```
context OclMessageExp
inv: sentSignal->notEmpty() implies
        arguments->forall (a | a.getType().conformsTo
              (self.sentSignal.signal.feature.oclAsType(StructuralFeature) )
                                  ->at (arguments->indexOf (a)).type))
```

[3] If the message is a call action, the operation must be an operation of the type of the target expression.

```
context OclMessageExp
inv: calledOperation->notEmpty() implies
       target.type.allOperations()->includes(calledOperation.operation)
```

[4] An OCL message has either a called operation or a sent signal.

```
context OclMessageExp
inv: calledOperation->size() + sentMessage->size() = 1
```

[5] The target of an OCL message cannot be a collection.

```
context OclMessageExp
inv: not target.type.oclIsKindOf (CollectionType)
```

OperationCallExp

[1] All the arguments must conform to the parameters of the referred operation

```
context OperationCallExp
inv: arguments->forall (a | a.type.conformsTo
                                       (self.refParams->at (arguments->indexOf (a)).type))
```

[2] There must be exactly as many arguments as the referred operation has parameters.

```
context OperationCallExp
inv: arguments->size() = refParams->size()
```

[3] An additional attribute refParams lists all parameters of the referred operation except the return and out parameter(s).

```
context OperationCallExp
def: attr refParams: Sequence(Parameter) = referredOperation.parameters->select (p |
                              p.kind <> ParameterDirectionKind::return or
                              p.kind <> ParameterDirectionKind::out)
```

PropertyCallExp

No additional well-formedness rules.

RealLiteralExp

[1] The type of a real Literal expression is the type Real.

```
context RealLiteralExp
inv: self.type.name = 'Real'
```

StringLiteralExp

[1] The type of a string Literal expression is the type String.

```
context StringLiteralExp
inv: self.type.name = 'String'
```

TupleLiteralExp

[1] The type of a TupleLiteralExp is a TupleType with the specified parts.

```
context TupleLiteralExp
inv: type.oclIsKindOf (TupleType)
     and
     tuplePart->forAll (tlep |
```

```
            type.oclAsType (TupleType).allAttributes()->exists (tp | tlep.attribute = tp))
      and
        tuplePart->size() = type.oclAsType (TupleType).allAttributes()->size()
```

[2] All tuple literal expression parts of one tuple literal expression have unique names.

```
context TupleLiteralExp
inv: tuplePart->isUnique (attribute.name)
```

TupleLiteralExpPart

[1] The type of the attribute is the type of the value expression.

```
context TupleLiteralExpPart
inv: attribute.type = value.type
```

UnspecifiedValueExp

No additional well-formedness rules.

VariableDeclaration

[1] For initialized variable declarations, the type of the initExpression must conform to the type of the declared variable.

```
context VariableDeclaration
inv: initExpression->notEmpty() implies initExpression.type.conformsTo (type)
```

VariableExp

[1] The type of a VariableExp is the type of the variable to which it refers.

```
context VariableExp
inv: type = referredVariable.type
```

3.3.8 Additional Operations on UML metaclasses

In the chapters 3 (“Abstract Syntax”), 4 (“Concrete Syntax”), 7 (“The Use of Ocl Expressions in UML Models”) and appendix 5 (“Semantics Described using UML”) many additional operations on UML metaclasses are used. They are defined in this section. The next section defines additional operations for the OCL metaclasses

Classifier

The operation commonSuperType results in the most specific common supertype of two classifiers.

```
context Classifier
def: oper commonSuperType (c : Classifier) : Classifier =
    Classifier.allInstances()->select (cst |
        c.conformsTo (cst) and
        self.conformsTo (cst) and
        not Classifier.allInstances()->exists (clst |
          c.conformsTo (clst) and
          self.conformsTo (clst) and
          clst.conformsTo (cst) and
          clst <> cst
        )
      )->any (true)
```

The following operations have been added to Classifier to lookup attributes, associationEnds and operations.

```
context Classifier
def: oper lookupAttribute(attName : String) : Attribute =
                          self.allAttributes->any(me | me.name = attName)
def: oper lookupAssociationEnd(name : String) : AssociationEnd =
                          self.allAssociationEnds->any (ae | ae.name = name)
def: oper lookupAssociationClass(name : String) : AssociationClass =
                          self.allAssociationClasses->any (ae | ae.name = name)
def: oper lookupOperation (name: String, paramTypes: Sequence(Classifier)): Operation =
                          self.allOperations->any (op | op.name = name and
                                                 op.hasMatchingSignature(paramTypes))
def: oper lookupSignal (sigName: String, paramTypes: Sequence(Classifier)): Operation =
                         self.allReceptions.signal->any (sig | sig.name = sigName and
                                                 sig.hasMatchingSignature(paramTypes))
```

Operations allAttributes, allOperations, etc. are defined in the UML semantics. The operation allReceptions is missing and defined here. The operation allReceptions results in a Set containing all Signals that the Classifier has as Receptions itself and all its inherited Attributes.

```
context Classifier

def: attr allReceptions : set(Reception) =
                       self.allFeatures->select(f | f.oclIsKindOf(Reception))
```

Operation

An additional operation is added to Operation, which checks whether its signature matches with a sequence of Clasifiers. Note that in making the match only parameters with direction kind ‘in’ are considered.

```
context Operation
def: oper hasMatchingSignature(paramTypes: Sequence(Classifier)) : Boolean =
     -- check that operation op has a signature that matches the given parameter lists
     = let sigParamTypes: Sequence(Classifier) = self.allAttributes.type in
           (
             ( sigParamTypes->size() = paramTypes->size() ) and
             ( Set{1..paramTypes->size()}->forAll ( i |
                 paramTypes->at (i).conformsTo (sigParamTypes->at (i))
               )
             )
           )
```

Parameter

The operation asAttribute results in an attribute that has the same name, type, etc. as the parameter.

```
context Parameter::asAttribute(): Attribute
pre: -- none
post: result.name                 = self.name
post: result.type                 = self.type
post: result.multiplicity         = 1
post: result.targetscope          = ScopeKind::instance
post: result.ownerscope           = ScopeKind::instance
post: result.ordering             = OrderingKind::unordered
post: result.visibility           = VisibilityKind::private
post: result.stereotype.name      = 'OclHelper'
```

An additional class operation is added to Parameter to return a Parameter.

```
context Parameter::make(n : String, c : Classifier, k : ParameterDirectionKind) :Parameter
post: result.name = n
post: result.kind = k
post: result.type = c
post: result.stereotype.name = 'OclHelper'
```

Signal

An additional operation is added to Signal, which checks whether its signature matches with a sequence of Clasifiers. Note that in making the match the parameters of the siugnal are its attributes.

```
context Signal
def: oper hasMatchingSignature(paramTypes: Sequence(Classifier)) : Boolean =
     -- check that signal has a signature that matches the given parameter lists
     = let opParamTypes: Sequence(Classifier) = self.parameter->select (p | p.kind <>
                                                     ParameterDirectionKind::return).type in
           (
             ( opParamTypes->size() = paramTypes->size() ) and
             ( Set{1..paramTypes->size()}->forAll ( i |
                 paramTypes->at (i).conformsTo (opParamTypes->at (i))
               )
             )
           )
```

State

[2] The operation getStateMachine() returns the statemachine to which a state belongs.

```
context State::getStateMachine() : StateMachine
post: result =
      if statemachine->notEmpty() then
         stateMachine
      else
         -- must be part of a composite state
         state.container.getStateMachine()
      endif
```

Transition

[1] The operation getStateMachine() returns the statemachine to which a transition belongs.

```
context Transition::getStateMachine() : StateMachine
post: result =
      if statemachine->notEmpty() then
         stateMachine
      else
         -- state is not empty
         state.getStateMachine()
      endif
```

3.3.9 Additional Operations on OCL metaclasses

In chapters 3 (“Abstract Syntax”), 4 (“Concrete Syntax”), 7 (“The Use of Ocl Expressions in UML Models”) and appendix 5 (“Semantics Described using UML”) many additional operations on OCL metaclasses are used. They are defined in this section. The previous section defines additional operations for the UML metaclasses

OclExpression

The following operation returns an operation call expression for the predefined *atPre()* operation with the self expression as its source.

```
OclExpression::withAtPre() : OperationCallExp
post: result.name = 'atPre'
post: result.arguments->isEmpty()
post: result.source = self
```

The following operation returns an operation call expression for the predefined *asSet()* operation with the self expression as its source.

```
OclExpression::withAsSet() : OperationCallExp
post: result.name = 'asSet'
post: result.arguments->isEmpty()
post: result.source = self
```

OclMessageArg

An additional operation is added to oclMessageArg to return the type of the argument.

```
context OclMessageArg
def: oper getType() : Classifier = if unspecified->notEmpty()
                                   then unspecified.type
                                   else expression.type
                                   endif
```

TupleType

An additional class operation is added to Tuple to return a new tuple. The name of a tupletype is defined in the abstract syntax chapter and need not to be specified here.

```
context TupleType::make(atts : sequence(Attribute) ) : TupleType
post: result.features = atts
post: result.stereotype.name = 'OclHelper'
```

VariableDeclaration

An additional operation is added to VariableDeclaration to return a corresponding Parameter.

```
context VariableDeclaration::asParameter() : Parameter
post: result.name = self.varName
post: result.kind = ParameterKind::in
post: result.type = self.type
```

An additional operation is added to VariableDeclaration to return a corresponding Attribute.

```
context VariableDeclaration::asAttribute() : Attribute
post: result.name                         = self.varName
post: result.type                         = self.type
post: result.multiplicity                 = 1
post: result.targetscope                  = ScopeKind::instance
post: result.ownerscope                   = ScopeKind::instance
post: result.ordering                     = OrderingKind::unordered
post: result.visibility                   = VisibilityKind::private
post: result.constraint.bodyExpression = self.initExpression
post: result.stereotype.name              = 'OclHelper'
```

3.3.10 Overview of class hierarchy of OCL Abstract Syntax metamodel

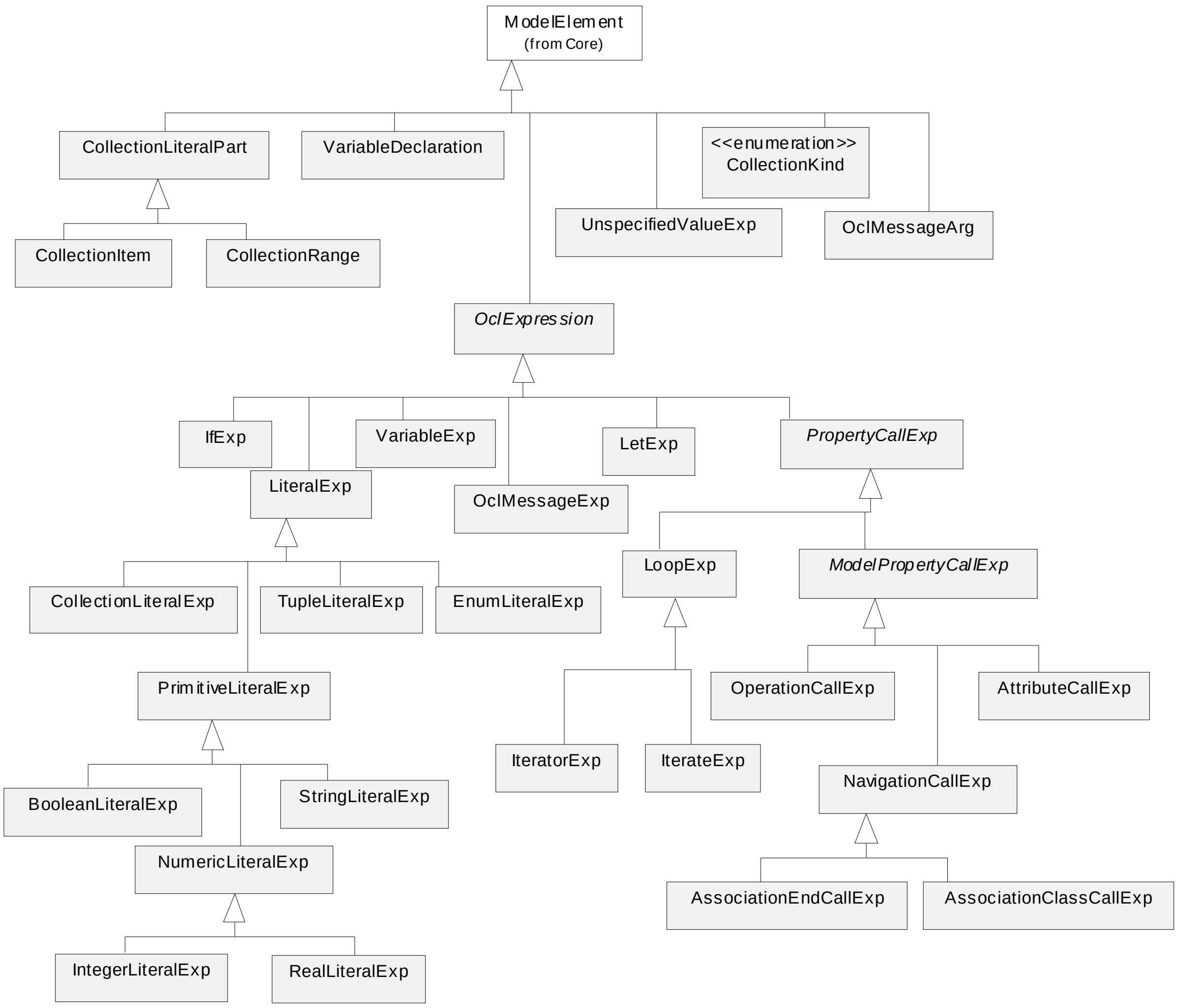


Figure 3-8 *Overview of the abstract syntax metamodel for Expressions*

4

Concrete Syntax

This section describes the concrete syntax of the OCL. This allows modelers to write down OCL expressions in a standardized way. A formal mapping from the concrete syntax to the abstract syntax from chapter 3 (“Abstract Syntax”) is given. Although not required by the UML 2.0 for OCL RfP, section 4.6 describes a mapping from the abstract syntax to the concrete syntax. This allows one to produce a standard human readable version of any OCL expression that is represented as an instance of the abstract syntax.

Section 4.1 (“Structure of the Concrete Syntax”) describes the structure of the grammar and the motivation for the use of an attribute grammar.

4.1 STRUCTURE OF THE CONCRETE SYNTAX

The concrete syntax of OCL is described in the form of an a full attribute grammar. Each production in an attribute grammar may have synthesized attributes attached to it. The value of synthesized attributes of elements on the left hand side of a production rule is always derived from attributes of elements at the right hand side of that production rule. Each production may also have inherited attributes attached to it. The value of inherited attributes of elements on the right hand side of a production rule is always derived from attributes of elements on the left hand side of that production.

In the attribute grammar that specifies the concrete syntax, every production rule is denoted using the EBNF formalism and annotated with synthesised and inherited attributes, and disambiguating rules. There are a number of special annotations:

Synthesized attributes. Each production rule has one synthesized attribute called *ast* (short for abstract syntax tree), that holds the instance of the OCL Abstract Syntax that is returned by the rule. The type of *ast* is different for every rule, but it always is an element of the abstract syntax. The type is stated with each production rule under the heading "Abstract Syntax Mapping". The *ast* attribute constitutes the formal mapping from concrete syntax to abstract syntax.

The motivation for the use of an attribute grammar is the easiness of the construction and the clarity of this mapping. Note that each name in the EBNF format of the production rule is postfixed with ’CS’ to clearly distinguish between the concrete syntax elements and their abstract syntax counterparts.

Inherited attributes. Each production rule has one inherited attribute called *env* (short for environment), that holds a list of names that are visible from the expression. All names are references to elements in the model. In fact, *env* is a name space environment for the expression or expression part denoted according to the production rule. The type of the *env* attribute is Environment, as shown in figure 4-1 on page 4-2. A number of operations are defined for this type. Their definitions and more details on the Environment type can be found in section 4.4 (“Environment definition”). The manner in which both the *ast* and *env* attributes are determined, is given using OCL expressions.

Note that the contents of the *env* attribute are fully determined by the context of the OCL expression. When an OCL expression is used as an invariant to class X, its environment will be different than in the case the expres-

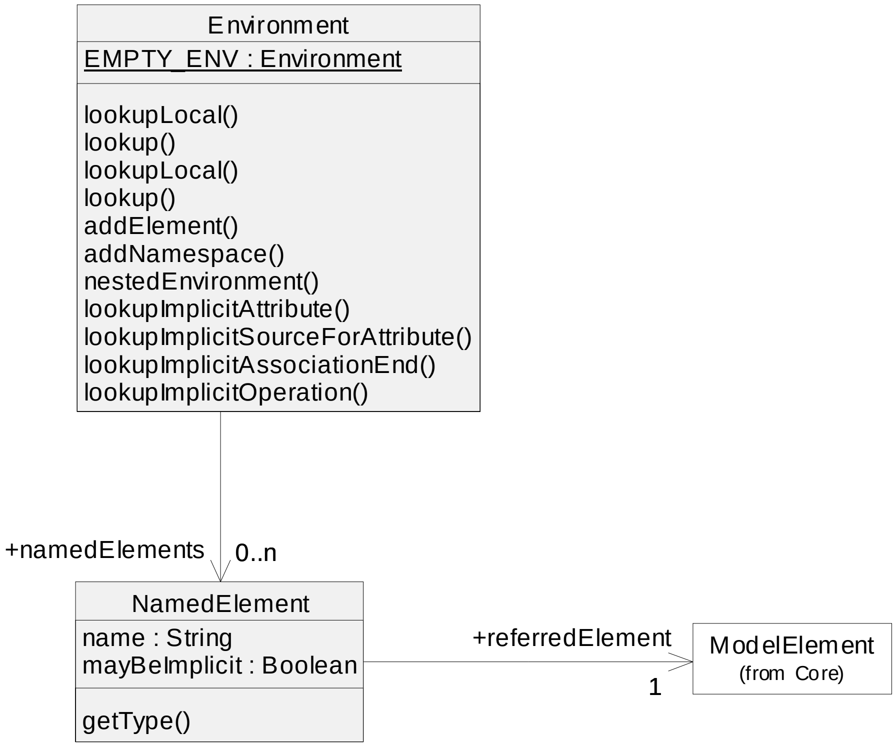


Figure 4-1 *The Environment type*

sion is used as a postcondition to an operation of class Y. In chapter 7 (“The Use of Ocl Expressions in UML Models”) the context of OCL expressions is defined in detail.

Multiple production rules. For some elements there is a choice of multiple production rules. In that case the EBNF format of each production rule is prefixed by a capital letter between square brackets. The same prefix is used for the corresponding determination rules for the *ast* and *env* attributes.

Multiple occurences of production names. In some production rules the same element name is used more than once. To distinguish between these occurences the names will be postfixed by a number in square brackets, as in the following example.

```
CollectionRangeCS ::= OclExpressionCS[1] ’..’ OclExpressionCS[2]
```

Disambiguating rules. Some of the production rules are syntactically ambiguous. For such productions disambiguating rules have been defined. Using these rules, each production and thus the complete grammar becomes nonambiguous. For example in parsing *a.b()*, there are at least four possible parsing solutions:

1. *a* is a VariableExpr (a reference to a let or an iterator variable)
2. *a* is an AttributeCallExp (self is implicit)
3. *a* is a NavigationCallExp (self is implicit)
4. *a* is a type and *b* a class operation (e.g., *allInstances()*)

A decision on which grammar production rule to use, can only be made when the environment of the expression is taken into account. The disambiguating rules describe these choices based on the environment and allow unambiguous parsing of *a.b()*. In this case the rules (in plain English) would be:

- If *a* is a classifier defined in the current scope, *a.b()* is class operation.
- If *a* is a defined variable in the current scope, *a* is a VariableExp.

- If not, check self and all iterator variables in scope. The inner-most scope for which as is either
  - an attribute with the name *a*, resulting in an AttributeCallExp,
  - or an opposite association-end with the name *a*, resulting in a NavigationCallExp,

  defines the meaning of *a.b()*.
- If neither of the above is true, the expression is illegal / incorrect and cannot be parsed.

Disambiguating rules may be based on the UML model to which the OCL expresion is attached (e.g does an attribute exist or not). Because of this, the UML model must be available when an OCL expression is parsed, otherwise it cannot be validated as a correct expression. The grammar is structured in such a way that at most one of the production rules will fullfil all the disambiguating rules, thus ensuring that the grammar as a whole is unambiguous. The disambiguating rules are written in OCL, and use some metaclasses and additional operations from the UML 1.4 semantics.

4.2 A NOTE TO TOOL BUILDERS

4.2.1 Parsing

The grammar in this chapter might not prove to be the most efficient way to directly construct a tool. Of course, a tool-builder is free to use a different parsing mechnism. He can e.g. first parse an OCL expression using a special concrete syntax tree, and do the semantic validation against a UML model in a second pass. Also, error correction or syntax directed editing might need hand-optimized grammars. This document does not prescribe any specific parsing approach. The only restriction is that at the end of all processing a tool should be able to produce the same well-formed instance of the abstract syntax, as would be produced by this grammar.

4.2.2 Visibility

The OCL specification puts no restrictions on visibility. In OCL, all modelelements are considered visible. The reason for this is to allow a modeler to specify constraints, even between 'hidden' elements. At the lowest implementation level this might be useful.

As a separate option OCL tools may enforce all UML visibility rules to support OCL expressions to be specified only over visible modelelements. Especially when a tool needs to generate code for runtime evaluation of OCL expressions, this visibility enforcement is necessary.

4.3 CONCRETE SYNTAX

ExpressionInOclCS

The ExpressionInOcl symbol has been added to setup the initial environment of an expression.

```
ExpressionInOclCS ::= OclExpressionCS
```

Abstract syntax mapping

```
ExpressionInOclCS.ast : OclExpression
```

Synthesized attributes

```
ExpressionInOclCS.ast = OclExpressionCS.ast
```

Inherited attributes

The environment of the OCL expression must be defined, but what exactly needs to be in the environment depends on the context of the OCL expression. The following rule is therefore not complete. It defines the env attribute by adding the self variable to an empty environment, as well as a Namespace containing all elements visible from self. (In section 7.2 (“The ExpressionInOcl Type”) the contextualClassifier will be defined for the various places where an ocl expression may occur.) In the context of a pre- or postcondition, the result variable as well as variable definitions for any named operation parameters can be added in a similar way.

```
OclExpressionCS.env =
      ExpressionInOclCS.contextualClassifier.namespace.getEnvironmentWithParents()
                 .addElement ('self', ExpressionInOclCS.contextualClassifier, true)
```

OclExpressionCS

An OclExpression has several production rules, one for each subclass of OclExpression. Note that UnspecifiedValueExp is handled explicitly in OclMessageArgCS, because that is the only place where it is allowed.

```
[A] OclExpressionCS ::= PropertyCallExpCS
[B] OclExpressionCS ::= VariableExpCS
[C] OclExpressionCS ::= LiteralExpCS
[D] OclExpressionCS ::= LetExpCS
[E] OclExpressionCS ::= OclMessageExpCS
[F] OclExpressionCS ::= IfExpCS
```

Abstract syntax mapping

```
OclExpressionCS.ast : OclExpression
```

Synthesized attributes

```
[A] OclExpressionCS.ast = PropertyCallExpCS.ast
[B] OclExpressionCS.ast = VariableExpCS.ast
[C] OclExpressionCS.ast = LiteralExpCS.ast
[D] OclExpressionCS.ast = LetExpCS.ast
[E] OclExpressionCS.ast = OclMessageExpCS.ast
[F] OclExpressionCS.ast = IfExpCS.ast
```

Inherited attributes

```
[A] PropertyCallExpCS.env   = OclExpressionCS.env
[B] VariableExpCS.env       = OclExpressionCS.env
[C] LiteralExpCS.env        = OclExpressionCS.env
[D] LetExpCS.env            = OclExpressionCS.env
[E] OclMessageExpCS.env     = OclExpressionCS.env
[F] IfExpCS.env             = OclExpressionCS.env
```

Disambiguating rules

The disambiguating rules are defined in the children.

VariableExpCS

A variable expression is just a name that refers to a variable.

```
VariableExpCS ::= simpleNameCS
```

Abstract syntax mapping

```
VariableExpCS.ast : VariableExpression
```

Synthesized attributes

```
VariableExpCS.ast.referredVariable =
              env.lookup(simpleNameCS.ast).referredElement.oclAsType(VariableDeclaration)
```

Inherited attributes

```
-- none
```

Disambiguating rules

[1] simpleName must be a name of a visible VariableDeclaration in the current environment.

```
env.lookup (simpleNameCS.ast).referredElement.oclIsKindOf (VariableDeclaration)
```

simpleNameCS

This production rule represents a single name. No special rules are applicable. The exact syntax of a String is undefined in UML 1.4, and remains undefined in OCL 2.0. The reason for this is internationalization.

```
simpleNameCS ::= <String>
```

Abstract syntax mapping

```
simpleNameGr.ast : String
```

Synthesized attributes

```
simpleNameGr.ast = <String>
```

Inherited attributes

```
-- none
```

Disambiguating rules

```
-- none
```

pathNameCS

This rule represents a path name, which is held in its *ast* as a sequence of Strings.

```
pathNameCS ::= simpleNameCS ('::' pathNameCS )?
```

Abstract syntax mapping

```
pathNameCS.ast : Sequence(String)
```

Synthesized attributes

```
pathNameCS.ast = Sequence{simpleNameCS.ast}->union(pathNameCS.ast)
```

Inherited attributes

```
-- none
```

Disambiguating rules

```
-- none
```

LiteralExpCS

This rule represents literal expressions.

```
[A] LiteralExpCS ::= EnumLiteralExpCS
[B] LiteralExpCS ::= CollectionLiteralExpCS
[C] LiteralExpCS ::= TupleLiteralExpCS
[D] LiteralExpCS ::= PrimitiveLiteralExpCS
```

Abstract syntax mapping

```
LiteralExpCS.ast : LiteralExp
```

Synthesized attributes

```
[A] LiteralExpCS.ast = EnumLiteralExpCS.ast
[B] LiteralExpCS.ast = CollectionLiteralExpCS.ast
[C] LiteralExpCS.ast = TupleLiteralExpCS.ast
[D] LiteralExpCS.ast = PrimitiveLiteralExpCS.ast
```

Inherited attributes

```
[A] EnumLiteralExpCS.env       = LiteralExpCS.env
[B] CollectionLiteralExpCS.env = LiteralExpCS.env
[C] TupleLiteralExpCS.env      = LiteralExpCS.env
[D] PrimitiveLiteralExpCS.env  = LiteralExpCS.env
```

Disambiguating rules

```
-- none
```

EnumLiteralExpCS

The rule represents Enumeration Literal expressions.

```
EnumLiteralExpCS ::= pathNameCS '::' simpleNameCS
```

Abstract syntax mapping

```
EnumLiteralExpCS.ast : EnumLiteralExp
```

Synthesized attributes

```
EnumLiteralExpCS.ast.type =
          env.lookupPathName (pathNameCS.ast).referredElement.oclAsType (Classifier)
EnumLiteralExpCS.ast.referredEnumLiteral =
          EnumLiteralExpCS.ast.type.oclAsType (Enumeration).literal->
                                     select (l  | l.name = simpleNameCS.ast )->any(true)
```

Inherited attributes

```
-- none
```

Disambiguating rules

[1] The specified name must indeed reference an enumeration:

```
not EnumLiteralExpCS.ast.type.oclIsUndefined() and
    EnumLiteralExpCS.ast.type.oclIsKindOf (Enumeration)
```

CollectionLiteralExpCS

This rule represents a collection literal expression.

```
CollectionLiteralExpCS ::= CollectionTypeIdentifierCS
                           '{' CollectionLiteralPartsCS? '}'
```

Abstract syntax mapping

```
CollectionLiteralExpCS.ast : CollectionLiteralExp
```

Synthesized attributes

```
CollectionLiteralExpCS.ast.parts = CollectionLiteralPartsCS.ast
CollectionLiteralExpCS.ast.kind  = CollectionTypeIdentifierCS.ast
```

Inherited attributes

```
CollectionTypeIdentifierCS.env = CollectionLiteralExpCS.env
CollectionLiteralPartsCS.env   = CollectionLiteralExpCS.env
```

Disambiguating rules

[1] In a literal the collectiuon type may not be Collection

```
CollectionTypeIdentifierCS.ast <> 'Collection'
```

CollectionTypeIdentifierCS

This rule represent the type indentifier in a collection literal expression. The Collection type is an abstract type on M1 level, so it has no corresponding literals.

```
[A] CollectionTypeIdentifierCS ::= 'Set'
[B] CollectionTypeIdentifierCS ::= 'Bag'
[C] CollectionTypeIdentifierCS ::= 'Sequence'
[D] CollectionTypeIdentifierCS ::= 'Collection'
```

Abstract syntax mapping

```
CollectionTypeIdentifierCS.ast : CollectionKind
```

Synthesized attributes

```
[A] CollectionTypeIdentifierCS.ast = CollectionKind::Set
[B] CollectionTypeIdentifierCS.ast = CollectionKind::Bag
[C] CollectionTypeIdentifierCS.ast = CollectionKind::Sequence
[D] CollectionTypeIdentifierCS.ast = CollectionKind::Collection
```

Inherited attributes

```
-- none
```

Disambiguating rules

```
-- none
```

CollectionLiteralPartsCS

This production rule describes a sequence of items that are the contents of a collection literal.

```
CollectionLiteralPartsCS[1] = CollectionLiteralPartCS
                              ( ',' CollectionLiteralPartsCS[2] )?
```

Abstract syntax mapping

```
CollectionLiteralPartsCS[1].ast : Sequence(CollectionLiteralPart)
```

Synthesized attributes

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

Inherited attributes

```
CollectionLiteralPartCS.env     = CollectionLiteralPartsCS[1].env
CollectionLiteralPartSCS[2].env = CollectionLiteralPartsCS[1].env
```

Disambiguating rules

```
-- none
```

CollectionLiteralPartCS

```
[A] CollectionLiteralPartCS ::= CollectionRangeCS
[B] CollectionLiteralPartCS ::= OclExpressionCS
```

Abstract syntax mapping

```
CollectionLiteralPartCS.ast : CollectionLiteralPart
```

Synthesized attributes

```
[A] CollectionLiteralPartCS.ast = CollectionRange.ast
[B] CollectionLiteralPartCS.ast.oclIsKindOf(CollectionItem) and
    CollectionLiteralPartCS.ast.oclAsType(CollectionItem).OclExpression =
                                                       OclExpressionCS.ast
```

Inherited attributes

```
[A] CollectionRangeCS.env = CollectionLiteralPartCS.env
[B] OclExpressionCS.env   = CollectionLiteralPartCS.env
```

Disambiguating rules

```
-- none
```

CollectionRangeCS

```
CollectionRangeCS ::= OclExpressionCS[1] '..' OclExpressionCS[2]
```

Abstract syntax mapping

```
CollectionRangeCS.ast : CollectionRange
```

Synthesized attributes

```
CollectionRangeCS.ast.first = OclExpressionCS[1].ast
CollectionRangeCS.ast.last  = OclExpressionCS[2].ast
```

Inherited attributes

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

Disambiguating rules

```
-- none
```

PrimitiveLiteralExpCS

This includes Real, Boolean, Integer and String literals. Exprecially String literals must take internationalisation into account and might need to remain undefined in this specification.

```
[A] PrimitiveLiteralExpCS ::= IntegerLiteralExpCS
[B] PrimitiveLiteralExpCS ::= RealLiteralExpCS
[C] PrimitiveLiteralExpCS ::= StringLiteralExpCS
[D] PrimitiveLiteralExpCS ::= BooleanLiteralExpCS
```

Abstract syntax mapping

```
PrimitiveLiteralExpCS.ast : PrimitiveLiteralExp
```

Synthesized attributes

```
[A] PrimitiveLiteralExpCS.ast = IntegerLiteralExpCS.ast
[B] PrimitiveLiteralExpCS.ast = RealLiteralExpCS.ast
[C] PrimitiveLiteralExpCS.ast = StringLiteralExpCS.ast
[D] PrimitiveLiteralExpCS.ast = BooleanLiteralExpCS.ast
```

Inherited attributes

```
-- none
```

Disambiguating rules

```
-- none
```

TupleLiteralExpCS

This rule represents tuple literal expressions.

```
TupleLiteralExpCS ::= ‘Tuple’ ‘{‘ variableDeclarationListCS ‘}’
```

Abstract syntax mapping

```
TupleLiteralExpCS.ast : TupleLiteralExp
```

Synthesized attributes

```
TupleLiteralExpCS.tuplePart = variableDeclarationListCS.ast
```

Inherited attributes

```
variableDeclarationListCS[1].env = TupleLiteralExpCS.env
```

Disambiguating rules

[1] The initExpression and type of all VariableDeclarations must exist.

```
TupleLiteralExpCS.tuplePart->forAll( varDecl |
    varDecl.initExpression->notEmpty() and not varDecl.type.oclIsUndefined() )
```

IntegerLiteralExpCS

This rule represents integer literal expressions.

```
IntegerLiteralExpCS ::= <String>
```

Abstract syntax mapping

```
IntegerLiteralExpCS.ast : IntegerLiteralExp
```

Synthesized attributes

```
IntegerLiteralExpCS.ast.integerSymbol = <String>.toInteger()
```

Inherited attributes

```
-- none
```

Disambiguating rules

```
-- none
```

RealLiteralExpCS

This rule represents real literal expressions.

```
RealLiteralExpCS ::= <String>
```

Abstract syntax mapping

```
RealLiteralExpCS.ast : RealLiteralExp
```

Synthesized attributes

```
RealLiteralExpCS.ast.realSymbol = <String>.toReal()
```

Inherited attributes

```
-- none
```

Disambiguating rules

```
-- none
```

StringLiteralExpCS

This rule represents string literal expressions.

```
StringLiteralExpCS ::= ''' <String> '''
```

Abstract syntax mapping

```
StringLiteralExpCS.ast : StringLiteralExp
```

Synthesized attributes

```
StringLiteralExpCS.ast.symbol = <String>
```

Inherited attributes

```
-- none
```

Disambiguating rules

```
-- none
```

BooleanLiteralExpCS

This rule represents boolean literal expressions.

```
[A] BooleanLiteralExpCS ::= 'true'
[B] BooleanLiteralExpCS ::= 'false'
```

Abstract syntax mapping

```
BooleanLiteralExpCS.ast : BooleanLiteralExp
```

Synthesized attributes

```
[A] BooleanLiteralExpCS.ast.booleanSymbol = true
[B] BooleanLiteralExpCS.ast.booleanSymbol = false
```

Inherited attributes

```
-- none
```

Disambiguating rules

```
-- none
```

PropertyCallExpCS

This rule represents property call expressions.

```
[A] PropertyCallExpCS ::= ModelPropertyCallExpCS
[B] PropertyCallExpCS ::= LoopExpCS
```

Abstract syntax mapping

```
PropertyCallExpCS.ast : PropertyCallExp
```

Synthesized attributes

```
[A] PropertyCallExpCS.ast = ModelPropertyCallCS.ast
[B] PropertyCallExpCS.ast = LoopExpCS.ast
```

Inherited attributes

```
[A] ModelPropertyCallCS.env  = PropertyCallExpCS.env
[B] LoopExpCS.env            = PropertyCallExpCS.env
```

Disambiguating rules

The disambiguating rules are defined in the children.

LoopExpCS

This rule represents loop expressions.

```
[A] LoopExpCS ::= IteratorExpCS
[B] LoopExpCS ::= IterateExpCS
```

Abstract syntax mapping

```
LoopExpCS.ast : LoopExp
```

Synthesized attributes

```
[A] LoopExpCS.ast = IteratorExpCS.ast
[B] LoopExpCS.ast = IterateExpCS.ast
```

Inherited attributes

```
[A] IteratorExpCS.env  = LoopExpCS.env
[B] IterateExpCS.env   = LoopExpCS.env
```

Disambiguating rules

```
-- none
```

IteratorExpCS

The first alternative is a straightforward Iterator expression, with optional iterator variable. The second and third alternatives are so-called implicit collect iterators. B is for operations and C for attributes, D for navigations and E for associationclasses.

```
[A] IteratorExpCS ::= OclExpressionCS[1] '->' simpleNameCS
                     '(' (VariableDeclarationCS[1],
                          (',' VariableDeclarationCS[2])? '|' )?
                        OclExpressionCS[2]
                     ')'
[B] IteratorExpCS ::= OclExpressionCS '.' simpleNameCS '('argumentsCS?')'
[C] IteratorExpCS ::= OclExpressionCS '.' simpleNameCS
[D] IteratorExpCS ::= OclExpressionCS '.' simpleNameCS
                                                  ('[' argumentsCS ']')?
[E] IteratorExpCS ::= OclExpressionCS '.' simpleNameCS
                                                  ('[' argumentsCS ']')?
```

Abstract syntax mapping

```
IteratorExpCS.ast : IteratorExp
```

Synthesized attributes

```
-- the ast needs to be determined bit by bit, first the source association of IteratorExp
[A] IteratorExpCS.ast.source        = OclExpressionCS[1].ast
-- next the iterator association of IteratorExp
```

```
-- when the variable declaration is present, its ast is the iterator of this iteratorExp
-- when the variable declaration is not present, the iterator has a default name and
-- type
-- In any case, the iterator does not have an init expression
[A] IteratorExpCS.ast.iterators->at(1).name = if VariableDeclarationCS[1]->isEmpty()
                                  then ''
                                  else VariableDeclarationCS[1].ast.name
                                  endif
[A] IteratorExpCS.ast.iterator->at(1).type =
                        if VariableDeclarationCS[1]->isEmpty() or
                          (VariableDeclarationCS[1]->notEmpty() and
                           VariableDeclarationCS[1].ast.type.oclIsUndefined() )
                        then
                           OclExpressionCS[1].type.oclAsType (CollectionType).elementType
                        else
                           VariableDeclarationCS[1].ast.type
                        endif
- The optional second iterator
[A] if VariableDeclarationCS[2]->isEmpty() then
       IteratorExpCS.ast.iterators->size() = 1
    else
       IteratorExpCS.ast.iterators->at(2).name = VariableDeclarationCS[2].ast.name
       and
       IteratorExpCS.ast.iterators->at(2).type =
                        if VariableDeclarationCS[2]->isEmpty() or
                          (VariableDeclarationCS[2]->notEmpty() and
                           VariableDeclarationCS[2].ast.type.oclIsUndefined() )
                        then
                           OclExpressionCS[1].type.oclAsType (CollectionType).elementType
                        else
                           VariableDeclarationCS[2].ast.type
                        endif
    endif
[A] IteratorExpCS.ast.iterators->forAll(initExpression->isEmpty())
-- next the name attribute and body association of the IteratorExp
[A] IteratorExpCS.ast.name   = simpleNameCS.ast and
[A] IteratorExpCS.ast.body   = OclExpressionCS[2].ast

-- Alternative B is an implicit collect of an operation over a collection
[B] IteratorExpCS.ast.iterator.type =
                          OclExpressionCS.ast.type.oclAsType (CollectionType).elementType
[B] IteratorExpCS.ast.source  = OclExpressionCS.ast
[B] IteratorExpCS.ast.name    = 'collect'
[B] -- the body of the implicit collect is the operation call referred to by 'name'
    IteratorExpCS.ast.body.oclIsKindOf (OperationCallExp) and
    let body : OperationCallExp = IteratorExpCS.ast.body.oclAsType(OperationCallExp)
    in
    body.arguments = argumentsCS.ast
    and
    body.source.oclIsKindOf(VariableExp)
    and
    body.source.oclAsType (VariableExp).referredVariable = IteratorExpCS.ast.iterator
    and
    body.referredOperation =
              OclExpressionCS.ast.type.oclAsType (CollectionType ).elementType
                          .lookupOperation( simpleNameCS.ast,
                                            if (argumentsCS->notEmpty())
                                            then arguments.ast->collect(type)
                                            else Sequence{} endif)
```

```
-- Alternative C/D is an implicit collect of an association or attribute over a collection
[C, D] IteratorExpCS.ast.iterator.type =
                           OclExpressionCS.ast.type.oclAsType (CollectionType).elementType
[C, D] IteratorExpCS.ast.source  = OclExpressionCS.ast
[C, D] IteratorExpCS.ast.name    = 'collect'
[C] -- the body of the implicit collect is the attribute referred to by 'name'
    let refAtt : Attribute = OclExpressionCS.ast.type.oclAsType (CollectionType).
                                          elementType.lookupAttribute( simpleNameCS.ast),
    in
    IteratorExpCS.ast.body.oclIsKindOf (AttributeCallExp) and
    let body : AttributeCallExp = IteratorExpCS.ast.body.oclAsType(AttributeCallExp)
    in
      body.source.oclIsKindOf(VariableExp)
      and
      body.source.oclAsType (VariableExp).referredVariable = IteratorExpCS.ast.iterator
      and
      body.referredAttribute = refAtt
[D] -- the body of the implicit collect is the navigation call referred to by 'name'
    let  refNav : AssociationEnd = OclExpressionCS.ast.type.oclAsType (CollectionType).
                                      elementType.lookupAssociationEnd(simpleNameCS.ast)
    in
      IteratorExpCS.ast.body.oclIsKindOf (AssociationEndCallExp) and
      let body : AssociationEndCallExp =
                                 IteratorExpCS.ast.body.oclAsType(AssociationEndCallExp)
      in
        body.source.oclIsKindOf(VariableExp)
        and
        body.source.oclAsType (VariableExp).referredVariable = IteratorExpCS.ast.iterator
        and
        body.referredAssociationEnd = refNav
        and
        body.ast.qualifiers = argumentsCS.ast
[E] -- the body of the implicit collect is the navigation to the association class
    -- referred to by 'name'
    let  refClass : AssociationClass =
                OclExpressionCS.ast.type.oclAsType (CollectionType).
                                     elementType.lookupAssociationClass(simpleNameCS.ast)
    in
      IteratorExpCS.ast.body.oclIsKindOf (AssociationClassCallExp) and
      let body : AssociationClassCallExp =
                                 IteratorExpCS.ast.body.oclAsType(AssociationClassCallExp)
      in
        body.source.oclIsKindOf(VariableExp)
        and
        body.source.oclAsType (VariableExp).referredVariable = IteratorExpCS.ast.iterator
        and
        body.referredAssociationClass = refNav
        and
        body.ast.qualifiers = argumentsCS.ast
```

Inherited attributes

```
[A] OclExpressionCS[1].env    = IteratorExpCS.env
[A] VariableDeclarationCS.env = IteratorExpCS.env
-- inside an iterator expression the body is evaluated with a new environment that
-- includes the iterator variable.
[A] OclExpressionCS[2].env    =
      IteratorExpCS.env.nestedEnvironment().addElement(VariableDeclarationCS.ast.varName,
                                                       VariableDeclarationCS.ast,
                                                       true)
[B] OclExpressionCS.env    = IteratorExpCS.env
[B] argumentsCS.env        = IteratorExpCS.env
[C] OclExpressionCS.env    = IteratorExpCS.env
```

```
[D] OclExpressionCS.env    = IteratorExpCS.env
```

Disambiguating rules

[1] [A] When the variable declaration is present, it may not have an init expression.

```
VariableDeclarationCS->notEmpty() implies
                              VariableDeclarationCS.ast.initExpression->isEmpty()
```

[2] [B] The source must be of a collection type.

```
OclExpressionCS.ast.type.oclIsKindOf(CollectionType)
```

[3] [C] The source must be of a collection type.

```
OclExpressionCS.ast.type.oclIsKindOf(CollectionType)
```

[4] [C] The referred attribute must be present.

```
refAtt->notEmpty()
```

[5] [D] The referred navifation must be present.

```
refNav->notEmpty()
```

IterateExpCS

```
IterateExpCS ::= OclExpressionCS[1] '->' 'iterate'
                 '(' (VariableDeclarationCS[1] ';')?
                      VariableDeclarationCS[2] '|'
                      OclExpressionCS[2]
                 ')'
```

Abstract syntax mapping

```
IterateExpCS.ast : IterateExp
```

Synthesized attributes

```
-- the ast needs to be determined bit by bit, first the source association of IterateExp
IterateExpCS.ast.source   = OclExpressionCS[1].ast
-- next the iterator association of IterateExp
-- when the first variable declaration is present, its ast is the iterator of this
-- iterateExp, when the variable declaration is not present, the iterator has a default
-- name and type,
-- in any case, the iterator has an empty init expression.
IterateExpCS.ast.iterator.name = if VariableDeclarationCS[1]->isEmpty() then ''
                                 else VariableDeclarationCS[1].ast.name
                                 endif
IterateExpCS.ast.iterator.type =
                       if VariableDeclarationCS[1]->isEmpty() or
                         (VariableDeclarationCS[1]->notEmpty() and
                          VariableDeclarationCS[1].ast.type.oclIsUndefined() )
                       then
                          OclExpressionCS[1].type.oclAsType (CollectionType).elementType
                       else
                          VariableDeclarationCS[1].ast.type
                       endif
IterateExpCS.ast.iterator.initExpression->isEmpty()
-- next the name attribute and body and result association of the IterateExp
IterateExpCS.ast.result   = VariableDeclarationCS[2].ast
IterateExpCS.ast.name     = 'iterate'
IterateExpCS.ast.body     = OclExpressionCS[2].ast
```

Inherited attributes

```
OclExpressionCS[1].env       = IteratorExpCS.env
VariableDeclarationCS[1].env = IteratorExpCS.env
VariableDeclarationCS[2].env = IteratorExpCS.env
-- Inside an iterate expression the body is evaluated with a new environment that includes
-- the iterator variable and the result variable.
```

```
OclExpressionCS[2].env       =
       IteratorExpCS.env.nestedEnvironment().addElement
                                    (VariableDeclarationCS[1].ast.varName,
                                     VariableDeclarationCS[1].ast,
                                     true).addElement
                                    (VariableDeclarationCS[2].ast.varName,
                                     VariableDeclarationCS[2].ast,
                                     true)
```

Disambiguating rules

[1] A result variable declaration must have a type and an initial value

```
not VariableDeclarationCS[2].ast.type.oclIsUndefined()
VariableDeclarationCS[2].ast.initExpression->notEmpty()
```

[2] When the first variable declaration is present, it may not have an init expression.

```
VariableDeclarationCS[1]->notEmpty() implies
                                  VariableDeclarationCS[1].ast.initExpression->isEmpty()
```

VariableDeclarationCS

In the variable declaration, the type and init expression are optional. When these are required, this is defined in the production rule where the variable declaration is used.

```
VariableDeclarationCS ::= simpleNameCS (':' typeCS)?
                          ( '=' OclExpressionCS )?
```

Abstract syntax mapping

```
VariableDeclarationCS.ast : VariableDeclaration
```

Synthesised attributes

```
VariableDeclarationCS.ast.name           = simpleNameCS.ast
VariableDeclarationCS.ast.initExpression = OclExpressionCS.ast
-- A well-formed VariableDeclaration must have a type according to the abstract syntax.
-- The value OclUndefined is used when no type has been given in the concrete syntax.
-- Production rules that use this need to check on this type.
VariableDeclarationCS.ast.type = if typeCS->notEmpty() then
                                    typeCS.ast
                                 else
                                    OclUndefined
                                 endif
```

Inherited attributes

```
OclExpressionCS.env = VariableDeclarationCS.env
typeCS.env          = VariableDeclarationCS.env
```

Disambiguating rules

```
-- none
```

TypeCS

A typename is either a Classifier, or a collection of some type.

```
[A] typeCS ::= pathNameCS
[B] typeCS ::= collectionTypeCS
[C] typeCS ::= tupleTypeCS
```

Abstract syntax mapping

```
typeCS.ast : Classifier
```

Synthesised attributes

```
[A] typeCS.ast =
       typeCS.env.lookupPathName(pathNameCS.ast).referredElement.oclAsType(Classifier)
[B] typeCS.ast = CollectionTypeCS.ast
```

```
[C] typeCS.ast = tupleTypeCS.ast
```

Inherited attributes

```
[B] collectionTypeCS.env = typeCS.env
[C] tupleTypeCS.env      = typeCS.env
```

Disambiguating rules

[1] [A] pathName must be a name of a Classifier in current environment.

```
typeCS.env.lookupPathName(pathNameCS.ast).referredElement.oclIsKindOf (Classifier)
```

collectionTypeCS

A typename is either a Classifier, or a collection of some type.

```
collectionTypeCS ::= collectionTypeIdentifierCS '(' typeCS ')'
```

Abstract syntax mapping

```
typeCS.ast : CollectionType
```

Synthesised attributes

```
collectionTypeCS.ast.elementType = typeCS.ast
-- We know that the 'ast' is a collectiontype, all we need to state now is which
-- specific collection type it is.
kind = CollectionKind::Set        implies collectionTypeCS.ast.oclIsKindOf (SetType     )
kind = CollectionKind::Sequence   implies collectionTypeCS.ast.oclIsKindOf (SequenceType)
kind = CollectionKind::Bag        implies collectionTypeCS.ast.oclIsKindOf (BagType     )
kind = CollectionKind::Collection implies collectionTypeCS.ast.oclIsKindOf
                                                                          (CollectionType)
```

Inherited attributes

```
typeCS.env = collectionTypeCS.env
```

Disambiguating rules

```
-- none
```

tupleTypeCS

This represents a tuple type declaration.

```
[C] typeCS ::= 'TupleType' '(' variableDeclarationListCS? ')'
```

Abstract syntax mapping

```
typeCS.ast : TupleType
```

Synthesised attributes

```
typeCS.ast =  TupleType::make( variableDeclarationListCS->collect( v | v.asAttribute() ))
```

Inherited attributes

```
variableDeclarationListCS.env = tupleTypeCS.env
```

Disambiguating rules

[1] Of all VariableDeclarations the initExpression must be empty and the type must exist.

```
variableDeclarationListCS.ast->forAll( varDecl |
    varDecl.initExpression->notEmpty() and varDecl.type->notEmpty() )
```

variableDeclarationListCS

This production rule represents the formal parameters of a tuple or attribute definition.

```
variableDeclarationListCS[1] = VariableDeclarationCS
                               (','variableDeclarationListCS[2] )?
```

Abstract syntax mapping

```
variableDeclarationListCS[1].ast : Sequence( VariableDeclaration )
```

Synthesized attributes

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

Inherited attributes

```
VariableDeclarationCS.env         = variableDeclarationListCS[1].env
variableDeclarationListCS[2].env  = variableDeclarationListCS[1].env
```

Disambiguating rules

```
-- none
```

ModelPropertyCallExpCS

A ModelPropertCall expression may have three different productions. Which one is chosen depends on the disambiguating rules defined in each of the alternatives.

```
[A] ModelPropertyCallExpCS ::= OperationCallExpCS
[B] ModelPropertyCallExpCS ::= AttributeCallExpCS
[C] ModelPropertyCallExpCS ::= NavigationCallExpCS
```

Abstract syntax mapping

```
ModelPropertyCallExpCS.ast : ModelPropertyCallExp
```

Synthesised attributes

The value of this production is the value of its child production.

```
[A] ModelPropertyCallExpCS.ast = OperationCallExpCS.ast
[B] ModelPropertyCallExpCS.ast = AttributeCallExpCS.ast
[C] ModelPropertyCallExpCS.ast = NavigationCallExpCS.ast
```

Inherited attributes

```
[A] OperationCallExpCS.env  = ModelPropertyCallExpCS.env
[B] AttributeCallExpCS.env  = ModelPropertyCallExpCS.env
[C] NavigationCallExpCS.env = ModelPropertyCallExpCS.env
```

Disambiguating rules

These are defined in the children.

OperationCallExpCS

An operation call has many different forms. A is used for infix, B for using an object as an implicit collection. C is a straightforward operation call, while D has an implicit source expression. E and F are like C and D, with the @pre addition. G covers the class operation call. Rule H is for unary prefix expressions.

```
[A] OperationCallExpCS ::= OclExpressionCS[1]
                                      simpleNameCS OclExpressionCS[2]
[B] OperationCallExpCS ::= OclExpressionCS '->' simpleNameCS '('
                                                          argumentsCS? ')'
[C] OperationCallExpCS ::= OclExpressionCS '.' simpleNameCS
                                                      '(' argumentsCS? ')'
[D] OperationCallExpCS ::= simpleNameCS  '(' argumentsCS? ')'
[E] OperationCallExpCS ::= OclExpressionCS '.' simpleNameCS
                                    isMarkedPreCS '(' argumentsCS? ')'
[F] OperationCallExpCS ::= simpleNameCS
                                    isMarkedPreCS '(' argumentsCS? ')'
[G] OperationCallExpCS ::= pathNameCS  '(' argumentsCS? ')'
[H] OperationCallExpCS ::= simpleNameCS OclExpressionCS
```

Abstract syntax mapping

```
OperationCallExpCS.ast  : OperationCallExp
```

Synthesised attributes

```
-- this rule is for binary operators as '+', '-', '*' etc. It has only one argument.
[A] OperationCallExpCS.ast.arguments  = Sequence{OclExpression2[2].ast}
    OperationCallExpCS.ast.source     = OclExpressionCS[1].ast
    OperationCallExpCS.ast.referredOperation =
               OclExpressionCS.ast.type.lookupOperation (
                                           simpleNameCS.ast,
                                           Sequence{OclExpression[2].ast.type} )
-- The source is either a collection or a single object used as a collection.
[B] OperationCallExpCS.ast.arguments = argumentsCS.ast
-- if the OclExpressionCS is a collectiontype, then the source is this OclExpressionCS.
-- Otherwise, the source must be build up by defining a singleton set containing
-- the OclExpressionCS. This is done though inserting a call to the standard
-- operation "asSet()"
   OperationCallExpCS.ast.source =
                     if OclExpressionCS.ast.type.oclIsKindOf(CollectionType)
                     then OclExpressionCS.ast
                     else OclExpressionCS.ast.withAsSet()
                     endif
---- The referred operation:
   OperationCallExpCS.ast.referredOperation =
       if OclExpressionCS.ast.type.oclIsKindOf (CollectionType)
       then  -- this is a collection operation called on a collection
         OclExpressionCS.ast.type.lookupOperation (simpleNameCS.ast,
                                                  if (argumentsCS->notEmpty())
                                                  then argumentsCS.ast->collect(type)
                                                  else Sequence{} endif )
       else
        -- this is a set operation called on an object => implicit Set with one element
        SetType.allInstances()->any (st |
            st.elementType = OclExpressionCS.ast.type).lookupOperation (
                                           simpleNameCS.ast,
                                           if (argumentsCS->notEmpty())
                                           then argumentsCS.ast->collect(type)
                                           else Sequence{} endif )
    endif
[C] OperationCallExpCS.ast.referredOperation =
                OclExpressionCS.ast.type.lookupOperation (simpleNameCS.ast,
                                                          if argumentsCS->notEmpty()
                                                         then arguments.ast->collect(type)
                                                          else Sequence{} endif)
    OperationCallExpCS.ast.arguments = argumentsCS.ast
    OperationCallExpCS.ast.source    = OclExpressionCS.ast
[D] OperationCallExpCS.ast.arguments        = argumentsCS.ast and
    OperationCallExpCS.ast.referredOperation =
                              env.lookupImplicitOperation(simpleName.ast,
                                                          if argumentsCS->notEmpty()
                                                         then arguments.ast->collect(type)
                                                          else Sequence{} endif)
    OperationCallExpCS.ast.source = env.lookupImplicitSourceForOperation(
                                                          simpleName.ast,
                                                          if argumentsCS->notEmpty()
                                                         then arguments.ast->collect(type)
                                                          else Sequence{} endif)


[E] -- incorporate the isPre() operation.
    OperationCallExpCS.ast.referredOperation =
                OclExpressionCS.ast.type.lookupOperation (simpleNameCS.ast,
                                                          if argumentsCS->notEmpty()
                                                         then arguments.ast->collect(type)
```

```
                                                            else Sequence{} endif)
        OperationCallExpCS.ast.arguments = argumentsCS.ast
        OperationCallExpCS.ast.source    = OclExpressionCS.ast.withAtPre()
    [F] -- incorporate atPre() operation with the implicit source
        OperationCallExpCS.ast.arguments         = argumentsCS.ast and
        OperationCallExpCS.ast.referredOperation =
                                 env.lookupImplicitOperation(simpleName.ast,
                                                            if argumentsCS->notEmpty()
                                                          then arguments.ast->collect(type)
                                                            else Sequence{} endif)
                                 )
        OperationCallExpCS.ast.source =
                          env.lookupImplicitSourceForOperation(simpleName.ast,
                                                            if argumentsCS->notEmpty()
                                                          then arguments.ast->collect(type)
                                                            else Sequence{} endif)
                                 ).withAtPre()
    [G] OperationCallExpCS.ast.arguments         = argumentsCS.ast and
        OperationCallExpCS.ast.referredOperation =
                                          env.lookupPathName(pathName.ast,
                                                            if argumentsCS->notEmpty()
                                                          then arguments.ast->collect(type)
                                                            else Sequence{} endif)
        OperationCallExpCS.ast.source->isEmpty()
    -- this rule is for unary operators as '-' and 'not' etc. It has no argument.
    [H] OperationCallExpCS.ast.arguments->isEmpty()
        OperationCallExpCS.ast.source     = OclExpressionCS.ast
        OperationCallExpCS.ast.referredOperation =
                     OclExpressionCS.ast.type.lookupOperation (
                                             simpleNameCS.ast,
                                             Sequence{} )
```

Inherited attributes

```
    [A] OclExpressionCS[1].env = OperationCallExpCS.env
    [A] OclExpressionCS[2].env = OperationCallExpCS.env
    [B] OclExpressionCS.env    = OperationCallExpCS.env
    [B] argumentsCS.env        = OperationCallExpCS.env
    [C] OclExpressionCS.env    = OperationCallExpCS.env
    [C] argumentsCS.env        = OperationCallExpCS.env
    [D] argumentsCS.env        = OperationCallExpCS.env
    [E] OclExpressionCS.env    = OperationCallExpCS.env
    [E] argumentsCS.env        = OperationCallExpCS.env
    [F] argumentsCS.env        = OperationCallExpCS.env
```

Disambiguating rules

[1] [A] The name of the referred Operation must be an operator

```
    Set{'+','-','*','/','and','or','xor','=','<=','>=','<','>'}->includes(simpleNameCS.ast)
```

[2] [A,B,C,D,E,F] The referred Operation must be defined for the type of source

```
    not OperationCallExpCS.ast.referredOperation.oclIsUndefined()
```

[3] [C] The name of the referred Operation cannot be an operator.

```
    Set{'+','-','*','/','and','or','xor','=','<=','>=','<','>'}->excludes(simpleNameCS.ast)
```

AttributeCallExpCS

This production rule results in an AttributeCallExp. In production [A] the source is explicit, while production [B] is used for an implicit source. Alternative C covers the use of a classifier scoped attribute.

```
[A] AttributeCallExpCS ::= OclExpressionCS '.' simpleNameCS isMarkedPreCS?
[B] AttributeCallExpCS ::= simpleNameCS isMarkedPreCS?
```

```
[C] AttributeCallExpCS ::= pathNameCS
```

Abstract syntax mapping

```
AttributeCallExpCS.ast : AttributeCallExp
```

Synthesised attributes

```
[A] AttributeCallExpCS.ast.referredAttribute =
                                 OclExpressionCS.ast.type.lookupAttribute(simpleNameCS.ast)
[A] AttributeCallExpCS.ast.source = if isMarkedPreCS->isEmpty()
                                    then  OclExpressionCS.ast
                                    else  OclExpressionCS.ast.withAtPre()
                                    endif
[B] AttributeCallExpCS.ast.referredAttribute =
                                             env.lookupImplicitAttribute(simpleNameCS.ast)
[B] AttributeCallExpCS.ast.source =
                       if isMarkedPreCS->isEmpty()
                       then env.findImplicitSourceForAttribute(simpleNameCS.ast)
                      else env.findImplicitSourceForAttribute(simpleNameCS.ast).withAtPre()
                       endif
[C] AttributeCallExpCS.ast.referredAttribute =
                          env.lookupPathName(pathNameCS.ast).oclAsType(Attribute)
```

Inherited attributes

```
[A] OclExpressionCS.env = AttributeCallExpCS.env
```

Disambiguating rules

[1] [A, B] ’simpleName’ is name of an Attribute of the type of source or if source is empty the name of an attribute of ’self’ or any of the iterator variables in (nested) scope. In OCL:

```
not AttributeCallExpCS.ast.referredAttribute.oclIsUndefined()
```

[2] [C] The pathName refers to a class attribute.

```
env.lookupPathName(pathNameCS.ast).oclIsKindOf(Attribute)
and
AttributeCallExpCS.ast.referredAttribute.ownerscope = ScopeKind::instance
```

NavigationCallExpCS

This production rule represents a navigation call expression.

```
[A] NavigationCallExpCS ::= AssociationEndCallExpCS
[B] NavigationCallExpCS ::= AssociationClassCallExpCS
```

Abstract syntax mapping

```
NavigationCallExpCS.ast : NavigationCallExp
```

Synthesised attributes

The value of this production is the value of its child production.

```
[A] NavigationCallExpCS.ast = AssociationEndCallExpCS.ast
[B] NavigationCallExpCS.ast = AssociationClassCallExpCS.ast
```

Inherited attributes

```
[A] AssociationEndCallExpCS.env    = NavigationCallExpCS.env
[B] AssociationClassCallExpCS.env  = NavigationCallExpCS.env
```

Disambiguating rules

These are defined in the children.

AssociationEndCallExpCS

This production rule represents a navigation through an association end. Rule A is the default, rule B is used with an implicit source, while rule C is used with qualifiers.

```
[A] AssociationEndCallExpCS ::= OclExpressionCS '.' simpleNameCS
                                ('[' argumentsCS ']')? isMarkedPreCS?
[B] AssociationEndCallExpCS ::= simpleNameCS
                                ('[' argumentsCS ']')? isMarkedPreCS?
```

Abstract syntax mapping

```
AssociationEndCallExpCS.ast : AssociationEndCallExp
```

Synthesised attributes

```
[A] AssociationEndCallExpCS.ast.referredAssociationEnd =
                  OclExpressionCS.ast.type.lookupAssociationEnd(simpleNameCS.ast)
    AssociationEndCallExpCS.ast.source = if isMarkedPreCS->isEmpty()
                                  then  OclExpressionCS.ast
                                  else  OclExpressionCS.ast.withAtPre()
                                  endif
[A] AssociationEndCallExpCS.ast.qualifiers = argumentsCS.ast

[B] AssociationEndCallExpCS.ast.referredAssociationEnd =
                           env.lookupImplicitAssociationEnd(simpleNameCS.ast)
    AssociationEndCallExpCS.ast.source =
              if isMarkedPreCS->isEmpty()
              then env.findImplicitSourceForAssociationEnd(simpleNameCS.ast)
             else env.findImplicitSourceForAssociationEnd(simpleNameCS.ast).withAtPre()
              endif
[B] AssociationEndCallExpCS.ast.qualifiers = argumentsCS.ast
```

Inherited attributes

```
[A] OclExpressionCS.env = AssociationEndCallExpCS.env
[A, B] argumentsCS.env  = AssociationEndCallExpCS.env
```

Disambiguating rules

[1] [A,B] 'simpleName' is name of an AssociationEnd of the type of source or if source is empty the name of an AssociationEnd of 'self' or any of the iterator variables in (nested) scope. In OCL:

```
not AssociationEndCallExpCS.ast.referredAssociationEnd.oclIsUndefined()
```

AssociationClassCallExpCS

This production rule represents a navigation to an association class.

```
[A] AssociationClassCallExpCS ::= OclExpressionCS '.' simpleNameCS
                                ('[' argumentsCS ']')? isMarkedPreCS?
[B] AssociationClassCallExpCS ::= simpleNameCS
                                ('[' argumentsCS ']')? isMarkedPreCS?
```

Abstract syntax mapping

```
AssociationClassCallExpCS.ast : AssociationClassCallExp
```

Synthesised attributes

```
[A] AssociationClassCallExpCS.ast.referredAssociationClass =
                OclExpressionCS.ast.type.lookupAssociationClass(simpleNameCS.ast)
    AssociationClassCallExpCS.ast.source = if isMarkedPreCS->isEmpty()
                                  then  OclExpressionCS.ast
                                  else  OclExpressionCS.ast.withAtPre()
                                  endif
[A] AssociationClassCallExpCS.ast.qualifiers = argumentsCS.ast

[B] AssociationClassCallExpCS.ast.referredAssociationClass =
                         env.lookupImplicitAssociationClass(simpleNameCS.ast)
    AssociationClassCallExpCS.ast.source =
          if isMarkedPreCS->isEmpty()
          then env.findImplicitSourceForAssociationClass(simpleNameCS.ast)
          else env.findImplicitSourceForAssociationClass(simpleNameCS.ast).withAtPre()
          endif
```

```
[B] AssociationClassCallExpCS.ast.qualifiers = argumentsCS.ast
```

Inherited attributes

```
[A] OclExpressionCS.env = AssociationClassCallExpCS.env
[A, B] argumentsCS.env  = AssociationClassCallExpCS.env
```

Disambiguating rules

[1] ’simpleName’ is name of an AssociationClass of the type of source.

```
not AssociationClassCallExpCS.ast.referredAssociationClass.oclIsUndefined()
```

isMarkedPreCS

This production rule represents the marking @pre in an ocl expression.

```
isMarkedPreCS ::= ’@’ ’pre’
```

Abstract syntax mapping

```
isMarkedPreCS.ast : Boolean
```

Synthesised attributes

```
self.ast = true
```

Inherited attributes

```
-- none
```

Disambiguating rules

```
-- none
```

argumentsCS

This production rule represents a sequence of arguments.

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

Abstract syntax mapping

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

Synthesised attributes

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

Inherited attributes

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

Disambiguating rules

```
-- none
```

LetExpCS

This production rule represents a let expression. The LetExpSubCS nonterminal has the purpose of allowing directly nested let expressions with the shorthand syntax, i.e. ending with one ’in’ keyword.

```
LetExpCS ::= ’let’ VariableDeclarationCS
                   LetExpSubCS
```

Abstract syntax mapping

```
LetExpCS.ast : LetExp
```

Synthesised attributes

```
LetExpCS.ast.variable = VariableDeclarationCS.ast
LetExpCS.ast.in       = LetExpSubCS.ast
```

Inherited attributes

```
LetExpSubCS.env = LetExpCS.env.nestedEnvironment().addElement(
                                              VariableDeclarationCS.ast.varName,
                                              VariableDeclarationCS.ast,
```

```
                                          false)
```

Disambiguating rules

[1] The variable name must be unique in the current scope

```
LetExpCS.env.lookup (VariableDeclarationCS.ast.varName).oclIsUndefined()
```

[2] A variable declaration inside a let must have a declared type and an initial value.

```
not VariableDeclarationCS.ast.type.oclIsUndefined() and
VariableDeclarationCS.ast.initExpression->notEmpty()
```

LetExpSubCS

```
[A] LetExpSubCS[1] ::= ',' VariableDeclarationCS LetExpSubCS[2]
[B] LetExpSubCS    ::= 'in' OclExpressionCS
```

Abstract syntax mapping

```
LetExpSubCS.ast : OclExpression
```

Synthesised attributes

```
[A] LetExpSubCS[1].ast.oclAsType(LetExp).variable      = VariableDeclarationCS.ast
[A] LetExpSubCS[1].ast.oclAsType(LetExp).OClExpression = LetExpSubCS[2].ast

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

Inherited attributes

```
[A] VariableDeclarationCS.env = LetExpSubCS[1].env
[A] LetExpSubCS[2].env = LetExpSubCS[1].env.nestedEnvironment().addElement(
                                          VariableDeclarationCS.ast.varName,
                                          VariableDeclarationCS.ast,
                                          false)

[B] OClExpressionCS.env = LetExpSubCS.env
```

Disambiguating rules

[A] The variable name must be unique in the current scope

```
LetExpSubCS[1].env.lookup (VariableDeclarationCS.ast.varName).oclIsUndefined()
```

[A] A variable declaration inside a let must have a declared type and an initial value.

```
not VariableDeclarationCS.ast.type.oclIsUndefined() and
VariableDeclarationCS.ast.initExpression->notEmpty()
```

OclMessageExpCS

The message Name must either be the name of a Signal, or the name of an Operation belonging to the target object(s).

```
[A] OclMessageExpCS ::= OclExpressionCS '^^'
                        simpleNameCS '(' OclMessageArgumentsCS? ')'

[B] OclMessageExpCS ::= OclExpressionCS '^'
                        simpleNameCS '(' OclMessageArgumentsCS? ')'
```

Abstract syntax mapping

```
[A] OclMessageExpCS.ast : OclMessageExp
[B] OclMessageExpCS.ast : OclMessageExp
```

Synthesised attributes

```
[A] OclMessageExpCS.ast.target    = OclExpressionCS.ast
[A] OclMessageExpCS.ast.arguments = OclMessageArgumentsCS.ast

-- first, find the sequence of types of the operation/signal parameters
[A] let params : Sequence(Classifier) = OclMessageArguments.ast->collect(messArg |
                                         messArg.getType() ),
```

```
-- try to find either the called operation or the sent signal
[A]     operation : Operation = OclMessageExpCS.ast.target.type.
                                   lookupOperation(simpleNameCS.ast, params),
        signal : Signal = OclMessageExpCS.ast.target.type.
                                       lookupSignal(simpleNameCS.ast, params)
    in
    OclMessageExpCS.ast.calledOperation = if operation->isEmpty()
                                          then OclUndefined
                                          else = operation
                                          endif
    OclMessageExpCS.ast.sentSignal = if signal->isEmpty()
                                     then OclUndefined
                                     else signal
                                     endif
[B]
-- OclExpression^simpleNameCS(OclMessageArguments) is identical to
-- OclExpression^^simpleNameCS(OclMessageArguments)->size() = 1
-- actual mapping: straigthforward, TBD...
```

Inherited attributes

```
OclExpressionCS.env       = OclMessageExpCS.env
OclMessageArgumentsCS.env = OclMessageExpCS.env
```

Disambiguating rules

```
-- none
```

OclMessageArgumentsCS

```
OclMessageArgumentsCS[1] ::= OclMessageArgCS
                             ( ',' OclMessageArgumentsCS[2] )?
```

Abstract syntax mapping

```
OclMessageArgumentsCS[1].ast : Sequence(OclMessageArg)
```

Synthesised attributes

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

Inherited attributes

```
OclMessageArgCS.env          = OclMessageArgumentsCS[1].env
OclMessageArgumentsCS[2].env = OclMessageArgumentsCS[1].env
```

Disambiguating rules

```
-- none
```

OclMessageArgCS

```
[A] OclMessageArgCS ::= '?' (':' typeCS)?
[B] OclMessageArgCS ::= OclExpressionCS
```

Abstract syntax mapping

```
OclMessageArgCS.ast : OclMessageArg
```

Synthesised attributes

```
[A] OclMessageArgCS.ast.expression->isEmpty()
[A] OclMessageArgCS.ast.unspecified->notEmpty()
[A] OclMessageArgCS.ast.type = typeCS.ast

[B] OclMessageArgCS.ast.unspecified->isEmpty()
[B] OclMessageArgCS.ast.expression = OclExpressionCS.ast
```

Inherited attributes

```
OclExpressionCS.env = OclMessageArgCS.env
```

Disambiguating rules

```
-- none
```

IfExpCS

```
IfExpCS ::= 'if'   OclExpression[1]
            'then' OclExpression[2]
            'else' OclExpression[3]
            'endif'
```

Abstract syntax mapping

```
IfExpCS.ast : IfExp
```

Synthesised attributes

```
IfExpCS.ast.condition      = OclExpression[1].ast
IfExpCS.ast.thenExpression = OclExpression[2].ast
IfExpCS.ast.elseExpression = OclExpression[3].ast
```

Inherited attributes

```
OclExpression[1].env = IfExpCS.env
OclExpression[2].env = IfExpCS.env
OclExpression[3].env = IfExpCS.env
```

Disambiguating rules

```
-- none
```

AttributeDefinitionListCS

This production rule represents a definition expression. The AttributeDefinitionListCS nonterminal has the purpose of defining additional attributes in OCL. Note that this maps directly to a UML attribute. If thge initExpression of the attribute definition is not empty this maps to a constraint that defines the derivation of the attribute value.

Also note that this can not be used in ordinary OCL expressions, only and uniquey in a <<definition>> constraint. See section 7.4 (“Concrete Syntax of Context Declarations”).

```
AttributeDefinitionCS ::= 'attr' VariableDeclarationListCS
```

Abstract syntax mapping

```
AttributeDefinitionCS.ast : Sequence(Attribute)
```

Synthesised attributes

```
AttributeDefinitionCS.ast = VariableDeclarationListCS.ast->collect(v | v.asAttribute() )
```

Inherited attributes

```
VariableDeclarationCS.env = AttributeDefinitionCS.env
```

Disambiguating rules

[1] An attribute must have a declared type.

```
not VariableDeclarationCS.ast->foeAll(type.oclIsUndefined())
```

OperationDefinitionListCS

This production rule represents a definition expression. The OperationDefinitionCS nonterminal has the purpose of defining additional operations in OCL. They map directly to a UML operation with a constraint that defines the derivation of the operation result value.

```
OperationDefinitionCS ::= 'oper' OperationListCS
```

Abstract syntax mapping

```
OperationDefinitionListCS.ast : Sequence( OperationDefinition )
```

Synthesized attributes

```
OperationDefinitionListCS.ast = OperationListCS.ast
```

Inherited attributes

```
operationListCS.env    = OperationDefinitionListCS[1].env
```

Disambiguating rules

```
-- none
```

operationListCS

This production rule represents a list of operation definitions, separated by the ’,’ separator..

```
operationListCS[1] = operationDefinitionCS
                              (’,’operationListCS[2] )?
```

Abstract syntax mapping

```
operationListCS[1].ast : Sequence( OperationDefinition )
```

Synthesized attributes

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

Inherited attributes

```
OperationDefinitionCS.env = operationListCS[1].env
operationListCS[2].env    = operationListCS[1].env
```

Disambiguating rules

```
-- none
```

OperationDefinitionCS

This production rule represents a definition expression. The OperationDefinitionCS nonterminal has the purpose of defining additional operations in OCL. They map directly to a UML operation with a constraint that defines the derivation of the attribute value.

```
OperationDefinitionCS ::= ’simpleNameCS ’(’ parametersCS? ’)’ ’:’ typeCS
                          (’=’ OclExpresionCS)?
```

Abstract syntax mapping

```
OperationDefinitionCS.ast : Operation
```

Synthesised attributes

```
OperationDefinitionCS.ast.name              = simpleNameCS.ast
OperationDefinitionCS.ast.parameter         = parametersCS.ast->append(
                                              Parameter.make(simpleName.ast,
                                                             typeCS.ast,
                                                         ParameterDirectionKind::return)
OperationDefinitionCS.ast.isQuery           = true
OperationDefinitionCS.ast.ownerscope        = ScopeKind::instance
OperationDefinitionCS.ast.visibility        = VisibilityKind::public
OperationDefinitionCS.ast.stereotype.name   = ’OclHelper’
OperationDefinitionCS.ast.constraint.bodyExpression  = OclExpressionCS.ast
OperationDefinitionCS.ast.constraint.stereotype.name = ’postcondition’
```

Inherited attributes

```
OclExpresionCS.env = OperationDefinitionCS.env
parameterCS.env    = OperationDefinitionCS.env
```

Disambiguating rules

```
-- none
```

parametersCS

This production rule represents the formal parameters of an operation .

```
parametersCS[1] = VariableDeclarationCS (',' parametersCS[2] )?
```

Abstract syntax mapping

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

Synthesized attributes

```
parameterCS[1].ast = Sequence { VariableDeclarationCS.ast.asParameter() }
                                                        ->union( parameterCS[2].ast)
```

Inherited attributes

```
-- none
```

Disambiguating rules

```
-- none
```

4.3.1 Comments

It is possible to include comments anywhere in a text composed according to the above concrete syntax. There will be no mapping of any comments to the abstract syntax. Comments are simply skipped when the text is being parsed. There are 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.

4.3.2 Operator Precedence

In the grammar, the precedence of the operators from highest to lowest is as follows:

- @pre
- dot and arrow operations: '.' and '->'
- unary 'not' and unary minus '-'
- '*' and '/'
- '+' and binary '-'
- 'if-then-else-endif'
- '<', '>', '<=', '>='
- '=', '<>'
- 'and', 'or' and 'xor'
- 'implies'

Parentheses '(' and ')' can be used to change precedence.

4.4 ENVIRONMENT DEFINITION

The Environment type used in the rules for the concrete syntax is defined according to the following invariants and additional operations. A diagrammatic view can be found in figure 4-1 on page 4-2. Environments can be nested, denoted by the existence of a parent environment. Each environment keeps a list of named elements, that have a name a reference to a ModelElement.

4.4.1 Environment

The definition of Environment has the following invariants and specifications of its operations.

[1] The attribute EMPTY_ENV is really just a helper to avoid having to say new Environment (...).

```
context Environment
inv EMPTY_ENV_Definition: EMPTY_ENV.namedElements->isEmpty()
```

[2] Find a named element in the current environment, not in its parents, based on a single name.

```
context Environment::lookupLocal(name : String) : NamedElement
post: result = namedElements->any(v | v.name = name)
```

[3] Find a named element in the current environment or recursively in its parent environment, based on a single name.

```
context Environment::lookup(name: String) : ModelElement
post: result = if not lookupLocal(name).oclIsUndefined() then
                  lookupLocal(name).referredElement
               else
                  parent.lookup(name)
               endif
```

[4] Find a named element in the current environment or recursively in its parent environment, based on a path name.

```
context Environment::lookupPathName(names: Sequence(String)) : ModelElement
post: let firstNamespace : ModelElement = lookupLocal( names->first() ).referredElement
in
      if firstNamespace.isOclKind(Namespace)
          -- indicates a sub namespace of the namespace in which self is present
      then
          result = self.nestedEnvironment().addNamespace(
                               firstNamespace ).lookupPathName( names->tail() )
      else
          -- search in surrounding namespace
          result = parent.lookupPathName( names )
      endif
```

[5] Add a new named element to the environment. Note that this operation is defined as a query operation so that it can be used in OCL constraints.

```
context Environment::addElement (name : String,
                                 elem : ModelElement, imp : Boolean) : Environment
pre : -- the name must not clash with names already existing in this environment
      self.lookupLocal(name).oclIsUndefined()
post: result.parent = self.parent and
      result.namedElements->includesAll (self.namedElements) and
      result.namedElements->count (v | v.oclIsNew()) = 1 and
      result.namedElements->forAll (v | v.oclIsNew() implies
                                          v.name = name and v.referredElement = elem)
                                          and
                                          v.mayBeImplicit = imp )
```

[6] Combine two environments resulting in a new environment. Note that this operation is defined as a query operation so that it can be used in OCL constraints.

```
context Environment::addEnvironment(env : Environment) : Environment
pre : -- the names must not clash with names already existing in this environment
      enf.namedElements->forAll(nm | self.lookupLocal(nm).oclIsUndefined() )
post: result.parent = self.parent and
      result.namedElements = self.namedElements->union(env.namedElements)
```

[7] Add all elements in the namespace to the environment.

```
context Environment::addNamespace(ns: Namespace) : Environment
post: result.namedElements = ns.getEnvironmentWithoutParents().namedElements->union(
                                                                  self.namedElements)
post: result.parent = self.parent
```

[8] This operation results in a new environment which has the current one as its parent.

```
context Environment::nestedEnvironment() : Environment
post: result.namedElements->isEmpty()
post: result.parent = self
post: result.oclIsNew()
```

[9] Lookup a given attribute name of an implicitly named element in the current environment, including its parents.

```
context Environment::lookupImplicitAttribute(name: String) : Attribute
pre: -- none
post: result =
              lookupImplicitSourceForAttribute(name).referredElement.oclAsType(Attribute)
```

[10] Lookup the implicit source belonging to a given attribute name in the current environment, including the parents.

```
context Environment::lookupImplicitSourceForAttribute(name: String) : NamedElement
pre: -- none
post: let foundElement : NamedElement =
        namedElements->select(mayBeImplicit)
            ->any( ne | not ne.getType().lookupAttribute(name).oclIsUndefined() ) in
      result = if foundAttribute.oclIsUndefined() then
                  self.parent.lookupImplicitSource ForAttribute(name)
               else
                  foundElement
               end
```

[11] Lookup up a given association end name of an implicitly named element in the current environment, including its parents.

```
context Environment::lookupImplicitAssociationEnd(name: String) : AssociationEnd
pre: -- none
post: let foundAssociationEnd : AssociationEnd =
        namedElements->select(mayBeImplicit)
             ->any( ne | not ne.getType().lookupAssociationEnd(name).oclIsUndefined() ) in
      result = if foundAssociationEnd.oclIsUndefined() then
                  self.parent.lookupImplicitAssociationEnd(name)
               else
                  foundAssociationEnd
               end
```

[12] Lookup up an operation of an implicitly named element with given name and parameter types in the current environment, including its parents.

```
context Environment::lookupImplicitOperation(name: String,
                                             params : Sequence(Classifier)) : Operation
pre: -- none
post: let foundOperation : Operation =
        namedElements->select(mayBeImplicit)
          ->any( ne | not ne.getType().lookupOperation(name, params).oclIsUndefined() ) in
      result = if foundOperation.oclIsUndefined() then
                  self.parent.lookupImplicitOperation(name)
               else
                  foundOperation
               end
```

4.4.2 NamedElement

A named element is a modelelement which is referred to by a name. A modelement itself has a name, but this is not always the name which is used to refer to it.

The operation getType() returns the type of the referred modelelement.

```
context NamedElement::getType() : Classifier
pre: -- none
post: referredElement.oclIsKindOf(VariableDeclaration) implies
                    result = referredElement.oclAsType(VariableDeclaration).type
post: referredElement.oclIsKindOf(Classifier) implies
                    result = referredElement
post: referredElement.oclIsKindOf(State) implies
                    result = -- TBD: when aligning with UML 2.0 Infrastructure
```

4.4.3 Namespace

The following additional operation returns the information of the contents of the namespace in the form of an Environment object, where Environment is the class defined in this chapter. Note that the *parent* association of Environment is not filled.

Because the definition of this operation is completely dependent on the UML metamodel, and this model will be considerably different in the 2.0 version, the definition is left to be done.

```
context Namespace::getEnvironmentWithoutParents() : Environment
post: self.isTypeOf(Classifier)  implies -- TBD when aligning with UML 2.0 Infrastrcuture
      -- include all class features and contained classifiers
post: self.isTypeOf(Package)     implies -- TBD when aligning with UML 2.0 Infrastrcuture
      -- include all classifiers and subpackages
post: self.isTypeOf(StateMachine)implies -- TBD when aligning with UML 2.0 Infrastrcuture
      -- include all states
post: self.isTypeOf(Subsystem)   implies -- TBD when aligning with UML 2.0 Infrastrcuture
      -- include all classifiers and subpackages
```

The following operation returns an Environment that contains a reference to its parent environment, which is itself created by this operation by means of a recursive call, and therefore contains a parent environment too.

```
context Namespace::getEnvironmentWithParents() : Environment
post: result.NamedElements = self.getEnvironmentWithoutParents()
post: if self.namespace->notEmpty() -- this namespace has an owning namespace
      then result.parent = self.namespace.getEnvironmentWithParents()
      else result.parent = OclUndefined
      endif
```

4.5 CONCRETE TO ABSTRACT SYNTAX MAPPING

The mapping from concrete to abstract syntax is described as part of the grammar. It is described by adding a synthesized attribute *ast* to each production which has the corresponding metaclass from the abstract syntax as its type. This allows the mapping to be fully formalized within the attribute grammar formalism.

4.6 ABSTRACT SYNTAX TO CONCRETE SYNTAX MAPPING

IIt is often useful to have a defined mapping from the abstract syntax to the concrete syntax. This mapping can be defined by applying the production rules in section 4.3 (“Concrete Syntax”) from left to right. As a general guideline nothing will be implicit (like e.g implicit collect, implicit use of object as set, etc.), and all iterator variables will be filled in completely. The mapping is not formally defined in this document but should be obvious.

5

Semantics Described using UML

This chapter describes the semantics of the OCL using the UML itself to describe the semantic domain and the mapping between semantic domain and abstract syntax. It explains the semantics of OCL in a manner based on the report *Unification of Static and Dynamic Semantics for UML* [Kleppe2001], which in its turn is based on the MML report [Clark2000]. The main difference between appendix A (“Semantics”), which describes the semantics in a formal manner, and this chapter is that this chapter defines a semantics for the ocl message expression.

5.1 INTRODUCTION

In section 3.3 (“The Expressions Package”) an OCL expression is defined as: "an expression that can be evaluated in a given environment", and in section 3.2 (“The Types Package”) it is stated that an "evaluation of the expression yields a value". The ‘meaning’ (semantics) of an OCL expression, therefore, can be defined as the value yielded by its evaluation in a given environment.

In order to specify the semantics of OCL expressions we need to define two things: (1) the set of possible values that evaluations of expressions may yield, and (2) evaluations and their environment. The set of possible values is called the *semantic domain*. The set of evaluations together with their associations with the concepts from the abstract syntax represent the mapping from OCL expressions to values from the semantic domain. Together the semantic domain and the evaluations with their environment will be called *domain* in this chapter.

The semantic domain is described in the form of a UML package, containing a UML class diagram, classes, associations, and attributes. The real semantic domain is the (infinite) set of instances that can be created according to this class diagram. To represent the evaluation of the OCL expressions in the semantic domain a second UML package is used. In it, a set of so-called *evaluation* classes is defined (in short *eval*). Each evaluation class is associated with a value (its result value), and a name space environment that binds names to values. Note that the UML model comprising both packages, resides on layer 1 of the OMG 4-layered architecture, while the abstract syntax defined in chapter 3 (“Abstract Syntax”), resides on layer 2.

The semantics of an OCL expression is given by association: each value defined in the semantic domain is associated with a type defined in the abstract syntax, each evaluation is associated with an expression from the abstract syntax. The value yielded by an OCL expression in a given environment, its ‘meaning’, is the result value of its evaluation within a certain name space environment. The semantics are also described in the form of a UML package called "AS-Domain-Mapping". Note that this package links the domain on layer 1 of the OMG 4-layered architecture with the abstract syntax on layer 2. The AS-Domain-Mapping package itself can not be positioned in one of the layers of the OMG 4-layered architecture. Note also that this package contains associations only, no new classes are defined.

Figure 5-1 on page 5-2 shows how the packages defined in this chapter relate to each other, and to the packages from the abstract syntax. It shows the following packages:

- The *Domain* package describes the values and evaluations. It is subdivided into two subpackages:
  - The *Values* package describes the semantic domain. It shows the values OCL expressions may yield as result.

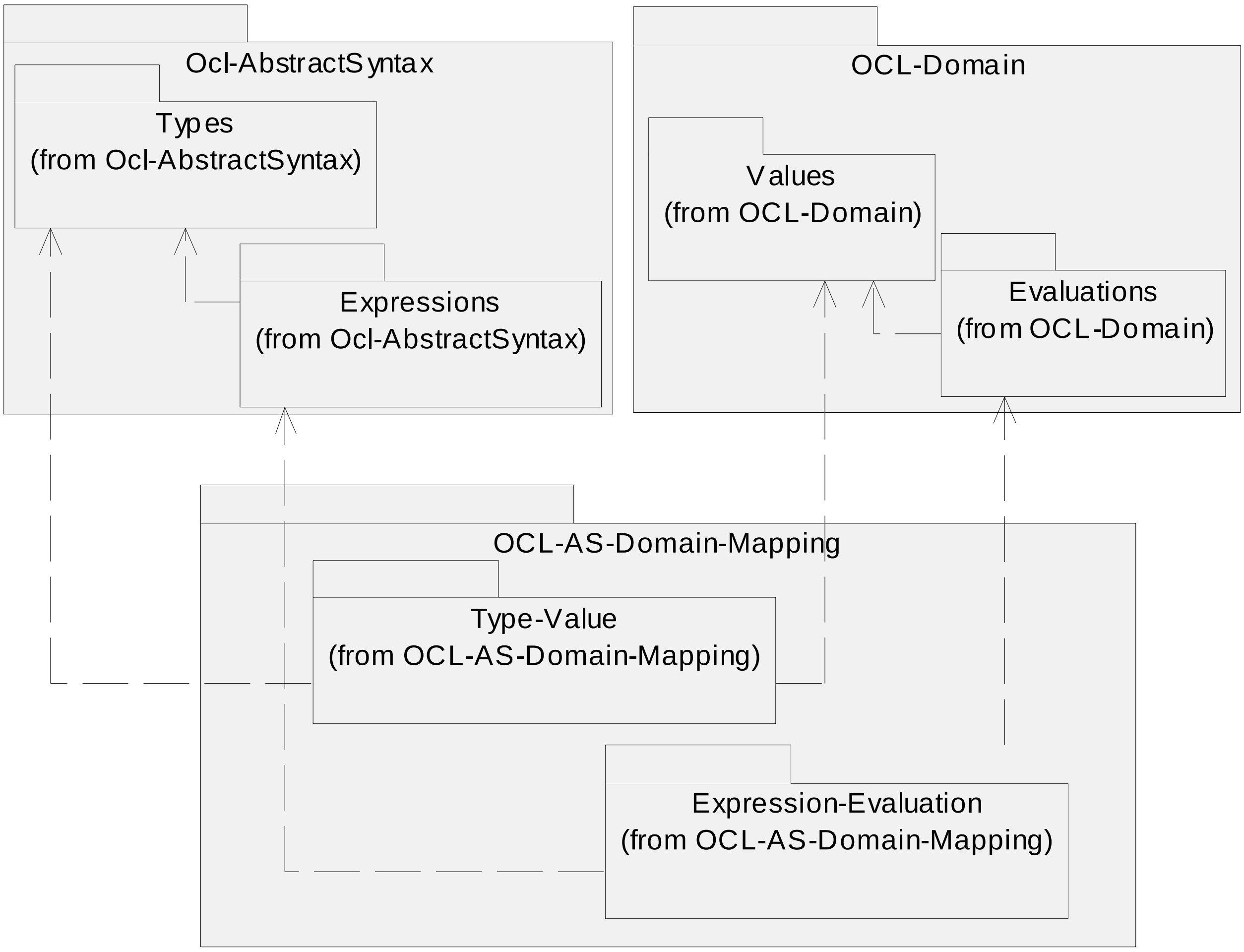


Figure 5-1 *Overview of packages in the UML-based semantics*

- The *Evaluations* package describes the evaluations of OCL expressions. It contains the rules that determine the result value for a given expression.
- The *AS-Domain-Mapping* package describes the associations of the values and evaluations with elements from the abstract syntax. It is subdivided into two subpackages:
  - The *Type-Value* package contains the associations between the instances in the semantics domain and the types in the abstract syntax.
  - The *Expression-Evaluation* package contains the associations between the evaluation classes and the expressions in the abstract syntax.

5.2 THE VALUES PACKAGE

OCL is an object language. A value can be either an object, which can change its state in time, or a data type, which can not change its state. The model in figure 5-2 on page 5-3 shows the values that form the semantic domain of an OCL expression. The basic type is the Value, which includes both objects and data values. There is a special subtype of Value called UndefinedValue, which is used to represent the undefined value for any Type in the abstract syntax.

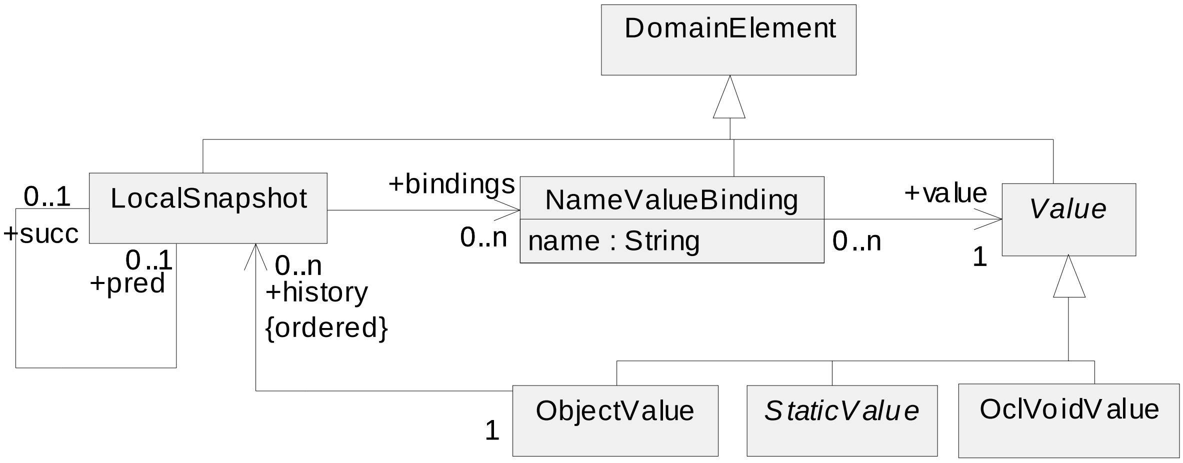


Figure 5-2 *The kernel values in the semantic domain*

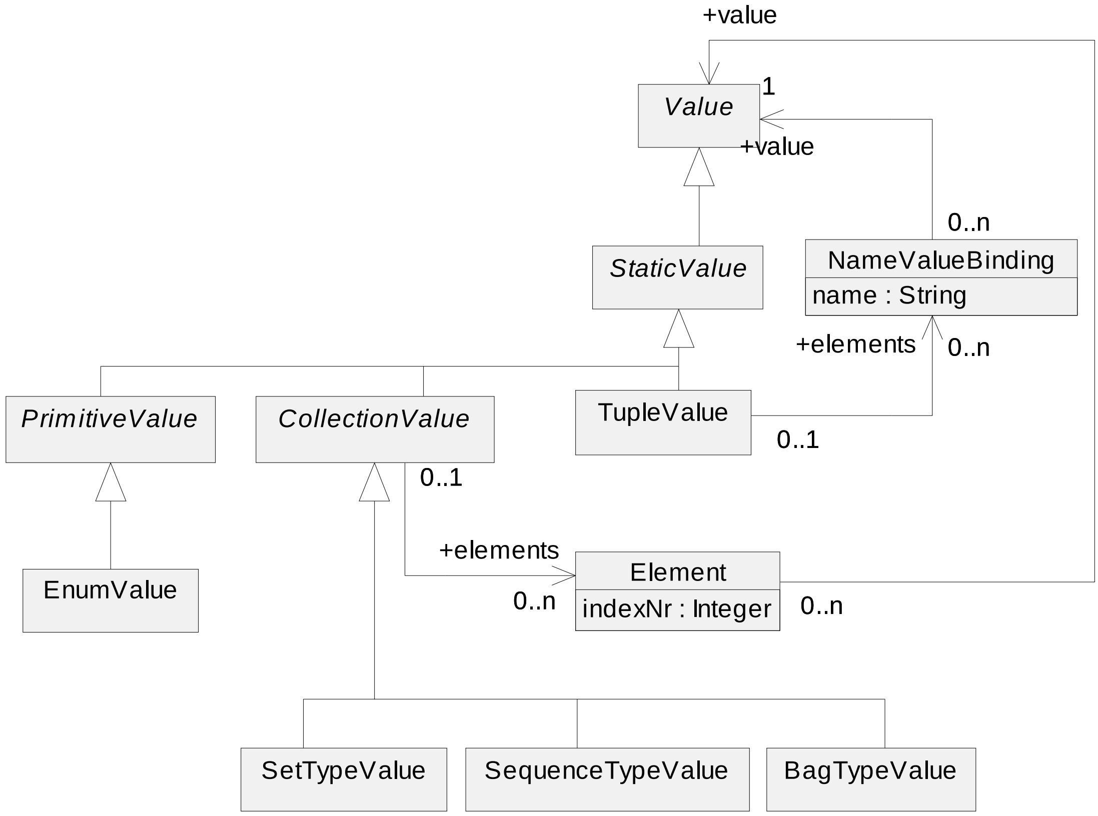


Figure 5-3 *The collection and tuple values in the semantic domain*

Figure 5-3 on page 5-3 shows a number of special data values, the collection and tuple values. To distinguish between instances of the Set, Bag, and Sequence types defined in the standard library, and the classes in this package that represent instances in the semantic domain, the names SetTypeValue, BagTypeValue, and SequenceTypeValue are used, instead of SetValue, BagValue, and SequenceValue.

The value resulting from an ocl message expression is shown in figure 5-4 on page 5-5. It links an ocl message value to the snapshot of an object.

5.2.1 Definitions of concepts for the Values package.

The section lists the definitions of concepts in the Values package in alphabetical order.

BagTypeValue

A bag type value is a collection value which is a multiset of values, where each value may occur multiple times in the bag. The values are unordered. In the metamodel, this list of values is shown as an association from *CollectionValue* (a generalization of *BagTypeValue*) to *Element*.

CollectionValue

A collection value is a list of values. In the metamodel, this list of values is shown as an association from *CollectionValue* to *Element*.

Associations

elements	The values of the elements in a collection.

DomainElement

A domain element is an element of the domain of OCL expressions. It is the generic superclass of all classes defined in this chapter, including Value and OclExpEval. It serves the same purpose as ModelElement in the UML meta model.

Element

An element represents a single component of a tuple value, or collection value. An element has an index number, and a value. The purpose of the index number is to uniquely identify the position of each element within the enclosing value, when it is used as an element of a SequenceValue.

LocalSnapshot

A local snapshot is a domain element that holds for one point in time the subvalues of an object value. It is always part of an ordered list of local snapshots of an object value, which is represented in the metamodel by the associations *pred*, *succ,* and *history.* An object value may also hold a sequence of *OclMessageValues*, which the object value has sent, and a sequence of *OclMessageValues*, which the object value has received. Both sequences can change in time, therefore they are included in a local snapshot. This is represented by the associations in the metamodel called *inputQ*, and *outputQ*.

A local snapshot has two attributes, *isPost* and *isPre*, that indicate whether this snapshot is taken at postcondition or precondition time of an operation execution. Within the history of an object value it is always possible to find the local snapshot at precondition time that corresponds with a given snapshot at postcondition time. The association *pre* (shown in figure 5-4 on page 5-5) is redundant, but added for convenience.

Associations

bindings	The set of name value bindings that hold the changes in time of the subvalues of the associated object value.
outputQ	The sequence of OclMessageValues that the associated ObjectValue at the certain point in time has sent, and are not yet put through to their targets.
inputQ	The sequence of OclMessageValues that the associated ObjectValue at the certain point in time has received, but not yet dealt with.

pred	The predecessor of this local snapshot in the history of an object value.
succ	The successor of this local snapshot in the history of an object value.
pre	If this snapshot is a snapshot at postcondition time of a certain operation execution, then *pre* is the associated snapshot at precondition time of the same operation in the history of an object value.

NameValueBinding

A name value binding is a domain element that binds a name to a value.

ObjectValue

An object value is a value that has an identity, and a certain structure of subvalues. Its subvalues may change over time, although the structure remains the same. Its identity may not change over time. In the metamodel, the structure is shown as a set of *NameValueBindings*. Because these bindings may change over time, the *ObjectValue* is associated with a sequence of *LocalSnapshots*, that hold a set of *NameValueBindings* at a certain point in time.

Associations

history	The sequence of local snapshots that hold the changes in time of the subvalues of this object value.

OclMessageValue

An ocl message value is a value that has as target and as source an object value. An ocl message value has a number of attributes. The name attribute corresponds to the *name* of the operation called, or signal sent. The *isSyncOperation*, *isAsyncOperation*, and *isSignal* attributes indicate respectively whether the message corresponds to a synchronous operation, an asynchrounous operation, or a signal.

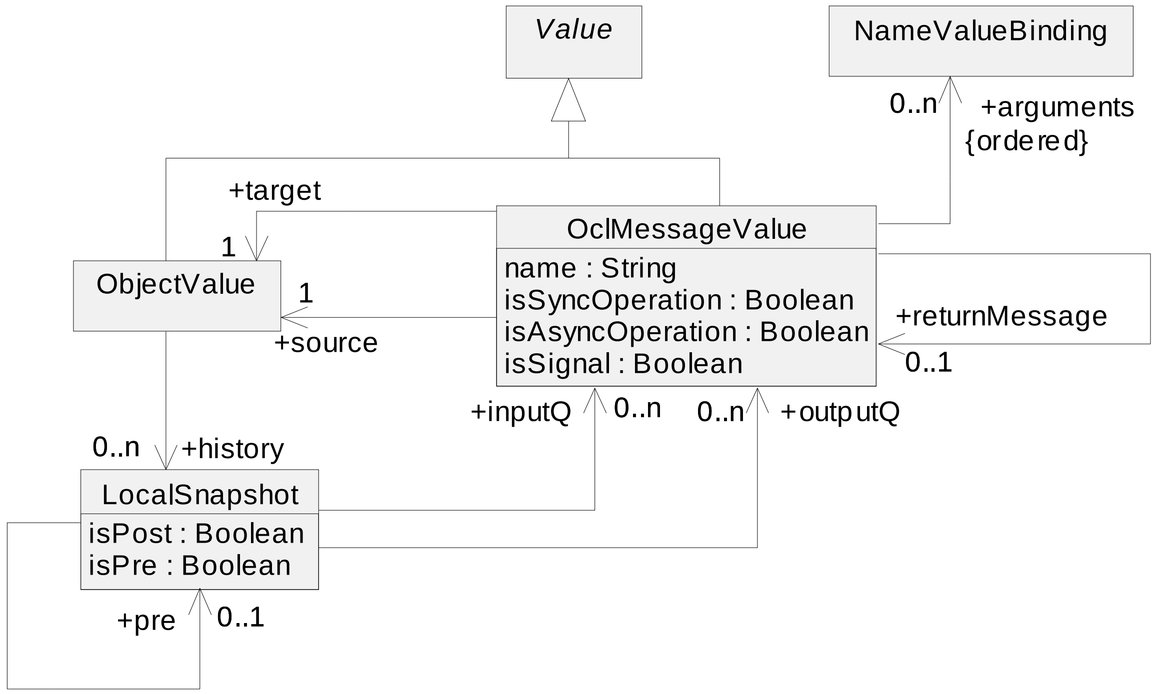


Figure 5-4 *The message values in the semantic domain*

Associations

arguments	A sequence of name value bindings that hold the arguments of the message from the source to the target.
source	The object value that has sent this signal.
target	The object value for which this signal has been intended.
returnMessage	The ocl message value that holds the values of the result and out parameters of a synchronous operation call in its arguments. Is only present if this message represents a synchronous operation call.

OclVoidValue

An undefined value is a value that represents void or undefined for any type.

PrimitiveValue

A primitive value is a predefined static value, without any relevant substructure (i.e., it has no parts).

SequenceTypeValue

A sequence type value is a collection value which is a list of values where each value may occur multiple times in the sequence. The values are ordered by their position in the sequence. In the metamodel, this list of values is shown as an association from *CollectionValue* (a generalization of *SequenceTypeValue*) to *Element*. The position of an element in the list is represented by the attribute *indexNr* of *Element*.

SetTypeValue

A set type value is a collection value which is a set of elements where each distinct element occurs only once in the set. The elements are not ordered. In the metamodel, this list of values is shown as an association from *CollectionValue* (a generalization of *SetTypeValue*) to *Element*.

StaticValue

A static value is a value that will not change over time.[1]

TupleValue

A tuple value (also known as record value) combines values of different types into a single aggregate value. The components of a tuple value are described by tuple parts each having a name and a value. In the metamodel, this is shown as an association from *TupleValue* to *NameValueBinding*.

Associations

elements	The names and values of the elements in a tuple value.

Value

A part of the semantic domain.

1. As *StaticValue* is the counterpart of the *DataType* concept in the abstract syntax, the name *DataValue* would be preferable. Because this name is used in the UML 1.4 specification to denote a model of a data value, the name *StaticValue* is used here.

5.2.2 Well-formedness rules for the Values Package

BagTypeValue

No additional well-formedness rules.

CollectionValue

No additional well-formedness rules.

DomainElement

No additional well-formedness rules.

Element

No additional well-formedness rules.

EnumValue

No additional well-formedness rules.

LocalSnapshot

[1] Only one of the attributes *isPost* and *isPre* may be true at the same time.

```
context LocalSnapshot
inv: isPost implies isPre = false
inv: ispre implies isPost = false
```

[2] Only if a snapshot is a postcondition snapshot it has an associated precondition snapshot.

```
context LocalSnapshot
inv: isPost implies pre->size() = 1
inv: not isPost implies pre->size() = 0
inv: self.pre->size() = 1 implies self.pre.isPre = true
```

NameValueBinding

No additional well-formedness rules.

ObjectValue

[1] The history of an object is ordered. The first element does not have a predecessor, the last does not have a successor.

```
context ObjectValue
inv: history->oclIsTypeOf( Sequence(LocalSnapShot) )
inv: history->last().succ->size = 0
inv: history->first().pre->size = 0
```

OclMessageValue

[1] Only one of the attributes *isSyncOperation*, *isAsyncOperation*, and *isSignal* may be true at the same time.

```
context OclMessageValue
inv: isSyncOperation implies isAsyncOperation = false and isSignal = false
```

```
inv: isAsyncOperation implies isSyncOperation = false and isSignal = false
inv: isSignal implies isSyncOperation = false and isAsyncOperation = false
```

[2] The return message is only present if, and only if the ocl message value is a synchronous operation call.

```
context OclMessageValue
inv: isSyncOperation implies returnMessage->size() = 1
inv: not isSyncOperation implies returnMessage->size() = 0
```

OclVoidValue

No additional well-formedness rules.

PrimitiveValue

No additional well-formedness rules.

SequenceTypeValue

[1] All elements belonging to a sequence value have unique index numbers.

```
self.element->isUnique(e : Element | e.indexNr)
```

SetTypeValue

[1] All elements belonging to a set value have unique values.

```
self.element->isUnique(e : Element | e.value)
```

StaticValue

No additional well-formedness rules.

TupleValue

[1] All elements belonging to a tuple value have unique names.

```
self.elements->isUnique(e : Element | e.name)
```

Value

No additional well-formedness rules.

5.2.3 Additional operations for the Values Package

LocalSnapshot

[1] The operation *allPredecessors* returns the collection of all snapshots before a snapshot, *allSuccessors* returns the collection of all snapshots after a snapshot.

```
context LocalSnapshot
def: let allPredecessors() : Sequence(LocalSnapshot) =
         if pred->notEmpty then
            pred->union(pred.allPredecessors())
         else
            Sequence {}
         endif
```

```
def: let allSuccessors() : Sequence(LocalSnapshot) =
         if succ->notEmpty then
            succ->union(succ.allSuccessors())
         else
            Sequence {}
```
endif

ObjectValue

[1] The operation *getCurrentValueOf* results in the value that is bound to the *name* parameter in the latest snapshot in the history of an object value. Note that the value may be the UndefinedValue.

```
context ObjectValue::getCurrentValueOf(n: String): Value
pre: -- none
post: result = history->last().bindings->any(name = n).value
```

[2] The operation *outgoingMessages* results in the sequence of *OclMessageValues* that have been in the output queue of the object between the last postcondition snapshot and its associated precondition snapshot.

```
context OclExpEval::outgoingMessages() : Sequence( OclMessageValue )
pre: -- none
post:
let end: LocalSnapshot =
      history->last().allPredecessors()->select( isPost = true )->first() in
let start: LocalSnapshot = end.pre  in
let inBetween: Sequence( LocalSnapshot ) =
      start.allSuccessors()->excluding( end.allSuccessors())->including( start ) in
         result = inBetween.outputQ->iterate (
         -- creating a sequence with all elements present once
         m : oclMessageValue;
         res: Sequence( OclMessageValue ) = Sequence{}
         |  if not res->includes( m )
            then res->append( m )
            else res
            endif )
```
endif

TupleValue

[1] The operation *getValueOf* results in the value that is bound to the *name* parameter in the tuple value.

```
context TupleValue::getValueOf(n: String): Value
pre: -- none
post: result = elements->any(name = n).value
```

5.2.4 Overview of the Values package

Figure 5-5 on page 5-10 shows an overview of the inheritance relationships between the classes in the Values package.

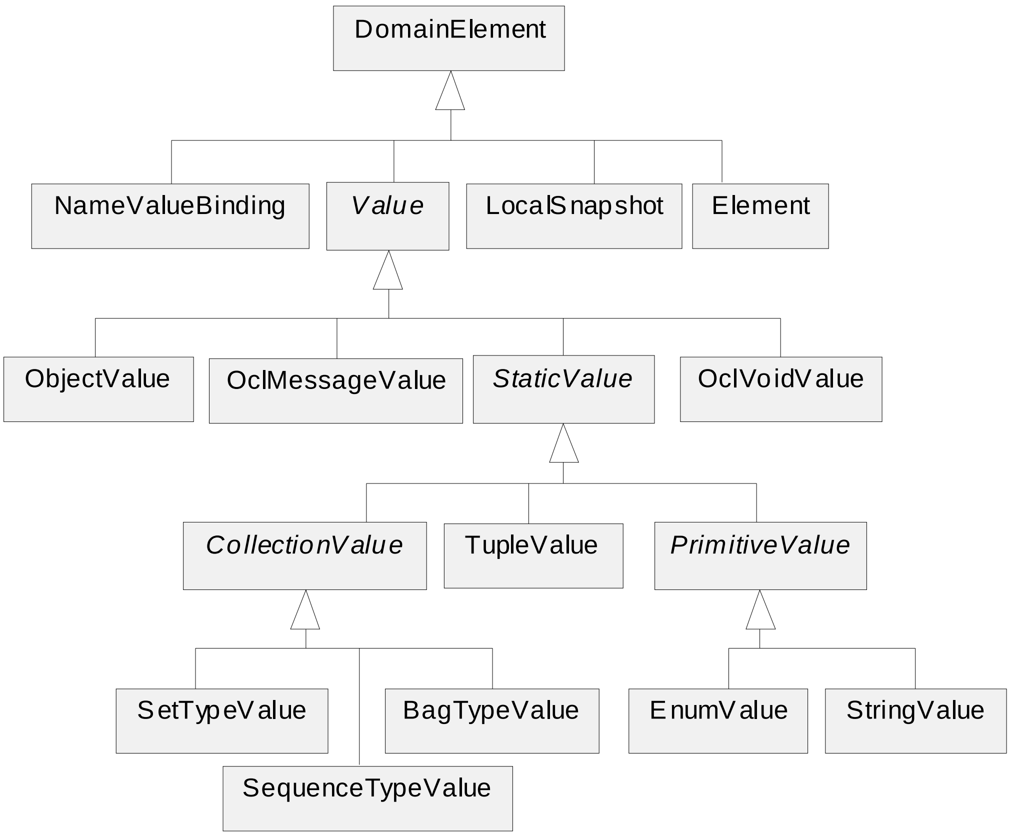


Figure 5-5 *The inheritance tree of classes in the Values package*

5.3 THE EVALUATIONS PACKAGE

This section defines the evaluations of OCL expressions. The evaluations package is a mirror image of the expressions package from the abstract syntax. Figure 5-6 on page 5-11 shows how the environment of an OCL expression evaluation is structured. The environment is determined by the placement of the expression within the UML model as discussed in chapter 7 (“The Use of Ocl Expressions in UML Models”). The calculation of the environment is done in the ExpressionInOclEval, which will be left undefined here.

Figure 5-7 on page 5-12 shows the core part of the Evaluations package. The basic elements in the package are the classes *OclEvaluation*, *PropertyCallExpEval* and *VariableExpEval*. An OclEvaluation always has a result value, and a name space that binds names to values. In figure 5-8 on page 5-15 the various subtypes of model propertycall evaluation are defined.

Most of the OCL expressions can be simply *evaluated*, i.e. their value can be determined based on a non-changing set of name value bindings. Operation call expressions, however, need the *execution* of the called operation. The semantics of the execution of an operation will be defined in the UML infrastructure. For our purposes it is enough to assume that an operation execution will add to the environment of an OCL expression the name ‘result’ bound to a certain value. In order not to become tangled in a mix of terms, the term *evaluation* is used in the following to denote both the ‘normal’ OCL evaluations and the executions of operation call expressions.

In sections 5.3.2 (“Model PropertyCall Evaluations”) to 5.3.6 (“Let expressions”) special subclasses of *OclExpEval* will be defined.

5.3.1 Definitions of concepts for the Evaluations package

The section lists the definitions of concepts in the Evaluations package in alphabetical order.

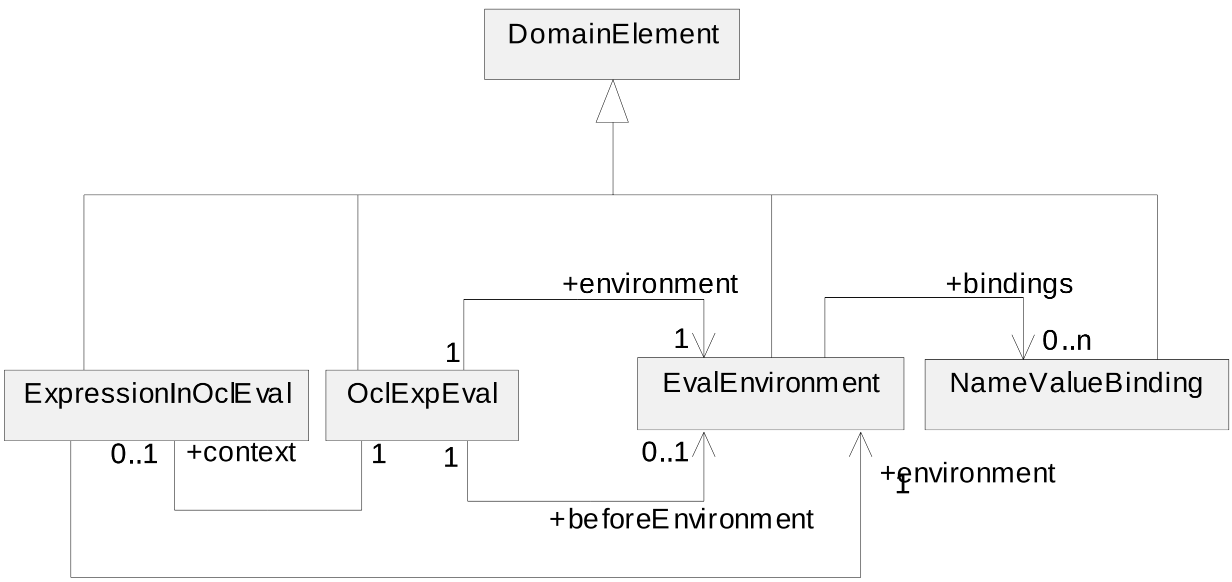


Figure 5-6 *The environment for ocl evaluations*

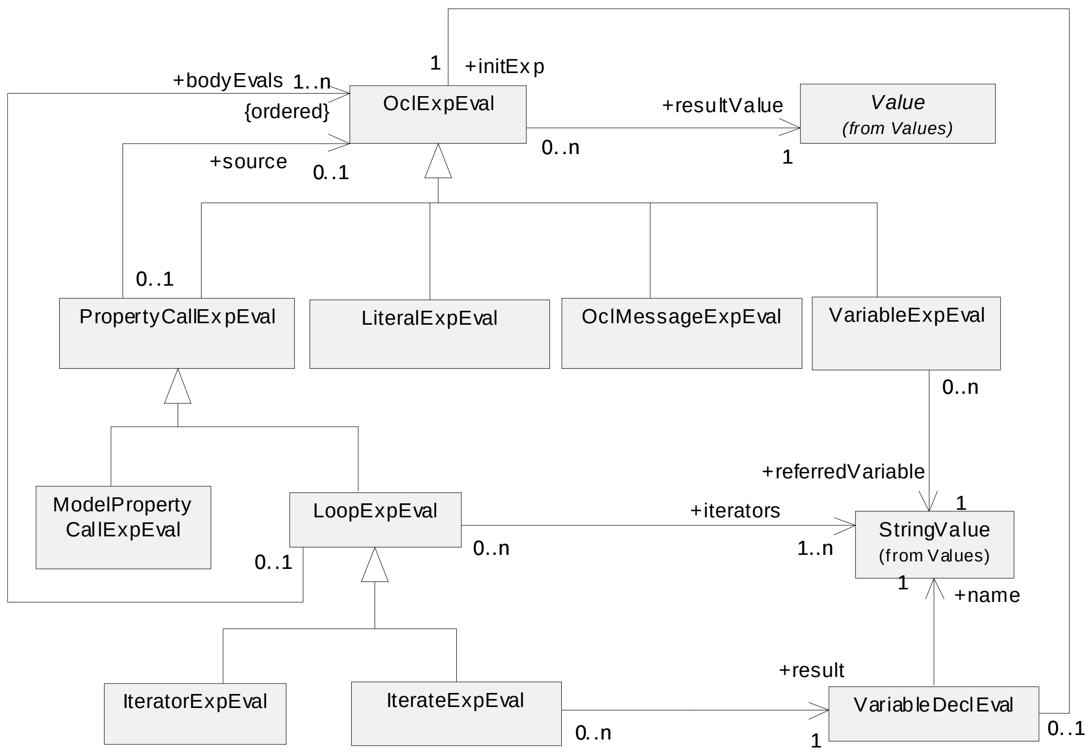


Figure 5-7 *Domain model for ocl evaluations*

EvalEnvironment

A EvalEnvironment is a set of NameValueBindings that form the environment in which an OCL expression is evaluated. A EvalEnvironment has three operations which are defined in section 5.3.8 (“Additional operations of the Evaluations package”).

Associations

bindings — The *NameValueBindings* that are the elements of this name space.

IterateExpEval

An *IterateExpEval* is an expression evaluation which evaluates its *body* expression for each element of a collection value, and accumulates a value in a *result* variable. It evaluates an IterateExp.

IteratorExpEval

An *IteratorExp* is an expression evaluation which evaluates its *body* expression for each element of a collection.

ExpressionInOclEval

An *ExpressionInOclEval* is an evaluation of the context of an OCL expression. It is the counterpart in the domain of the ExpressionInOcl metaclass defined in chapter 7 (“The Use of Ocl Expressions in UML Models”). It is merely included here to be able to determine the environment of an OCL expression.

LiteralExpEval

A Literal expression evaluation is an evaluation of a Literal expression.

LoopExpEval

A loop expression evaluation is an evaluation of a Loop expression.

Associations

bodyEvals	The *oclExpEvaluations* that represent the evaluation of the body expression for each element in the source collection.
iterators	The names of the iterator variables in the loop expression.

ModelPropertyCallExpEval

A model property call expression evaluation is an evaluation of a ModelPropertyCallExp. In figure 5-8 on page 5-15 the various subclasses of *ModelPropertyCallExpEval* are shown.

Operations

atPre	The *atPre* operation returns true if the property call is marked as being evaluated at precondition time.

OclExpEval

An ocl expression evaluation is an evaluation of an *OclExpression*. It has a result value, and it is associated with a set of name-value bindings, called *environment*. These bindings represent the values that are visible for this evaluation, and the names by which they can be referenced. A second set of name-value bindings is used to evaluate any sub expression for which the operation *atPre* returns true, called *beforeEnvironment*.

Note that as explained in chapters 4 ("Concrete Syntax") and 7 ("The Use of Ocl Expressions in UML Models"), these bindings need to be established, based on the placement of the OCL expression within the UML model. A binding for an invariant will not need the beforeEnvironment, and it will be different from a binding of the same expression when used as precondition.

Associations

environment	The set of name value bindings that is the context for this evaluation of an ocl expression.
beforeEnvironment	The set of name value bindings at the precondition time of an operation, to evaluate any sub expressions of type *ModelPropertyCallExp* for which the operation *atPre* returns true.
resultValue	The value that is the result of evaluating the *OclExpression*.

OclMessageExpEval

An ocl message expression evaluation is defined in section 5.3.4 ("Ocl Message Expression Evaluations"), but included in this diagram for completeness.

PropertyCallExpEval

A property call expression evaluation is an evaluation of a *PropertyCallExp*.

Associations

source	The result value of the source expression evaluation is the instance that performs the property call.

VariableDeclEval

A variable declaration evaluation represents the evaluation of a variable declaration. Note that this is not a subtype of OclExpEval, therefore it has no resultValue.

Associations

name	The name of the variable.
initExp	The value that will be initially bound to the name of this evaluation.

VariableExpEval

A variable expression evaluation is an evaluation of a *VariableExp*, which in effect is the search of the value that is bound to the variable name within the environment of the expression.

Associations

variable	The name that refers to the value that is the result of this evaluation.

5.3.2 Model PropertyCall Evaluations

The subtypes of *ModelPropertyCallExpEval* are shown in figure 5-8, and are defined in this section in alphabetical order.

AssociationClassCallExpEval

An association end call expression evaluation is an evaluation of a *AssociationClassCallExp*, which in effect is the search of the value that is bound to the associationClass name within the expression environment.

Associations

referredAssociationClass	The name of the *AssociationClass* to which the corresponding *AssociationClassCallExp* is a reference.

AssociationEndCallExpEval

An association end call expression evaluation is an evaluation of a *AssociationEndCallExp*, which in effect is the search of the value that is bound to the associationEnd name within the expression environment.

Associations

referredAssociationEnd	The name of the *AssociationEnd* to which the corresponding *NavigationCallExp* is a reference.

AttributeCallExpEval

An attribute call expression evaluation is an evaluation of an *AttributeCallExp*, which in effect is the search of the value that is bound to the attribute name within the expression environment.

Associations

referredAttribute	The name of the *Attribute* to which the corresponding *AttributeCallExp* is a reference.

NavigationCallExpEval

A navigation call expression evaluation is an evaluation of a *NavigationCallExp*.

Associations

navigationSource	The name of the *AssociationEnd* of which the corresponding *NavigationCallExp* is the source.

OperationCallExp

An operation call expression evaluation is an evaluation of an *OperationCallExp*.

Associations

arguments	The arguments denote the arguments to the operation call. This is only useful when the operation call is related to an *Operation* that takes parameters.
referredOperation	The name of the *Operation* to which this *OperationCallExp* is a reference. This is an *Operation* of a *Classifier* that is defined in the UML model.

5.3.3 If Expression Evaluations

If expression evaluations are shown in figure 5-9, and defined in this section.

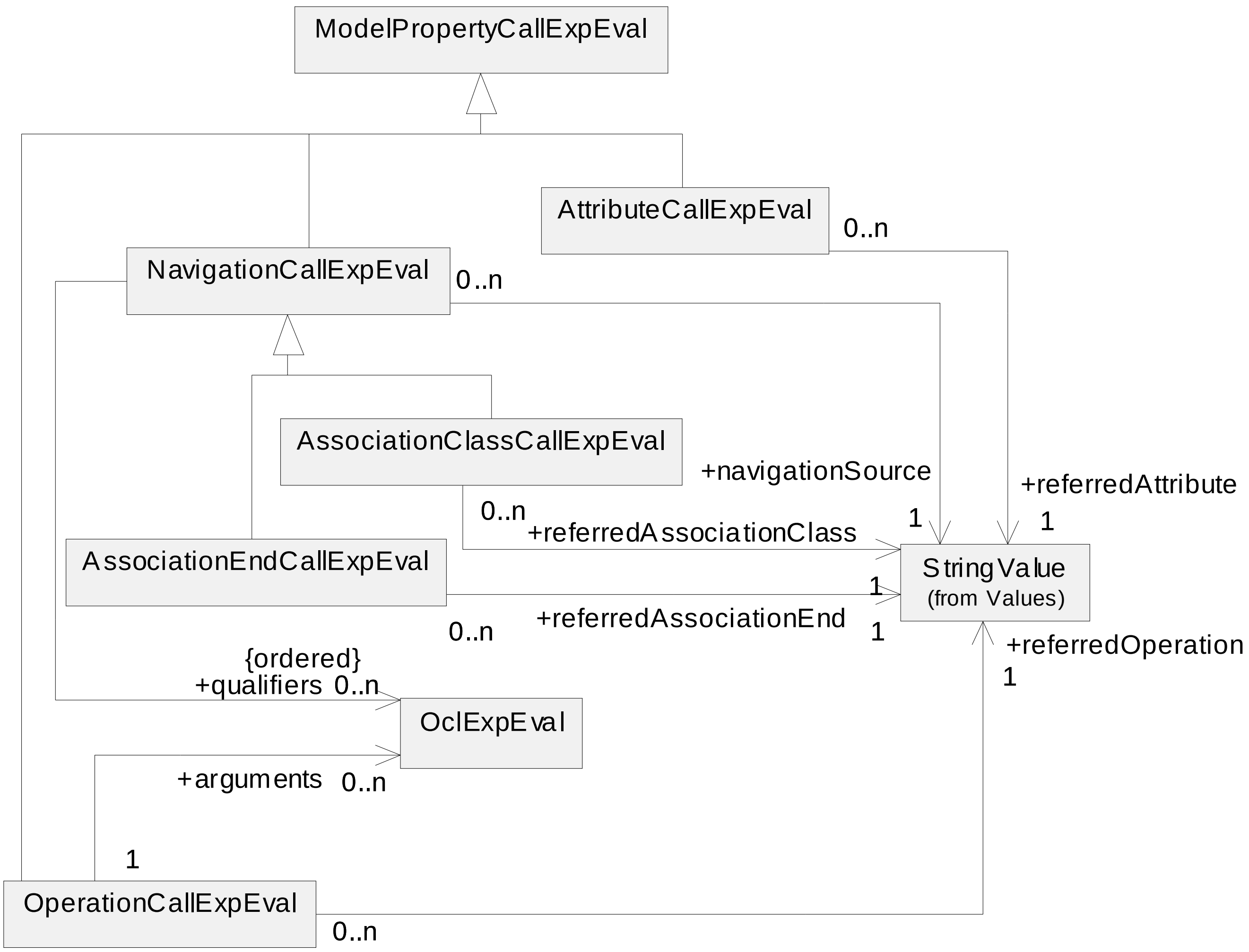


Figure 5-8 *Domain model for ModelPropertyCallExpEval and subtypes*

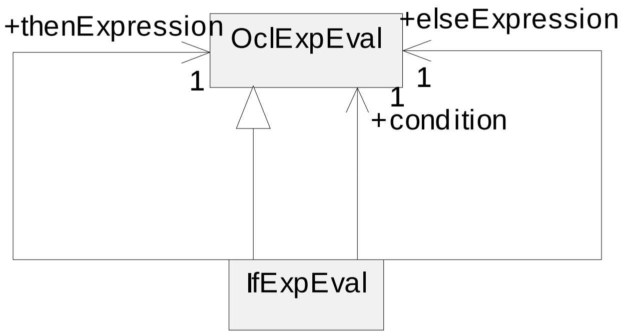


Figure 5-9 *Domain model for if expression*

IfExpEval

An IfExpEval is an evaluation of an IfExp.

Associations

condition	The OclExpEval that evaluates the condition of the corresponding IfExpression.
thenExpression	The OclExpEval that evaluates the thenExpression of the corresponding IfExpression.
elseExpression	The OclExpEval that evaluates the elseExpression of the corresponding IfExpression.

5.3.4 Ocl Message Expression Evaluations

Ocl message expressions are used to specify the fact that an object has, or will sent some message to another object at a some moment in time. Ocl message expresssion evaluations are shown in figure 5-10 on page 5-16, and defined in this section.

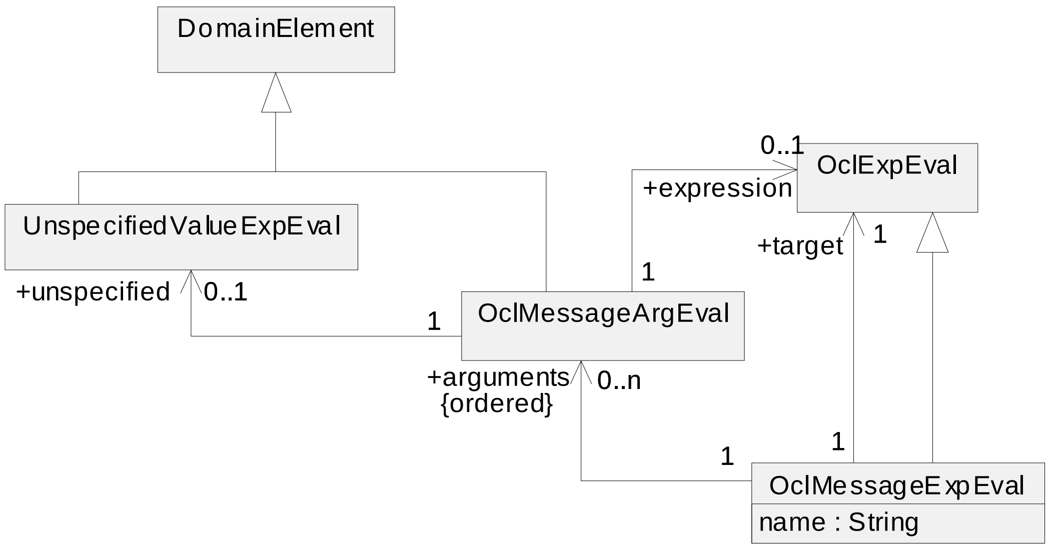


Figure 5-10 *Domain model for message evaluation*

OclMessageArgEval

An ocl message argument evaluation is an evaluation of a *OclMessageArg.* It represents the evaluation of the actual parameters to the *Operation* or *Signal*. An argument of a message expression is either an ocl expression, or a variable declaration.

Associations

variable	The *OclExpEval* that represents the evaluation of the argument, in case the argument is a *VariableDeclaration*.
expression	The *OclExpEval* that represents the evaluation of the argument, in case the argument is an *OclExpression*.

OclMessageExpEval

An ocl message expression evaluation is an evaluation of a *OclMessageExp.* As explained in [Kleppe2000] the only demand we can put on the ocl message expression is that the OclMessageValue it represents (either an operation call, or a UML signal), has been at some time between 'now' and a reference point in time in the output queue of the sending instance. The 'now' timepoint is the point in time at which this evaluation is performed. This point is represented by the *environment* link of the *OclMessageExpEval* (inherited from *OclExpEval*).

Associations

target	The *OclExpEval* that represents the evaluation of the target instance or instances on which the action is perfomed.
arguments	The *OclMessageArgEvals* that represent the evaluation of the actual parameters to the *Operation* or *Message*.

UnspecifiedValueExpEval

An unspecified value expression evaluation is an evaluation of an UnSpecifiedValueExp. It results in a randomly picked instance of the type of the expression.

5.3.5 Literal Expression Evaluations

This section defines the different types of literal expression evaluations in OCL, as shown in figure 5-11 on page 5-18. Again it is a complete mirror image of the abstract syntax.

BooleanLiteralExpEval

A boolean literal expression evaluation represents the evaluation of a boolean literal expression.

CollectionItemEval

A collection item evaluation represents the evaluation of a collection item.

CollectionLiteralExpEval

A collection literal expression evaluation represents the evaluation of a collection literal expression.

CollectionLiteralPartEval

A collection literal part evaluation represents the evaluation of a collection literal part.

CollectionRangeEval
A collection range evaluation represents the evaluation of a collection range.

EnumLiteralExpEval
An enumeration literal expression evaluation represents the evaluation of an enumeration literal expression.

IntegerLiteralExpEval
A integer literal expression evaluation represents the evaluation of a integer literal expression.

NumericLiteralExpEval
A numeric literal expression evaluation represents the evaluation of a numeric literal expression.

PrimitiveLiteralExpEval
A primitive literal expression evaluation represents the evaluation of a primitive literal expression.

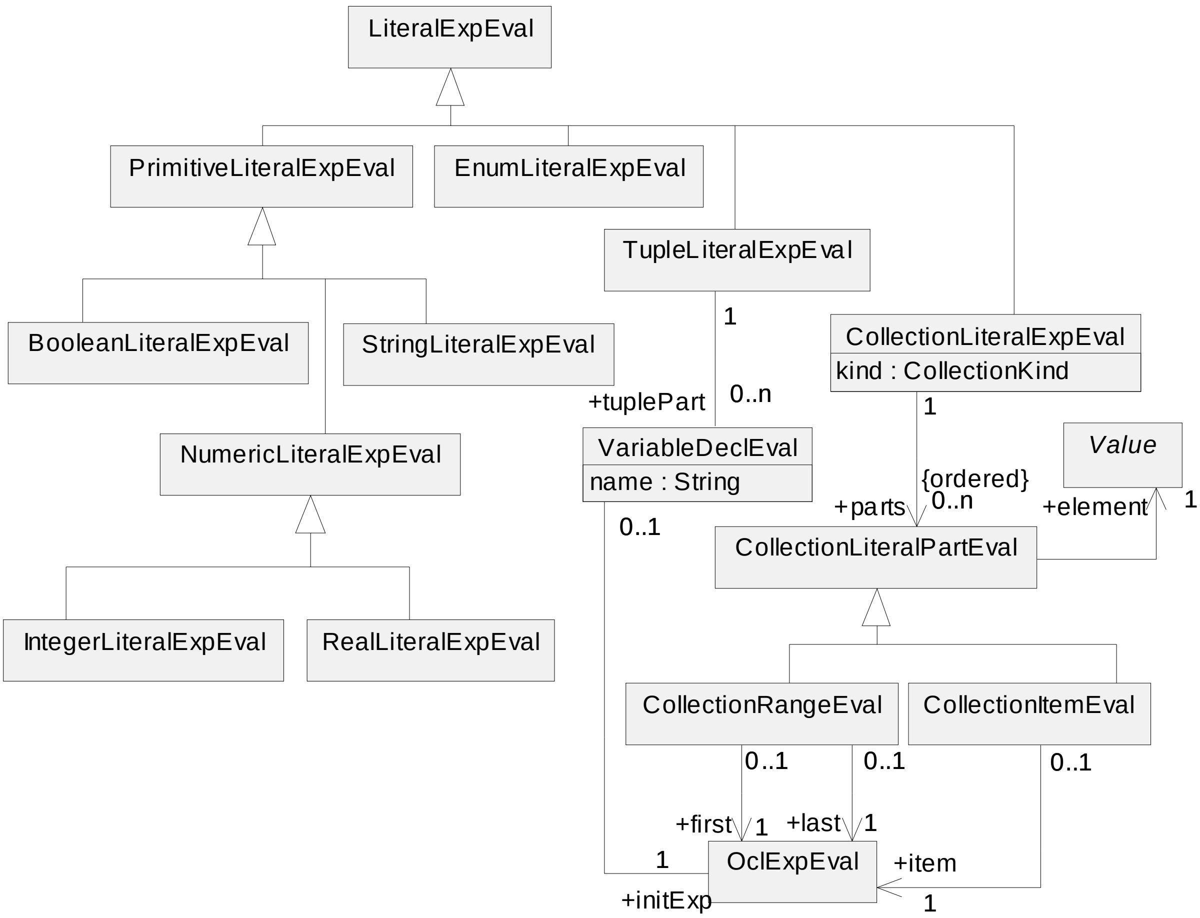


Figure 5-11 *Domain model for literal expressions*

RealLiteralExpEval

A real literal expression evaluation represents the evaluation of a real literal expression.

StringLiteralExpEval

A string literal expression evaluation represents the evaluation of a string literal expression.

TupleLiteralExpEval

A tuple literal expression evaluation represents the evaluation of a tuple literal expression.

TupleLiteralExpPartEval

A tuple literal expression part evaluation represents the evaluation of a tuple literal expression part.

5.3.6 Let expressions

Let expressions define new variables. The structure of the let expression evaluation is shown in figure 5-12 on

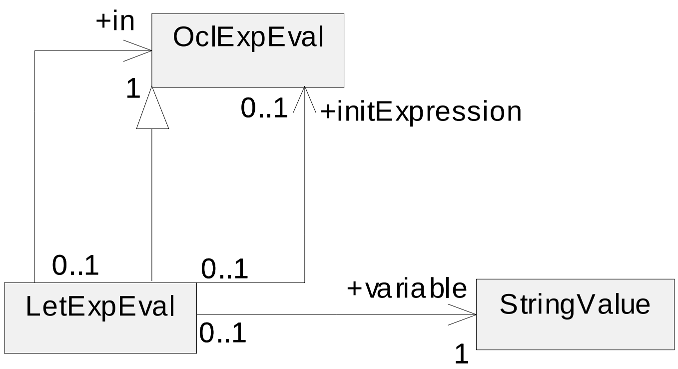


Figure 5-12 *Domain model for let expression*

page 5-19.

LetExpEval

A Let expression evaluation is an evaluation of a Let expression that defines a new variable with an initial value. A Let expression evaluation changes the environment of the *in* expression evaluation.

Associations

variable	The name of the variable that is defined.
in	The expression in whose environment the defined variable is visible.
initExpression	The expression that represents the initial value of the defined variable.

5.3.7 Well-formedness Rules of the Evaluations package

The metaclasses defined in the evaluations package have the following well-formednes rules. These rules state how the result value is determined. This defines the semantics of the OCL expressions.

AssociationClassCallExpEval

[1] The result value of an association class call expression is the value bound to the name of the association class to which it refers. Note that the determination of the result value when qualifiers are present is specified in section 5.4.3 (“Well-formedness rules for the AS-Domain-Mapping.exp-eval Package”). The operation getCurrentValueOf is an operation defined on ObjectValue in section 5.2.3 (“Additional operations for the Values Package”).

```
context AssociationClassCallExpEval inv:
qualifiers->size = 0 implies
     resultValue =
              source.resultValue.getCurrentValueOf(referredAssociationClass.name)
```

AssociationEndCallExpEval

[1] The result value of an association end call expression is the value bound to the name of the attribute to which it refers. Note that the determination of the result value when qualifiers are present is specified in section 5.4.3 (“Well-formedness rules for the AS-Domain-Mapping.exp-eval Package”).

```
context  AssociationEndCallExpEval inv:
qualifiers->size = 0 implies
     resultValue =
              source.resultValue.getCurrentValueOf(referredAssociationEnd.name)
```

AttributeCallExpEval

[1] The result value of an attribute call expression is the value bound to the name of the attribute to which it refers.

```
context AttributeCallExpEval inv:
resultValue = if source.resultValue->isOclType( ObjectValue) then
                 source.resultValue->asOclType( ObjectValue )
                                   .getCurrentValueOf(referredAttribute.name)
              else -- must be a tuple value
                 source.resultValue->asOclType( TupleValue )
                                   .getValueOf(referredAttribute.name)
              endif
```

BooleanLiteralExpEval

No extra well-formedness rules. The manner in which the resultValue is determined is given in section 5.4.3 (“Well-formedness rules for the AS-Domain-Mapping.exp-eval Package”).

CollectionItemEval

[1] The value of a collection item is the result value of its *item* expression. The environment of this *item* expression is equal to the environment of the collection item evaluation.

```
context CollectionItemEval
inv: element = item.resultValue
inv: item.environment = self.environment
```

CollectionLiteralExpEval

[1] The environment of its parts is equal to the environment of the collection literal expression evaluation.

```
context CollectionLiteralExpEval
inv: parts->forAll( p | p.environment = self.environment )
```

[2] The result value of a collection literal expression evaluation is a collection literal value, or one of its subtypes.

```
context CollectionLiteralExpEval inv:
resultValue.isOclKind( CollectionValue )
```

[3] The number of elements in the result value is equal to the number of elements in the collection literal parts, taking into account that a collection range can result in many elements.

```
context CollectionLiteralExpEval inv:
resultValue.elements->size() = parts->collect( element )->size()->sum()
```

[4] The elements in the result value are the elements in the collection literal parts, taking into account that a collection range can result in many elements.

```
context CollectionLiteralExpEval inv:
let allElements = parts->collect( element )->flatten() in
    Sequence{1..allElements->size()}->forAll( i: Integer |
                resultValue.elements->at(i).name = '' and
                resultValue.elements->at(i).value = allElements->at(i) and
                self.kind = CollectionKind::Sequence implies
                     resultValue.elements->at(i).indexNr = i )
```

CollectionLiteralPartEval

No extra well-formedness rules. The manner in which its value is determined is given by its subtypes.

CollectionRangeEval

[1] The value of a collection range is the range of integer numbers between the result value of its *first* expression and its *last* expression.

```
context CollectionRangeEval
inv: element.isOclType( Sequence(Integer) ) and
     element = getRange( first->asOclType(Integer), last->asOclType(Integer) )
```

EnumLiteralExpEval

No extra well-formedness rules.

EvalEnvironment

[1] All names in a name space must be unique.

```
context EvalEnvironment inv:
bindings->collect(name)->forAll( name: String | bindings->collect(name)->isUnique(name))
```

ExpressionInOclEval

No extra well-formedness rules.

IfExpEval

[1] The result value of an if expression is the result of the thenExpression if the condition is true, else it is the result of the elseExpression.

```
context IfExpEval inv:
resultValue = if condition then thenExpression.resultValue else elseExpression.resultValue
```

[2] The environment of the condition, thenExpression and elseExpression are both equal to the environment of the if expression.

```
context IfExpEval
inv: condition.environment = environment
inv: thenExpression.environment = environment
inv: elseExpression.environment = environment
```

IntegerLiteralExpEval

No extra well-formedness rules. The manner in which the resultValue is determined is given in section 5.4.3 (“Well-formedness rules for the AS-Domain-Mapping.exp-eval Package”).

IterateExpEval

[1] All sub evaluations have a different environment. The first sub evaluation will start with an environment in which all iterator variables are bound to the first element of the source, plus the result variable which is bound to the init expression of the variable declaration in which it is defined.

```
context IterateExpEval
inv: let bindings: Sequence( NameValueBindings ) =
         iterators->collect( i |
              NameValueBinding( i.varName, source->asSequence()->first() )
     in
        bodyEvals->at(1).environment = self.environment->addAll( bindings )
               ->add( NameValueBinding( result.name, result.initExp.resultValue ))
```

[2] The environment of any sub evaluation is the same environment as the one from its previous sub evaluation, taking into account the bindings of the iterator variables, plus the result variable which is bound to the result value of the last sub evaluation.

```
inv: let SS: Integer = source.value->size()
in if iterators->size() = 1 then
         Sequence{2..SS}->forAll( i: Integer |
             bodyEvals->at(i).environment = bodyEvals->at(i-1).environment
                   ->replace( NameValueBinding( iterators->at(1).varName,
                                                source.value->asSequence()->at(i)))
                   ->replace( NameValueBinding( result.varName,
                                                bodyEvals->at(i-1).resultValue )))
   else -- iterators->size() = 2
         Sequence{2..SS*SS}->forAll( i: Integer |
              bodyEvals->at(i).environment = bodyEvals->at(i-1).environment
                   ->replace( NameValueBinding( iterators->at(1).varName,
                         source->asSequence()->at(i.div(SS) + 1) ))
                   ->replace( NameValueBinding( iterators->at(2).varName,
                         source.value->asSequence()->at(i.mod(SS))))
                   ->replace( NameValueBinding( result.varName,
                                                bodyEvals->at(i-1).resultValue )))
   endif
```

[3] The result value of an IteratorExpEval is the result of the last of its body evaluations.

```
context IteratorExpEval
inv: resultValue = bodyEvals->last().resultValue
```

IteratorExpEval

The *IteratorExp* in the abstract syntax is merely a placeholder for the occurence of one of the predefined iterator expressions in the standard library (see chapter 6 (“The OCL Standard Library”)). These predefined iterator expressions are all defined in terms of an iterate expression. The semantics defined for the iterate expression are sufficient to define the iterator expression. No well-formedness rules for IteratorExpEval are defined.

LetExpEval

[1] A let expression results in the value of its *in* expression.

```
context LetExpEval inv:
resultValue = in.resultValue
```

[2] A let expression evaluation adds a name value binding that binds the *variable* to the value of its *initExpression,* to the environment of its *in* expression.

```
context LetExpEval
inv: in.environment = self.environment
   ->add( NameValueBinding( variable.varName, variable.initExpression.resultValue ))
```

[3] The environment of the *initExpression* is equal to the environment of this Let expression evaluation.

```
context LetExpEval
inv: initExpression.environment = self.environment
```

LiteralExpEval

No extra well-formedness rules.

LoopExpEval

The result value of a loop expression evaluation is determined by its subtypes.

[1] There is an OclExpEval (a sub evaluation) for combination of values for the iterator variables. Each iterator variable will run through every element of the source collection.

```
context LoopExpEval
inv: bodyEvals->size() =
                   if iterators->size() = 1 then
                         source.value->size()
                   else -- iterators->size() = 2
                         source.value->size() * source.value->size()
                   endif
```

[2] All sub evaluations (in the sequence *bodyEvals*) have a different environment. The first sub evaluation will start with an environment in which all iterator variables are bound to the first element of the source. Note that this is an arbitrary choice, one could easily well start with the last element of the source, or any other combination.

```
context LoopExpEval
inv: let bindings: Sequence( NameValueBindings ) =
         iterators->collect( i |
              NameValueBinding( i.varName, source->asSequence()->first() )
     in
        bodyEvals->at(1).environment = self.environment->addAll( bindings )
```

[3] All sub evaluations (in the sequence *bodyEvals*) have a different environment. The environment is the same environment as the one from the previous bodyEval, where the iterator variable or variables are bound to the subsequent elements of the source.

```
context LoopExpEval
inv:
let SS: Integer = source.value->size()
in if iterators->size() = 1 then
         Sequence{2..SS}->forAll( i: Integer |
             bodyEvals->at(i).environment = bodyEvals->at(i-1).environment
                   ->replace( NameValueBinding( iterators->at(1).varName,
                        source.value->asSequence()->at(i) )))
   else -- iterators->size() = 2
         Sequence{2..SS*SS}->forAll( i: Integer |
```

```
            bodyEvals->at(i).environment = bodyEvals->at(i-1).environment
                 ->replace( NameValueBinding( iterators->at(1).varName,
                      source->asSequence()->at(i.div(SS) + 1) ))
                 ->replace( NameValueBinding( iterators->at(2).varName,
                      source.value->asSequence()->at(i.mod(SS)) )) ) ))
    endif
```

ModelPropertyCallExpEval

Result value is determined by its subtypes.

[1] The environment of an ModelPropertyCall expression is equal to the environment of its source.

```
context ModelPropertyCallExpEval inv:
environment = source.environment
```

NavigationCallExpEval

[1] When the navigation call expression has qualifiers, the result value is limited to those elements for which the qualifier value equals the value of the attribute.

```
-- To be done.
```

NumericLiteralExpEval

No extra well-formedness rules. Result value is determined by its subtypes.

OclExpEval

The result value of an ocl expression is determined by its subtypes.

[1] The environment of an OclExpEval is determined by its context, i.e. the ExpressionInOclEval.

```
context OclExpEval
inv: environment = context.environment
```

[2] Every OclExpEval has an environment in which at most one self instance is known.

```
context OclExpEval
inv: environment->select( name = 'self' )->size() = 1
```

OclMessageExpEval

[1] The result value of an ocl message expression is an ocl message value.

```
context OclMessageExpEval
inv: resultValue->isTypeOf( OclMessageValue )
```

[2] The result value of an ocl message expression is the sequence of the outgoing messages of the 'self' object that matches the expression. Note that this may result in an empty sequence when the expression does not match to any of the outgoing messages.

```
context OclMessageExpEval
inv: resultValue =
        environment.getValueOf( 'self' ).outgoingMessages->select( m |
             m.target = target.resultValue and
             m.name = self.name and
             self.arguments->forAll( expArg: OclMessageArgEval |
                not expArg.resultValue.oclIsUndefined() implies
                   m.arguments->exists( messArg | messArg.value = expArg.value ))
```

[3] The source of the resulting ocl message value is equal to the 'self' object of the ocl message expression.

```
context OclMessageExpEval
inv: resultValue.source = environment.getValueOf( 'self' )
```

[4] The isSent attribute of the resulting ocl message value is true only if the message value is in the outgoing messages of the 'self' object.

```
context OclMessageExpEval
inv:
if resultValue.oclIsUndefined()
       resultValue.isSent = false
else
       resultValue.isSent = true
endif
```

[5] The target of an ocl message expression is an object value.

```
context OclMessageExpEval
inv: target.resultValue->isTypeOf( ObjectValue )
```

[6] The environment of all arguments, and the environment of the target expression are equal to the environment of this ocl message value.

```
context OclMessageExpEval
inv: arguments->forAll( a | a.environment = self.environment )
inv: target.environment = self.environment
```

OclMessageArgEval

[1] An ocl message argument evaluation has either an ocl expression evaluation, or an unspecified value expression evaluation, not both.

```
context OclMessageArgEval inv:
expression->size() = 1 implies unspecified->size() = 0
expression->size() = 0 implies unspecified->size() = 1
```

[2] The result value of an ocl message argument is determined by the result value of its expression, or its unspecified value expression.

```
context OclMessageArgEval inv:
if expression->size() = 1
then resultValue = expression.resultValue
else resultValue = unspecified.resultValue
endif
```

[3] The environment of the expression and unspecified value are equal to the environment of this ocl message argument.

```
context OclMessageArgEval
inv: expression.environment = self.environment
inv: unspecified.environment = self.environment
```

OperationCallExpEval

The definition of the semantics of the operation call expression depends on the definition of operation call execution in the UML semantics. This is part of the UML infrastructure specification, and will not be defined here. For the semantics of the OperationCallExp it suffices to know that the execution of an operation call will produce a result of the correct type. The latter will be specified in section 5.4 ("The AS-Domain-Mapping Package").

[1] The environments of the arguments of an operation call expression are equal to the environment of this call.

```
context  OperationCallExpEval inv:
arguments->forall( a | a.environment = self.environment )
```

PropertyCallExpEval

The result value and environment are determined by its subtypes.

[1] The environment of the source of an property call expression is equal to the environment of this call.

```
context  PropertyCallExpEval inv:
source.environment = self.environment
```

PrimitiveLiteralExpEval

No extra well-formedness rules. The result value is determined by its subtypes.

RealLiteralExpEval

No extra well-formedness rules. The manner in which the resultValue is determined is given in section 5.4.3 (“Well-formedness rules for the AS-Domain-Mapping.exp-eval Package”).

StringLiteralExpEval

No extra well-formedness rules. The manner in which the resultValue is determined is given in section 5.4.3 (“Well-formedness rules for the AS-Domain-Mapping.exp-eval Package”).

TupleLiteralExpEval

[1] The result value of a tuple literal expression evaluation is a tuple value whose elements correspond to the parts of the tuple literal expression evaluation.

```
context TupleLiteralExpEval inv:
resultValue.isOclType( TupleValue ) and
tuplePart->size() = resultValue.elements->size() and
Sequence{1..tuplePart->size()}->forAll( i: Integer |
                 resultValue.elements->at(i).name = tuplePart.name and
                resultValue.elements->at(i).value = tuplePart.initExpression.resultValue )
```

UnspecifiedValueExpEval

The result of an unspecified value expression is a randomly picked instance of the type of the expression. This rule will be defined in 5.4.3 (“Well-formedness rules for the AS-Domain-Mapping.exp-eval Package”).

VariableDeclEval

No extra well-formedness rules.

VariableExpEval

[1] The result of a VariableExpEval is the value bound to the name of the variable to which it refers.

```
context VariableExpEval inv:
resultValue = environment.getValueOf(referredVariable.varName)
```

5.3.8 Additional operations of the Evaluations package

EvalEnvironment

[1] The operation *getValueOf* results in the value that is bound to the *name* parameter in the bindings of a name space. Note that the value may be the UndefinedValue.

```
context EvalEnvironment::getValueOf(n: String): Value
pre: -- none
post: result = bindings->any(name = n).value
```

[2] The operation *replace* replaces the value of a name, by the value given in the *nvb* parameter.

```
context EvalEnvironment::replace(nvb: NameValueBinding): EvalEnvironment
pre: -- none
post: result.bindings = self.bindings
         ->excluding( self.bindings->any( name = nvb.name) )->including( nvb )
```

[3] The operation *add* adds the name and value indicated by the NameValueBinding given by the *nvb* parameter.

```
context EvalEnvironment::add(nvb: NameValueBinding): EvalEnvironment
pre: -- none
post: result.bindings = self.bindings->including( nvb )
```

[4] The operation *addAll* adds all NameValueBindings in the *nvbs* parameter.

```
context EvalEnvironment::add(nvbs: Collection(NameValueBinding)): EvalEnvironment
pre: -- none
post: result.bindings = self.bindings->union( nvbs )
```

CollectionRangeEval

[1] The operation *getRange()* returns a sequence of integers that contains all integer in the collection range.

```
context CollectionRangeEval::getRange(first, last: Integer): Sequence(Integer)
pre: -- none
post: result = if first = last then
                  first->asSequence()
               else
                  first->asSequence()->union(getRange(first + 1, last))

              endif
```

5.3.9 Overview of the Values package

Figure 5-13 on page 5-28 shows an overview of the inheritance relationships between the classes in the Values package.

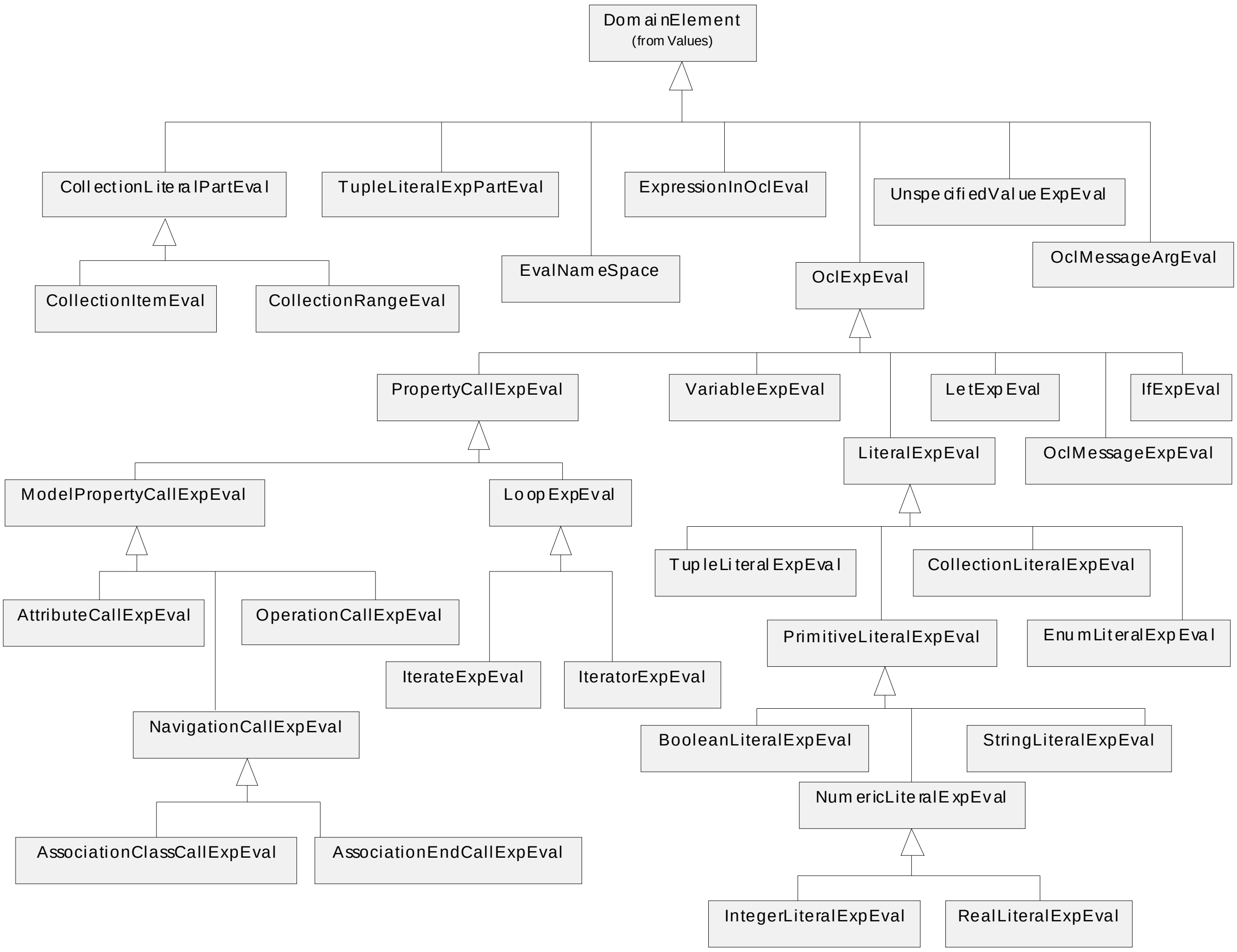


Figure 5-13 *The inheritance tree of classes in the Evaluations package*

5.4 THE AS-DOMAIN-MAPPING PACKAGE

The figures 5-15 on page 5-30 and 5-14 on page 5-29 show the associations between the abstract syntax concepts and the domain concepts defined in this chapter. Each domain concept has a counterpart called *model* in the abstract syntax. Each *model* has one or more instances in the semantic domain. Note that in particular every OCL expression can have more than one evaluation. Still every evaluation has only one value. For example, the "asSequence" applied to a Set may have n! evaluations, which each give a different permutation of the elements in the set, but each evaluation has exactly one result value.

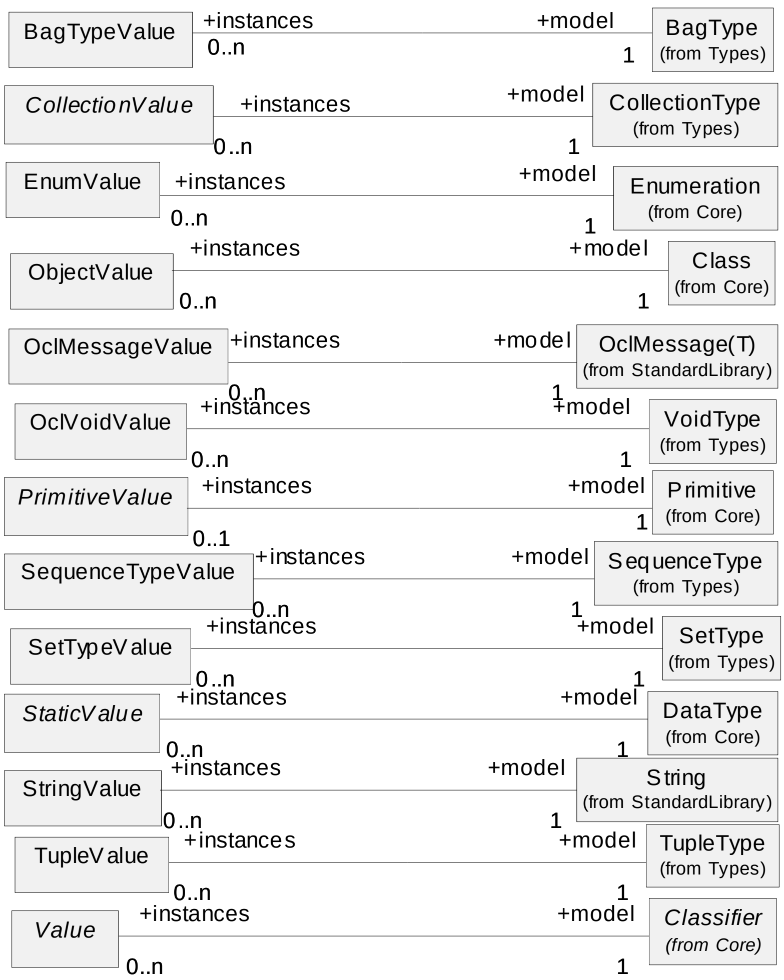


Figure 5-14 *Associations between values and the types defined in the abstract syntax.*

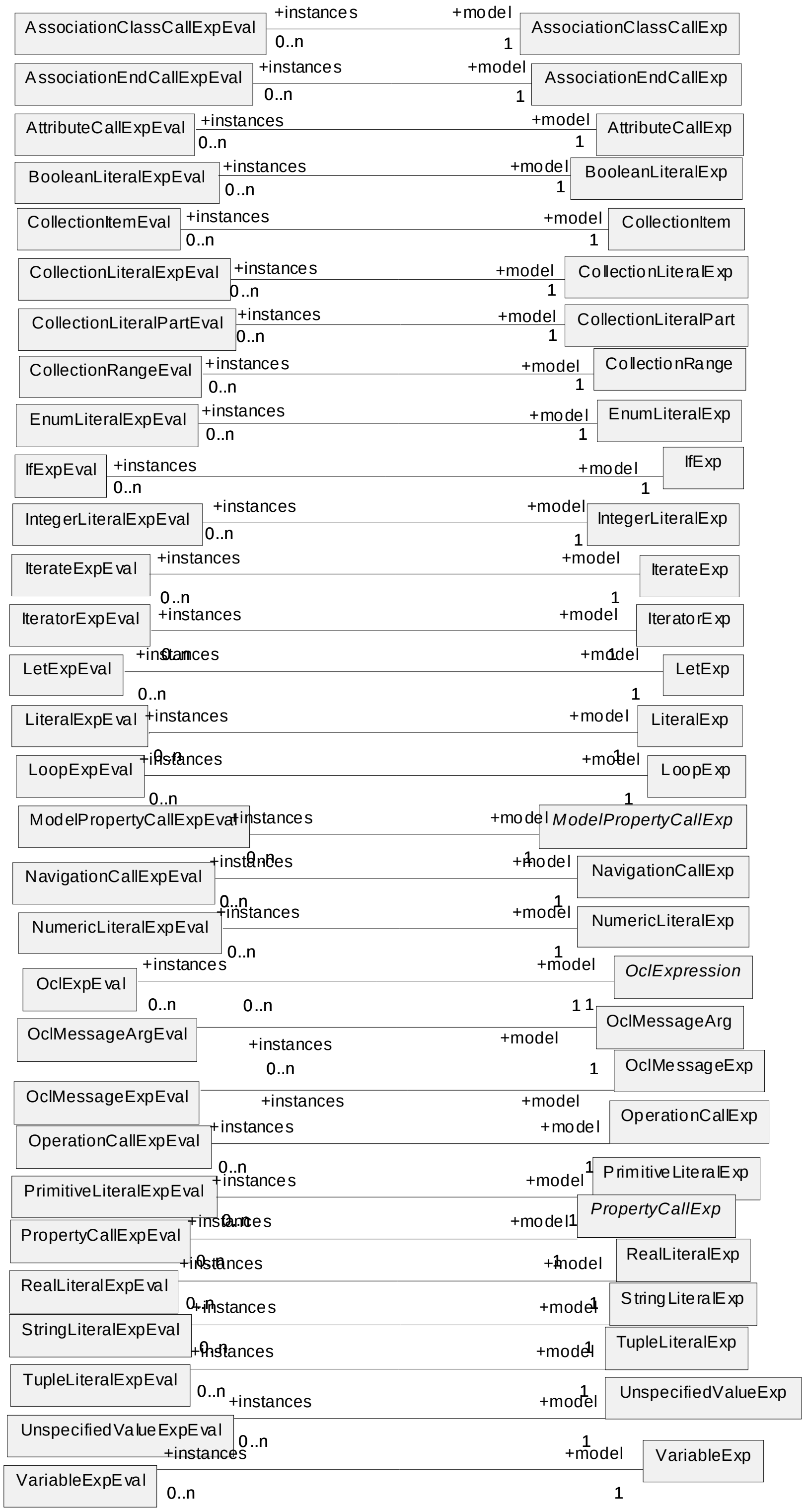


Figure 5-15 *Associations between evaluations and abstract syntax concepts*

5.4.1 Well-formedness rules for the AS-Domain-Mapping.type-value Package

CollectionValue

[1] All elements in a collection value must have a type that conforms to the elementType of its corresponding CollectionType.

```
context CollectionValue inv:
elements->forAll( e: Element | e.value.model.conformsTo( model.elementType ) )
```

DomainElement

No additional well-formedness rules.

Element

No additional well-formedness rules.

EnumValue

No additional well-formedness rules.

ObjectValue

[1] All bindings in an object value must correspond to attributes or associations defined in the object's Classifier.

```
context ObjectValue inv:
history->forAll( snapshot | snapshot.bindings->forAll( b |
           self.model.allAttributes()->exists (attr | b.name = attr.name)
           or
           self.model.allAssociationEnds()->exists ( role | b.name = role.name) ) )
```

OclMessageValue

No additional well-formedness rules.

PrimitiveValue

No additional well-formedness rules.

SequenceTypeValue

No additional well-formedness rules.

SetTypeValue

No additional well-formedness rules.

StaticValue

No additional well-formedness rules.

TupleValue

[1] The elements in a tuple value must have a type that conforms to the type of the corresponding tuple parts.

```
context TupleValue inv:
elements->forAll( elem |
   let correspondingPart: Attribute =
      self.model.allAttributes()->select( part | part.name = elem.name ) in
   elem.value.model.conformsTo( correspondingPart.type ) )
```

UndefinedValue

No additional well-formedness rules.

Value

No additional well-formedness rules.

5.4.2 Additional operations for the AS-Domain-Mapping.type-value Package

Value

[1] The additional operation *isInstanceOf* returns true if this value is an instance of the parameter classifier.

```
context Value::isInstanceOf( c: Classifier ): Boolean
pre: -- none
post: result = self.model.conformsTo( c )
```

5.4.3 Well-formedness rules for the AS-Domain-Mapping.exp-eval Package

AssociationClassCallExpEval

[1] The string that represents the referredAssociationClass in the evaluation must be equal to the name of the referredAssociationClass in the corresponding expression.

```
context AssociationClassCallExpEval inv:
referredAssociationClass = model.referredAssociationClass.name
```

[2] The result value of an association class call expression evaluation that has qualifiers, is determined according to the following rule. The ‘normal’ determination of result value is already given in section 5.3.7 (“Well-formedness Rules of the Evaluations package”).

```
let
-- the attributes that are the formal qualifiers. Because and association class has two or
-- more association ends, we must select the qualifiers from the other end(s), not from
-- the source of this expression. We allow only 2-ary associations.
   formalQualifiers : Sequence(Attribute) =
              self.model.referredAssociationClass.connection->any( c |
                                   c <> self.navigationSource).qualifier.asSequence() ,

-- the attributes of the class at the qualified end. Here we already assume that an
-- AssociationEnd will be owned by a Classifier, as will most likely be the case in the
-- UML 2.0 Infrastructure.
   objectAttributes: Sequence(Attribute) =
              self.model.referredAssociationClass.connection->any( c |
                     c <> self.navigationSource).owner.feature->select( f |
                           f.isOclType( Attribute ).asSequence() ,
```

```
-- the rolename of the qualified association end
   qualifiedEnd: String = self.model.referredAssociationClass.connection->any( c |
                    c <> self.navigationSource).name ,

-- the values for the qualifiers given in the ocl expression
   qualifierValues : Sequence( Value ) = self.qualifiers.asSequence()

-- the objects from which a subset must be selected through the qualifiers
   normalResult =
              source.resultValue.getCurrentValueOf(referredAssociationClass.name)

in
-- if name of attribute of object at qualified end equals name of formal qualifier then
-- if value of attribute of object at qualified end equals the value given in the exp
-- then select this object and put it in the resultValue of this expression.

qualifiers->size <> 0 implies
normalResult->select( obj |
     Sequence{1..formalQualifiers->size()}->forAll( i |
          objectAttributes->at(i).name = formalQualifiers->at(i).name and
          obj.qualifiedEnd.getCurrentValueOf( objectAttributes->at(i).name ) =
                                                    qualifiersValues->at(i) ))
```

AssociationEndCallExpEval

[1] The string that represents the referredAssociationEnd in the evaluation must be equal to the name of the referredAssociationEnd in the corresponding expression.

```
context AssociationEndCallExpEval inv:
referredAssociationEnd = model.referredAssociationEnd.name
```

[2] The result value of an association end call expression evaluation that has qualifiers, is determined according to the following rule. The 'normal' determination of result value is already given in section 5.3.7 ("Well-formedness Rules of the Evaluations package").

```
let
-- the attributes that are the formal qualifiers
   formalQualifiers : Sequence(Attribute) = self.model.referredAssociationEnd.qualifier ,

-- the attributes of the class at the qualified end
   objectAttributes: Sequence(Attribute) =
      (if self.resultValue.model.isOclKind( Collection ) implies
      then self.resultValue.model.oclAsType( Collection ).elementType->
                                   collect( feature->asOclType( Attribute ) )
      else self.resultValue.model->collect( feature->asOclType( Attribute ) )
      endif).asSequence() ,

-- the values for the qualifiers given in the ocl expression
   qualifierValues : Sequence( Value ) = self.qualifiers.asSequence()

-- the objects from which a subset must be selected through the qualifiers
   normalResult =
              source.resultValue.getCurrentValueOf(referredAssociationEnd.name)

in
-- if name of attribute of object at qualified end equals name of formal qualifier then
-- if value of attribute of object at qualified end equals the value given in the exp
-- then select this object and put it in the resultValue of this expression.

qualifiers->size <> 0 implies
normalResult->select( obj |
```

```
Sequence{1..formalQualifiers->size()}->forAll( i |
     objectAttributes->at(i).name = formalQualifiers->at(i).name and
     obj.getCurrentValueOf( objectAttributes->at(i).name ) =
                                            qualifiersValues->at(i) ))
```

AttributeCallExpEval

[1] The string that represents the referredAttribute in the evaluation must be equal to the name of the referredAttribute in the corresponding expression.

```
context AttributeCallExpEval inv:
referredAttribute = model.referredAttribute.name
```

BooleanLiteralExpEval

[1] The result value of a boolean literal expression is equal to the literal expression itself (‘true’ or ‘false’). Because the booleanSymbol attribute in the abstract syntax is of type Boolean as defined in the MOF, and resultValue is of type Primitive as defined in this chapter, a conversion is neccessary. For the moment, we assume the additional operation MOFbooleanToOCLboolean() exists. This will need to be re-examined when the MOF and/or UML Infrastructure submissions are finalised.

```
context BooleanLiteralExpEval inv:
resultValue = model.booleanSymbol.MOFbooleanToOCLboolean()
```

CollectionItemEval

No extra well-formedness rules.

CollectionLiteralExpEval

No extra well-formedness rules.

CollectionLiteralPartEval

No extra well-formedness rules.

CollectionRangeEval

No extra well-formedness rules.

EvalEnvironment

Because there is no mapping of name space to an abstract syntax concept, there are no extra well-formedness rules.

LiteralExpEval

No extra well-formedness rules.

LoopExpEval

No extra well-formedness rules.

EnumLiteralExpEval

[1] The result value of an EnumLiteralExpEval must be equal to one of the literals defined in its type.

```
context EnumLiteralExpEval inv:
model.type->includes( self.resultValue )
```

IfExpEval

[1] The condition evaluation corresponds with the condition of the expression, and likewise for the thenExpression and the else Expression.

```
context IfExpEval inv:
condition.model = model.condition
thenExpression.model = model.thenExpression
elseExpression.model = model.elseExpression
```

IntegerLiteralExpEval

```
context IntegerLiteralExpEval inv:
resultValue = model.integerSymbol
```

IterateExpEval

[1] The model of the result of an iterate expression evaluation is equal to the model of the result of the associated IterateExp.

```
context IterateExpEval
inv: result.model = model.result )
```

IteratorExpEval

No extra well-formedness rules.

LetExpEval

[1] All parts of a let expression evaluation correspond to the parts of its associated LetExp.

```
context LetExpEval inv:
in.model = model.in and
initExpression.model = model.initExpression and
variable = model.variable.varName
```

LoopExpEval

[1] All sub evaluations have the same model, which is the body of the associated LoopExp.

```
context LoopExpEval
inv: bodyEvals->forAll( model = self.model )
```

ModelPropertyCallExpEval

No extra well-formedness rules.

NumericLiteralExpEval

No extra well-formedness rules.

NavigationCallExpEval

[1] The string that represents the navigation source in the evaluation must be equal to the name of the navigationSource in the corresponding expression.

```
context NavigationCallExpEval inv:
navigationSource = model.navigationSource.name
```

[2] The qualifiers of a navigation call expression evaluation must correspond with the qualifiers of the associated expression.

```
context NavigationCallExpEval inv:
Sequence{1..qualifiers->size()}->forAll( i |
         qualifiers->at(i).model = model.qualifiers->at(i).type )
```

OclExpEval

[1] The result value of the evaluation of an ocl expression must be an instance of the type of that expression.

```
context OclExpEval
inv: resultValue.isInstanceOf( model.type )
```

OclMessageExpEval

[1] An ocl message expression evaluation must correspond with its message expression.

```
context OclMessageExpEval
inv: target.model = model.target
inv: Set{1..arguments->size()}->forall (i | arguments->at(i) = model.arguments->at(i) )
```

[2] The name of the resulting ocl message value must be equal to the name of the operation or signal indicated in the message expression.

```
context OclMessageExpEval inv:
if model.operation->size() = 1
then resultValue.name = model.operation.name
else resultValue.name = model.signal.name
endif
```

[3] The *isSignal*, *isSyncOperation*, and *isAsyncOperation* attributes of the result value of an ocl message expression evaluation must correspond to the operation indicated in the ocl message expression.

```
context OclMessageExpEval
inv:  if model.calledOperation->size() = 1
      then model.calledOperation.isAsynchronous = true implies
                resultValue.isAsyncOperation = true
      else -- message represents sending a signal
             resultValue.isSignal = true
      endif
```

[4] The arguments of an ocl message expression evaluation must correspond to the formal input parameters of the operation, or the attributes of the signal indicated in the ocl message expression.

```
context OclMessageExpEval
inv: model.calledOperation->size() = 1 implies
       Sequence{1.. arguments->size()} ->forAll( i |
          arguments->at(i).variable->size() = 1 implies
               model.calledOperation.operation.parameter->
                    select( kind = ParameterDirectionKind::in )->at(i).name =
                                                        arguments->at(i).variable
          and
          arguments->at(i).expression->size() = 1 implies
               model.calledOperation.operation.parameter->
                    select( kind = ParameterDirectionKind::in )at(i).type =
```

```
                                                    arguments->at(i).expression.model
inv: model.sentSignal->size() = 1 implies
       Sequence{1.. arguments->size()} ->forAll( i |
          arguments->at(i).variable->size() = 1 implies
               model.sentSignal.signal.feature->select(
                                              arguments->at(i).variable )->notEmpty()
          and
          arguments->at(i).expression->size() = 1 implies
               model.sentSignal.signal.feature.oclAsType(StructuralFeature).type =
                                              arguments->at(i).expression.model
```

[5] The arguments of the return message of an ocl message expression evaluation must correspond to the names given by the formal output parameters, and the result type of the operation indicated in the ocl message expression. Note that the Parameter type is defined in the UML 1.4 foundation package.

```
context OclMessageExpEval
inv: let returnArguments: Sequence{ NameValueBindings ) =
                                        resultValue.returnMessage.arguments ,
         formalParameters: Sequence{ Parameter } =
                                        model.calledOperation.operation.parameter
in
     resultValue.returnMessage->size() = 1 and model.calledOperation->size() = 1 implies
     -- 'result' must be present and have correct type
          returnArguments->any( name = 'result' ).value.model =
             formalParameters->select( kind = ParameterDirectionKind::return ).type
     and
     -- all 'out' parameters must be present and have correct type
     Sequence{1.. returnArguments->size()} ->forAll( i |
          returnArguments->at(i).name =
             formalParameters->select( kind = ParameterDirectionKind::out )->at(i).name
          and
          returnArguments->at(i).value.model =
            formalParameters->select( kind = ParameterDirectionKind::out )->at(i).type )
```

OclMessageArgEval

[1] An ocl message argument evaluation must correspond with its argument expression.

```
context OclMessageArgEval
inv: model.variable->size() = 1
         implies variable->size() = 1 and variable.symbol = model.variable.name
inv: model.expression->size() = 1
         implies expression and expression.model = model.expression
```

OperationCallExpEval

[1] The result value of an operation call expression will have the type given by the Operation being called, if the operation has no out or in/out parmeters, else the type will be a tuple containing all out, in/out parameters and the result value.

```
context  OperationCallEval inv:
let outparameters : Set( Parameter ) = referedOperation.parameter->select( p |
                                  p.kind = ParameterDirectionKind::in/out or
                                  p.kind = ParameterDirectionKind::out)
in
   if outparameters->isEmpty()
   then resultValue.model = model.referredOperation.parameter
                                  ->select( kind = ParameterDirectionKind::result ).type
   else resultValue.model.oclIsType( TupleType ) and
        outparameters->forAll( p |
          resultValue.model.attribute->exist( a | a.name = p.name and a.type = p.type ))
```

```
endif
```

[2] The string that represents the referred operation in the evaluation must be equal to the name of the referredOperation in the corresponding expression.

```
context OperationCallExpEval inv:
referredOperation = model.referredOperation.name
```

[3] The arguments of an operation call expression evaluation must correspond with the arguments of its associated expression.

```
context OperationCallExpEval inv:
Sequence{1..arguments->size}->forAll( i |
         arguments->at(i).model = model.arguments->at(i) )
```

PropertyCallExpEval

[1] The source of the evaluation of a property call corresponds to the source of its associated expression.

```
context PropertyCallExpEval inv:
source.model = model.source
```

PrimitiveLiteralExpEval

No extra well-formedness rules.

RealLiteralExpEval

```
context RealLiteralExpEval inv:
resultValue = model.realSymbol
```

StringLiteralExpEval

```
context StringLiteralExpEval inv:
resultValue = model.stringSymbol
```

TupleLiteralExpEval

```
context TupleLiteralExpEval inv:
model.tuplePart = tuplePart.model
```

UnspecifiedValueExpEval

[1] The result of an unspecified value expression is a randomly picked instance of the type of the expression.

```
context UnspecifiedValueExpEval
inv: resultValue = model.type.allInstances()->any( true )
inv: resultValue.model = model.type
```

VariableDeclEval

```
context VariableDeclEval inv:
model.initExpression = initExpression.model
```

VariableExpEval

No extra well-formedness rules.

6

The OCL Standard Library

This section describes the OCL Standard Library of predefined types, their operations, and predefined expression templates in the OCL. This section contains all standard types defined within OCL, including all the operations defined on those types. For each operation the signature and a description of the semantics is given. Within the description, the reserved word ‘result’ is used to refer to the value that results from evaluating the operation. In several places, post conditions are used to describe properties of the result. When there is more than one postcondition, all postconditions must be true. A similar thing is true for multiple preconditions. If these are used, the operation is only defined if all preconditions evaluate to true.

6.1 INTRODUCTION

The structure, syntax and semantics of the OCL is defined in the sections 3 (“Abstract Syntax”), 4 (“Concrete Syntax”) and A (“Semantics”). This section adds another part to the OCL definition: a library of predefined types and operations. Any implementation of OCL must include this library package. This approach has also been taken by e.g. the Java definition, where the language definition and the standard libraries are both mandatory parts of the complete language definition.

The OCL standard library defines a number of types, which are shown in figure 6-1 on page 6-2. It includes several primitive types: Integer, Real, String and Boolean. These are familiar from many other languages. The second part of the standard library consists of the collection types. They are Bag, Set, Sequence and Collection, where Collection is an abstract type. Note that all types defined in the OCL standard library are instances of an abstract syntax class. The OCL standard library exists at the modeling level, also referred to as the M1 level, where the abstract syntax is the metalevel or M2 level.

Next to definitions of types the OCL standard library defines a number of template expressions. Many operations defined on collections, map not on the abstract syntax metaclass ModelPropertyCallExp, but on the IteratorExp. For each of these a template expression that defines the name and format of the expression, is defined in section 6.6 (“Predefined Iterator Expressions”).

6.2 THE OCLANY, OCLVOID, AND OCLMESSAGE TYPES

OclAny

The type OclAny is the supertype of all types in the UML model and the primitive types in the OCL Standard Library. The collection types from the OCL Standard Library are not subtypes of OclAny. Properties of OclAny are available on each object in all OCL expressions. OclAny is itself an instance of the metatype Classifier.

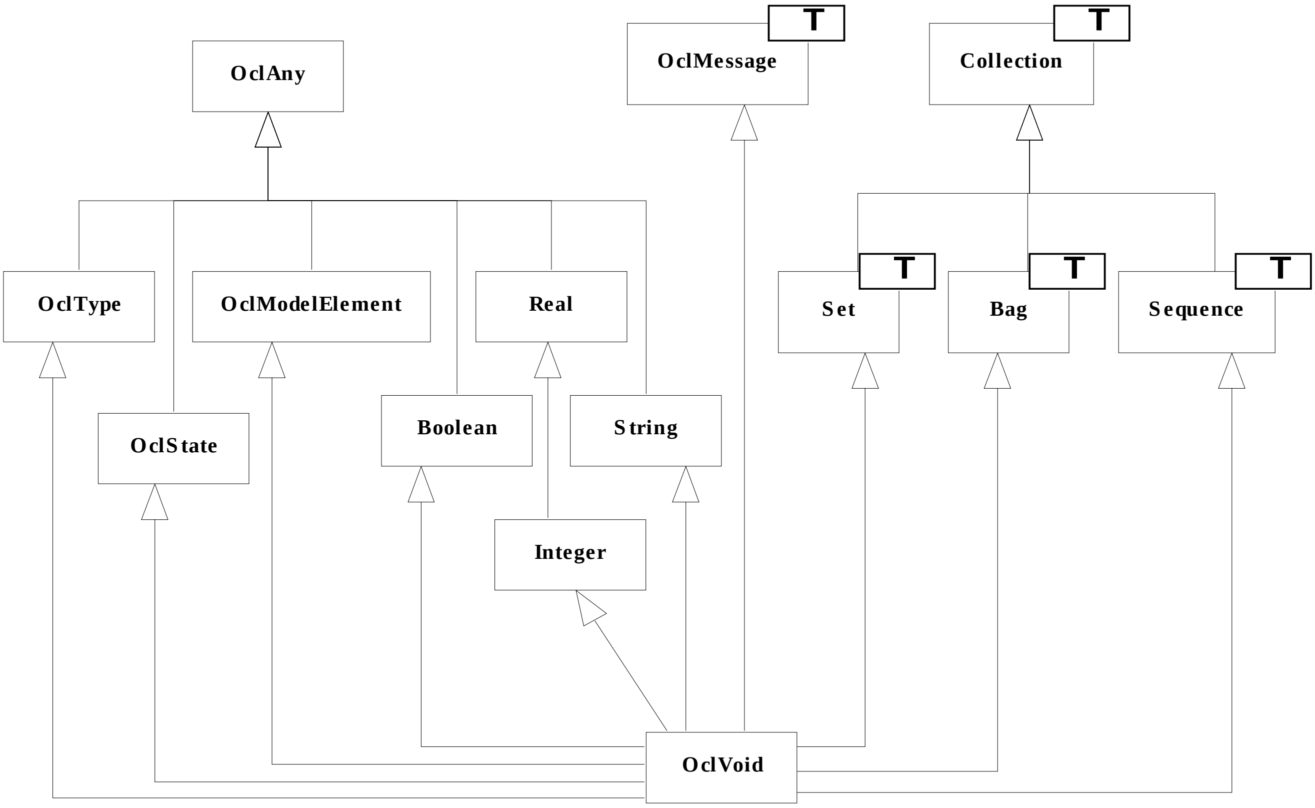


Figure 6-1 *The types defined in the OCL standard library*

All classes in a UML model inherit all operations defined on OclAny. To avoid name conflicts between properties in the model and the properties inherited from OclAny, all names on the properties of OclAny start with ‘ocl.’ Although theoretically there may still be name conflicts, they can be avoided. One can also use the oclAsType() operation to explicitly refer to the OclAny properties.

Operations of OclAny, where the instance of OclAny is called *object*.

OclMessage

This section contains the definition of the standard type *OclMessage*. As defined in this section, each ocl message type is actually a template type with one parameter. ‘T’ denotes the parameter. A concrete ocl message type is created by substituting an operation or signal for the T.

The predefined type *OclMessage* is an instance of *OclMessageType*. Every *OclMessage* is fully determined by either the operation, or signal given as parameter. Note that there is conceptually an undefined (infinite) number of these types, as each is determined by a different operation or signal. These types are unnamed. Every type has as attributes the name of the operation or signal, and either all formal parameters of the operation, or all attributes of the signal. OclMessage is itself an instance of the metatype OclMessageType.

OclMessage has a number of predefined operations, as shown in the OCL Standard Library.

OclVoid

The type OclVoid is a type that conforms to all other types. It has one single instance called OclUndefined. Any propertycall applied on OclUndefined results in OclUndefined, except for the operation oclIsUndefined(). OclVoid is itself an instance of the metatype Classifier.

6.2.1 Operations and well-formedness rules

OclAny

= (object2 : OclAny) : Boolean

True if *self* is the same object as *object2*. Infix operator.

post: result = (self = object2)

<> (object2 : OclAny) : Boolean

True if *self* is a different object from *object2*. Infix operator.

```
post: result = not (self = object2)
```

oclIsNew() : Boolean

Can only be used in a postcondition. Evaluates to true if the *self* is created during performing the operation. I.e. it didn't exist at precondition time.

post: self@pre.oclIsUndefined()

oclIsUndefined() : Boolean

Evaluates to true if the *self* is equal to OclUndefined.

post: result = self.isTypeOf(OclVoid)

OclMessage

hasReturned() : Boolean

True if type of template parameter is an operation call, and the called operation has returned a value. This implies the fact that the message has been sent. False in all other cases.

```
post: --
```

result() : *<<The return type of the called operation>>*

Returns the result of the called operation, if type of template parameter is an operation call, and the called operation has returned a value. Otherwise the undefined value is returned.

```
pre: hasReturned()
```

isSignalSent() : Boolean

Returns true if the OclMessage represents the sending of a UML Signal.

isOperationCall() : Boolean

Returns true if the OclMessage represents the sending of a UML Operation call.

OclVoid

oclIsUndefined() : Boolean

Evaluates to true if the *object* is equal to OclUndefined.

post: result = true

[1] OclVoid has only one instance.

```
context OclVoid inv:
OclVoid.allinstances()->size() = 1
```

6.3 MODELELEMENT TYPES

This section defines several enumeration types that allow the modeler to refer to elements defined in the UML model.

OclModelElement

An OclModelElement is an enumeration. For each element in a UML model there is a corresponding enumeration literal. OclModelElement is itself an instance of the metatype Enumeration (from UML Core).

OclType

An OclType is an enumeration. For each Classifier in a UML model there is a corresponding enumeration literal. OclType is itself an instance of the metatype Enumeration (from UML Core).

OclState

An OclState is an enumeration. For each State in a UML model there is a corresponding enumeration literal. OclState is itself an instance of the metatype Enumeration (from UML Core).

6.3.1 Operations and well-formedness rules

This section contains thye operatiins and well-formedness rules of the model element types.

OclModelElement

= (object : OclType) : Boolean

True if *self* is the same object as *object*.

<> (object : OclType) : Boolean

True if *self* is a different object from *object*.

```
post: result = not (self = object)
```

OclType

= (object : OclType) : Boolean

True if *self* is the same object as *object*.

<> (object : OclType) : Boolean

True if *self* is a different object from *object*.

```
post: result = not (self = object)
```

OclState

= (object : OclState) : Boolean

True if *self* is the same object as *object*.

<> (object : OclState) : Boolean

True if *self* is a different object from *object*.

```
post: result = not (self = object)
```

6.4 PRIMITIVE TYPES

The primitive types defined in the OCL standard library are Integer, Real, String and Boolean. They are all instance of the metaclass Primitive from the UML core package.

Real

The standard type Real represents the mathematical concept of real. Note that Integer is a subclass of Real, so for each parameter of type Real, you can use an integer as the actual parameter. Real is itself an instance of the metatype Primitive (from UML Core).

Integer

The standard type Integer represents the mathematical concept of integer. Integer is itself an instance of the metatype Primitive (from UML Core).

String

The standard type String represents strings, which can be both ASCII or Unicode. String is itself an instance of the metatype Primitive (from UML Core).

Boolean

The standard type Boolean represents the common true/false values. Boolean is itself an instance of the metatype Primitive (from UML Core).

6.4.1 Operations and well-formedness rules

This section contains the operatiins and well-formedness rules of the primitive types.

Real

Note that Integer is a subclass of Real, so for each parameter of type Real, you can use an integer as the actual parameter.

+ (r : Real) : Real

The value of the addition of *self* and *r*.

- (r : Real) : Real

The value of the subtraction of *r* from *self*.

*** (r : Real) : Real**

The value of the multiplication of *self* and *r*.

- : Real

The negative value of *self*.

/ (r : Real) : Real

The value of *self* divided by *r*.

abs() : Real

The absolute value of *self*.

```
post: if self < 0 then result = - self else result = self endif
```

floor() : Integer

The largest integer which is less than or equal to *self*.

```
post: (result <= self) and (result + 1 > self)
```

round() : Integer

The integer which is closest to *self*. When there are two such integers, the largest one.

```
post: ((self - result).abs() < 0.5) or ((self - result).abs() = 0.5 and (result > self))
```

max(r : Real) : Real

The maximum of *self* and *r*.

```
post: if self >= r then result = self else result = r endif
```

min(r : Real) : Real

The minimum of *self* and *r*.

```
post: if self <= r then result = self else result = r endif
```

< (r : Real) : Boolean

True if *self* is less than *r*.

> (r : Real) : Boolean

True if *self* is greater than *r*.

```
post: result = not (self <= r)
```

<= (r : Real) : Boolean

True if *self* is less than or equal to *r*.

```
post: result = ((self = r) or (self < r))
```

>= (r : Real) : Boolean

True if *self* is greater than or equal to *r*.

```
post: result = ((self = r) or (self > r))
```

Integer

- : Integer

The negative value of *self*.

+ (i : Integer) : Integer

The value of the addition of *self* and *i*.

- (i : Integer) : Integer

The value of the subtraction of *i* from *self*.

*** (i : Integer) : Integer**

he value of the multiplication of *self* and *i*.

/ (i : Integer) : Real

The value of *self* divided by *i*.

abs() : Integer

The absolute value of *self*.

```
post: if self < 0 then result = - self else result = self endif
```

div(i : Integer) : Integer

The number of times that *i* fits completely within *self*.

```
pre : i <> 0
post: if self / i >= 0 then result = (self / i).floor()
                       else result = -((-self/i).floor())
      endif
```

mod(i : Integer) : Integer

The result is *self* modulo *i*.

```
post: result = self - (self.div(i) * i)
```

max(i : Integer) : Integer

The maximum of *self* an *i*.

```
post: if self >= i then result = self else result = i endif
```

min(i : Integer) : Integer

The minimum of *self* an *i*.

```
post: if self <= i then result = self else result = i endif
```

String

size() : Integer

The number of characters in *self*.

concat(s : String) : String

The concatenation of *self* and *s*.

```
post: result.size() = self.size() + string.size()
post: result.substring(1, self.size() ) = self
post: result.substring(self.size() + 1, result.size() ) = s
```

substring(lower : Integer, upper : Integer) : String

The sub-string of *self* starting at character number *lower*, up to and including character number *upper*. Character numbers run from 1 to *self.size()*.

```
pre: 1 <= lower
pre: lower <= upper
pre: upper <= self.size()
```

toInteger() : Integer

Converts *self* to an Integer value.

toReal() : Real

Converts *self* to a Real value.

Boolean

or (b : Boolean) : Boolean

True if either *self* or *b* is true.

xor (b : Boolean) : Boolean

True if either *self* or *b* is true, but not both.

```
post: (self or b) and not (self = b)
```

and (b : Boolean) : Boolean

True if both *b1* and *b* are true.

not : Boolean

True if *self* is false.

```
post: if self then result = false else result = true endif
```

implies (b : Boolean) : Boolean

True if *self* is false, or if *self* is true and *b* is true.

```
post: (not self) or (self and b)
```

6.5 COLLECTION-RELATED TYPES

This section defines the collection types and their operations. As defined in this section, each collection type is actually a template type with one parameter. 'T' denotes the parameter. A concrete collection type is created by substituting a type for the T. So Set (Integer) and Bag (Person) are collection types.

Collection

Collection is the abstract supertype of all collection types in the OCL Standard Library. Each occurrence of an object in a collection is called an *element*. If an object occurs twice in a collection, there are two elements. This section defines the properties on Collections that have identical semantics for all collection subtypes. Some operations may be defined within the subtype as well, which means that there is an additional postcondition or a more specialized return value. Collection is itself an instance of the metatype CollectionType.

The definition of several common operations is different for each subtype. These operations are not mentioned in this section.

The semantics of the collection operations is given in the form of a postcondtion that uses the *IterateExp* ot the *IteratorExp* construct. The semantics of those constructs is defined in section A ("Semantics"). In several cases the postcondtion refers to other collection operations, which in turn are defined in terms of the *IterateExp* or *IteratorExp* constructs.

Set

The Set is the mathematical set. It contains elements without duplicates. Set is itself an instance of the metatype SetType.

Bag

A bag is a collection with duplicates allowed. That is, one object can be an element of a bag many times. There is no ordering defined on the elements in a bag. Bag is itself an instance of the metatype BagType.

Sequence

A sequence is a collection where the elements are ordered. An element may be part of a sequence more than once. Sequence is itself an instance of the metatype SequenceType.

6.5.1 Operations and well-formedness rules

This section contains the operations and well-formedness rules of the collection types.

Collection

size() : Integer

The number of elements in the collection *self*.

```
post: result = self->iterate(elem; acc : Integer = 0 | acc + 1)
```

includes(object : T) : Boolean

True if *object* is an element of *self*, false otherwise.

```
post: result = (self->count(object) > 0)
```

excludes(object : T) : Boolean

True if *object* is not an element of *self*, false otherwise.

```
post: result = (self->count(object) = 0)
```

count(object : T) : Integer

The number of times that *object* occurs in the collection *self.*

```
post: result = self->iterate( elem; acc : Integer = 0 |
             if elem = object then acc + 1 else acc endif)
```

includesAll(c2 : Collection(T)) : Boolean

Does *self* contain all the elements of *c2* ?

```
post: result = c2->forAll(elem | self->includes(elem))
```

excludesAll(c2 : Collection(T)) : Boolean

Does *self* contain none of the elements of *c2* ?

```
post: result = c2->forAll(elem | self->excludes(elem))
```

isEmpty() : Boolean

Is *self* the empty collection?

```
post: result = ( self->size() = 0 )
```

notEmpty() : Boolean

Is *self* not the empty collection?

```
post: result = ( self->size() <> 0 )
```

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 and be both associative: (a+b)+c = a+(b+c), and commutative: a+b = b+a. Integer and Real fulfill this condition.

```
post: result = self->iterate( elem; acc : T = 0 | acc + elem )
```

Set

union(s : Set(T)) : Set(T)

The union of *self* and *s*.

```
post: result->forAll(elem | self->includes(elem) or s->includes(elem))
post: self  ->forAll(elem | result->includes(elem))
post: s     ->forAll(elem | result->includes(elem))
```

union(bag : Bag(T)) : Bag(T)

The union of *self* and *bag*.

```
post: result->forAll(elem | result->count(elem) = self->count(elem) + bag->count(elem))
post: self->forAll(elem | result->includes(elem))
post: bag ->forAll(elem | result->includes(elem))
```

= (s : Set(T)) : Boolean

Evaluates to true if *self* and *s* contain the same elements.

```
post: result = (self->forAll(elem | s->includes(elem)) and
                                           s->forAll(elem | self->includes(elem)) )
```

intersection(s : Set(T)) : Set(T)

The intersection of *self* and *s* (i.e, the set of all elements that are in both *self* and *s*).

```
post: result->forAll(elem | self->includes(elem) and s->includes(elem))
post: self->forAll(elem | s   ->includes(elem) = result->includes(elem))
post: s   ->forAll(elem | self->includes(elem) = result->includes(elem))
```

intersection(bag : Bag(T)) : Set(T)

The intersection of *self* and *bag.*

```
post: result = self->intersection( bag->asSet )
```

– (s : Set(T)) : Set(T)

The elements of *self*, which are not in *s*.

```
post: result->forAll(elem | self->includes(elem) and s->excludes(elem))
```

```
post: self  ->forAll(elem | result->includes(elem) = s->excludes(elem))
```

including(object : T) : Set(T)

The set containing all elements of *self* plus *object*.

```
post: result->forAll(elem | self->includes(elem) or (elem = object))
post: self-  >forAll(elem | result->includes(elem))
post: result->includes(object)
```

excluding(object : T) : Set(T)

The set containing all elements of *self* without *object*.

```
post: result->forAll(elem | self->includes(elem) and (elem <> object))
post: self-  >forAll(elem | result->includes(elem) = (object <> elem))
post: result->excludes(object)
```

symmetricDifference(s : Set(T)) : Set(T)

The sets containing all the elements that are in *self* or *s*, but not in both.

```
post: result->forAll(elem | self->includes(elem) xor s->includes(elem))
post: self->forAll(elem | result->includes(elem) = s   ->excludes(elem))
post: s   ->forAll(elem | result->includes(elem) = self->excludes(elem))
```

count(object : T) : Integer

The number of occurrences of *object* in *self*.

```
post: result <= 1
```

flatten() : Set(T2)

If the element type is not a collection type this result in the same *self*. If the element type is a collection type, the result is the set containing all the elements of all the elements of *self*.

```
post: result = if self.type.elementType.oclIsKindOf(CollectionType) then
                  self->iterate(c; acc : Set() = Set{} |
                       acc->union(c->asSet() ) )
               else
                  self
               endif
```

asSet() : Set(T)

A Set identical to self. This operation exists for convenience reasons.

```
post: result = self
```

asSequence() : Sequence(T)

A Sequence that contains all the elements from *self*, in undefined order.

```
post: result->forAll(elem | self->includes(elem))
post: self->forAll(elem | result->count(elem) = 1)
```

asBag() : Bag(T)

The Bag that contains all the elements from *self*.

```
post: result->forAll(elem | self->includes(elem))
post: self->forAll(elem | result->count(elem) = 1)
```

Bag

= (bag : Bag(T)) : Boolean

True if *self* and *bag* contain the same elements, the same number of times.

```
post: result = (self->forAll(elem | self->count(elem) = bag->count(elem)) and
                bag->forAll(elem | bag->count(elem) = self->count(elem)) )
```

union(bag : Bag(T)) : Bag(T)

The union of *self* and *bag*.

```
post: result->forAll( elem | result->count(elem) = self->count(elem) + bag->count(elem))
post: self  ->forAll( elem | result->count(elem) = self->count(elem) + bag->count(elem))
post: bag   ->forAll( elem | result->count(elem) = self->count(elem) + bag->count(elem))
```

union(set : Set(T)) : Bag(T)

The union of *self* and *set*.

```
post: result->forAll(elem | result->count(elem) = self->count(elem) + set->count(elem))
post: self  ->forAll(elem | result->count(elem) = self->count(elem) + set->count(elem))
post: set   ->forAll(elem | result->count(elem) = self->count(elem) + set->count(elem))
```

intersection(bag : Bag(T)) : Bag(T)

The intersection of *self* and *bag*.

```
post: result->forAll(elem |
      result->count(elem) = self->count(elem).min(bag->count(elem)) )
post: self->forAll(elem |
      result->count(elem) = self->count(elem).min(bag->count(elem)) )
post: bag->forAll(elem |
      result->count(elem) = self->count(elem).min(bag->count(elem)) )
```

intersection(set : Set(T)) : Set(T)

The intersection of *self* and *set*.

```
post: result->forAll(elem|result->count(elem) = self->count(elem).min(set->count(elem)) )
post: self  ->forAll(elem|result->count(elem) = self->count(elem).min(set->count(elem)) )
post: set   ->forAll(elem|result->count(elem) = self->count(elem).min(set->count(elem)) )
```

including(object : T) : Bag(T)

The bag containing all elements of *self* plus *object*.

```
post: result->forAll(elem |
        if elem = object then
           result->count(elem) = self->count(elem) + 1
        else
           result->count(elem) = self->count(elem)
        endif)
post: self->forAll(elem |
        if elem = object then
           result->count(elem) = self->count(elem) + 1
        else
           result->count(elem) = self->count(elem)
        endif)
```

excluding(object : T) : Bag(T)

The bag containing all elements of self apart from all occurrences of *object*.

```
post: result->forAll(elem |
        if elem = object then
           result->count(elem) = 0
        else
           result->count(elem) = self->count(elem)
        endif)
post: self->forAll(elem |
        if elem = object then
           result->count(elem) = 0
        else
           result->count(elem) = self->count(elem)
        endif)
```

count(object : T) : Integer

The number of occurrences of *object* in *self*.

flatten() : Bag(T2)

If the element type is not a collection type this result in the same bag. If the element type is a collection type, the result is the bag containing all the elements of all the elements of *self*.

```
post: result = if self.type.elementType.oclIsKindOf(CollectionType) then
                 self->iterate(c; acc : Bag() = Bag{} |
                      acc->union(c->asBag() ) )
              else
```

```
                  self
               endif
```

asBag() : Bag(T)

A Bag identical to *self*. This operation exists for convenience reasons.

```
post: result = self
```

asSequence() : Sequence(T)

A Sequence that contains all the elements from *self*, in undefined order.

```
post: result->forAll(elem | self->count(elem) = result->count(elem))
post: self  ->forAll(elem | self->count(elem) = result->count(elem))
```

asSet() : Set(T)

The Set containing all the elements from *self*, with duplicates removed.

```
post: result->forAll(elem | self  ->includes(elem))
post: self  ->forAll(elem | result->includes(elem))
```

Sequence

count(object : T) : Integer

The number of occurrences of *object* in *self*.

= (s : Sequence(T)) : Boolean

True if *self* contains the same elements as *s* in the same order.

```
post: result = Sequence{1..self->size()}->forAll(index : Integer |
                                 self->at(index) = s->at(index))
                                 and
                                 self->size() = s->size()
```

union (s : Sequence(T)) : Sequence(T)

The sequence consisting of all elements in *self*, followed by all elements in *s*.

```
post: result->size() = self->size() + s->size()
post: Sequence{1..self->size()}->forAll(index : Integer |
                                                    self->at(index) = result->at(index))
post: Sequence{1..s->size()}->forAll(index : Integer |
                                         s->at(index) =  result->at(index + self->size() )))
```

flatten() : Sequence(T2)

If the element type is not a collection type this result in the same *self*. If the element type is a collection type, the result is the seuqnce containing all the elements of all the elements of *self*. The order of the elements is partial.

```
post: result = if self.type.elementType.oclIsKindOf(CollectionType) then
                  self->iterate(c; acc : Sequence() = Sequence{} |
                       acc->union(c->asSequence() ) )
               else
                  self
               endif
```

append (object: T) : Sequence(T)

The sequence of elements, consisting of all elements of *self*, followed by *object*.

```
post: result->size() = self->size() + 1
post: result->at(result->size() ) = object
post:   Sequence{1..self->size() }->forAll(index : Integer |
         result->at(index) = self ->at(index))
```

prepend(object : T) : Sequence(T)

The sequence consisting of *object*, followed by all elements in *self*.

```
post: result->size = self->size() + 1
post: result->at(1) = object
post:   Sequence{1..self->size()}->forAll(index : Integer |
```

```
        self->at(index) = result->at(index + 1))
```

insertAt(index : Integer, object : T) : Sequence(T)

The sequence consisting of *self* with *object* inserted at position *index*.

```
post: result->size = self->size() + 1
post: result->at(index) = object
post: Sequence{1..(index - 1)}->forAll(i : Integer |
        self->at(i) = result->at(i))
post: Sequence{(index + 1)..self->size()}->forAll(i : Integer |
        self->at(i) = result->at(i + 1))
```

subSequence(lower : Integer, upper : Integer) : Sequence(T)

The sub-sequence of *self* starting at number *lower*, up to and including element number *upper*.

```
pre : 1 <= lower
pre : lower <= upper
pre : upper <= self->size()
post: result->size() = upper -lower + 1
post: Sequence{lower..upper}->forAll( index |
         result->at(index - lower + 1) =
                          self->at(index))
```

at(i : Integer) : T

The *i-th* element of sequence.

```
pre : i >= 1 and i <= self->size()
```

indexOf(obj : T) : Integer

The index of object *obj* in the sequence.

```
pre  : self->includes(obj)
post : self->at(i) = obj
```

first() : T

The first element in *self*.

```
post: result = self->at(1)
```

last() : T

The last element in *self*.

```
post: result = self->at(self->size() )
```

including(object : T) : Sequence(T)

The sequence containing all elements of *self* plus *object* added as the last element.

```
post: result = self.append(object)
```

excluding(object : T) : Sequence(T)

The sequence 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() - self->count(object)
post: result = self->iterate(elem; acc : Sequence(T)
     = Sequence{}|
         if elem = object then acc else acc->append(elem) endif )
```

asBag() : Bag(T)

The Bag containing all the elements from *self*, including duplicates.

```
post: result->forAll(elem | self->count(elem) = result->count(elem) )
post: self->forAll(elem | self->count(elem) = result->count(elem) )
```

asSequence() : Sequence(T)

The Sequence identical to the object itself. This operation exists for convenience reasons.

```
post: result = self
```

asSet() : Set(T)

The Set containing all the elements from *self*, with duplicated removed.

```
post: result->forAll(elem | self  ->includes(elem))
post: self  ->forAll(elem | result->includes(elem))
```

6.6 PREDEFINED ITERATOR EXPRESSIONS

This section defines the standard OCL iterator expressions. In the abstract syntax these are all instances of IteratorExp. These iterator expressions always have a collection expression as their source, as is defined in the well-formedness rules in section 3 (“Abstract Syntax”). The defined iterator expressions are shown per source collection type. The semantics of each iterator expression is defined through a mapping from the iterator to the ’*iterate*’ construct. this means that the semantics of the iterator expressions does not need to be defined seperately in the sementics sections.

Whenever a new itertor is added to the library, the mapping to the *iterate* expression must be defined. If this is not done, the semantics of the new iterator is undefined.

In all of the following OCL expressions, the lefthand side of the equals sign is the *IteratorExp* to be defined, and the righthand side of the equals sign is the equivalent as an *IterateExp*. The names *source*, *body* and *iterator* refer to the role names in the abstract syntax:

source	The source expression of the IteratorExp
body	The body expression of the IteratorExp
iterator	The iterator variable of the IteratorExp
result	The result variable of the IterateExp

Extending the standard library with iterator expressions

When new iterator expressions are added to the standard library, there mapping to existing constructs should be fully defines. If this is done, the semantics of the new iterator expression will be defined.

6.6.1 Mapping rules for predefined iterator expressions

This section contains the operations and well-formedness rules of the collection types.

Collection

exists

Results in true if *body* evaluates to true for at least one element in the *source* collection.

```
source->exists(iterators | body) =
                     source->iterate(iterators; result : Boolean = false | result or body)
```

forAll

Results in true if the *body* expression evaluates to true for each element in the *source* collection; otherwise, result is false.

```
source->forAll(iterators | body ) =
                   source->iterate(iterators; result : Boolean = true | result and body)
```

isUnique

Results in true if *body* evaluates to a different value for each element in the *source* collection; otherwise, result is false.

```
source->isUnique(iterators | body) =
    source->collect(iterators|body)->forAll(x, y | x <> y )
```

sortedBy

Results in the Sequence containing all elements of the *source* collection. The element for which *body* has the lowest value comes first, and so on. The type of the *body* expression must have the < operation defined. The < operation must return a Boolean value and must be transitive i.e. if a < b and b < c then a < c.

```
source->sortedBy(iterator | body) =
    iterate( iterator ; result : Sequence(T) : Sequence {} |
         if result->isEmpty() then
           result.append(iterator)
         else
           let position : Integer = result->select(item | item > iterator)->first()
           in
             result.insertAt(position, iterator)
         endif
```

sortedBy may have at most one iterator variable.

any

Returns any element in the *source* collection for which *body* evaluates to true. If there is more than one element for which *body* is true, one of them is returned. There must be at least one element fulfilling *body*, otherwise the result of this IteratorExp is OclUndefined.

```
source->any(iterator | body) =
       source->select(iterator | body)->asSequence()->first()
```

any may have at most one iterator variable.

one

Results in true if there is exactly one element in the *source* collection for which *body* is true.

```
source->one(iterator | body) =
      source->select(iterator | body)->size() = 1
```

one may have at most one iterator variable.

collect

The Collection of elements which results from applying *body* to every member of the *source* set. The result is flattened. Notice that this is based on *collectNested*, which can be of different type depending on the type of *source*. *collectNested* is defined individually for each subclass of *CollectionType*.

```
source->collect (iterators | body) = source->collectNested (iterators | body)->flatten()
```

Set

The standard iterator expression with source of type Set(T) are:

select

The subset of *set* for which *expr* is true.

```
source->select(iterator | body) =
         source->iterate(iterator; result : Set(T) = Set{} |
                          if body then result->including(iterator)
                                  else result
                           endif)
```

select may have at most one iterator variable.

reject

The subset of the *source* set for which *body* is false.

```
source->reject(iterator | body) =
        source->select(iterator | not body)
```

reject may have at most one iterator variable.

collectNested

The Bag of elements which results from applying *body* to every member of the *source* set.

```
source->collect(iterators | body) =
        source->iterate(iterators; result : Bag(body.type) = Bag{} |
                        result->including(body ) )
```

Bag

The standard iterator expression with source of type Bag(T) are:

select

The sub-bag of the *source* bag for which *body* is true.

```
source->select(iterator | body) =
        source->iterate(iterator; result : Bag(T) = Bag{} |
                        if body then result->including(iterator)
                                else result
                        endif)
```

select may have at most one iterator variable.

reject

The sub-bag of the *source* bag for which *body* is false.

```
source->reject(iterator | body) =
        source->select(iterator | not body)
```

reject may have at most one iterator variable.

collectNested

The Bag of elements which results from applying *body* to every member of the *source* bag.

```
source->collect(iterators | body) =
        source->iterate(iterators; result : Bag(body.type) = Bag{} |
                        result->including(body ) )
```

Sequence

The standard iterator expressions with source of type Sequence(T) are:

select(expression : OclExpression) : Sequence(T)

The subsequence of the *source* sequence for which *body* is *true*.

```
source->select(iterator | body) =
        source->iterate(iterator; result : Sequence(T) = Sequence{} |
                        if body then result->including(iterator)
                                else result
                        endif)
```

select may have at most one iterator variable.

reject

The subsequence of the *source* sequence for which *body* is false.

```
source->reject(iterator | body) =
        source->select(iterator | not body)
```

reject may have at most one iterator variable.

collectNested

The Sequence of elements which results from applying *body* to every member of the *source* sequence.

```
source->collect(iterators | body) =
        source->iterate(iterators; result : Sequence(body.type) = Sequence{} |
                        result->append(body ) )
```

7

The Use of Ocl Expressions in UML Models

This section describes the various manners in which OCL expressions can be used in UML models.

7.1 INTRODUCTION

In principle, everywhere in the UML specification where the term *expression* is used, an OCL expression can be used. In UML 1.4 OCL expressions could be used e.g. for invariants, preconditions and postconditons, but other placements are possible too. The meaning of the value, which results from the evaluation of the OCL expression, depends on its placement within the UML model.

In this specification the structure of an expression, and its evaluation are separated from the usage of the expression. Chapter 3 (“Abstract Syntax”) defines the structure of an expression, and chapter A (“Semantics”) defines the evaluation. In chapter 4 (“Concrete Syntax”) it was allready noted that the contents of the name space environment of an OCL expression are fully determined by the placement of the OCL expression in the model. In that chapter an inherited attribute *env* was introduced for every production rule in the attribute grammar to represent this name space environment.

This section specifies a number of predefined places where OCL expressions can be used, their associated meaning, and the contents of the name space environment. The modeler has to define her/his own meaning, if OCL is used at a place in the UML model which is not defined in this section.

For every occurence of an OCL expression three things need to be separated: the placement, the contextual classifier, and the self instance of an OCL expression.

- The *placement* is the position where the OCL expression is used in the UML model, e.g. as invariant connected to class Person.
- The *contextual classifier* defines the namespace in which the expression is evaluated. For example, the contextual classifier of a precondition is the classifier that is the owner of the operation for which the precondition is defined. Visible within the precondition are all model element that are visible in the contextual classifier.
- The *self instance* is the reference to the object that evaluates the expression. It is always an instance of the contextual classifier. Note that evaluation of an OCL expression may result in a different value for every instance of the contextual classifier.

In the next section a number of placements are stated explicitly. For each the contextual classifier is defined, and well-formedness rules are given, that exactly define the place where the OCL expression is attached to the UML model.

UML 2.0 Alignment

The definition of the contextualClassifier and ExpressionInOcl depends to a large extend on the UML 2.0 definition. Therefore this section will need to be finished after the UML 2.0 definition has been frozen. Therefore not all rules in this section are completely finished, they need to be re-done anyway.

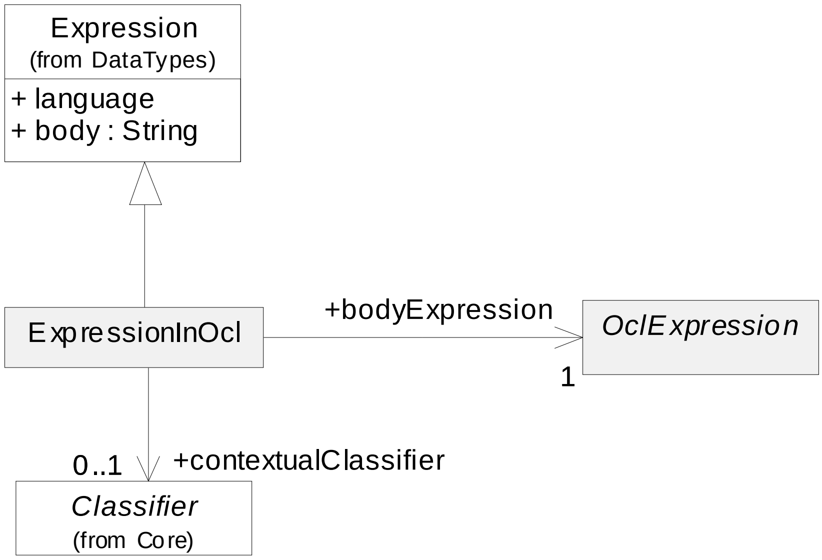


Figure 7-1 *Metaclass ExpressionInOcl added to the UML metamodel*

7.2 THE EXPRESSIONINOCL TYPE

Because in the abstract syntax OclExpression is defined recursively, we need a new metaclass to represent the top of the abstract syntax tree that represents an OCL expression. This metaclass is called *ExpressionInOcl*, and it is defined to be a subclass of the Expression metaclass from the UML core, as shown in figure 7-1. In UML (1.4) the Expression metaclass has an attribute language which may have the value 'OCL'. The body attribute contains a text representation of the actual expression. The *bodyExpression* association of ExpressionInOcl is an association to the OCL expression as represented by the OCL Abstract syntax metamodel. The *body* attribute (inherited from Expression) may still be used to store the string representation of the OCL expression. The language attribute (also inherited form Expression) has the value 'OCL'.

ExpressionInOcl

An expression in OCL is an expression that is written in OCL. The value of the *language* attribute is therefore always equal to 'OCL'.

Associations

contextualClassifier	The classifier that is the context of the OCL expression. Self is always an instance of this classifier. The *contextualClassifier* is always a derived association. In the remainder of this chapter many derivations will be given.
bodyExpression	The *bodyExpression* is an *OclExpression* that is the root of the actual OCL expression, which is described fully by the OCL abstract syntax metamodel.

7.2.1 Well-formedness rules

ExpressionInOcl

[1] This expression is always written in OCL

```
context ExpressionInOcl
inv: language = 'OCL'
```

7.3 STANDARD PLACEMENTS OF OCL EXPRESSIONS

This section defines the standard places where OCL expressions may occur, and defines for each case the value for the contextual classifier. Note that this list of places is not exhausting, and can be enhanced.

How to extend the use of OCL at other places

At many places in the UML where an Expression is used, one can write this expression in OCL. To define the use of OCL at such a place, the main task is to define what the ciontextual classifier is. When that is given, the OCL expression is fully defined. This section defines a number of often used placemants of OCL expressions.

7.3.1 Definition

A definition constraint is a constraint that is linked to a Classifier. It may only consist of one or more LetExps. The variable or function defined by the Let expression can be used in an identical way as an attribute or operation of the Classifier. Their visibility is equal to that of a public attribute or operation. The purpose of a definition constraint is to define reusable sub-expressions for use in other OCL expressions.

The placement of a definition constraint in the UML metamodel is shown in figure 7-2. The following well-formedness rule must hold. This rule also defines the value of the contextual Classifier.

Well-formedness rules

[1] The ExpressinInOcl is a definition constraint if it has the stereotype <<definition>> (A) and the constraint is attached to only one model element (B) and the constraint is attached to a Classifier (C).

```
context ExpressionInOcl
def: attr isDefinitionConstraint : Boolean =
                self.constraint.stereotype.name = 'definition'                      -- A
                and
                self.constraint.constrainedElement->size() = 1                      -- B
                and
                self.constraint.constrainedElement.any(true).oclIsKindOf(Classifier) -- C
```

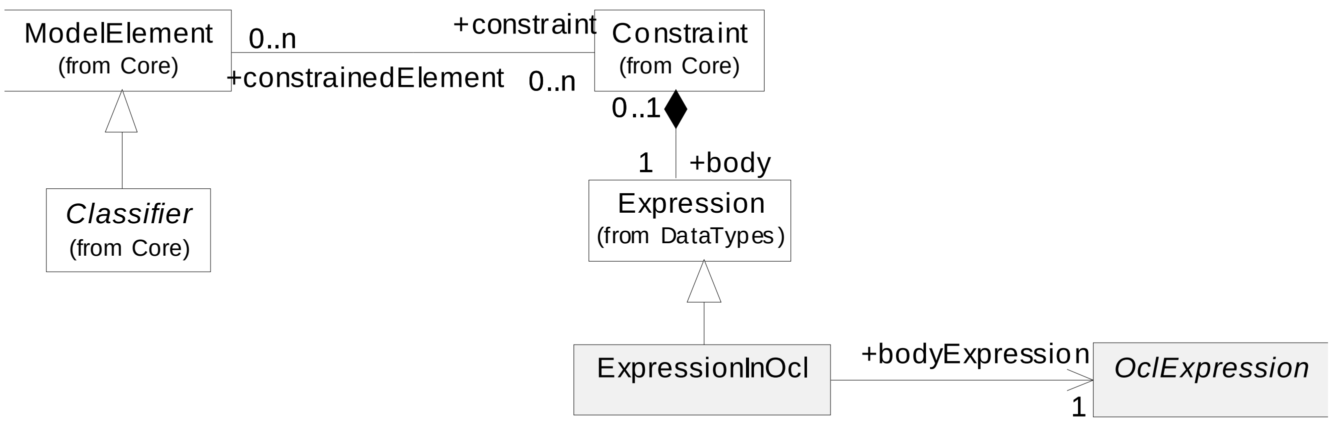


Figure 7-2 *Situation of Ocl expression used as definition or invariant*

[2] For a definition constraint the contextual classifier is the constrained element.

```
context ExpressionInOcl
inv: isDefinitionConstraint implies
             contextualClassifier =
                       self.constraint.constrainedElement.any(true).oclAsType(Classifier)
```

[3] Inside a definition constraint the use of @pre is not allowed.

```
context ExpressionInOcl
inv: --
```

7.3.2 Invariant

An invariant constraint is a constraint that is linked to a Classifier. The purpose of an invariant constraint is to specify invariants for the Classifier. An invariant constraint consists of an OCL expression of type Boolean. The expression must be true for each instance of the classifier at any moment in time. Only when an instance is executing an operation, this does not need to evaluate to true.

The placement of an invariant constraint in the UML metamodel is equal to the placement of a definition constraint, which is shown in figure 7-2. The following well-formedness rule must hold. This rule also defines the value of the contextual Classifier.

Well-formedness rules

[1] The constraint has the stereotype <<invariant>> (A) and the constraint is attached to only one model element (B) the constraint is attached to a Classifier (C). The contextual classifier is the constrained element and the type of the OCL expression must be Boolean.

```
context ExpressionInOcl
inv: self.constraint.stereotype.name = 'invariant'                          -- A
     and
     self.constraint.constrainedElement->size() = 1                         -- B
     and
     self.constraint.constrainedElement.any(true).oclIsKindOf(Classifier)   -- C
     implies
         contextualClassifier =
                   self.constraint.constrainedElement->any(true).oclAsType(Classifier)
         and
         self.bodyExpression.type.name = 'Boolean'
```

[2] Inside an invariant constraint the use of @pre is not allowed.

```
context ExpressionInOcl
inv: --
```

7.3.3 Precondition

A precondition is a constraint that may be linked to an Operation of a Classifier. The purpose of a precondition is to specify the conditions that must hold before the operation executes. A precondition consists of an OCL expression of type Boolean. The expression must evaluate to true whenever the operation starts executing, but only for the instance that will execute the operation.

The placement of a precondition in the UML metamodel is shown in figure 7-3. The following well-formedness rule must hold. This rule also defines the value of the contextual Classifier.

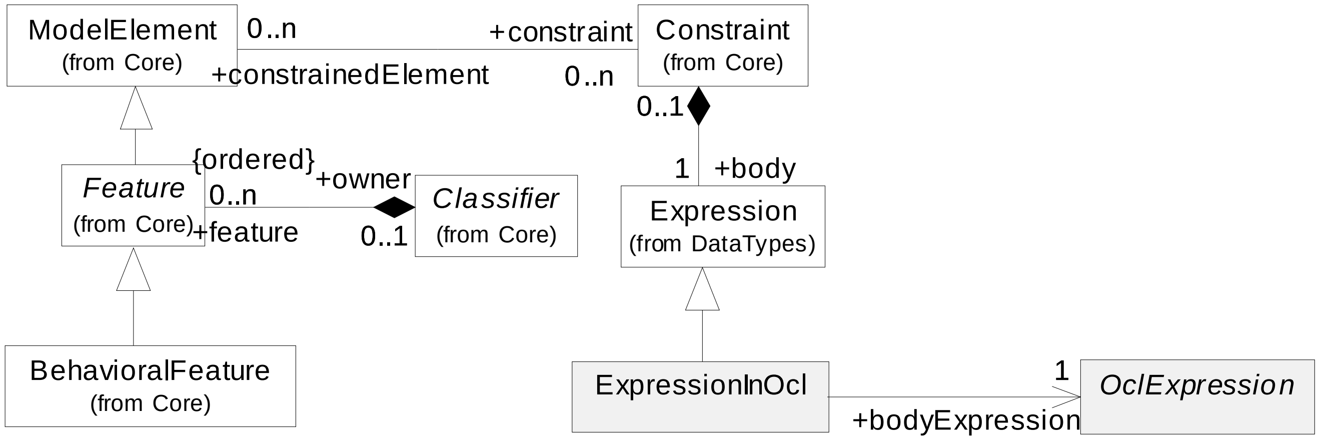


Figure 7-3 *An OCL ExpressionInOcl used as a pre- or post-condition.*

Well-formedness rules

[1] The Constraint has the stereotype <<precondition>> (A), and is attached to only one model element (B), and to a BehavioralFeature (C), which has an owner (D). The contextual classifier is the owner of the operation to which the constraint is attached, and the type of the OCL expression must be Boolean

```
context Expression
inv: self.constraint.stereotype.name = 'precondition'                          -- A
     and
     self.constraint.constrainedElement->size() = 1                            -- B
     and
    self.constraint.constrainedElement->any(true).oclIsKindOf(BehavioralFeature)   -- C
     and
     self.constraint.constrainedElement->any(true)                             -- D
                                      .oclAsType(BehavioralFeature).owner->size() = 1
     implies
         contextualClassifier =
                       self.constraint.constrainedElement->any(true)
                                                  .oclAsType(BehavioralFeature).owner
         and
         self.bodyExpression.type.name = 'Boolean'
```

[2] Inside a precondtion constraint the use of @pre is not allowed.

```
context ExpressionInOcl
inv: --
```

7.3.4 Postcondition

Like a precondition, a postcondition is a constraint that may be linked to an Operation of a Classifier. The purpose of a postcondition is to specify the conditions that must hold after the operation executes. A postcondition consists of an OCL expression of type Boolean. The expression must evaluate to true at the moment that the operation stops executing, but only for the instance that has just executed the operation. Within an OCL expression used in a postcondition, the "@pre" mark can be used to refer to values at precondition time. The variable *result* refers to the return value of the operation if there is any.

The placement of a postcondition in the UML metamodel is equal to the placement of a precondition, which is shown in figure 7-3. The following well-formedness rule must hold. This rule also defines the value of the contextual Classifier.

Wellformedness rules

[1] The Constraint has the stereotype <<postcondition>> (A), and it is attached to only one model element (B), that is an BehavioralFeature (C), which has an owner (D). The contextual classifier is the owner of the operation to which the constraint is attached, and the type of the OCL expression must be Boolean

```
context Expression
inv: self.constraint.stereotype.name = 'postcondition'                          -- A
     and
     self.constraint.constrainedElement->size() = 1                             -- B
     and
     self.constraint.constrainedElement->any(true).oclIsKindOf(BehavioralFeature)     -- C
     and
     self.constraint.constrainedElement->any(true)                              -- D
                                  .oclAsType(BehavioralFeature).owner->size() = 1
     implies
         contextualClassifier =
             self.constraint.constrainedElement->any().oclAsType(BehavioralFeature).owner
         and
         self.bodyExpression.type.name = 'Boolean'
```

7.3.5 Attribute initial value

An attribute initial value is an expression that may be linked to an Attribute of a Classifier. An OCL expression acting as the initial value of an attribute must conform to the defined type of the attribute. The OCL expression is evaluated at the creation time of the instance that owns the attribute for this created instance.

The placement of an attribute initial value in the UML metamodel is shown in figure 7-4. The following well-formedness rule must hold. This rule also defines the value of the contextual Classifier.

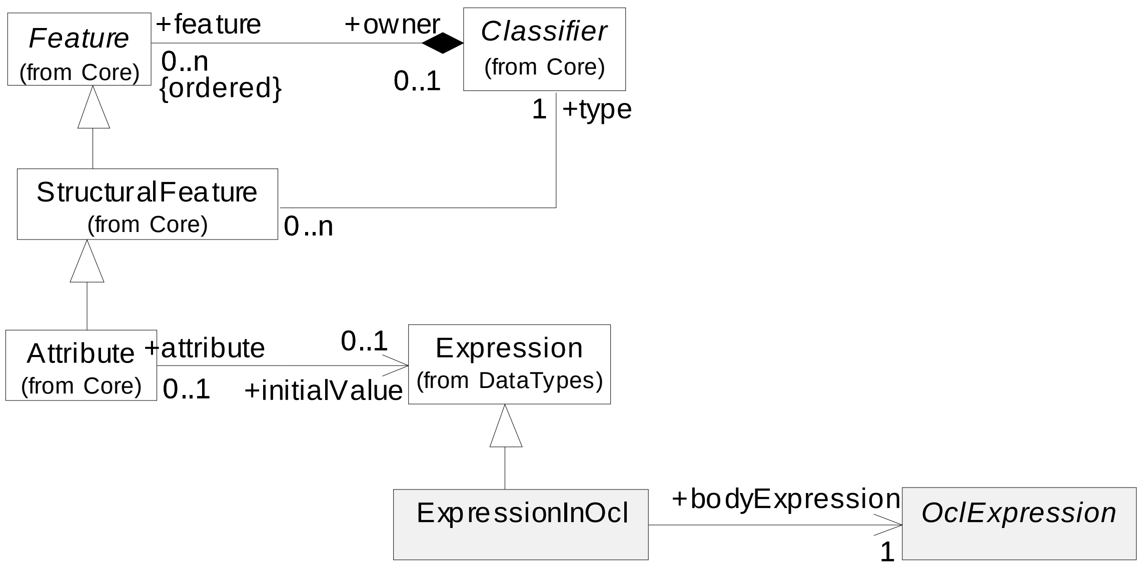


Figure 7-4 *Expression used to define the inital value of an attribute*

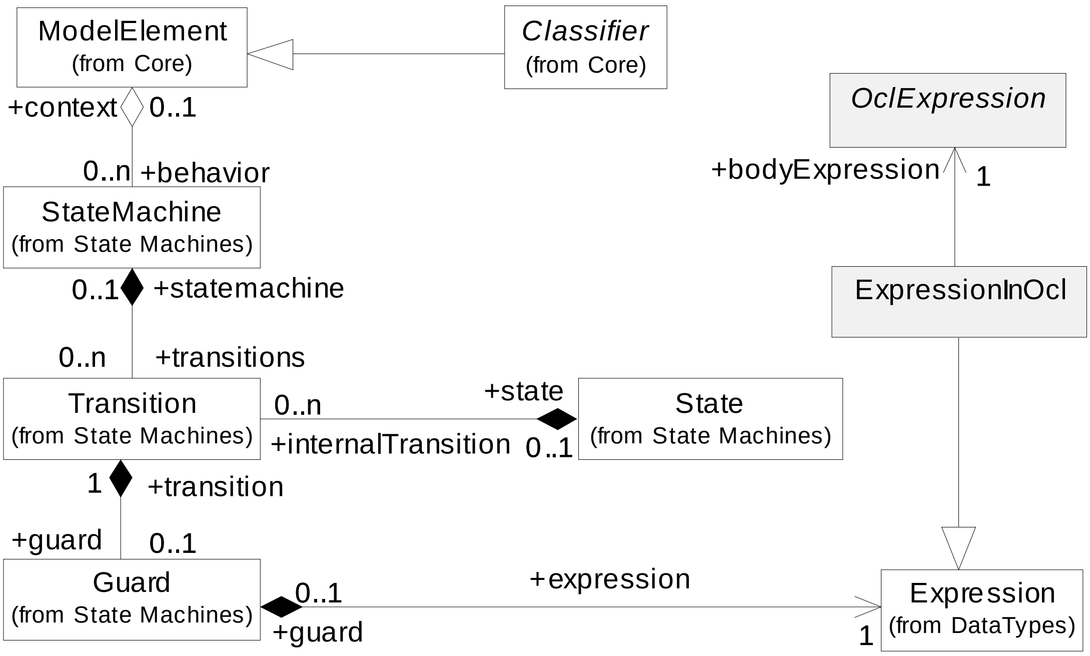


Figure 7-5 *An OCL expression used as a Guard expression*

Well-formedness rules

[1] The Expression is the initial value of an attribute (A), and the Attribute has an owner (B). The contextual classifier is the owner of the attribute, and the type of the OCL expression must conform to the type of the attribute.

```
context ExpressionInOcl
inv: self.attribute->notEmpty()                                                  -- A
     and
     self.attribute.owner->size() = 1                                            -- B
     implies
         contextualClassifier = self.attribute.owner
         and
         self.bodyExpression.type.conformsTo(self.attribute.type)
```

[2] Inside an initial attribute value the use of @pre is not allowed.

```
context ExpressionInOcl
inv: --
```

7.3.6 Guard

A guard is an expression that may be linked to a Transition in a StateMachine. An OCL expression acting as the guard of a transition restricts the transition. An OCL expression acting as value of a guard is of type Boolean. The expresion is evaluated at the moment that the transition attached to the guard is attempted.

The placement of a guard in the UML metamodel is shown in figure 7-5. The following well-formedness rule must hold. In order to state the rule a number of additional operations are defined. The rule also defines the value of the contextual Classifier.

Well-formedness rules

[1] The statemachine in which the guard appears must have a context (A), that is a Classifier (B). The contextual classifier is the owner of the statemachine, and the type of the OCL expression must be Boolean.

```
context ExpressionInOcl
inv: self.guard.transition.getStateMachine().context-> notEmpty()              -- A
     and
     self.guard.transition.getStateMachine().context.oclIsKindOf(Classifier)      -- B
     implies
       contextualClassifier =
            self.guard.transition.getStateMachine().context.oclAsType(Classifier)
       and
       self.bodyExpression.type.name = 'Boolean'
```

[2] Inside an guard the use of @pre is not allowed.

```
context ExpressionInOcl
inv: --
```

7.4 CONCRETE SYNTAX OF CONTEXT DECLARATIONS

This section describes the concrete syntax for specifying the context of the different types of usage of OCL expressions. It makes use of grammar rules defined in chapter 4 (“Concrete Syntax”). Here too, every production rule is associated to the abstract syntax by the type of the attribute *ast*. However, we must sometimes refer to the abstract syntax of the UML to find the right type for each production.

Visibility rules etc. must be defined in the UML metamodel. Here we assume that every classifier has an operation *visibleElements()*, which returns an instance of type Environment, as defined in chapter 4 (“Concrete Syntax”).

NB Note that the context declarations as described in this section are not needed when the OCL expressions are attached directly to the UML model. This concerete syntax for context declarations is only there to facilitate separate OCl expressions in text files.

because of the assumption that the concrete syntax below is used separate from the UML model, we assume the existence of an operation *getClassifier()* on the UML model that allows us to find a Classifier anywhere in the corresponding model. The signature of this operation is defined as follows:

```
context Model::findClassifier( pathName : Sequence(String) ) : Classifier
```

The *pathName* needs not be a fully qualified name (it may be), as long as it can uniquely identify the classifier siomewhere in the UML model. If a classifier name occurrs more than once, it needs to be qualified with its owning package (recusriveley) until the qualified name is unique. If more than one classifier is found, the operation returns *OclUndefined*. The variable *Model* is used to refer to the UML Model.It is used as *Model.findClassifier()*.

contextDeclarationCS

This production rule represents all different context declarations.

```
[A] contextDeclarationCS ::= attributeContextDeclCS
[B] contextDeclarationCS ::= guardContextDeclCS
[C] contextDeclarationCS ::= constraintContextDeclCS
```

Abstract syntax mapping

```
contextDeclarationCS.ast : ModelElement
```

Synthesized attributes

```
[A] contextDeclarationCS.ast = attributeContextDeclCS.ast
[B] contextDeclarationCS.ast = guardContextDeclCS.ast
```

```
[C] contextDeclarationCS.ast = constraintContextDeclCS.ast
```

Inherited attributes

```
[A] attributeContextDeclCS.env  = Environment.EMPTY_ENV
[B] guardContextDeclCS.env      = Environment.EMPTY_ENV
[C] constraintContextDeclCS.env = Environment.EMPTY_ENV
```

Disambiguating rules

```
-- none
```

attributeContextDeclCS

This production rule represents a context declaration for an attribute initial value. The path name refers to the attribute, the type states the type of the attribute.

```
attributeContextDeclCS ::= 'context' pathNameCS simpleName
                           ':' typeCS 'init' ':'  OclExpression
```

Abstract syntax mapping

```
attributeContextDeclCS.ast : ExpressionInOcl
```

Synthesized attributes

```
let contextualClassifier = Model.findClassifier(pathNameCS.ast) in

attributeContextDeclCS.ast.bodyExpression       = OclExpression.ast
attributeContextDeclCS.ast.contextualClassifier = contextualClassifier
attributeContextDeclCS.ast.attribute            =
                            contextualClassifier.lookupAttribute(simpleNameCS.ast)
```

Inherited attributes

```
OclExpressionCS.env = attributeContextDeclCS.env->addNameSpace(contextualClassifier)
                                                 ->addElement('self',
                                                              contextualClassifier
                                                               true)
```

Disambiguating rules

```
-- none
```

guardContextDeclCS

This production rule represents a context declarations for a guard in a statemachine. This concrete syntax will not often be used, because guards can for more easily be notated in the statemachine diagram. It is specified for reasons of completeness. The path name refers to the statemachine, the first simple name refers to the *from* state of a transition, the second simple name refers to the *to* state of that transition.

```
guardContextDeclCS ::= 'context' pathNameCS '::' simpleNameCS[1]
                             -> simpleNameCS[2] 'guard' ':'
                                    OclExpressionCs
```

Abstract syntax mapping

```
guardContextDeclCS.ast : ExpresssionInOcl
```

Synthesized attributes

```
let contextualClassifier = Model.findClassifier(pathNameCS.ast) in

guardContextDeclCS.ast.bodyExpression = OclExpressionCS.ast
-- To finish this, we need to be able to find the states with names simpleNameCS[1]
-- and simpleNameCS[2] and the guard of the transition between them. That is where the
-- ExpressionInOcl must be attached to the UML model.
-- TBD  when aligning with UML 2.0 Superstructure
```

Inherited attributes

```
--
```

Disambiguating rules

```
-- none
```

constraintContextDeclCS

This production rule represents a context declarations for a definition or constraint, either invariant, or pre- or postcondition.

```
[A] constraintContextDeclCS ::= invariantDeclCS
[B] constraintContextDeclCS ::= definitionDeclCS
[C] constraintContextDeclCS ::= operationDeclCS
```

invariantContextDeclCS

This production rule represents a context declarations for an invariant.

```
invariantContextDeclCS ::= 'context' pathNameCS
                              'inv' (simpleNameCS)? ':' OclExpressionCS
```

Abstract syntax mapping

```
invariantContextDeclCS.ast : Constraint
```

Synthesized attributes

```
let contextualClassifier = Model.findClassifier(pathNameCS.ast) in

invariantContextDeclCS.ast.constrainedElement        = contextualClassifier
invariantContextDeclCS.ast.body.bodyExpression       = OclExpression.ast
invariantContextDeclCS.ast.stereotype.name           = 'invariant'
invariantContextDeclCS.ast.body.contextualClassifier = contextualClassifier
```

Inherited attributes

```
OclExpressionCS.env = invariantContextDeclCS.env->addNameSpace( contextualClassifier)
                                                ->addElement('self',
                                                              contextualClassifier
                                                              true)
```

Disambiguating rules

```
-- none
```

definitionContextDeclCS

This production rule represents a context declarations for a definition.

```
[A] definitionContextDeclCS ::= 'context' pathNameCS
                                'def' (simpleNameCS)? ':'
                                 AttributeDefinitionListCS?
                                 OperationDefinitionListCS?
```

Abstract syntax mapping

```
definitionContextDeclCS.ast : Sequence(Feature)
```

Synthesized attributes

```
let contextualClassifier = Model.findClassifier(pathNameCS.ast) in

definitionContextDeclCS.ast = AttributeDefinitionListCS.ast->union(
                                                      OperationDefinitionCS.ast)
definitionContextDeclCS.ast.owner           = contextualClassifier
definitionContextDeclCS.ast.stereotype.name = 'definition'
```

Inherited attributes

```
AttributeDefinitionListCS.env = definitionContextDeclCS.env->addNameSpace(
                                                            contextualClassifier)
```

```
                                                         ->addElement('self',
                                                                    contextualClassifier
                                                                    true)
  OperationDefinitionListCS.env = definitionContextDeclCS.env->addNameSpace(
                                                                contextualClassifier)
                                                         ->addElement('self',
                                                                    contextualClassifier
                                                                    true)
```

Disambiguating rules

```
  -- none
```

operationContextDeclCS

This production rule represents a context declarations for a pre- or postcondition.

```
[A] operationContextDeclCS ::= 'context' operationCS
                               'pre' (simpleNameCS)? ':' OclExpressionCS
[B] operationContextDeclCS ::= 'context' operationCS
                               'post' (simpleNameCS)? ':' OclExpressionCS
```

Abstract syntax mapping

```
  operationContextDeclCS.ast : Constraint
```

Synthesized attributes

```
  let contextualClassifier = operationCS.ast.owner.oclAstype(Classifier) in

  [A,B] operationContextDeclCS.ast.body.bodyExpression = OclExpression.ast
  -- the operation to which this pre or postcondition is attached
  [A,B] operationContextDeclCS.ast.constrainedElement = operationCS.ast
  -- the stereotype of the constraint
  [A] operationContextDeclCS.ast.stereotype.name = 'precondition'
  [B] operationContextDeclCS.ast.stereotype.name = 'postcondition'
```

Inherited attributes

```
  [A] OclExpressionCS.env   = operationContextDeclCS.env
                              ->addNameSpace(contextualClassifier)
                              ->addElement('self', contextualClassifier,  true)
                              -- add all in and in/out parameters
                              ->addEnvironment(
    operationCS.ast.parameter->iterate(par; answer : Environment = Environment::EMPTY_ENV |
                if par.kind = ParameterDirectionKind::in or
                   par.kind = ParameterDirectionKind::inout
                then
                   answer.addElement(par.name, par.type, false) )
                else
                   answer
                endif )
   )
  [B] OclExpressionCS.env   = operationContextDeclCS.env
                              ->addNameSpace(contextualClassifier)
                              ->addElement ('result', operationCS.returnType, false)
                              ->addElement('self', contextualClassifier, true)
                              -- add all out and in/out parameters
                              ->addEnvironment(
    operationCS.ast.parameter->iterate(par; answer : Environment = Environment::EMPTY_ENV |
                if par.kind = ParameterDirectionKind::out or
                   par.kind = ParameterDirectionKind::inout
                then
                   answer.addElement(par.name, par.type, false) )
                else
                   answer
                endif )
```

```
)
```

Disambiguating rules

```
-- none
```

operationCS

This production rule represents an operation in a context declaration.

```
operationCS ::= pathNameCS '::' simpleNameCS '(' parametersCS? ')' ':'
typeCS?
```

Abstract syntax mapping

```
operationCS.ast       : Operation
```

Synthesized attributes

```
let contextualClassifier = Model.findClassifier(pathNameCS.ast) in

operationCS.ast = contextualClassifier
                              .lookupOperation( simpleNameCS.ast, parametersCS.ast.type)
```

Inherited attributes

```
parametersCS.env = operationCS.env->addNameSpace(contextualClassifier)
                                                        ->addElement('self',
                                                                     contextualClassifier
                                                                     true)
```

Disambiguating rules

```
-- none
```

A SEMANTICS

This section formally defines the syntax and semantics of OCL. Most of the material in this section is based on work presented in [Ric02]. This section is organized as follows. Section A.1 defines the concept of object models. Object models provide information used as context for OCL expressions and constraints. Section A.2 defines the type system of OCL and the set of standard operations. Finally, Section A.3 defines the syntax and semantics of OCL expressions.

A.1 OBJECT MODELS

In this section, the notion of an *object model* is formally defined. An object model provides the context for OCL expressions and constraints. A precise understanding of object models is required before a formal definition of OCL expressions can be given. Section A.1.1 proceeds with a formal definition of the syntax of object models. The semantics of object models is defined in Section A.1.2. This section also defines the notion of system states as snapshots of a running system.

A.1.1 SYNTAX OF OBJECT MODELS

In this section, we formally define the syntax of object models. Such a model has the following components:

- a set of classes,
- a set of attributes for each class,
- a set of operations for each class,
- a set of associations with role names and multiplicities,
- a generalization hierarchy over classes.

Additionally, types such as *Integer*, *String*, *Set*(*Real*) are available for describing types of attributes and operation parameters. In the following, each of the model components is considered in detail. The following definitions are combined in Section A.1.1.7 to give a complete definition of the syntax of object models. For naming model components, we assume in this section an alphabet $\mathcal{A}$ and a set of finite, non-empty names $\mathcal{N} \subseteq \mathcal{A}^+$ over alphabet $\mathcal{A}$ to be given.

A.1.1.1 TYPES

Types are considered in depth in Section A.2. For now, we assume that there is a signature $\Sigma = (T, \Omega)$ with T being a set of type names, and Ω being a set of operations over types in T. The set T includes the basic types *Integer*, *Real*, *Boolean*, and *String*. These are the predefined basic types of OCL. All type domains include an undefined value that allows to operate with unknown or “null” values. Operations in Ω include, for example, the usual arithmetic operations $+, -, *, /$, etc. for integers. Furthermore, collection types are available for describing collections of values, for example, *Set*(*String*), *Bag*(*Integer*), and *Sequence*(*Real*). Structured values are described by tuple types with named components, for example, *Tuple*(*name:String, age:Integer*).

A.1.1.2 CLASSES

The central concept of UML for modeling entities of the problem domain is the class. A class provides a common description for a set of objects sharing the same properties.

DEFINITION A.1 (CLASSES)
The set of classes is a finite set of names $\text{CLASS} \subseteq \mathcal{N}$. □

Each class $c \in \text{CLASS}$ induces an *object type* $t_c \in T$ having the same name as the class. A value of an object type refers to an object of the corresponding class. The main difference between classes and object types is that the interpretation of the latter includes a special undefined value.

Note that for a definition of the semantics of OCL, UML's distinction between classes and interfaces does not matter. OCL specifies constraints for instances of a given interface specification. Whether this specification is stated in the form of a class or interface definition makes no difference.

A.1.1.3 ATTRIBUTES

Attributes are part of a class declaration in UML. Objects are associated with attribute values describing properties of the object. An attribute has a name and a type specifying the domain of attribute values.

DEFINITION A.2 (ATTRIBUTES)
Let $t \in T$ be a type. The attributes of a class $c \in \text{CLASS}$ are defined as a set ATT_c of signatures $a : t_c \rightarrow t$ where the attribute name a is an element of $\mathcal{N}$, and $t_c \in T$ is the type of class c. □

All attributes of a class have distinct names. In particular, an attribute name may not be used again to define another attribute with a different type.

$$\forall t, t' \in T : (a : t_c \rightarrow t \in \text{ATT}_c \text{ and } a : t_c \rightarrow t' \in \text{ATT}_c) \implies t = t'$$

Attributes with the same name may, however, appear in different classes that are not related by generalization. Details are given in Section A.1.1.6 where we discuss generalization. The set of attribute names and class names need not be disjoint.

A.1.1.4 OPERATIONS

Operations are part of a class definition. They are used to describe behavioral properties of objects. The effect of an operation may be specified in a declarative way with OCL pre- and postconditions. Section A.3 discusses pre- and postconditions in detail. Furthermore, operations performing computations without side effects can be specified with OCL. In this case, the computation is determined by an explicit OCL expression. This is also discussed in Section A.3. Here, we focus on the syntax of operation signatures declaring the interface of user-defined operations. In contrast, other kinds of operations which are not explicitly defined by a modeler are, for example, navigation operations derived from associations. These are discussed in the next section and in Section A.2.

DEFINITION A.3 (OPERATIONS)
Let t and $t_1, \ldots, t_n$ be types in T. Operations of a class $c \in \text{CLASS}$ with type $t_c \in T$ are defined by a set OP_c of signatures $\omega : t_c \times t_1 \times \cdots \times t_n \rightarrow t$ with operation symbols ω being elements of $\mathcal{N}$. □

The name of an operation is determined by the symbol ω. The first parameter t_c denotes the type of the class instance to which the operation is applied. An operation may have any number of parameters but only a single return type. In general, UML allows multiple return values. We currently do not support this feature in OCL.

A.1.1.5 ASSOCIATIONS

Associations describe structural relationships between classes. Generally, classes may participate in any number of associations, and associations may connect two or more classes.

DEFINITION A.4 (ASSOCIATIONS)
The set of associations is given by

i. a finite set of names $\text{ASSOC} \subseteq \mathcal{N}$,

ii. a function $\text{associates} : \begin{cases} \text{ASSOC} \to \text{CLASS}^+ \\ as \mapsto \langle c_1, \ldots, c_n \rangle \text{ with } (n \geq 2) \end{cases}$.

□

The function associates maps each association name $as \in \text{ASSOC}$ to a finite list $\langle c_1, \ldots, c_n \rangle$ of classes participating in the association. The number n of participating classes is also called the *degree* of an association; associations with degree n are called n-ary associations. For many problems the use of binary associations is often sufficient. A *self-association* (or recursive association) sa is a binary association where both ends of the association are attached to the same class c such that $\text{associates}(sa) = \langle c, c \rangle$. The function associates does not have to be injective. Multiple associations over the same set of classes are possible.

ROLE NAMES

Classes may appear more than once in an association each time playing a different role. For example, in a self-association PhoneCall on a class *Person* we need to distinguish between the person having the role of a caller and another person being the callee. Therefore we assign each class participating in an association a unique role name. Role names are also important for OCL navigation expressions. A role name of a class is used to determine the navigation path in this kind of expressions.

DEFINITION A.5 (ROLE NAMES)
Let $as \in \text{ASSOC}$ be an association with $\text{associates}(as) = \langle c_1, \ldots, c_n \rangle$. Role names for an association are defined by a function

$$\text{roles} : \begin{cases} \text{ASSOC} \to \mathcal{N}^+ \\ as \mapsto \langle r_1, \ldots, r_n \rangle \text{ with } (n \geq 2) \end{cases}$$

where all role names must be distinct, i.e.,

$$\forall i, j \in \{1, \ldots, n\} : i \neq j \implies r_i \neq r_j \ .$$

□

The function $\text{roles}(as) = \langle r_1, \ldots, r_n \rangle$ assigns each class c_i for $1 \leq i \leq n$ participating in the association a unique role name r_i. If role names are omitted in a class diagram, implicit names are constructed in UML by using the name of the class at the target end and changing its first letter to lower case. As mentioned above, explicit role names are mandatory for self-associations.

Additional syntactical constraints are required for ensuring the uniqueness of role names when a class is part of many associations. We first define a function $\mathrm{participating}$ that gives the set of associations a class participates in.

$$\mathrm{participating} : \begin{cases} \textsc{Class} \to \mathcal{P}(\textsc{Assoc}) \\ c \mapsto \{as \mid as \in \textsc{Assoc} \wedge \mathrm{associates}(as) = \langle c_1, \ldots, c_n \rangle \\ \qquad\qquad \wedge \exists i \in \{1, \ldots, n\} : c_i = c\} \end{cases}$$

The following function $\mathrm{navends}$ gives the set of all role names reachable (or *navigable*) from a class over a given association.

$$\mathrm{navends} : \begin{cases} \textsc{Class} \times \textsc{Assoc} \to \mathcal{P}(\mathcal{N}) \\ (c, as) \mapsto \{r \mid \mathrm{associates}(as) = \langle c_1, \ldots, c_n \rangle \\ \qquad\qquad \wedge \mathrm{roles}(as) = \langle r_1, \ldots, r_n \rangle \\ \qquad\qquad \wedge \exists i, j \in \{1, \ldots, n\} : (i \neq j \wedge c_i = c \wedge r_j = r)\} \end{cases}$$

The set of role names that are reachable from a class along all associations the class participates in can then be determined by the following function.

$$\mathrm{navends}(c) : \begin{cases} \textsc{Class} \to \mathcal{P}(\mathcal{N}) \\ c \mapsto \bigcup_{as \in \mathrm{participating}(c)} \mathrm{navends}(c, as) \end{cases}$$

MULTIPLICITIES

An association specifies the possible existence of links between objects of associated classes. The number of links that an object can be part of is specified with *multiplicities*. A multiplicity specification in UML can be represented by a set of natural numbers.

DEFINITION A.6 (MULTIPLICITIES)
Let $as \in \textsc{Assoc}$ be an association with $\mathrm{associates}(as) = \langle c_1, \ldots, c_n \rangle$. The function $\mathrm{multiplicities}(as) = \langle M_1, \ldots, M_n \rangle$ assigns each class c_i participating in the association a non-empty set $M_i \subseteq \mathbb{N}_0$ with $M_i \neq \{0\}$ for all $1 \leq i \leq n$. □

The precise meaning of multiplicities is defined as part of the interpretation of object models in Section A.1.2.

REMARK: AGGREGATION AND COMPOSITION

Special forms of associations are aggregation and composition. In general, aggregations and compositions impose additional restrictions on relationships. An aggregation is a special kind of binary association representing a *part-of* relationship. The aggregate is marked with a hollow diamond at the association end in class diagrams. An aggregation implies the constraint that an object cannot be part of itself. Therefore, a link of an aggregation may not connect the same object. In case of chained aggregations, the chain may not contain cycles.

An even stronger form of aggregation is composition. The composite is marked with a filled diamond at the association end in class diagrams. In addition to the requirements for aggregations, a part may only belong to at most one composite.

These seemingly simple concepts can have quite complex semantic issues [AFGP96, Mot96, Pri97, GR99, HSB99, BHS99, BHSOG01]. Here, we are concerned only with syntax. The syntax of aggregations and compositions is very similar to associations. Therefore, we do not add an extra concept to our formalism. As a convention, we always use the first component in an association for a class playing the role of an aggregate or composite. The semantic restrictions then have to be expressed as an explicit constraint. A systematic way for mapping aggregations and compositions to simple associations plus OCL constraints is presented in [GR99].

A.1.1.6 GENERALIZATION

A generalization is a taxonomic relationship between two classes. This relationship specializes a general class into a more specific class. Specialization and generalization are different views of the same concept. Generalization relationships form a hierarchy over the set of classes.

DEFINITION A.7 (GENERALIZATION HIERARCHY)
A generalization hierarchy $\prec$ is a partial order on the set of classes CLASS. □

Pairs in $\prec$ describe generalization relationships between two classes. For classes $c_1, c_2 \in \text{CLASS}$ with $c_1 \prec c_2$, the class c_1 is called a *child class* of c_2, and c_2 is called a *parent class* of c_1.

FULL DESCRIPTOR OF A CLASS

A child class implicitly inherits attributes, operations and associations of its parent classes. The set of properties defined in a class together with its inherited properties is called a *full descriptor* in UML. We can formalize the full descriptor in our framework as follows. First, we define a convenience function for collecting all parents of a given class.

$$\text{parents} : \begin{cases} \text{CLASS} \to \mathcal{P}(\text{CLASS}) \\ c \mapsto \{c' \mid c' \in \text{CLASS} \wedge c \prec c'\} \end{cases}$$

The full set of attributes of class c is the set ATT_c^* containing all inherited attributes and those that are defined directly in the class.

$$\text{ATT}_c^* = \text{ATT}_c \cup \bigcup_{c' \in \text{parents}(c)} \text{ATT}_{c'}$$

We define the set of inherited user-defined operations analogously.

$$\text{OP}_c^* = \text{OP}_c \cup \bigcup_{c' \in \text{parents}(c)} \text{OP}_{c'}$$

Finally, the set of navigable role names for a class and all of its parents is given as follows.

$$\text{navends}^*(c) = \text{navends}(c) \cup \bigcup_{c' \in \text{parents}(c)} \text{navends}(c')$$

DEFINITION A.8 (FULL DESCRIPTOR OF A CLASS)
The full descriptor of a class $c \in \text{CLASS}$ is a structure $\text{FD}_c = (\text{ATT}_c^*, \text{OP}_c^*, \text{navends}^*(c))$ containing all attributes, user-defined operations, and navigable role names defined for the class and all of its parents. □

The UML standard requires that properties of a full descriptor must be distinct. For example, a class may not define an attribute that is already defined in one of its parent classes. These constraints are captured more precisely by the following well-formedness rules in our framework. Each constraint must hold for each class $c \in \text{CLASS}$.

1. Attributes are defined in exactly one class.

$$\forall (a : t_c \to t,\ a' : t_{c'} \to t' \in \text{ATT}_c^*) : \\ (a = a' \implies t_c = t_{c'} \wedge t = t') \qquad \textbf{(WF-1)}$$

2. In a full class descriptor, an operation may only be defined once. The first parameter of an operation signature indicates the class in which the operation is defined. The following condition guarantees that each operation in a full class descriptor is defined in a single class.

$$\forall(\omega : t_c \times t_1 \times \cdots \times t_n \to t,\ \omega : t_{c'} \times t_1 \times \cdots \times t_n \to t' \in \mathrm{OP}^*_c) : \\ (t_c = t_{c'}) \qquad \textbf{(WF-2)}$$

3. Role names are defined in exactly one class.

$$\forall c_1, c_2 \in \mathrm{parents}(c) \cup \{c\} : \\ (c_1 \neq c_2 \implies \mathrm{navends}(c_1) \cap \mathrm{navends}(c_2) = \emptyset) \qquad \textbf{(WF-3)}$$

4. Attribute names and role names must not conflict. This is necessary because in OCL the same notation is used for attribute access and navigation by role name. For example, the expression `self.x` may either be a reference to an attribute `x` or a reference to a role name `x`.

$$\forall(a : t_c \to t \in \mathrm{ATT}^*_c) : \forall r \in \mathrm{navends}^*(c) : \\ (a \neq r) \qquad \textbf{(WF-4)}$$

Note that operations may have the same name as attributes or role names because the concrete syntax of OCL allows us to distinguish between these cases. For example, the expression `self.age` is either an attribute or role name reference, but a call to an operation age without parameters is written as `self.age()`.

A.1.1.7 FORMAL SYNTAX

We combine the components introduced in the previous section to formally define the syntax of object models.

DEFINITION A.9 (SYNTAX OF OBJECT MODELS)
The syntax of an object model is a structure

$$\mathcal{M} = (\mathrm{CLASS}, \mathrm{ATT}_c, \mathrm{OP}_c, \mathrm{ASSOC}, \mathrm{associates}, \mathrm{roles}, \mathrm{multiplicities}, \prec)$$

where

i. CLASS is a set of classes (Definition A.1).

ii. ATT_c is a set of operation signatures for functions mapping an object of class c to an associated attribute value (Definition A.2).

iii. OP_c is a set of signatures for user-defined operations of a class c (Definition A.3).

iv. ASSOC is a set of association names (Definition A.4).

   (a) associates is a function mapping each association name to a list of participating classes (Definition A.4).
   (b) roles is a function assigning each end of an association a role name (Definition A.5).
   (c) multiplicities is a function assigning each end of an association a multiplicity specification (Definition A.6).

v. $\prec$ is a partial order on CLASS reflecting the generalization hierarchy of classes (Definitions A.7 and A.8).

□

A.1.2 INTERPRETATION OF OBJECT MODELS

In the previous section, the syntax of object models has been defined. An interpretation of object models is presented in the following.

A.1.2.1 OBJECTS

The domain of a class $c \in \text{CLASS}$ is the set of objects that can be created by this class and all of its child classes. Objects are referred to by unique object identifiers. In the following, we will make no conceptual distinction between objects and their identifiers. Each object is uniquely determined by its identifier and vice versa. Therefore, the actual representation of an object is not important for our purposes.

DEFINITION A.10 (OBJECT IDENTIFIERS)

i. The set of object identifiers of a class $c \in \text{CLASS}$ is defined by an infinite set $\text{oid}(c) = \{\underline{c}_1, \underline{c}_2, \dots\}$.

ii. The domain of a class $c \in \text{CLASS}$ is defined as $I_{\text{CLASS}}(c) = \bigcup\{\text{oid}(c') \mid c' \in \text{CLASS} \wedge c' \preceq c\}$.

□

In the following, we will omit the index for a mapping I when the context is obvious. The concrete scheme for naming objects is not important as long as every object can be uniquely identified, i.e., there are no different objects having the same name. We sometimes use single letters combined with increasing indexes to name objects if it is clear from the context to which class these objects belong.

GENERALIZATION

The above definition implies that a generalization hierarchy induces a subset relation on the semantic domain of classes. The set of object identifiers of a child class is a subset of the set of object identifiers of its parent classes. With other words, we have

$$\forall c_1, c_2 \in \text{CLASS} : c_1 \prec c_2 \implies I(c_1) \subseteq I(c_2) \ .$$

From the perspective of programming languages this closely corresponds to the domain-inclusion semantics commonly associated with subtyping and inheritance [CW85]. Data models for object-oriented databases such as the generic OODB model presented in [AHV95] also assume an inclusion semantics for class extensions. This requirement guarantees two fundamental properties of generalizations. First, an object of a child class has (inherits) all the properties of its parent classes because it *is* an instance of the parent classes. Second, this implies that an object of a more specialized class can be used anywhere where an object of a more general class is expected (principle of substitutability) because it has at least all the properties of the parent classes. In general, the interpretation of classes is pairwise disjoint if two classifiers are not related by generalization and do not have a common child.

A.1.2.2 LINKS

An association describes possible connections between objects of the classes participating in the association. A connection is also called a link in UML terminology. The interpretation of an association is a relation describing the set of all possible links between objects of the associated classes and their children.

DEFINITION A.11 (LINKS)
Each association $as \in \text{ASSOC}$ with $\text{associates}(as) = \langle c_1, \ldots, c_n \rangle$ is interpreted as the Cartesian product of the sets of object identifiers of the participating classes: $I_{\text{ASSOC}}(as) = I_{\text{CLASS}}(c_1) \times \cdots \times I_{\text{CLASS}}(c_n)$. A *link* denoting a connection between objects is an element $l_{as} \in I_{\text{ASSOC}}(as)$. □

A.1.2.3 SYSTEM STATE

Objects, links and attribute values constitute the state of a system at a particular moment in time. A system is in different states as it changes over time. Therefore, a system state is also called a snapshot of a running system. With respect to OCL, we can in many cases concentrate on a single system state given at a discrete point in time. For example, a system state provides the complete context for the evaluation of OCL invariants. For pre- and postconditions, however, it is necessary to consider two consecutive states.

DEFINITION A.12 (SYSTEM STATE)
A system state for a model $\mathcal{M}$ is a structure $\sigma(\mathcal{M}) = (\sigma_{\text{CLASS}}, \sigma_{\text{ATT}}, \sigma_{\text{ASSOC}})$.

i. The finite sets $\sigma_{\text{CLASS}}(c)$ contain all objects of a class $c \in \text{CLASS}$ existing in the system state: $\sigma_{\text{CLASS}}(c) \subset \text{oid}(c)$.

ii. Functions σ_{ATT} assign attribute values to each object: $\sigma_{\text{ATT}}(a) : \sigma_{\text{CLASS}}(c) \to I(t)$ for each $a : t_c \to t \in \text{ATT}^*_c$.

iii. The finite sets σ_{ASSOC} contain links connecting objects. For each $as \in \text{ASSOC}$: $\sigma_{\text{ASSOC}}(as) \subset I_{\text{ASSOC}}(as)$. A link set must satisfy all multiplicity specifications defined for an association (the function $\pi_i(l)$ projects the ith component of a tuple or list l, whereas the function $\bar{\pi}_i(l)$ projects *all but* the ith component):

$$\forall i \in \{1, \ldots, n\}, \forall l \in \sigma_{\text{ASSOC}}(as) :$$
$$|\{l' \mid l' \in \sigma_{\text{ASSOC}}(as) \land (\bar{\pi}_i(l') = \bar{\pi}_i(l))\}| \in \pi_i(\text{multiplicities}(as))$$

□

A.2 OCL TYPES AND OPERATIONS

OCL is a strongly typed language. A type is assigned to every OCL expression and typing rules determine in which ways well-formed expressions can be constructed. In addition to those types introduced by UML models, there are a number of predefined OCL types and operations available for use with any UML model. This section formally defines the type system of OCL. Types and their domains are fixed, and the abstract syntax and semantics of operations is defined.

Our general approach to defining the type system is as follows. Types are associated with a set of operations. These operations describe functions combining or operating on values of the type domains. In our approach, we use a data signature $\Sigma = (T, \Omega)$ to describe the syntax of types and operations. The semantics of types in T and operations in Ω is defined by a mapping that assigns each type a domain and each operation a function. The definition of the syntax and semantics of types and operations will be developed and extended in several steps. At the end of this section, the complete set of types is defined in a single data signature.

Section A.2.1 defines the basic types *Integer*, *Real*, *Boolean* and *String*. Enumeration types are defined in Section A.2.3. Section A.2.4 introduces object types that correspond to classes in a model. Collection and tuple types are discussed in Section A.2.5. The special types *OclAny* and *OclState* are considered in Section A.2.6. Section A.2.7 introduces subtype relationships forming a type hierarchy. All types and operations are finally summarized in a data signature defined in Section A.2.8.

A.2.1 BASIC TYPES

Basic types are *Integer*, *Real*, *Boolean* and *String*. The syntax of basic types and their operations is defined by a signature $\Sigma_B = (T_B, \Omega_B)$. T_B is the set of basic types, Ω_B is the set of signatures describing operations over basic types.

DEFINITION A.13 (SYNTAX OF BASIC TYPES)
The set of basic types T_B is defined as $T_B = \{\mathit{Integer}, \mathit{Real}, \mathit{Boolean}, \mathit{String}\}$. □

Next we define the semantics of basic types by mapping each type to a domain.

DEFINITION A.14 (SEMANTICS OF BASIC TYPES)
Let $\mathcal{A}^*$ be the set of finite sequences of characters from a finite alphabet $\mathcal{A}$. The semantics of a basic type $t \in T_B$ is a function I mapping each type to a set:

- $I(\mathit{Integer}) = \mathbb{Z} \cup \{\bot\}$
- $I(\mathit{Real}) = \mathbb{R} \cup \{\bot\}$
- $I(\mathit{Boolean}) = \{\text{true}, \text{false}\} \cup \{\bot\}$
- $I(\mathit{String}) = \mathcal{A}^* \cup \{\bot\}$.

□

The basic type *Integer* represents the set of integers, *Real* the set of real numbers, *Boolean* the truth values true and false, and *String* all finite strings over a given alphabet. Each domain also contains a special undefined value which is motivated in the next section.

A.2.1.1 ERROR HANDLING

Each domain of a basic type t contains a special value $\bot$. This value represents an undefined value which is useful for two purposes.

1. An undefined value may, for example, be assigned to an attribute of an object. In this case the undefined value helps to model the situation where the attribute value is not yet known (for example, the email address of a customer is unknown at the time of the first contact, but will be added later) or does not apply to this specific object instance (e.g., the customer does not have an email address). This usage of undefined values is well-known in database modeling and querying with SQL [Dat90, EN94]), in the Extended ER-Model [Gog94], and in the object specification language TROLL *light* [Her95].
2. An undefined value can signal an error in the evaluation of an expression. An example for an expression that is defined by a partial function is the division of integers. The result of a division by zero is undefined. The problems with partial functions can be eliminated by including an undefined value $\bot$ into the domains of types. For all operations we can then extend their interpretation to total functions.

The interpretation of operations is considered strict unless there is an explicit statement in the following. Hence, an undefined argument value causes an undefined operation result. This ensures the propagation of error conditions.

A.2.1.2 OPERATIONS

There are a number of predefined operations on basic types. The set Ω_B contains the signatures of these operations. An operation signature describes the name, the parameter types, and the result type of an operation.

DEFINITION A.15 (SYNTAX OF OPERATIONS)
The syntax of an operation is defined by a signature $\omega : t_1 \times \cdots \times t_n \to t$. The signature contains the operation symbol ω, a list of parameter types $t_1, \ldots, t_n \in T$, and a result type $t \in T$. □

Table A.1 shows a schema defining most predefined operations over basic types. The left column contains partially parameterized signatures in Ω_B. The right column specifies variations for the operation symbols or types in the left column.

The set of predefined operations includes the usual arithmetic operations $+, -, *, /$, etc. for integers and real numbers, division (div) and modulo (mod) of integers, sign manipulation ($-$, abs), conversion of *Real* values to *Integer* values (floor, round), and comparison operations ($<, >, \leq, \geq$).

Operations for equality and inequality are presented later in Section A.2.2, since they apply to all types. Boolean values can be combined in different ways (and, or, xor, implies), and they can be negated (not). For strings the length of a string (size) can be determined, a string can be projected to a substring and two strings can be concatenated (concat). Finally, assuming a standard alphabet like ASCII or Unicode, case translations are possible with toUpper and toLower.

Some operation symbols (such as $+$ and $-$) are overloaded, that is there are signatures having the same operation symbol but different parameters (concerning number or type) and possibly different result types. Thus in general, the full argument list has to be considered in order to identify a signature unambiguously.

The operations in Table A.1 all have at least one parameter. There is another set of operations in Ω_B which do not have parameters. These operations are used to produce constant values of basic types. For example, the integer value 12 can be generated by the operation $12 :\to$ *Integer*. Similar operations exist for the other basic types. For each value, there is an operation with no parameters and an operation symbol that corresponds to the common notational representation of this value.

Signature	Schema parameters
$\omega : \mathit{Integer} \times \mathit{Integer} \rightarrow \mathit{Integer}$	$\omega \in \{+, -, *, \max, \min\}$
$\mathit{Integer} \times \mathit{Real} \rightarrow \mathit{Real}$	
$\mathit{Real} \times \mathit{Integer} \rightarrow \mathit{Real}$	
$\mathit{Real} \times \mathit{Real} \rightarrow \mathit{Real}$	
$\omega : \mathit{Integer} \times \mathit{Integer} \rightarrow \mathit{Integer}$	$\omega \in \{\mathrm{div}, \mathrm{mod}\}$
$/ : t_1 \times t_2 \rightarrow \mathit{Real}$	$t_1, t_2 \in \{\mathit{Integer}, \mathit{Real}\}$
$- : t \rightarrow t$	$t \in \{\mathit{Integer}, \mathit{Real}\}$
$\mathrm{abs} : t \rightarrow t$	
$\mathrm{floor} : t \rightarrow \mathit{Integer}$	
$\mathrm{round} : t \rightarrow \mathit{Integer}$	
$\omega : t_1 \times t_2 \rightarrow \mathit{Boolean}$	$\omega \in \{<, >, \leq, \geq\}$, $t_1, t_2 \in \{\mathit{Integer}, \mathit{Real}, \mathit{String}, \mathit{Boolean}\}$
$\omega : \mathit{Boolean} \times \mathit{Boolean} \rightarrow \mathit{Boolean}$	$\omega \in \{\mathrm{and}, \mathrm{or}, \mathrm{xor}, \mathrm{implies}\}$
$\mathrm{not} : \mathit{Boolean} \rightarrow \mathit{Boolean}$	
$\mathrm{size} : \mathit{String} \rightarrow \mathit{Integer}$	
$\mathrm{concat} : \mathit{String} \times \mathit{String} \rightarrow \mathit{String}$	
$\mathrm{toUpper} : \mathit{String} \rightarrow \mathit{String}$	
$\mathrm{toLower} : \mathit{String} \rightarrow \mathit{String}$	
$\mathrm{substring} : \mathit{String} \times \mathit{Integer} \times \mathit{Integer} \rightarrow \mathit{String}$	

Table A.1: Schema for operations on basic types

A.2.1.3 SEMANTICS OF OPERATIONS

DEFINITION A.16 (SEMANTICS OF OPERATIONS)
The semantics of an operation with signature $\omega : t_1 \times \cdots \times t_n \rightarrow t$ is a total function $I(\omega : t_1 \times \cdots \times t_n \rightarrow t) : I(t_1) \times \cdots \times I(t_n) \rightarrow I(t)$. □

When we refer to an operation, we usually omit the specification of the parameter and result types and only use the operation symbol if the full signature can be derived from the context.

The next example shows the interpretation of the operation $+$ for adding two integers. The operation has two arguments $i_1, i_2 \in I(\mathit{Integer})$. This example also demonstrates the strict evaluation semantics for undefined arguments.

$$I(+)(i_1, i_2) = \begin{cases} i_1 + i_2 & \text{if } i_1 \neq \bot \text{ and } i_2 \neq \bot, \\ \bot & \text{otherwise.} \end{cases}$$

We can define the semantics of the other operations in Table A.1 analogously. The usual semantics of the boolean operations and, or, xor, implies, and not, is extended for dealing with undefined argument values. Table A.2 shows the interpretation of boolean operations following the proposal in [CKM+99] based on three-valued logic.

Since the semantics of the other basic operations for *Integer*, *Real*, and *String* values is rather obvious, we will not further elaborate on them here.

b_1	b_2	b_1 and b_2	b_1 or b_2	b_1 xor b_2	b_1 implies b_2	not b_1
false	false	false	false	false	true	true
false	true	false	true	true	true	true
true	false	false	true	true	false	false
true	true	true	true	false	true	false
false	$\perp$	false	$\perp$	$\perp$	true	true
true	$\perp$	$\perp$	true	$\perp$	$\perp$	false
$\perp$	false	false	$\perp$	$\perp$	$\perp$	$\perp$
$\perp$	true	$\perp$	true	$\perp$	true	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Table A.2: Semantics of boolean operations

A.2.2 COMMON OPERATIONS ON ALL TYPES

At this point, we introduce some operations that are defined on all types (including those which are defined in subsequent sections). The equality of values of the same type can be checked with the operation $=_t$: $t \times t \rightarrow$ *Boolean*. Furthermore, the semantics of $=_t$ is defined to be strict. For two values $v_1, v_2 \in I(t)$, we have

$$I(=_t)(v_1, v_2) = \begin{cases} \text{true} & \text{if } v_1 = v_2, \text{ and } v_1 \neq \perp \text{ and } v_2 \neq \perp, \\ \perp & \text{if } v_1 = \perp \text{ or } v_2 = \perp, \\ \text{false} & \text{otherwise.} \end{cases}$$

A test for inequality $\neq_t$: $t \times t \rightarrow$ *Boolean* can be defined analogously. It is also useful to have an operation that allows to check whether an arbitrary value is well-defined or undefined. This can be done with the operations $\text{isDefined}_t : t \rightarrow$ *Boolean* and $\text{isUndefined}_t : t \rightarrow$ *Boolean* for any type $t \in T$. The semantics of these operations is given for any $v \in I(t)$ by:

$$I(\text{isDefined}_t)(v) = (v \neq \perp)$$
$$I(\text{isUndefined}_t)(v) = (v = \perp)$$

A.2.3 ENUMERATION TYPES

Enumeration types are user-defined types. An enumeration type is defined by specifying a name and a set of literals. An enumeration value is one of the literals used for its type definition.

The syntax of enumeration types and their operations is defined by a signature $\Sigma_E = (T_E, \Omega_E)$. T_E is the set of enumeration types and Ω_E the set of signatures describing the operations on enumeration types.

DEFINITION A.17 (SYNTAX OF ENUMERATION TYPES)
An enumeration type $t \in T_E$ is associated with a finite non-empty set of enumeration literals by a function $\text{literals}(t) = \{e_{1_t}, \ldots, e_{n_t}\}$. □

An enumeration type is interpreted by the set of literals used for its declaration.

DEFINITION A.18 (SEMANTICS OF ENUMERATION TYPES)
The semantics of an enumeration type $t \in T_E$ is a function $I(t) = \text{literals}(t) \cup \{\perp\}$. □

A.2.3.1 OPERATIONS

There is only a small number of operations defined on enumeration types: the test for equality or inequality of two enumeration values. The syntax and semantics of these general operations was defined in Section A.2.2 and applies to enumeration types as well.

In addition, the operation $\text{allInstances}_t :\to \mathit{Set}(t)$ is defined for each $t \in T_E$ to return the set of all literals of the enumeration:

$$\forall t \in T_E : I(\text{allInstances}_t()) = \text{literals}(t)$$

A.2.4 OBJECT TYPES

A central part of a UML model are classes that describe the structure of objects in a system. For each class, we define a corresponding object type describing the set of possible object instances. The syntax of object types and their operations is defined by a signature $\Sigma_C = (T_C, \Omega_C)$. T_C is the set of object types, and Ω_C is the set of signatures describing operations on object types.

DEFINITION A.19 (SYNTAX OF OBJECT TYPES)
Let $\mathcal{M}$ be a model with a set CLASS of class names. The set T_C of object types is defined such that for each class $c \in \text{CLASS}$ there is a type $t \in T_C$ having the same name as the class c. □

We define the following two functions for mapping a class to its type and vice versa.

$$\begin{aligned} \text{typeOf} &: \text{CLASS} \to T_C \\ \text{classOf} &: T_C \to \text{CLASS} \end{aligned}$$

The interpretation of classes is used for defining the semantics of object types. The set of object identifiers $I_{\text{CLASS}}(c)$ was introduced in Definition A.10 on page 7.

DEFINITION A.20 (SEMANTICS OF OBJECT TYPES)
The semantics of an object type $t \in T_C$ with $\text{classOf}(t) = c$ is defined as $I(t) = I_{\text{CLASS}}(c) \cup \{\bot\}$. □

In summary, the domain of an object type is the set of object identifiers defined for the class and its children. The undefined value that is only available with the type – not the class – allows us to work with values not referring to any existing object. This is useful, for example, when we have a navigation expression pointing to a class with multiplicity `0..1`. The result of the navigation expression is a value referring to the actual object only if a target object exists. Otherwise, the result is the undefined value.

A.2.4.1 OPERATIONS

There are four different kinds of operations that are specific to object types.

- *Predefined operations*: These are operations which are implicitly defined in OCL for all object types.
- *Attribute operations*: An attribute operation allows access to the attribute value of an object in a given system state.

- *Object operations*: A class may have operations that do not have side effects. These operations are marked in the UML model with the tag *isQuery*. In general, OCL expressions could be used to define object operations. The semantics of an object operation is therefore given by the semantics of the associated OCL expression.
- *Navigation operations*: An object may be connected to other objects via association links. A navigation expression allows to follow these links and to retrieve connected objects.

PREDEFINED OPERATIONS

For all classes $c \in \text{CLASS}$ with object type $t_c = \text{typeOf}(c)$ the operations

$$\text{allInstances}_{t_c} : \rightarrow \textit{Set}(t_c)$$

are in Ω_C. The semantics is defined as

$$I(\text{allInstances}_{t_c} : \rightarrow \textit{Set}(t_c)) = \sigma_{\text{CLASS}}(c) \enspace .$$

This interpretation of allInstances is safe in the sense that its result is always limited to a finite set. The extension of a class is always a finite set of objects.

ATTRIBUTE OPERATIONS

Attribute operations are declared in a model specification by the set ATT_c for each class c. The set contains signatures $a : t_c \rightarrow t$ with a being the name of an attribute defined in the class c. The type of the attribute is t. All attribute operations in ATT_c are elements of Ω_C. The semantics of an attribute operation is a function mapping an object identifier to a value of the attribute domain. An attribute value depends on the current system state.

DEFINITION A.21 (SEMANTICS OF ATTRIBUTE OPERATIONS)
An attribute signature $a : t_c \rightarrow t$ in Ω_C is interpreted by an attribute value function $I_{\text{ATT}}(a : t_c \rightarrow t) : I(t_c) \rightarrow I(t)$ mapping objects of class c to a value of type t.

$$I_{\text{ATT}}(a : t_c \rightarrow t)(\underline{c}) = \begin{cases} \sigma_{\text{ATT}}(a)(\underline{c}) & \text{if } \underline{c} \in \sigma_{\text{CLASS}}(c), \\ \bot & \text{otherwise.} \end{cases}$$

□

Note that attribute functions are defined for all possible objects. The attempt to access an attribute of a non-existent object results in an undefined value.

OBJECT OPERATIONS

Object operations are declared in a model specification. For side effect-free operations the computation can often be described with an OCL expression. The semantics of a side effect-free object operation can then be given by the semantics of the OCL expression associated with the operation. We give a semantics for object operations in Section A.3 when OCL expressions are introduced.

NAVIGATION OPERATIONS

A fundamental concept of OCL is navigation along associations. Navigation operations start from an object of a source class and retrieve all connected objects of a target class. In general, every n-ary association induces a total of $n \cdot (n-1)$ directed navigation operations, because OCL navigation operations only consider two classes of an association at a time. For defining the set of navigation operations of a given class, we have to consider all associations the class is participating in. A corresponding function named $\mathrm{participating}$ was defined on page 4.

DEFINITION A.22 (SYNTAX OF NAVIGATION OPERATIONS)
Let $\mathcal{M}$ be a model

$$\mathcal{M} = (\textsc{Class}, \textsc{Att}_c, \textsc{Op}_c, \textsc{Assoc}, \mathrm{associates}, \mathrm{roles}, \mathrm{multiplicities}, \prec) \enspace .$$

The set $\Omega_{\text{nav}}(c)$ of navigation operations for a class $c \in \textsc{Class}$ is defined such that for each association $as \in \mathrm{participating}(c)$ with $\mathrm{associates}(as) = \langle c_1, \ldots, c_n \rangle$, $\mathrm{roles}(as) = \langle r_1, \ldots, r_n \rangle$, and $\mathrm{multiplicities}(as) = \langle M_1, \ldots, M_n \rangle$ the following signatures are in $\Omega_{\text{nav}}(c)$.

For all $i, j \in \{1, \ldots, n\}$ with $i \neq j$, $c_i = c$, $t_{c_i} = \mathrm{typeOf}(c_i)$, and $t_{c_j} = \mathrm{typeOf}(c_j)$

i. if $n = 2$ and $M_j - \{0, 1\} = \emptyset$ then $r_{j_{(as, r_i)}} : t_{c_i} \rightarrow t_{c_j} \in \Omega_{\text{nav}}(c)$,

ii. if $n > 2$ or $M_j - \{0, 1\} \neq \emptyset$ then $r_{j_{(as, r_i)}} : t_{c_i} \rightarrow \mathit{Set}(t_{c_j}) \in \Omega_{\text{nav}}(c)$.

All navigation operations are elements of Ω_C. □

As discussed in Section A.1, we use unique role names instead of class names for navigation operations in order to avoid ambiguities. The index of the navigation operation name specifies the association to be navigated along as well as the source role name of the navigation path. The result type of a navigation over binary associations is the type of the target class if the multiplicity of the target is given as `0..1` or `1` (i). All other multiplicities allow an object of the source class to be linked with multiple objects of the target class. Therefore, we need a set type to represent the navigation result (ii). Non-binary associations always induce set-valued results since a multiplicity at the target end is interpreted in terms of *all* source objects. However, for a navigation operation, only a single source object is considered.

Navigation operations are interpreted by navigation functions. Such a function has the effect of first selecting all those links of an association where the source object occurs in the link component corresponding to the role of the source class. The resulting links are then projected onto those objects that correspond to the role of the target class.

DEFINITION A.23 (SEMANTICS OF NAVIGATION OPERATIONS)
The set of objects of class c_j linked to an object $\underline{c}_i$ via association as is defined as

$$L(as)(\underline{c}_i) = \{\underline{c}_j \mid (\underline{c}_1, \ldots, \underline{c}_i, \ldots, \underline{c}_j, \ldots, \underline{c}_n) \in \sigma_{\textsc{Assoc}}(as)\}$$

The semantics of operations in $\Omega_{\text{nav}}(c)$ is then defined as

i. $I(r_{j_{(as,r_i)}} : t_{c_i} \rightarrow t_{c_j})(\underline{c}_i) = \begin{cases} \underline{c}_j & \text{if } \underline{c}_j \in L(as)(\underline{c}_i), \\ \perp & \text{otherwise.} \end{cases}$

ii. $I(r_{j_{(as,r_i)}} : t_{c_i} \rightarrow \mathit{Set}(t_{c_j}))(\underline{c}_i) = L(as)(\underline{c}_i)$.

□

A.2.5 COLLECTION AND TUPLE TYPES

We call a type that allows the aggregation of several values into a single value a complex type. OCL provides the complex types *Set*(t), *Sequence*(t), and *Bag*(t) for describing collections of values of type t. There is also a supertype *Collection*(t) which describes common properties of these types. The OCL collection types are homogeneous in the sense that all elements of a collection must be of the same type t. This restriction is slightly relaxed by the substitution rule for subtypes in OCL (see Section A.2.7). The rule says that the actual elements of a collection must have a type which is a subtype of the declared element type. For example, a *Set*(*Person*) may contain elements of type *Customer* or *Employee*.

A.2.5.1 SYNTAX AND SEMANTICS

Since complex types are parameterized types, we define their syntax recursively by means of type expressions.

DEFINITION A.24 (TYPE EXPRESSIONS)
Let $\hat{T}$ be a set of types and $l_1, \ldots, l_n \in \mathcal{N}$ a set of disjoint names. The set of type expressions $T_{\text{Expr}}(\hat{T})$ over $\hat{T}$ is defined as follows.

i. If $t \in \hat{T}$ then $t \in T_{\text{Expr}}(\hat{T})$.

ii. If $t \in T_{\text{Expr}}(\hat{T})$ then $\textit{Set}(t), \textit{Sequence}(t), \textit{Bag}(t) \in T_{\text{Expr}}(\hat{T})$.

iii. If $t \in T_{\text{Expr}}(\hat{T})$ then $\textit{Collection}(t) \in T_{\text{Expr}}(\hat{T})$.

iv. If $t_1, \ldots, t_n \in T_{\text{Expr}}(\hat{T})$ then $\textit{Tuple}(l_1 : t_1, \ldots, l_n : t_n) \in T_{\text{Expr}}(\hat{T})$.

□

The definition says that every type $t \in \hat{T}$ can be used as an element type for constructing a set, sequence, bag, or collection type. The components of a tuple type are marked with labels $l_1, \ldots, l_n$. Complex types may again be used as element types for constructing other complex types. The recursive definition allows unlimited nesting of type expressions.

For the definition of the semantics of type expressions we make the following conventions. Let $\mathcal{F}(S)$ denote the set of all finite subsets of a given set S, S^* is the set of all finite sequences over S, and $\mathcal{B}(S)$ is the set of all finite multisets (bags) over S.

DEFINITION A.25 (SEMANTICS OF TYPE EXPRESSIONS)
Let $\hat{T}$ be a set of types where the domain of each $t \in \hat{T}$ is $I(t)$. The semantics of type expressions $T_{\text{Expr}}(\hat{T})$ over $\hat{T}$ is defined for all $t \in \hat{T}$ as follows.

i. $I(t)$ is defined as given.

ii. $I(\textit{Set}(t)) = \mathcal{F}(I(t)) \cup \{\bot\}$,
$I(\textit{Sequence}(t)) = (I(t))^* \cup \{\bot\}$,
$I(\textit{Bag}(t)) = \mathcal{B}(I(t)) \cup \{\bot\}$.

iii. $I(\textit{Collection}(t)) = I(\textit{Set}(t)) \cup I(\textit{Sequence}(t)) \cup I(\textit{Bag}(t))$.

iv. $I(\textit{Tuple}(l_1 : t_1, \ldots, l_n : t_n)) = I(t_1) \times \cdots \times I(t_n) \cup \{\bot\}$.

□

In this definition, we observe that the interpretation of the type *Collection*(t) subsumes the semantics of the set, sequence and bag type. In OCL, the collection type is described as a supertype of *Set*(t), *Sequence*(t) and *Bag*(t). This construction greatly simplifies the definition of operations having a similar semantics for each of the concrete collection types. Instead of explicitly repeating these operations for each collection type, they are defined once for *Collection*(t). Examples for operations which are "inherited" in this way are the size and includes operations which determine the number of elements in a collection or test for the presence of an element in a collection, respectively.

A.2.5.2 OPERATIONS

CONSTRUCTORS

The most obvious way to create a collection value is by explicitly enumerating its element values. We therefore define a set of generic operations which allow us to construct sets, sequences, and bags from an enumeration of element values. For example, the set $\{1, 2, 5\}$ can be described in OCL by the expression `Set{1,2,5}`, the list $\langle 1, 2, 5\rangle$ by `Sequence{1,2,5}`, and the bag $\{\!\{2, 2, 7\}\!\}$ by `Bag{2,2,7}`. A shorthand notation for collections containing integer intervals can be used by specifying lower and upper bounds of the interval. For example, the expression `Sequence{3..6}` denotes the sequence $\langle 3, 4, 5, 6\rangle$. This is only syntactic sugar because the same collection can be described by explicitly enumerating all values of the interval.

Operations for constructing collection values by enumerating their element values are called *constructors*. For types $t \in T_{\text{Expr}}(\hat{T})$ constructors in $\Omega_{T_{\text{Expr}}(\hat{T})}$ are defined below. A parameter list $t \times \cdots \times t$ denotes n ($n \geq 0$) parameters of the same type t. We define constructors mkSet_t, mkSequence_t, and mkBag_t not only for any type t but also for any finite number n of parameters.

- $\text{mkSet}_t : t \times \cdots \times t \rightarrow \textit{Set}(t)$
- $\text{mkSequence}_t : t \times \cdots \times t \rightarrow \textit{Sequence}(t)$
- $\text{mkBag}_t : t \times \cdots \times t \rightarrow \textit{Bag}(t)$

The semantics of constructors is defined for values $v_1, \dots, v_n \in I(t)$ by the following functions.

- $I(\text{mkSet}_t)(v_1, \dots, v_n) = \{v_1, \dots, v_n\}$
- $I(\text{mkSequence}_t)(v_1, \dots, v_n) = \langle v_1, \dots, v_n\rangle$
- $I(\text{mkBag}_t)(v_1, \dots, v_n) = \{\!\{v_1, \dots, v_n\}\!\}$

A tuple constructor in OCL specifies values and labels for all components, for example, `Tuple{number:3, fruit:'apple', flag:true}`. A constructor for a tuple with component types $t_1, \dots, t_n \in T_{\text{Expr}}(\hat{T})$ ($n \geq 1$) is given in abstract syntax by the following operation.

- $\text{mkTuple} : l_1 : t_1 \times \cdots \times l_n : t_n \rightarrow \textit{Tuple}(l_1 : t_1, \dots, l_n : t_n)$

The semantics of tuple constructors is defined for values $v_i \in I(t_i)$ with $i = 1, \dots, n$ by the following function.

- $I(\text{mkTuple})(l_1 : v_1, \dots, l_n : v_n) = (v_1, \dots, v_n)$

Note that constructors having element values as arguments are deliberately defined not to be strict. A collection value therefore may contain undefined values while still being well-defined.

COLLECTION OPERATIONS

The definition of operations of collection types comprises the set of all predefined collection operations. Operations common to the types *Set*(t), *Sequence*(t), and *Bag*(t) are defined for the supertype *Collection*(t). Table A.3 shows the operation schema for these operations. For all $t \in T_{\text{Expr}}(\hat{T})$, the signatures resulting from instantiating the schema are included in $\Omega_{T_{\text{Expr}}(\hat{T})}$. The right column of the table illustrates the intended set-theoretic interpretation. For this purpose, C, C_1, C_2 are values of type *Collection*(t), and v is a value of type t.

Signature	Semantics
size : *Collection*(t) $\rightarrow$ *Integer*	$\|C\|$
count : *Collection*(t) $\times\, t \rightarrow$ *Integer*	$\|C \cap \{v\}\|$
includes : *Collection*(t) $\times\, t \rightarrow$ *Boolean*	$v \in C$
excludes : *Collection*(t) $\times\, t \rightarrow$ *Boolean*	$v \notin C$
includesAll : *Collection*(t) $\times$ *Collection*(t) $\rightarrow$ *Boolean*	$C_2 \subseteq C_1$
excludesAll : *Collection*(t) $\times$ *Collection*(t) $\rightarrow$ *Boolean*	$C_2 \cap C_1 = \emptyset$
isEmpty : *Collection*(t) $\rightarrow$ *Boolean*	$C = \emptyset$
notEmpty : *Collection*(t) $\rightarrow$ *Boolean*	$C \neq \emptyset$
sum : *Collection*(t) $\rightarrow t$	$\sum_{i=1}^{\|C\|} c_i$

Table A.3: Operations for type *Collection*(t)

The operation schema in Table A.3 can be applied to sets (sequences, bags) by substituting *Set*(t) (*Sequence*(t), *Bag*(t)) for all occurrences of type *Collection*(t). A semantics for the operations in Table A.3 can be easily defined for each of the concrete collection types *Set*(t), *Sequence*(t), and *Bag*(t). The semantics for the operations of *Collection*(t) can then be reduced to one of the three cases of the concrete types because every collection type is either a set, a sequence, or a bag. Consider, for example, the operation count : *Set*(t) $\times\, t \rightarrow$ *Integer* that counts the number of occurrences of an element v in a set s. The semantics of count is

$$I(\text{count} : \mathit{Set}(t) \times t \rightarrow \mathit{Integer})(s, v) = \begin{cases} 1 & \text{if } v \in s, \\ 0 & \text{if } v \notin s, \\ \bot & \text{if } s = \bot. \end{cases}$$

Note that count is not strict. A set may contain the undefined value so that the result of count is 1 if the undefined value is passed as the second argument, for example, count$(\{\bot\}, \bot) = 1$ and count$(\{1\}, \bot) = 0$.

For bags (and very similar for sequences), the meaning of count is

$$\begin{aligned} &I(\text{count} : \mathit{Bag}(t) \times t \rightarrow \mathit{Integer})(\{\!\{v_1, \dots, v_n\}\!\}, v) \\ &= \begin{cases} 0 & \text{if } n = 0, \\ I(\text{count})(\{\!\{v_2, \dots, v_n\}\!\}, v) & \text{if } n > 0 \text{ and } v_1 \neq v, \\ I(\text{count})(\{\!\{v_2, \dots, v_n\}\!\}, v) + 1 & \text{if } n > 0 \text{ and } v_1 = v. \end{cases} \end{aligned}$$

As explained before, the semantics of count for values of type *Collection*(t) can now be defined in terms of the semantics of count for sets, sequences, and bags.

$$\begin{aligned} &I(\text{count} : \mathit{Collection}(t) \times t \rightarrow \mathit{Integer})(c, v) \\ &= \begin{cases} I(\text{count} : \mathit{Set}(t) \times t \rightarrow \mathit{Integer})(c, v) & \text{if } c \in I(\mathit{Set}(t)), \\ I(\text{count} : \mathit{Sequence}(t) \times t \rightarrow \mathit{Integer})(c, v) & \text{if } c \in I(\mathit{Sequence}(t)), \\ I(\text{count} : \mathit{Bag}(t) \times t \rightarrow \mathit{Integer})(c, v) & \text{if } c \in I(\mathit{Bag}(t)), \\ \bot & \text{otherwise.} \end{cases} \end{aligned}$$

SET OPERATIONS

Operations on sets include the operations listed in Table A.3. These are inherited from *Collection*(t). Operations which are specific to sets are shown in Table A.4 where S, S_1, S_2 are values of type *Set*(t), B is a value of type *Bag*(t) and v is a value of type t.

Signature	Semantics
union : *Set*(t) $\times$ *Set*(t) $\rightarrow$ *Set*(t)	$S_1 \cup S_2$
union : *Set*(t) $\times$ *Bag*(t) $\rightarrow$ *Bag*(t)	$S \cup B$
intersection : *Set*(t) $\times$ *Set*(t) $\rightarrow$ *Set*(t)	$S_1 \cap S_2$
intersection : *Set*(t) $\times$ *Bag*(t) $\rightarrow$ *Set*(t)	$S \cap B$
$-$: *Set*(t) $\times$ *Set*(t) $\rightarrow$ *Set*(t)	$S_1 - S_2$
symmetricDifference : *Set*(t) $\times$ *Set*(t) $\rightarrow$ *Set*(t)	$(S_1 \cup S_2) - (S_1 \cap S_2)$
including : *Set*(t) $\times$ t $\rightarrow$ *Set*(t)	$S \cup \{v\}$
excluding : *Set*(t) $\times$ t $\rightarrow$ *Set*(t)	$S - \{v\}$
asSequence : *Set*(t) $\rightarrow$ *Sequence*(t)	
asBag : *Set*(t) $\rightarrow$ *Bag*(t)	

Table A.4: Operations for type *Set*(t)

Note that the semantics of the operation asSequence is nondeterministic. Any sequence containing only the elements of the source set (in arbitrary order) satisfies the operation specification in OCL.

BAG OPERATIONS

Operations for bags are shown in Table A.5. The operation asSequence is nondeterministic also for bags.

Signature	Semantics
union : *Bag*(t) $\times$ *Bag*(t) $\rightarrow$ *Bag*(t)	$B_1 \cup B_2$
union : *Bag*(t) $\times$ *Set*(t) $\rightarrow$ *Bag*(t)	$B \cup S$
intersection : *Bag*(t) $\times$ *Bag*(t) $\rightarrow$ *Bag*(t)	$B_1 \cap B_2$
intersection : *Bag*(t) $\times$ *Set*(t) $\rightarrow$ *Set*(t)	$B \cap S$
including : *Bag*(t) $\times$ t $\rightarrow$ *Bag*(t)	$B \cup \{\!\{v\}\!\}$
excluding : *Bag*(t) $\times$ t $\rightarrow$ *Bag*(t)	$B - \{\!\{v\}\!\}$
asSequence : *Bag*(t) $\rightarrow$ *Sequence*(t)	
asSet : *Bag*(t) $\rightarrow$ *Set*(t)	

Table A.5: Operations for type *Bag*(t)

SEQUENCE OPERATIONS

Sequence operations are displayed in Table A.6. The intended semantics again is shown in the right column of the table. S, S_1, S_2 are sequences occurring as argument values, v is a value of type t, and i, j are arguments of type *Integer*. The length of sequence S is n. The operator $\circ$ denotes the concatenation of lists, $\pi_i(S)$ projects the ith element of a sequence S, and $\pi_{i,j}(S)$ results in a subsequence of S starting with the ith element up to and including the jth element. The result is $\bot$ if an index is out of range. $S - \langle v \rangle$ produces a sequence equal to S but with all elements equal to v removed. Note that the operations append and including are also defined identically in the OCL standard.

Signature		Semantics
union :	*Sequence*(t) × *Sequence*(t) → *Sequence*(t)	$S_1 \circ S_2$
append :	*Sequence*(t) × t → *Sequence*(t)	$S \circ \langle e \rangle$
prepend :	*Sequence*(t) × t → *Sequence*(t)	$\langle e \rangle \circ S$
subSequence :	*Sequence*(t) × *Integer* × *Integer* → *Sequence*(t)	$\pi_{i,j}(S)$
at :	*Sequence*(t) × *Integer* → t	$\pi_i(S)$
first :	*Sequence*(t) → t	$\pi_1(S)$
last :	*Sequence*(t) → t	$\pi_n(S)$
including :	*Sequence*(t) × t → *Sequence*(t)	$S \circ \langle e \rangle$
excluding :	*Sequence*(t) × t → *Sequence*(t)	$S - \langle e \rangle$
asSet :	*Sequence*(t) → *Set*(t)	
asBag :	*Sequence*(t) → *Bag*(t)	

Table A.6: Operations for type *Sequence*(t)

FLATTENING OF COLLECTIONS

Type expressions as introduced in Definition A.24 allow arbitrarily deep nested collection types. We pursue the following approach for giving a precise meaning to collection flattening. First, we keep nested collection types because they do not only make the type system more orthogonal, but they are also necessary for describing the input of the flattening process. Second, we define flattening by means of an explicit function making the effect of the flattening process clear. There may be a shorthand notation omitting the flatten operation in concrete syntax which would expand in abstract syntax to an expression with an explicit flattening function.

Flattening in OCL does apply to all collection types. We have to consider all possible combinations first. Table A.7 shows all possibilities for combining *Set*, *Bag*, and *Sequence* into a nested collection type. For each of the different cases, the collection type resulting from flattening is shown in the right column. Note that the element type t can be any type. In particular, if t is also a collection type the indicated rules for flattening can be applied recursively until the element type of the result is a non-collection type.

Nested collection type	Type after flattening
Set(*Sequence*(t))	*Set*(t)
Set(*Set*(t))	*Set*(t)
Set(*Bag*(t))	*Set*(t)
Bag(*Sequence*(t))	*Bag*(t)
Bag(*Set*(t))	*Bag*(t)
Bag(*Bag*(t))	*Bag*(t)
Sequence(*Sequence*(t))	*Sequence*(t)
Sequence(*Set*(t))	*Sequence*(t)
Sequence(*Bag*(t))	*Sequence*(t)

Table A.7: Flattening of nested collections.

A signature schema for a flatten operation that removes one level of nesting can be defined as

$$\text{flatten} : C_1(C_2(t)) \rightarrow C_1(t)$$

where C_1 and C_2 denote any collection type name *Set*, *Sequence*, or *Bag*. The meaning of the flatten operations can be defined by the following generic iterate expression. The semantics of OCL iterate expressions is defined in Section A.3.1.2.

```
<collection-of-type-C1(C2(t))>->iterate(e1 : C2(t);
  acc1 : C1(t) = C1{} |
  e1->iterate(v : t;
    acc2 : C1(t) = acc1 |
    acc2->including(v)))
```

The following example shows how this expression schema is instantiated for a bag of sets of integers, that is, $C_1 = \textit{Bag}$, $C_2 = \textit{Set}$, and $t = \textit{Integer}$. The result of flattening the value `Bag{Set{3,2},Set{1,2,4}}` is `Bag{1,2,2,3,4}`.

```
Bag{Set{3,2},Set{1,2,4}}->iterate(e1 : Set(Integer);
  acc1 : Bag(Integer) = Bag{} |
  e1->iterate(v : Integer;
    acc2 : Bag(Integer) = acc1 |
    acc2->including(v)))
```

It is important to note that flattening sequences of sets and bags (see the last two rows in Table A.7) is potentially nondeterministic. For these two cases, the flatten operation would have to map each element of the (multi-) set to a distinct position in the resulting sequence, thus imposing an order on the elements which did not exist in the first place. Since there are types (e.g. object types) which do not define an order on their domain elements, there is no obvious mapping for these types. Fortunately, these problematic cases do not occur in standard navigation expressions. Furthermore, these kinds of collections can be flattened if the criteria for ordering the elements is explicitly specified.

TUPLE OPERATIONS

An essential operation for tuple types is the projection of a tuple value onto one of its components. An element of a tuple with labeled components can be accessed by specifying its label.

- $\text{element}_{l_i} : \textit{Tuple}(l_1 : t_1, \ldots, l_i : t_i, \ldots, l_n : t_n) \to t_i$
- $I(\text{element}_{l_i} : \textit{Tuple}(l_1 : t_1, \ldots, l_i : t_i, \ldots, l_n : t_n) \to t_i)(v_1, \ldots, v_i, \ldots, v_n) = v_i$

A.2.6 SPECIAL TYPES

Special types in OCL that do not fit into the categories discussed so far are *OclAny*, *OclState* and *OclVoid*.

- *OclAny* is the supertype of all other types except for the collection types. The exception has been introduced in UML 1.3 because it considerably simplifies the type system [CKM+99]. A simple set inclusion semantics for subtype relationships as proposed in the next section would not be possible due to cyclic domain definitions if *OclAny* were the supertype of *Set*(*OclAny*).
- *OclState* is a type very similar to an enumeration type. It is only used in the operation oclInState for referring to state names in a state machine. There are no operations defined on this type. *OclState* is therefore not treated specially.
- *OclVoid* is the subtype of all other types. The only value of this type is the undefined value. Notice that there is no problem with cyclic domain definitions as $\bot$ is an instance of every type.

DEFINITION A.26 (SPECIAL TYPES)
The set of special types is $T_S = \{\mathit{OclAny}, \mathit{OclVoid}\}$.

Let $\hat{T}$ be the set of basic, enumeration, and object types $\hat{T} = T_B \cup T_E \cup T_C$. The domain of OclAny is given as $I(\mathit{OclAny}) = \left(\bigcup_{t \in \hat{T}} I(t)\right) \cup \{\bot\}$.

The domain of OclVoid is $I(\mathit{OclVoid}) = \{\bot\}$. □

Operations on *OclAny* include equality (=) and inequality (<>) which already have been defined for all types in Section A.2.2. The operations oclIsKindOf, oclIsTypeOf, and oclAsType expect a type as argument. We define them as part of the OCL expression syntax in the next section. The operation oclIsNew is only allowed in postconditions and will be discussed in Section A.3.2.

For *OclVoid*, the constant operation undefined $:\to$ *OclVoid* results in the undefined value $\bot$. The semantics is given by $I(\text{undefined}) = \bot$.

A.2.7 TYPE HIERARCHY

The type system of OCL supports inclusion polymorphism [CW85] by introducing the concept of a *type hierarchy*. The type hierarchy is used to define the notion of *type conformance*. Type conformance is a relationship between two types, expressed by the *conformsTo ()* operation from the abstract syntax metamodel. A valid OCL expression is an expression in which all the types conform. The consequence of type conformance can be loosely stated as: a value of a conforming type *B* may be used wherever a value of type *A* is required.

The type hierarchy reflects the subtype/supertype relationship between types. The following relationships are defined in OCL.

1. *Integer* is a subtype of *Real*.
2. All types, except for the collection and tuple types, are subtypes of *OclAny*.
3. *Set*(t), *Sequence*(t), and *Bag*(t) are subtypes of *Collection*(t).
4. *OclVoid* is subtype of all other types.
5. The hierarchy of types introduced by UML model elements mirrors the generalization hierarchy in the UML model.

Type conformance is a relation which is identical to the subtype relation introduced by the type hierarchy. The relation is reflexive and transitive.

DEFINITION A.27 (TYPE HIERARCHY)
Let T be a set of types and T_C a set of object types with $T_C \subset T$. The relation $\leq$ is a partial order on T and is called the *type hierarchy* over T. The type hierarchy is defined for all $t, t', t'' \in T$ and all $t_c, t'_c \in T_C, n, m \in \mathbb{N}$ as follows.

i. $\leq$ is (a) reflexive, (b) transitive, and (c) antisymmetric:
   (a) $t \leq t$
   (b) $t'' \leq t' \wedge t' \leq t \implies t'' \leq t$
   (c) $t' \leq t \wedge t \leq t' \implies t = t'$.

ii. *Integer* $\leq$ *Real*.

iii. $t \leq$ *OclAny* for all $t \in (T_B \cup T_E \cup T_C)$.

iv. *OclVoid* $\leq t$.

v. *Set*$(t) \leq$ *Collection*(t),
Sequence$(t) \leq$ *Collection*(t), and
Bag$(t) \leq$ *Collection*(t).

vi. If $t' \leq t$ then *Set*$(t') \leq$ *Set*(t), *Sequence*$(t') \leq$ *Sequence*(t), *Bag*$(t') \leq$ *Bag*(t), and
Collection$(t') \leq$ *Collection*(t).

vii. If $t'_i \leq t_i$ for $i = 1, \ldots, n$ and $n \leq m$ then
Tuple$(l_1 : t'_1, \ldots, l_n : t'_n, \ldots, l_m : t'_m) \leq$ *Tuple*$(l_1 : t_1, \ldots, l_n : t_n)$.

viii. If $\mathrm{classOf}(t'_c) \prec \mathrm{classOf}(t_c)$ then $t'_c \leq t_c$.

□

If a type t' is a subtype of another type t (i.e. $t' \leq t$), we say that t' *conforms* to t. Type conformance is associated with the principle of substitutability. A value of type t' may be used wherever a value of type t is expected. This rule is defined more formally in Section A.3.1 which defines the syntax and semantics of expressions.

The principle of substitutability and the interpretation of types as sets suggest that the type hierarchy should be defined as a subset relation on the type domains. Hence, for a type t' being a subtype of t, we postulate that the interpretation of t' is a subset of the interpretation of t. It follows that every operation ω accepting values of type t has the same semantics for values of type t', since $I(\omega)$ is already well-defined for values in $I(t')$:

$$\text{If } t' \leq t \text{ then } I(t') \subseteq I(t) \text{ for all types } t', t \in T.$$

A.2.8 DATA SIGNATURE

We now have available all elements necessary to define the final data signature for OCL expressions. The signature provides the basic set of syntactic elements for building expressions. It defines the syntax and semantics of types, the type hierarchy, and the set of operations defined on types.

DEFINITION A.28 (DATA SIGNATURE)
Let $\hat{T}$ be the set of non-collection types: $\hat{T} = T_B \cup T_E \cup T_C \cup T_S$. The syntax of a data signature over an object model $\mathcal{M}$ is a structure $\Sigma_\mathcal{M} = (T_\mathcal{M}, \leq, \Omega_\mathcal{M})$ where

i. $T_\mathcal{M} = T_{\mathrm{Expr}}(\hat{T})$,

ii. $\leq$ is a type hierarchy over $T_\mathcal{M}$,

iii. $\Omega_\mathcal{M} = \Omega_{T_{\mathrm{Expr}}(\hat{T})} \cup \Omega_B \cup \Omega_E \cup \Omega_C \cup \Omega_S$.

The semantics of $\Sigma_\mathcal{M}$ is a structure $I(\Sigma_\mathcal{M}) = (I(T_\mathcal{M}), I(\leq), I(\Omega_\mathcal{M}))$ where

i. $I(T_\mathcal{M})$ assigns each $t \in T_\mathcal{M}$ an interpretation $I(t)$.

ii. $I(\leq)$ implies for all types $t', t \in T_\mathcal{M}$ that $I(t') \subseteq I(t)$ if $t' \leq t$.

iii. $I(\Omega_\mathcal{M})$ assigns each operation $\omega : t_1 \times \cdots \times t_n \to t \in \Omega_\mathcal{M}$ a total function
$I(\omega) : I(t_1) \times \cdots \times I(t_n) \to I(t)$.

□

A.3 OCL EXPRESSIONS AND CONSTRAINTS

The core of OCL is given by an expression language. Expressions can be used in various contexts, for example, to define constraints such as class invariants and pre-/postconditions on operations. In this section, we formally define the syntax and semantics of OCL expressions, and give precise meaning to notions like context, invariant, and pre-/postconditions.

Section A.3.1 defines the abstract syntax and semantics of OCL expressions and shows how other OCL constructs can be derived from this language core. The context of expressions and other important concepts such as invariants, queries, and shorthand notations are discussed. Section A.3.2 defines the meaning of operation specifications with pre- and postconditions.

A.3.1 EXPRESSIONS

In this section, we define the syntax and semantics of expressions. The definition of expressions is based upon the data signature we developed in the previous section. A data signature $\Sigma_{\mathcal{M}} = (T_{\mathcal{M}}, \leq, \Omega_{\mathcal{M}})$ provides a set of types $T_{\mathcal{M}}$, a relation $\leq$ on types reflecting the type hierarchy, and a set of operations $\Omega_{\mathcal{M}}$. The signature contains the initial set of syntactic elements upon which we build the expression syntax.

A.3.1.1 SYNTAX OF EXPRESSIONS

We define the syntax of expressions inductively so that more complex expressions are recursively built from simple structures. For each expression the set of free occurrences of variables is also defined. Also, each section in the definition corresponds to a subclass of OCLExpression in the abstract syntax. The mapping is indicated.

DEFINITION A.29 (SYNTAX OF EXPRESSIONS)
Let $\Sigma_{\mathcal{M}} = (T_{\mathcal{M}}, \leq, \Omega_{\mathcal{M}})$ be a data signature over an object model $\mathcal{M}$. Let $\text{Var} = \{\text{Var}_t\}_{t \in T_{\mathcal{M}}}$ be a family of variable sets where each variable set is indexed by a type t. The syntax of expressions over the signature $\Sigma_{\mathcal{M}}$ is given by a set $\text{Expr} = \{\text{Expr}_t\}_{t \in T_{\mathcal{M}}}$ and a function $\text{free} : \text{Expr} \to \mathcal{F}(\text{Var})$ that are defined as follows.

i. If $v \in \text{Var}_t$ then $\boldsymbol{v} \in \text{Expr}_t$ and $\text{free}(v) := \{v\}$. This maps into the **VariableExp** class in the abstract syntax.

ii. If $v \in \text{Var}_{t_1}, e_1 \in \text{Expr}_{t_1}, e_2 \in \text{Expr}_{t_2}$ then **let** $\boldsymbol{v = e_1}$ **in** $\boldsymbol{e_2} \in \text{Expr}_{t_2}$ and $\text{free}(\text{let } v = e_1 \text{ in } e_2) := \text{free}(e_2) - \{v\}$. This maps into **LetExpression** in the abstract syntax. $v = e_1$ is the **VariableDeclaration** referred through the *variable* association; e_2 is the **OclExpression** referred through association end *in*. e_1 is the **OclExpression** referred from the **VariableDeclaration** through the *initExpression* association.

iii. (a) If $t \in T_{\mathcal{M}}$ and $\omega :\to t \in \Omega_{\mathcal{M}}$ then $\boldsymbol{\omega} \in \text{Expr}_t$ and $\mathbf{undefined} \in \text{Expr}_{\mathit{OclVoid}}$ and $\text{free}(\omega) := \emptyset$ and $\text{free}(\text{undefined}) := \emptyset$. This maps into the **ConstantExp** class and its subclasses from the abstract syntax.

(b) If $\omega : t_1 \times \cdots \times t_n \to t \in \Omega_{\mathcal{M}}$ and $e_i \in \text{Expr}_{t_i}$ for all $i = 1, \ldots, n$ then $\boldsymbol{\omega(e_1, \ldots, e_n)} \in \text{Expr}_t$ and $\text{free}(\omega(e_1, \ldots, e_n)) := \text{free}(e_1) \cup \cdots \cup \text{free}(e_n)$. This maps into **ModelPropertyCallExp** and its subclasses, with e_1 representing the *source* and e_2 to e_n the *arguments*.

iv. If $e_1 \in \text{Expr}_{\text{Boolean}}$ and $e_2, e_3 \in \text{Expr}_t$ then **if** $\boldsymbol{e_1}$ **then** $\boldsymbol{e_2}$ **else** $\boldsymbol{e_3}$ **endif** $\in \text{Expr}_t$ and $\text{free}(\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ endif}) := \text{free}(e_1) \cup \text{free}(e_2) \cup \text{free}(e_3)$. This corresponds to the **IfExpression** in the abstract syntax. e_1 is the **OclExpression** referred through *condition*, e_2 corresponds to the *thenExpression* association, and e_3 maps into the **OclExpression** *elseExpression*.

v. If $e \in \text{Expr}_t$ and $t' \leq t$ or $t \leq t'$ then $(\boldsymbol{e}$ **asType** $\boldsymbol{t'}) \in \text{Expr}_{t'}$, $(\boldsymbol{e}$ **isTypeOf** $\boldsymbol{t'}) \in \text{Expr}_{\text{Boolean}}$, $(\boldsymbol{e}$ **isKindOf** $\boldsymbol{t'}) \in \text{Expr}_{\text{Boolean}}$ and $\text{free}((e \text{ asType } t')) := \text{free}(e)$, $\text{free}((e \text{ isTypeOf } t')) := \text{free}(e)$, $\text{free}((e \text{ isKindOf } t')) := \text{free}(e)$. This maps into some special instances of **OclOperationWithTypeArgument**.

vi. If $e_1 \in \text{Expr}_{\mathit{Collection}(t_1)}$, $v_1 \in \text{Var}_{t_1}, v_2 \in \text{Var}_{t_2}$, and $e_2, e_3 \in \text{Expr}_{t_2}$ then $\boldsymbol{e_1 \rightarrow \textbf{iterate}(v_1; v_2 = e_2 \mid e_3)} \in \text{Expr}_{t_2}$ and $\text{free}(e_1 \rightarrow \text{iterate}(v_1; v_2 = e_2 \mid e_3)) := (\text{free}(e_1) \cup \text{free}(e_2) \cup \text{free}(e_3)) - \{v_1, v_2\}$. This is a representation of the **IterateExp**. e_1 is the *source*, $v_2 = e2$ is the **VariableDeclaration** which is referred to through the *result* association in the abstract syntax. v_1 corresponds to the *iterator* **VariableDeclaration**. Finally, e_3 is the **OclExpression** *body*. Instances of **IteratorExp** are defined in the OCL Standard Library.

An expression of type t' is also an expression of a more general type t. For all $t' \leq t$: if $e \in \text{Expr}_{t'}$ then $e \in \text{Expr}_t$. □

A variable expression (i) refers to the value of a variable. Variables (including the special variable `self`) may be introduced by the context of an expression, as part of an iterate expression, and by a let expression. Let expressions (ii) do not add to the expressiveness of OCL but help to avoid repetitions of common sub-expressions. Constant expressions (iiia) refer to a value from the domain of a type. Operation expressions (iiib) apply an operation from $\Omega_{\mathcal{M}}$. The set of operations includes:

- predefined data operations: `+`, `-`, `*`, `<`, `>`, `size`, `max`
- attribute operations: `self.age`, `e.salary`
- side effect-free operations defined by a class:
  `b.rentalsForDay(...)`
- navigation by role names: `self.employee`

As demonstrated by the examples, an operation expression may also be written in OCL path syntax as $e_1.\omega(e_2, \ldots, e_n)$. This notational style is common in many object-oriented languages. It emphasizes the role of the first argument as the “receiver” of a “message”. If e_1 denotes a collection value, an arrow symbol is used in OCL instead of the period: $e_1 \rightarrow \omega(e_2, \ldots, e_n)$. Collections may be bags, sets, or lists.

An if-expression (iv) provides an alternative selection of two expressions depending on the result of a condition given by a boolean expression.

An asType expression (v) can be used in cases where static type information is insufficient. It corresponds to the `oclAsType` operation in OCL and can be understood as a cast of a source expression to an equivalent expression of a (usually) more specific target type. The target type must be related to the source type, that is, one must be a subtype of the other. The isTypeOf and isKindOf expressions correspond to the `oclIsTypeOf` and `oclIsKindOf` operations, respectively. An expression $(e \text{ isTypeOf } t')$ can be used to test whether the type of the value resulting from the expression e has the type t' given as argument. An isKindOf expression $(e \text{ isKindOf } t')$ is not as strict in that it is sufficient for the expression to become true if t' is a supertype of the type of the value of e. Note that in previous OCL versions these type casts and tests were defined as operations with parameters of type *OclType*. Here, we technically define them as first class expressions which has the benefit that we do not need the metatype *OclType*. Thus the type system is kept simple while preserving compatibility with standard OCL syntax.

An iterate expression (vi) is a general loop construct which evaluates an argument expression e_3 repeatedly for all elements of a collection which is given by a source expression e_1. Each element of the collection is bound in turn to the variable v_1 for each evaluation of the argument expression. The argument expression e_3 may contain the variable v_1 to refer to the current element of the collection. The result variable v_2 is initialized with the expression e_2. After each evaluation of the argument expression e_3, the result is bound to the variable v_2. The final value of v_2 is the result of the whole iterate expression.

The iterate construct is probably the most important kind of expression in OCL. Many other OCL constructs (such as `select`, `reject`, `collect`, `exists`, `forAll`, and `isUnique`) can be equivalently defined in terms of an iterate expression (see Section A.3.1.3).

Following the principle of substitutability, the syntax of expressions is defined such that wherever an expression $e \in \mathrm{Expr}_t$ is expected as part of another expression, an expression with a more special type $t', (t' \leq t)$ may be used. In particular, operation arguments and variable assignments in let and iterate expressions may be given by expressions of more special types.

A.3.1.2 SEMANTICS OF EXPRESSIONS

The semantics of expressions is made precise in the following definition. A context for evaluation is given by an environment $\tau = (\sigma, \beta)$ consisting of a system state σ and a variable assignment $\beta : \mathrm{Var}_t \to I(t)$. A system state σ provides access to the set of currently existing objects, their attribute values, and association links between objects. A variable assignment β maps variable names to values.

DEFINITION A.30 (SEMANTICS OF EXPRESSIONS)
Let Env be the set of environments $\tau = (\sigma, \beta)$. The semantics of an expression $e \in \mathrm{Expr}_t$ is a function $I[\![e]\!] : \mathrm{Env} \to I(t)$ that is defined as follows.

i. $I[\![v]\!](\tau) = \beta(v)$.

ii. $I[\![\text{let } v = e_1 \text{ in } e_2]\!](\tau) = I[\![e_2]\!](\sigma, \beta\{v/I[\![e_1]\!](\tau)\})$.

iii. $I[\![\text{undefined}]\!](\tau) = \bot$ and $I[\![\omega]\!](\tau) = I(\omega)$

iv. $I[\![\omega(e_1, \dots, e_n)]\!](\tau) = I(\omega)(\tau)(I[\![e_1]\!](\tau), \dots, I[\![e_n]\!](\tau))$.

v. $I[\![\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ endif}]\!](\tau) = \begin{cases} I[\![e_2]\!](\tau) & \text{if } I[\![e_1]\!](\tau) = \text{true}, \\ I[\![e_3]\!](\tau) & \text{if } I[\![e_1]\!](\tau) = \text{false}, \\ \bot & \text{otherwise.} \end{cases}$

vi. $I[\![(e \text{ asType } t')]\!](\tau) = \begin{cases} I[\![e]\!](\tau) & \text{if } I[\![e]\!](\tau) \in I(t'), \\ \bot & \text{otherwise.} \end{cases}$

$I[\![(e \text{ isTypeOf } t')]\!](\tau) = \begin{cases} \text{true} & \text{if } I[\![e]\!](\tau) \in I(t') - \bigcup_{t'' < t'} I(t''), \\ \text{false} & \text{otherwise.} \end{cases}$

$I[\![(e \text{ isKindOf } t')]\!](\tau) = \begin{cases} \text{true} & \text{if } I[\![e]\!](\tau) \in I(t'), \\ \text{false} & \text{otherwise.} \end{cases}$

vii. $I[\![e_1 \to \text{iterate}(v_1; v_2 = e_2 \mid e_3)]\!](\tau) = I[\![e_1 \to \text{iterate}'(v_1 \mid e_3)]\!](\tau')$ where $\tau' = (\sigma, \beta')$ and $\tau'' = (\sigma, \beta'')$ are environments with modified variable assignments

$$\beta' := \beta\{v_2/I[\![e_2]\!](\tau)\}$$
$$\beta'' := \beta'\{v_2/I[\![e_3]\!](\sigma, \beta'\{v_1/x_1\})\}$$

and iterate$'$ is defined as:

(a) If $e_1 \in \text{Expr}_{\textit{Sequence}(t_1)}$ then

$$I[\![\, e_1 \rightarrow \text{iterate}'(v_1 \mid e_3) \,]\!](\tau') = \begin{cases} I[\![\, v_2 \,]\!](\tau') \\ \quad \text{if } I[\![\, e_1 \,]\!](\tau') = \langle\rangle, \\ I[\![\, \text{mkSequence}_{t_1}(x_2, \dots, x_n) \rightarrow \text{iterate}'(v_1 \mid e_3) \,]\!](\tau'') \\ \quad \text{if } I[\![\, e_1 \,]\!](\tau') = \langle x_1, \dots, x_n \rangle. \end{cases}$$

(b) If $e_1 \in \text{Expr}_{\textit{Set}(t_1)}$ then

$$I[\![\, e_1 \rightarrow \text{iterate}'(v_1 \mid e_3) \,]\!](\tau') = \begin{cases} I[\![\, v_2 \,]\!](\tau') \\ \quad \text{if } I[\![\, e_1 \,]\!](\tau') = \emptyset, \\ I[\![\, \text{mkSet}_{t_1}(x_2, \dots, x_n) \rightarrow \text{iterate}'(v_1 \mid e_3) \,]\!](\tau'') \\ \quad \text{if } I[\![\, e_1 \,]\!](\tau') = \{x_1, \dots, x_n\}. \end{cases}$$

(c) If $e_1 \in \text{Expr}_{\textit{Bag}(t_1)}$ then

$$I[\![\, e_1 \rightarrow \text{iterate}'(v_1 \mid e_3) \,]\!](\tau') = \begin{cases} I[\![\, v_2 \,]\!](\tau') \\ \quad \text{if } I[\![\, e_1 \,]\!](\tau') = \emptyset, \\ I[\![\, \text{mkBag}_{t_1}(x_2, \dots, x_n) \rightarrow \text{iterate}'(v_1 \mid e_3) \,]\!](\tau'') \\ \quad \text{if } I[\![\, e_1 \,]\!](\tau') = \{\!\{x_1, \dots, x_n\}\!\}. \end{cases}$$

□

The semantics of a variable expression (i) is the value assigned to the variable. A let expression (ii) results in the value of the sub-expression e_2. Free occurrences of the variable v in e_2 are bound to the value of the expression e_1. An operation expression (iv) is interpreted by the function associated with the operation. Each argument expression is evaluated separately. The state σ is passed to operations whose interpretation depends on the system state. These include, for example, attribute and navigation operations as defined in Section A.2.4.

The computation of side effect-free operations can often be described with OCL expressions. We can extend the definition to allow object operations whose effects are defined in terms of OCL expressions. The semantics of a side effect-free operation can then be given by the semantics of the OCL expression associated with the operation. Recall that object operations in $\text{O}\textsc{p}_c$ are declared in a model specification. Let $\text{oclexp} : \text{O}\textsc{p}_c \rightarrow \text{Expr}$ be a partial function mapping object operations to OCL expressions. We define the semantics of an operation with an associated OCL expression as

$$I[\![\, \omega(p_1 : e_1, \dots, p_n : e_n) \,]\!](\tau) = I[\![\, \text{oclexp}(\omega) \,]\!](\tau')$$

where $p_1, \dots, p_n$ are parameter names, and $\tau' = (\sigma, \beta')$ denotes an environment with a modified variable assignment defined as

$$\beta' := \beta\{p_1/I[\![\, e_1 \,]\!](\tau), \dots, p_n/I[\![\, e_n \,]\!](\tau)\} \ .$$

Argument expressions are evaluated and assigned to parameters that bind free occurrences of $p_1, \dots, p_n$ in the expression $\text{oclexp}(\omega)$. For a well-defined semantics, we need to make sure that there is no infinite recursion resulting from an expansion of the operation call. A strict solution that can be statically checked is to forbid any occurrences of ω in $\text{oclexp}(\omega)$. However, allowing recursive operation calls considerably adds to the expressiveness of OCL. We therefore allow recursive invocations as long as the recursion is finite. Unfortunately, this property is generally undecidable.

The result of an if-expression (v) is given by the then-part if the condition is true. If the condition is false, the else-part is the result of the expression. An undefined condition makes the whole expression undefined. Note that when an expression in one of the alternative branches is undefined, the whole expression may still have a well-defined result. For example, the result of the following expression is 1.

```
if true then 1 else 1 div 0 endif
```

The result of a cast expression (vi) using asType is the value of the expression, if the value lies within the domain of the specified target type, otherwise it is undefined. A type test expression with isTypeOf is true if the expression value lies exactly within the domain of the specified target type without considering subtypes. An isKindOf type test expression is true if the expression value lies within the domain of the specified target type or one of its subtypes. Note that these type cast and test expressions also work with undefined values since every value – including an undefined one – has a well-defined type.

An iterate expression (vii) loops over the elements of a collection and allows the application of a function to each collection element. The function results are successively combined into a value that serves as the result of the whole iterate expression. This kind of evaluation is also known in functional style programming languages as *fold* operation (see, e.g., [Tho99]).

In Definition A.30, the semantics of iterate expressions is given by a recursive evaluation scheme. Information is passed between different levels of recursion by modifying the variable assignment β appropriately in each step. The interpretation of iterate starts with the initialization of the accumulator variable. The recursive evaluation following thereafter uses a simplified version of iterate, namely an expression iterate$'$ where the initialization of the accumulator variable is left out, since this sub-expression needs to be evaluated only once. If the source collection is not empty, (1) an element from the collection is bound to the iteration variable, (2) the argument expression is evaluated, and (3) the result is bound to the accumulator variable. These steps are all part of the definition of the variable assignment β''. The recursion terminates when there are no more elements in the collection to iterate over. The constructor operations mkSequence_t, mkBag_t, and mkSet_t (see page 17) are in $\Omega_{\mathcal{M}}$ and provide the abstract syntax for collection literals like `Set{1,2}` in concrete OCL syntax.

The result of an iterate expression applied to a set or bag is deterministic only if the inner expression is both commutative and associative.

A.3.1.3 DERIVED EXPRESSIONS BASED ON ITERATE

A number of important OCL constructs such as `exists`, `forAll`, `select`, `reject`, `collect`, and `isUnique` are defined in terms of iterate expressions. The following schema shows how these expressions can be translated to equivalent iterate expressions. A similar translation can be found in [Cla99].

$$I[\![\, e_1 \rightarrow \text{exists}(v_1 \mid e_3)\,]\!](\tau) = \\ \quad I[\![\, e_1 \rightarrow \text{iterate}(v1; v2 = \text{false} \mid v_2 \text{ or } e_3)\,]\!](\tau)$$

$$I[\![\, e_1 \rightarrow \text{forAll}(v_1 \mid e_3)\,]\!](\tau) = \\ \quad I[\![\, e_1 \rightarrow \text{iterate}(v1; v2 = \text{true} \mid v_2 \text{ and } e_3)\,]\!](\tau)$$

$$I[\![\, e_1 \rightarrow \text{select}(v_1 \mid e_3)\,]\!](\tau) = \\ \quad I[\![\, e_1 \rightarrow \text{iterate}(v1; v2 = e_1 \mid \\ \qquad \text{if } e_3 \text{ then } v_2 \text{ else } v_2 \rightarrow \text{excluding}(v_1) \text{ endif})\,]\!](\tau)$$

$$I[\![\, e_1 \rightarrow \text{reject}(v_1 \mid e_3)\,]\!](\tau) = \\ \quad I[\![\, e_1 \rightarrow \text{iterate}(v1; v2 = e_1 \mid \\ \qquad \text{if } e_3 \text{ then } v_2 \rightarrow \text{excluding}(v_1) \text{ else } v_2 \text{ endif})\,]\!](\tau)$$

$$I[\![\, e_1 \rightarrow \text{collect}(v_1 \mid e_3)\,]\!](\tau) = \\ \quad I[\![\, e_1 \rightarrow \text{iterate}(v1; v2 = \text{mkBag}_{\textit{type-of-}e_3}() \mid v_2 \rightarrow \text{including}(e_3))\,]\!](\tau)$$

$$I[\![\, e_1 \rightarrow \text{isUnique}(v_1 \mid e_3)\,]\!](\tau) =$$

$$I[\![\, e_1 \rightarrow \text{iterate}(v1; v2 = \text{true} \mid v_2 \text{ and } e_1 \rightarrow \text{count}(v_1) = 1) \,]\!](\tau)$$

A.3.1.4 EXPRESSION CONTEXT

An OCL expression is always written in some syntactical context. Since the primary purpose of OCL is the specification of constraints on a UML model, it is obvious that the model itself provides the most general kind of context. In our approach, the signature $\Sigma_{\mathcal{M}}$ contains types (e.g., object types) and operations (e.g., attribute operations) that are "imported" from a model, thus providing a context for building expressions that depend on the elements of a specific model.

On a much smaller scale, there is also a notion of context in OCL that simply introduces variable declarations. This notion is closely related to the syntax for constraints written in OCL. A context clause declares variables in invariants, and parameters in pre- and postconditions.

A *context of an invariant* is a declaration of variables. The variable declaration may be implicit or explicit. In the implicit form, the context is written as

```
context C inv:
  <expression>
```

In this case, the `<expression>` may use the variable `self` of type C as a free variable. In the explicit form, the context is written as

```
context v1 : C1, ..., vn : Cn inv:
  <expression>
```

The `<expression>` may use the variables $v_1, \ldots, v_n$ of types $C_1, \ldots, C_n$ as free variables.

A *context of a pre-/postcondition* is a declaration of variables. In this case, the context is written as

```
context C :: op(p1 : T1, ..., pn : Tn) : T
  pre: P
  post: Q
```

This means that the variable `self` (of type *C*) and the parameters $p_1, \ldots, p_n$ may be used as free variables in the precondition P and the postcondition Q. Additionally, the postcondition may use `result` (of type *T*) as a free variable. The details are explained in Section A.3.2.

A.3.1.5 INVARIANTS

An invariant is an expression with boolean result type and a set of (explicitly or implicitly declared) free variables $v_1 : C_1, \ldots, v_n : C_n$ where $C_1, \ldots, C_n$ are classifier types. An invariant

```
context v1 : C1, ..., vn : Cn inv:
  <expression>
```

is equivalent to the following expression without free variables that must be valid in all system states.

```
C1.allInstances->forAll(v1 : C1 |
   ...
   Cn.allInstances->forAll(vn : Cn |
     <expression>
   )
   ...
)
```

A system state is called valid with respect to an invariant if the invariant evaluates to true. Invariants with undefined result invalidate a system state.

A.3.2 PRE- AND POSTCONDITIONS

The definition of expressions in the previous section is sufficient for invariants and queries where we have to consider only single system states. For pre- and postconditions, there are additional language constructs in OCL which enable references to the system state before the execution of an operation and to the system state that results from the operation execution. The general syntax of an operation specification with pre- and postconditions is defined as

```
context C :: op(p1 : T1, ..., pn : Tn)
  pre: P
  post: Q
```

First, the context is determined by giving the signature of the operation for which pre- and postconditions are to be specified. The operation `op` which is defined as part of the classifier *C* has a set of typed parameters $\text{PARAMS}_{\text{op}} = \{p_1, \ldots, p_n\}$. The UML model providing the definition of an operation signature also specifies the direction kind of each parameter. We use a function $kind : \text{PARAMS}_{\text{op}} \rightarrow \{\text{in}, \text{out}, \text{inout}, \text{return}\}$ to map each parameter to one of these kinds. Although UML makes no restriction on the number of return parameters, there is usually only at most one return parameter considered in OCL which is referred to by the keyword `result` in a postcondition. In this case, the signature is also written as $C :: \texttt{op}(p_1 : T_1, \ldots, p_{n-1} : T_{n-1}) : T$ with T being the type of the `result` parameter.

The precondition of the operation is given by an expression P, and the postcondition is specified by an expression Q. P and Q must have a boolean result type. If the precondition holds, the contract of the operation guarantees that the postcondition is satisfied after completion of `op`. Pre- and postconditions form a pair. A condition defaults to true if it is not explicitly specified.

A.3.2.1 EXAMPLE

Before we give a formal definition of operation specifications with pre- and postconditions, we demonstrate the fundamental concepts by means of an example. Figure A.1 shows a class diagram with two classes *A* and *B* that are related to each other by an association R. Class *A* has an operation `op()` but no attributes. Class *B* has an attribute c and no operations. The implicit role names `a` and `b` at the link ends allow navigation in OCL expressions from a *B* object to the associated *A* object and vice versa.

Figure A.2 shows an example for two consecutive states of a system corresponding to the given class model. The object diagrams show instances of classes *A* and *B* and links of the association R. The left object diagram shows the state before the execution of an operation, whereas the right diagram shows the state after the operation has been executed. The effect of the operation can be described by the following changes in the post-state: (1) the

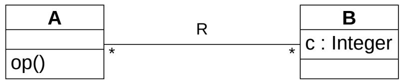
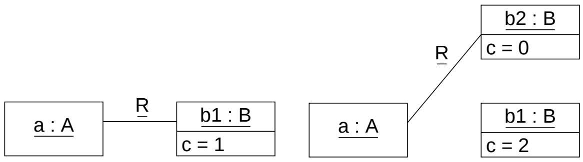


Figure A.1: Example class diagram



(a) Pre-state with objects $\underline{a}$ and $\underline{b}_1$.

(b) Post-state. Object $\underline{b}_2$ did not exist in the pre-state.

Figure A.2: Object diagrams showing a pre- and a post-state

value of the attribute c in object $\underline{b}_1$ has been incremented by one, (2) a new object $\underline{b}_2$ has been created, (3) the link between $\underline{a}$ and $\underline{b}_1$ has been removed, and (4) a new link between $\underline{a}$ and $\underline{b}_2$ has been established.

For the following discussion, consider the OCL expression `a.b.c` where `a` is a variable denoting the object $\underline{a}$. The expression navigates to the associated object of class B and results in the value of the attribute c. Therefore, the expression evaluates to 1 in the pre-state shown in Figure A.2(a). As an example of how the OCL modifier `@pre` may be used in a postcondition to refer to properties of the previous state, we now look at some variations of the expression `a.b.c` that may appear as part of a postcondition. For each case, the result is given and explained.

- `a.b.c = 0`
  Because the expression is completely evaluated in the post-state, the navigation from $\underline{a}$ leads to the $\underline{b}_2$ object. The value of the attribute c of $\underline{b}_2$ is 0 in Figure A.2(b).

- `a.b@pre.c = 2`
  This expression refers to both the pre- and the post-state. The previous value of `a.b` is a reference to object $\underline{b}_1$. However, since the `@pre` modifier only applies to the expression `a.b`, the following reference to the attribute c is evaluated in the post-state of $\underline{b}_1$, even though $\underline{b}_1$ is not connected anymore to $\underline{a}$. Therefore, the result is 2.

- `a.b@pre.c@pre = 1`
  In this case, the value of the attribute c of object $\underline{b}_1$ is taken from the pre-state. This expression is semantically equivalent to the expression `a.b.c` in a precondition.

- `a.b.c@pre =` $\perp$
  The expression `a.b` evaluated in the post-state yields a reference to object $\underline{b}_2$ which is now connected to $\underline{a}$. Since $\underline{b}_2$ has just been created by the operation, there is no previous state of $\underline{b}_2$. Hence, a reference to the previous value of attribute c is undefined.

Note that the `@pre` modifier may only be applied to operations not to arbitrary expressions. An expression such as `(a.b)@pre` is syntactically illegal.

OCL provides the standard operation `oclIsNew` for checking whether an object has been created during the execution of an operation. This operation may only be used in postconditions. For our example, the following conditions indicate that the object $\underline{b}_2$ has just been created in the post-state and $\underline{b}_1$ already existed in the pre-state.

- `a.b.oclIsNew = true`
- `a.b@pre.oclIsNew = false`

A.3.2.2 SYNTAX AND SEMANTICS OF POSTCONDITIONS

All common OCL expressions can be used in a precondition P. Syntax and semantics of preconditions are defined exactly like those for plain OCL expressions in Section A.3.1. Also, all common OCL expressions can be used in a postcondition Q. Additionally, the `@pre` construct, the special variable `result`, and the operation `oclIsNew` may appear in a postcondition. In the following, we extend Definition A.29 for the syntax of OCL expressions to provide these additional features.

DEFINITION A.31 (SYNTAX OF EXPRESSIONS IN POSTCONDITIONS)
Let op be an operation with a set of parameters $\text{PARAMS}_{\text{op}}$. The set of parameters includes at most one parameter of kind "return". The basic set of expressions in postconditions is defined by repeating Definition A.29 while substituting all occurrences of Expr_t with Post-Expr_t. Furthermore, we define that

- Each non-return parameter $p \in \text{PARAMS}_{\text{op}}$ with a declared type t is available as variable: $p \in \text{Var}_t$.
- If $\text{PARAMS}_{\text{op}}$ contains a parameter of kind "return" and type t then `result` is a variable: `result` $\in \text{Var}_t$.
- The operation `oclIsNew` : $c \to$ *Boolean* is in $\Omega_{\mathcal{M}}$ for all object types $c \in T_{\mathcal{M}}$.

The syntax of expressions in postconditions is extended by the following rule.

vii. If $\omega : t_1 \times \cdots \times t_n \to t \in \Omega_{\mathcal{M}}$ and $e_i \in \text{Post-Expr}_{t'}$ for all $i = 1, \ldots, n$ then $\boldsymbol{\omega}_{\text{@pre}}(\boldsymbol{e_1}, \ldots, \boldsymbol{e_n}) \in \text{Post-Expr}_t$.

□

All general OCL expressions may be used in a postcondition. Moreover, the basic rules for recursively constructing expressions do also apply. Operation parameters are added to the set of variables. For operations with a return type, the variable `result` refers to the operation result. The set of operations is extended by `oclIsNew` which is defined for all object types. Operations $\omega_{\text{@pre}}$ are added for allowing references to the previous state (vii). The rule says that the `@pre` modifier may be applied to all operations, although, in general, not all operations do actually depend on a system state (for example, operations on data types). The result of these operations will be the same in all states. Operations which do depend on a system state are, e.g., attribute access and navigation operations.

For a definition of the semantics of postconditions, we will refer to *environments* describing the previous state and the state resulting from executing the operation. An environment $\tau = (\sigma, \beta)$ is a pair consisting of a system state σ and a variable assignment β (see Section A.3.1.2). The necessity of including variable assignments into environments will be discussed shortly. We call an environment $\tau_{\text{pre}} = (\sigma_{\text{pre}}, \beta_{\text{pre}})$ describing a system state and variable assignments before the execution of an operation a *pre-environment*. Likewise, an environment $\tau_{\text{post}} = (\sigma_{\text{post}}, \beta_{\text{post}})$ after the completion of an operation is called a *post-environment*.

DEFINITION A.32 (SEMANTICS OF POSTCONDITION EXPRESSIONS)
Let Env be the set of environments. The semantics of an expression $e \in \text{Post-Expr}_t$ is a function $I[\![e]\!] : \text{Env} \times \text{Env} \to I(t)$. The semantics of the basic set of expressions in postconditions is defined by repeating Definition A.30 while substituting all occurrences of Expr_t with Post-Expr_t. References to $I[\![e]\!](\tau)$ are replaced by $I[\![e]\!](\tau_{\text{pre}}, \tau_{\text{post}})$ to include the pre-environment. Occurrences of τ are changed to τ_{post} which is the default environment in a postcondition.

- For all $p \in \text{PARAMS}_{\text{op}} : I[\![p]\!](\tau_{\text{pre}}, \tau_{\text{post}}) = \beta_{\text{post}}(p)$.
  - Input parameters may not be modified by an operation: kind$(p) =$ in implies $\beta_{\text{pre}}(p) = \beta_{\text{post}}(p)$.
  - Output parameters are undefined on entry: kind$(p) =$ out implies $\beta_{\text{pre}}(p) = \bot$.
- $I[\![\texttt{result}]\!](\tau_{\text{pre}}, \tau_{\text{post}}) = \beta_{\text{post}}(\texttt{result})$.
- $I[\![\texttt{oclIsNew}]\!](\tau_{\text{pre}}, \tau_{\text{post}})(\underline{c}) = \begin{cases} \text{true} & \text{if } \underline{c} \notin \sigma_{\text{pre}}(c) \text{ and } \underline{c} \in \sigma_{\text{post}}(c), \\ \text{false} & \text{otherwise.} \end{cases}$

vii. $I[\![\omega_{@\text{pre}}(e_1, \ldots, e_n)]\!](\tau_{\text{pre}}, \tau_{\text{post}}) = I(\omega)(\tau_{\text{pre}})(I[\![e_1]\!](\tau_{\text{pre}}, \tau_{\text{post}}), \ldots, I[\![e_n]\!](\tau_{\text{pre}}, \tau_{\text{post}}))$

□

Standard expressions are evaluated as defined in Definition A.30 with the post-environment determining the context of evaluation. Input parameters do not change during the execution of the operation. Therefore, their values are equal in the pre- and post-environment. The value of the `result` variable is determined by the variable assignment of the post-environment. The `oclIsNew` operation yields true if an object did not exist in the previous system state. Operations referring to the previous state are evaluated in context of the pre-environment (vii). Note that the operation arguments may still be evaluated in the post-environment. Therefore, in a nested expression, the environment only applies to the current operation, whereas deeper nested operations may evaluate in a different environment.

With these preparations, the semantics of an operation specification with pre- and postconditions can be precisely defined as follows. We say that a precondition P *satisfies* a pre-environment τ_{pre} – written as $\tau_{\text{pre}} \models P$ – if the expression P evaluates to true according to Definition A.30. Similarly, a postcondition Q satisfies a pair of pre- and post-environments, if the expression Q evaluates to true according to Definition A.32:

$$\begin{aligned} \tau_{\text{pre}} \models P \quad &\text{iff} \quad I[\![P]\!](\tau_{\text{pre}}) = \text{true} \\ (\tau_{\text{pre}}, \tau_{\text{post}}) \models Q \quad &\text{iff} \quad I[\![Q]\!](\tau_{\text{pre}}, \tau_{\text{post}}) = \text{true} \end{aligned}$$

DEFINITION A.33 (SEMANTICS OF OPERATION SPECIFICATIONS)
The semantics of an operation specification is a set $R \subseteq \text{Env} \times \text{Env}$ defined as

$$[\![\ \textbf{context}\ C :: \text{op}(p_1 : T_1, \ldots, p_n : T_n) \ \ \textbf{pre:}\ P \ \ \textbf{post:}\ Q \ ]\!] \ = \ R$$

where R is the set of all pre- and post-environment pairs such that the pre-environment τ_{pre} satisfies the precondition P and the pair of both environments satisfies the postcondition Q:

$$R = \{(\tau_{\text{pre}}, \tau_{\text{post}}) \mid \tau_{\text{pre}} \models P \ \wedge \ (\tau_{\text{pre}}, \tau_{\text{post}}) \models Q\}$$

□

The satisfaction relation for Q is defined in terms of both environments since the postcondition may contain references to the previous state. The set R defines all legal transitions between two states corresponding to the effect of an operation. It therefore provides a framework for a correct implementation.

DEFINITION A.34 (SATISFACTION OF OPERATION SPECIFICATIONS)
An operation specification with pre- and postconditions is satisfied by a program S in the sense of total correctness if the computation of S is a total function $f_S : \text{dom}(R) \rightarrow \text{im}(R)$ and $\text{graph}(f_S) \subseteq R$. □

In other words, the program S accepts each environment satisfying the precondition as input and produces an environment that satisfies the postcondition. The definition of R allows us to make some statements about the specification. In general, a reasonable specification implies a non-empty set R allowing one or more different implementations of the operation. If $R = \emptyset$, then there is obviously no implementation possible. We distinguish two cases: (1) no environment satisfying the precondition exists, or (2) there are environments making the precondition true, but no environments do satisfy the postcondition. Both cases indicate that the specification is inconsistent with the model. Either the constraint or the model providing the context should be changed. A more restrictive definition might even prohibit the second case.

BIBLIOGRAPHY

[AFGP96] A. Artale, E. Franconi, N. Guarino, and L. Pazzi. Part-whole relations in object-centered systems: An overview. *Data & Knowledge Engineering*, 20(3):347–383, November 1996.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[BHS99] F. Barbier and B. Henderson-Sellers. Object metamodelling of the whole-part relationship. In C. Mingins, editor, *Proceedings of TOOLS Pacific 1999*. IEEE Computer Society, 1999.

[BHSOG01] F. Barbier, B. Henderson-Sellers, A. L. Opdahl, and M. Gogolla. The whole-part relationship in the Unified Modeling Language: A new approach. In K. Siau and T. Halpin, editors, *Unified Modeling Language: Systems Analysis, Design and Development Issues*, chapter 12, pages 185–209. Idea Publishing Group, 2001.

[CKM+99] S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills. The Amsterdam manifesto on OCL. Technical Report TUM-I9925, Technische Universität München, December 1999.

[Cla99] T. Clark. Type checking UML static diagrams. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 503–517. Springer, 1999.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[Dat90] C. J. Date. *An Introduction to Database Systems – Vol. I*. Addison-Wesley, Readings (MA), 1990.

[EN94] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. The Benjamin/Cummings Publishing Company, Inc., 2 edition, 1994.

[Gog94] M. Gogolla. *An Extended Entity-Relationship Model – Fundamentals and Pragmatics*, volume 767 of *LNCS*. Springer, Berlin, 1994.

[GR99] M. Gogolla and M. Richters. Transformation rules for UML class diagrams. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998, Selected Papers*, volume 1618 of *LNCS*, pages 92–106. Springer, 1999.

[Her95] R. Herzig. *Zur Spezifikation von Objektgesellschaften mit TROLL light*. VDI-Verlag, Düsseldorf, Reihe 10 der Fortschritt-Berichte, Nr. 336, 1995. (Dissertation, Naturwissenschaftliche Fakultät, Technische Universität Braunschweig, 1994).

[HSB99] B. Henderson-Sellers and F. Barbier. Black and white diamonds. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 550–565. Springer, 1999.

[Mot96] R. Motschnig-Pitrik. Analyzing the notions of attribute, aggregate, part and member in data/knowledge modeling. *The Journal of Systems and Software*, 33(2):113–122, May 1996.

[Pri97] S. Pribbenow. What's a part? On formalizing part-whole relations. In *Foundations of Computer Science: Potential – Theory – Cognition*, volume 1337 of *LNCS*, pages 399–406. Springer, 1997.

[Ric02] M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. Ph.D. thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.

[Tho99] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 2nd edition, 1999.

B

Interchange Format

B.1 This appendix is intentionally left blank.

This section contains the interchange format for OCL. This XMI DTD should be generated from the metamodel.

Comment – This needs to be done when the final submission is finished.

Comment – Note that even the concrete syntax could be used as a simple interchange format, because it only consists of standard text strings. However. accepting tools would need to (re)parse the concrete syntax. The benefit will be that tools that do not support OCL (it is a optional compliance point within UML) can still create and interchange OCL as text.

C

References

[Warmer98] Jos Warmer en Anneke Kleppe, *The Object Constraint Language: precise modeling with UML*, Addison-Wesley, 1999

[Kleppe2000] Anneke Kleppe and Jos Warmer, *Extending OCL to include Actions*, in Andy Evans, Stuart Kent and Bran Selic (editors), <<UML>>2000 - *The Unified Modeling Language. Advancing the Standard. Third International Conference,* York, UK, October 2000, Proceedings, volume 1939 of *LNCS*. Springer, 2000

[Clark2000] Tony Clark, Andy Evans, Stuart Kent, Steve Brodsky, Steve Cook, *A feasibility Study in Rearchitecting UML as a Family of Languages using a Precise OO Meta-Modelling Approach, version 1.0*, September 2000, available from www.puml.org

[Richters1999] Mark Richters and Martin Gogolla, *A metamodel for OCL*, in Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference,* Fort Collins, CO, USA, October 28-30. 1999, Proceedings, volume 1723 of *LNCS*. Springer, 1999.

[Richters1998] Mark Richters and Martin Gogolla. *On formalizing the UML Object Constraint Language OCL*. In Tok Wang Ling, Sudha Ram, and Mong Li Lee, editors, *Proc. 17th Int. Conf. Conceptual Modeling (ER'98)*, volume 1507 of *LNCS*, pages 449–464. Springer, 1998..

[Kleppe2001] Anneke Kleppe and Jos Warmer, *Unification of Static and Dynamic Semantics of UML:* a *Study in redefining the Semantics of the UML using the pUML OO Meta Modelling Approach,* available for donwload at: http://www.klasse.nl/english/uml/uml-semantics.html

[Akehurst2001] D.H. Akehirst and B. Bordbar, *On Querying UML Data Models with OCL,* proceeding of the UML 2001 conference

Comment – TBD: This list of references is not complete.

Index

N

O

P

Q

R