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

Initial Submission, Version 1.0
August 20, 2001

OMG Document ad/2001-08-01

Submitters

Boldsoft
Rational Software Corporation
IONA

Supporters

Klasse Objecten
Kings College
University of Bremen
Dresden University of Technology
Kabira Technologies, Inc.

Contents

List of figures

List of tables

Foreword

COPYRIGHT WAIVER

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

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

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

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

Conrad Bock (conrad.bock@kabira.com)
Kabira Technologies, Inc

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.

1.2 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 then attempts to capture all of these goals as the team itself defined them.

1.2.1 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.

- Define a MOF-compliant metamodel for OCL. This metamodel should define the concepts and semantics of OCL and act as an abstract syntax for the language.
- (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.
- The separation between the metamodel and the concrete syntax allows for alternative concrete syntaxes (e.g. visual constraint diagrams).

1.2.2 OCL Expressibility and Usability

OCL 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.

- OCL is currently defined as a language for describing constraints. OCL 2.0 will specify the Object Constraint Languages as a general object query language that can be used wherever expressions over UML models are required.

- Additional concepts will be added to OCL to allow for a more complete specification of components.
- Additional concepts will be added to OCL to allow the specification of behavioral constraints. This will allow, e.g. to specify stimulus response rules.
- 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.
- The simplicity and usability requirements that the OCL 1.4 definition is built upon are the major guideline for the OCL 2.0.

1.2.3 OCL Semantics

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

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

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.

1.3 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.

1.3.1 Abstract syntax

1. Collections can be nested. This is different from OCL 1.4 where collections were always implicitly flattened. Flattening was only applied to the collect and iterate operations in OCL. For the collect we now distinguish two different operations:
   - collectNested() which is identical to the OCL 1.4 collect without flattening applied.
   - flatten(), which flattens a nested collection.
   - collect(), which is identical to collectNested()->flatten()

   The current syntax should retain the meaning of automatic flattening, because most OCL is currently written in paper form only and this needs to be backward compatible.
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 removed. 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 has become Classifier from the UML kernel.
6. OCL should be extendible. We plan to (re)use any extension mechanism as described in the UML infrastructure submission.
7. We want to enable full use of OCL as a query language. Therefore the concept of Tuple is added to OCL.

1.3.2 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.
3. The OCL 2.0 grammar looks very different from the OCL grammar in UML 1.4, but describes the same concrete syntax. The grammar has been derived directly from the abstract syntax. To understand how this grammar was developed this section describes the rules that have been used. The reason for this approach is that we wanted a concrete syntax that was easily mappable to the abstract syntax.
   - Each metaclass becomes a non-terminal in the grammar. Every other grammar rule/symbol must be reachable from these.
   - A superclass becomes a choice rule with all of its subclasses as a choice.
   - For any leaf class the rules 2 - 5 apply
   - If a superclass can have instances itself, a special "concrete" Superclass non-terminal is added to the choice as described at 1.
   - Each type of each attribute becomes a token/symbol in the grammar.
   - Each attribute becomes a non-terminal in the grammar with the type of the attribute as the production.
   - Each association in the abstract syntax metamodel will get a specified direction.
   - Each association-end at the target side of the association as specified by the direction becomes a non-terminal in the grammar
   - If an associated class is ’just a reference’, but not being defined in the grammar, some other symbolic non-terminal will be used. This rule is when the associated metaclass is not defined at the specified location in the grammar, but only referenced. E.g. the association from *VariableEpr* to *VariableDeclaration* and the associations from the different *ModelPropertyCallExp* to their *referredFeatures*. In these cases a special ’lookup’ function needs to find the instance of the referred metaclass.
   - Multiplicity in the abstract syntax metamodel are reflected directly in the grammar

1.3.3 Semantics

1. The semantic description is based on a formal mathematical model.
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 an appendix.

Thne equivalance of the two semantic descriptions has not been formally established. Wherever they are conflicting, the description in section 5 (“Semantics”) is normative.

1.3.4 OCL Standard Library

The so-called predefined types and operations in UML 1.4 are now defined as the OCL Standard Library. This also 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.

1.4 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.

1.4.1 General Requirements

5.1.2; This specification of OCL includes detailed semantics and a normative formalism defining it's 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 it does expect that 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.

5.1.6; no known changes to any existing OMG specification.

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 (excepting 5.2.5) where not expressly accounted for in the development of this specification.

1.4.2 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 bi-directional mapping from the abstract syntax to the defined canonical concrete 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 OclType and the type ExpressionType have been removed. The operations that use the OclType are still available to the OCL user, but mapped to a special construct inb the abstract syntax. Therefore the OCL user will not notice this change.

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

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

1.4.3 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 it's expressive power, these are clearly defined in the body of the document.

1.4.4 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 initial submission does not cover the requirements for, or the approach to compliance testing of OCL implementations.

1.5 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.

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

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”) describes the underlying semantics of OCL. This is done using a mathematical formalism.

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 the UML”) describes a number of places within the UML where OCL expressions can be used. Appendix B (“Interchange Format”) is currently a place holder for an interchange format, which can be defined along the same lines as XMI. Appendix A (“Semantics Described using UML”) describes an informal semantics for OCL. This appendix, however is not normative, but given to the readers who need a non-mathematics description for the semantics of OCL.

1.6 EXPECTED CHANGES FOR THE FINAL SUBMISSION

Due to different reasons the specification of OCL, as described in this document, is not fully complete yet. This section describes the major changes that we expect to make before the final submission. Apart from these, we also expect to make minor changes to enhance the consistency and correctness of the specification.

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 final submission 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 unified, including a clarification of how multiple pre- and/or post-conditions are merged into one operation specification.

References to Pre-values in Postconditions

Free mixing of references to pre and post state in post-condition navigation expressions may be difficult to implement for tool builders. A warning for tool builders shall be included.

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 that 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 the UML”).

Concrete Syntax for Tuples

There is no concrete syntax yet for dealing with tuples. This needs to be added.

Flattening of Collections

The automatic flattening of collections in UML 1.4 is a deep flatten. We need to check whether the new "flatten()" operation should be a shallow (which it currently is) or a deep flatten, or whether we need both.

Undefined

The undefined value should be handled properly. In the UML 1.4 OCL specification this is too shallow. A literal "Undefined" and/or an operation "isUndefined()" will probably be needed.

Collection Constructors

Collection constructors are not handles in the abstract and the concrete syntax. This needs to be added.

Collection Operations

The collection operations in the standard library need to be checked for completeness. Additional operations might be added.

Concrete Syntax for Context Declarations

The concrete syntax for context declarations from UML 1.4 OCL are not defined in this initial submission. This concrete syntax needs to be added. It is expected to look identical to that of the UML 1.4 OCL specification.

OCL Language Description

Section 2 (“OCL Language Description”) is not complete with respect to newly added features as e.g. the action clause, non-flattened collections, and tuples. This section needs to be updated to describe all OCL features.

Classifier Scoped Operations/Attributes

Features with Classifier scope are currently not supported in the abstract syntax. Theirsource is not an expression, but a Classifier. They need to be added to the abstract syntax.

Formal Semantics of Action Expression

The formal semantics of the Action expression is not covered yet. In the UML description of the semantics in section A.3.4 (“Action Expression Evaluations”) a definition can be found.

Formal Semantics of Qualified Associations

The semantics of qualified association sis not defined yet. This needs to be done.

Overlap in Chapters 5 (“Semantics”) and 6 (“The OCL Standard Library”)

Several of the operations defined in the OCL Standard Library are also defined in the semantics chapter. This overlap will be resolved by removing these operations from the semantics section.

Index

The index is incomplete.

2

OCL Language Description

This chapter introduces the Object Constraint Language (OCL), a formal language used to express constraints. These typically specify invariant conditions that must hold for the system being modeled. 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. UML modelers can use OCL to specify application-specific constraints in their models.

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 string 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 expression 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, not everything in it is promised to be 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 “The OCL Standard Library” on page 1).

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 constraints on operations
- for any expression over a UML model

2.2 INTRODUCTION

2.2.1 Legend

Text written in the courier 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.

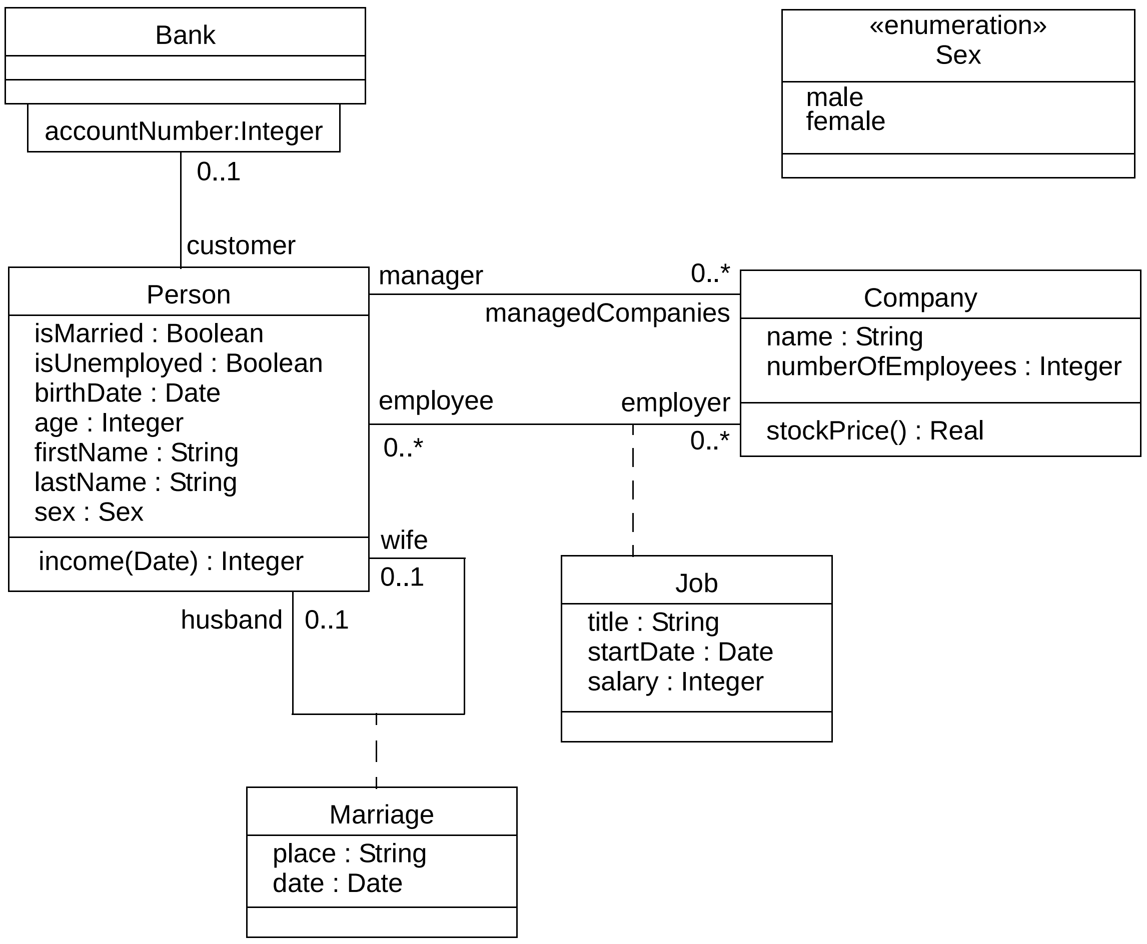


Figure 2-1 *Class Diagram Example*

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 , 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 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 metamodel, this name is an 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 Method. 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 precondiation 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 attributes 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 conatin any number package statements, thus allowing all invariant, preconitions and postconditions to be written down and stored in one file. This file may co-exist with a UML model as a separate entity.

2.3.6 General 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.

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. Some basic types used in the examples in this document, 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 6.3 (“Primitive Types”) for a complete list of all operations.

The complete list of operations provided for each type is described at the end of this chapter. Collection, Set, Bag and Sequence 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`	`toUpper(), concat()`

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 Sex with values 'female' or 'male' they can be used as follows:

```
context Person inv: sex = Sex::male
```

2.4.3 Let Expressions and «definition» Constraints

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

```
context Person inv:
let income : Integer = self.job.salary->sum()
let hasTitle(t : String) : Boolean =
             self.job->exists(title = t) in
if isUnemployed then
   self.income < 100
else
   self.income >= 100 and self.hasTitle(‘manager’)
endif
```

A let expression may be included in an invariant or pre- or postcondition. It is then only known within this specific constraint. To enable reuse of let variables/operations one can use a Constraint with the stereotype «definition», in which let variables/operations are defined. This «definition» Constraint must be attached to a Classifier and may only contain let definitions. 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. In essence, such variables and operations are psuedo-attributes and psuedo-operations of the classifier. They are used in an OCL expression in exactly the same way as attributes or operations are used. The textual notation for a «definition» Constraint uses the keyword ‘def’ as shown below:

```
context Person def:
let income : Integer = self.job.salary->sum()
let 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. Also, the names of all let variables and operations connected with a Classifier must be unique.

2.4.4 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 type *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 value types are listed in table 3.

Type	Conforms to/Is a subtype of
`Set(T)`	`Collection(T)`
`Sequence(T)`	`Collection(T)`
`Bag(T)`	`Collection(T)`
`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 -16 for the complete conformance rules for collections.

Table 0-1 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 0-1 Valid expressions

2.4.5 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 “Undefined Values” on page -9).

2.4.6 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.

2.4.7 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.

2.4.8 Keywords

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

```
if
then
else
endif
not
let
or
and
xor
implies
endpackage
package
context
def
inv
pre
post
in
```

2.4.9 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
```

2.4.10 Undefined Values

Comment – Needs to be changed to OCL 2.0

Whenever an OCL expression is being evaluated, there is a possibility that one or more of the queries in the expression are undefined. If this is the case, then the complete expression will be undefined.

There are two exceptions to this for the Boolean operators:

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

The above two rules are valid irrespective of the order of the arguments and the above rules are valid whether or not the value of the other sub-expression is known.

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

2.5.1 Properties

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

```
context AType inv:
self.property
```

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

2.5.2 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 basic 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 as shown in the invariant above.

2.5.3 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 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. The type of *result* is the return type of the operation, which is Integer in the above example.

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
```

2.5.4 Properties: Association Ends 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.rolename
```

The value of this expression is the set of objects on the other side of the *rolename* 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 Rolenames

When a rolename is missing at one of the ends of an association, the name of the type at the association end, starting with a lowercase character, is used as the rolename. If this results in an ambiguity, the rolename is mandatory. This is 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.

The following example shows how a property of a collection can be used.

```
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.sex = Sex::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
```

2.5.5 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.

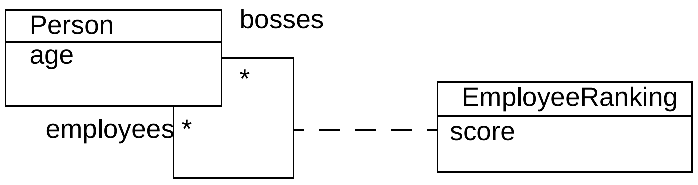


Figure 2-2 *Navigating recursive association classes*

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]
```

2.5.6 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 (->).

2.5.7 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.

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

This results in a Set(Person) containing all customers of the Bank.

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

This results in one Person, having accountnumber 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.

2.5.8 Using Pathnames for Packages

Within UML, different 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
```

2.5.9 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 0-1 shows an example where such a construct is needed.

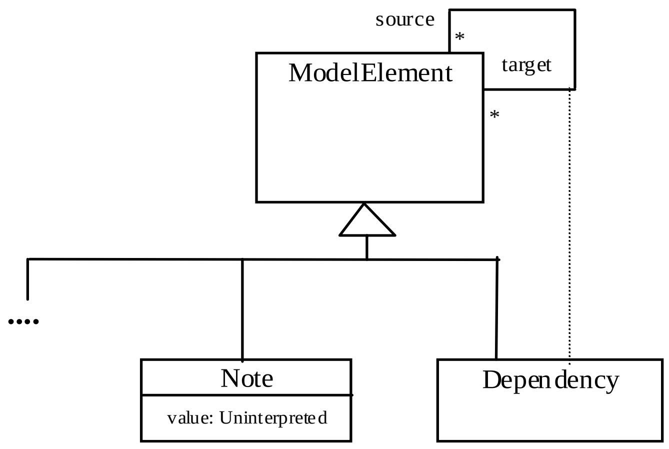


Figure 0-1 Accessing Overridden Properties Example

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 role-name, so this is always ambiguous. Using *oclAsType()* we can distinguish between them with:

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

2.5.10 Predefined properties on All Objects

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

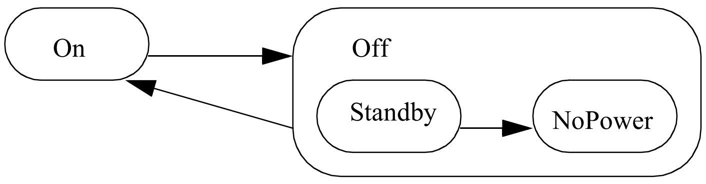
```
oclIsTypeOf(t : OclType)  : Boolean
oclIsKindOf(t : OclType)  : Boolean
oclInState(s : OclState)  : Boolean
oclIsNew()         : Boolean
oclAsType(t : OclType) : instance of OclType
```

The operation is *oclTypeOf* 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 semicolon ::, 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.

2.5.11 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 each type 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 at the snapshot in time that the expression is evaluated.

NB: The use of *allInstances* has some problems and its use is discouraged in most cases. The first problem is best explained by looking at the types like Integer, Real and String. For these types the meaning of *allInstances* is undefined. What does it mean for an Integer to exist? The evaluation of the expression *Integer.allInstances()* results in an infinite set and is therefore undefined within OCL. The second problem with *allInstances* is that the existence of objects must be considered within some overall context, like a system or a model. This overall context must be defined, which is not done within OCL. A recommended style is to model the overall contextual system explicitly as an object within the system and navigate from that object to its containing instances without using *allInstances*.

2.5.12 Collections

Single navigation results in a Set, combined navigations in a Bag, and navigation over associations adorned with {ordered} results in a Sequence. Therefore, the collection types 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. 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 at the end of this chapter.

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)
```

2.5.13 Collections of Collections

Comment – Needs to be changed to OCL 2.0

Within OCL, all Collections of Collections are flattened automatically; therefore, the following two expressions have the same value:

```
Set{ Set{1, 2}, Set{3, 4}, Set{5, 6} }
Set{ 1, 2, 3, 4, 5, 6 }
```

2.5.14 Collection Type Hierarchy and Type Conformance Rules

In addition to the type conformance rules in “Type Conformance” on page -7, 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*.
- 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.

2.5.15 Previous Values in Postconditions

As stated in “Pre- and Postconditions” on page -4, OCL can be used to specify pre- and post-conditions on operations and methods in UML. In a postcondition, the expression can refer to two sets of values for each property of an object:

- 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 on 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
```

The above operation can also be specified by a postcondition and a precondition together:

```
context Company::hireEmployee(p : Person)
pre : not employee->includes(p)
post: employees->includes(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 Undefined. Also, referring to the previous value of an object that has been created during execution of the operation results in Undefined.

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 delivers 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( forename = 'Jack' )
inv:  self.employee->forAll( p | p.forename = 'Jack' )
inv:  self.employee->forAll( p : Person | p.forename = 'Jack' )
```

These invariants evaluate to true if the forename feature of each employee is equal to 'Jack.'

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 |
            e1 <> e2 implies e1.forename <> e2.forename)
```

```
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 feature 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>
    }
}
```

Although the Java pseudo code uses a 'next element', the *iterate* operation is defined for each collection type and 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.

2.6.6 Iterators in Collection Operations

The collection operations that take an OclExpression as parameter may all have an optional iterator declaration. For any operation name *op*, the syntax options are:

```
collection->op( iter : Type | OclExpression )
collection->op( iter | OclExpression )
collection->op( OclExpression )
```

2.6.7 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*. Propoerty *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:

```
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 core)’ and shown with a transparant fill color. All metaclasses defined as part of the OCL abstract syntax are shows 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 types that can occur in an OCL expression. 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.

The type OclType has been removed from the type hierarchy. This means that a Classifier is not a valid OCL expression any more.

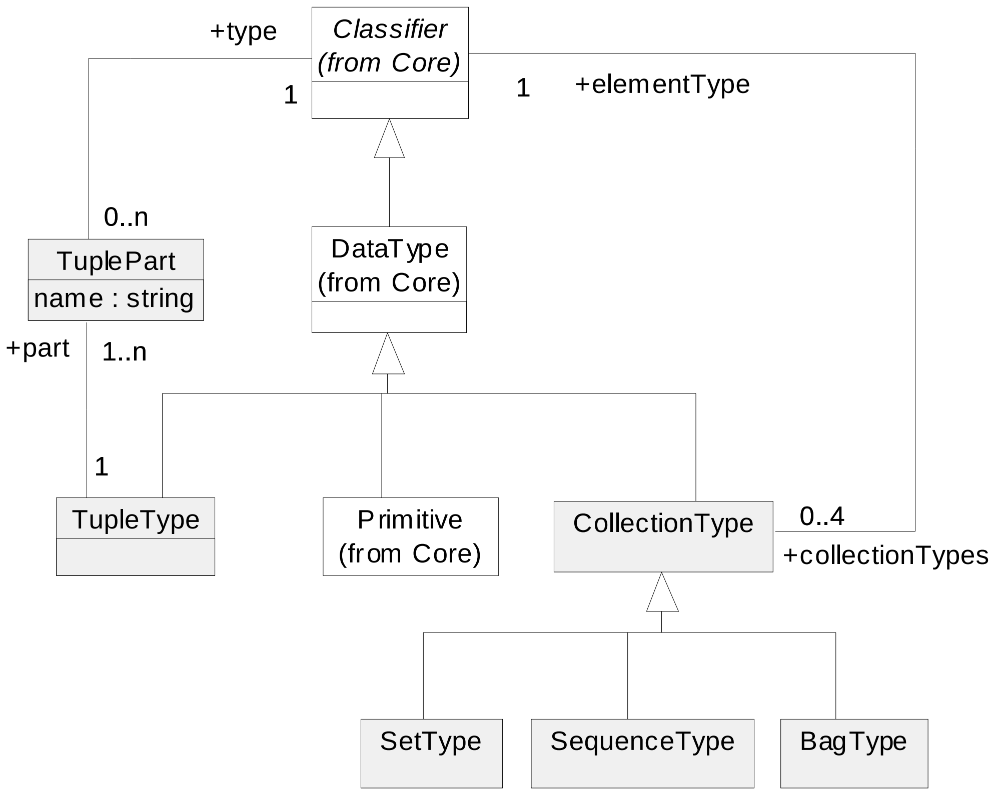


Figure 3-1 *Abstract syntax kernel metamodel for OCL Types*

BagType

A bag type 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 bag type is the declaration of the type of its elements. In the metamodel, this is shown as an association from *CollectionType* (a generalization of *BagType*) to *Classifier.*

CollectionType

A collection type describes a list of elements of a particular given type. Collection types are Set, Sequence and Bag 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 which may be nested arbitrarily deep.

Associations

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

SequenceType

A sequence type 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 sequence type is the declaration of the type of its elements. In the metamodel, this is shown as an association from *CollectionType* (a generalization of *SequenceType*) to *Classifier.*

SetType

A set type 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 set type is the declaration of the type of its elements. In the metamodel, this is shown as an association from *CollectionType* (a generalization of *SetType*) to *Classifier.*

TupleType

A tuple type (also known as record type) combines different types into a single aggregate type. The components of a tuple type are described by tuple parts 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 tuple type may contain other tuple types and collection types.

TuplePart

A tuple part represents a single component of a tuple type. A part has a name and a type. The purpose of the name is to uniquely identify each component.

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 using OCL.

BagType

[1] Different bag types conform to each other if there elements 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 there elements 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 there elements 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.

```
context TupleType
inv: TupleType.allInstances()->forAll (t |
       ( t.part->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)
          self.part->exists(stp|stp.name = tp.name) and
          -- make sure that all tupleparts with the same name conforms.
          self.part->forAll(stp | (stp.name = tp.name) and stp.type.conformsTo(tp.type))
       )
       implies
            self.conformsTo(t)
     ) )
```

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.oclIsTypeOf(BagType) implies
         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.oclIsTypeOf(CollectionType) implies
         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
```

SequenceType

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

```
inv: self.oclIsTypeOf(SequenceType) implies
         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.oclIsTypeOf(SetType) implies
         self.name = ’Set(’ + self.elementType.name + ’)’
```

Tuple

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

```
context Tuple
inv: self.part->isUnique(tp : TuplePart | tp.name)
```

TuplePart

No additional well-formedness rule.

3.3 THE EXPRESSIONS PACKAGE

This section defined the abstract syntax of the expressions package. This defines the structure that OCL expressions can have.

3.3.1 Expressions Core

Figure 3-2 on page 6 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 object. In this

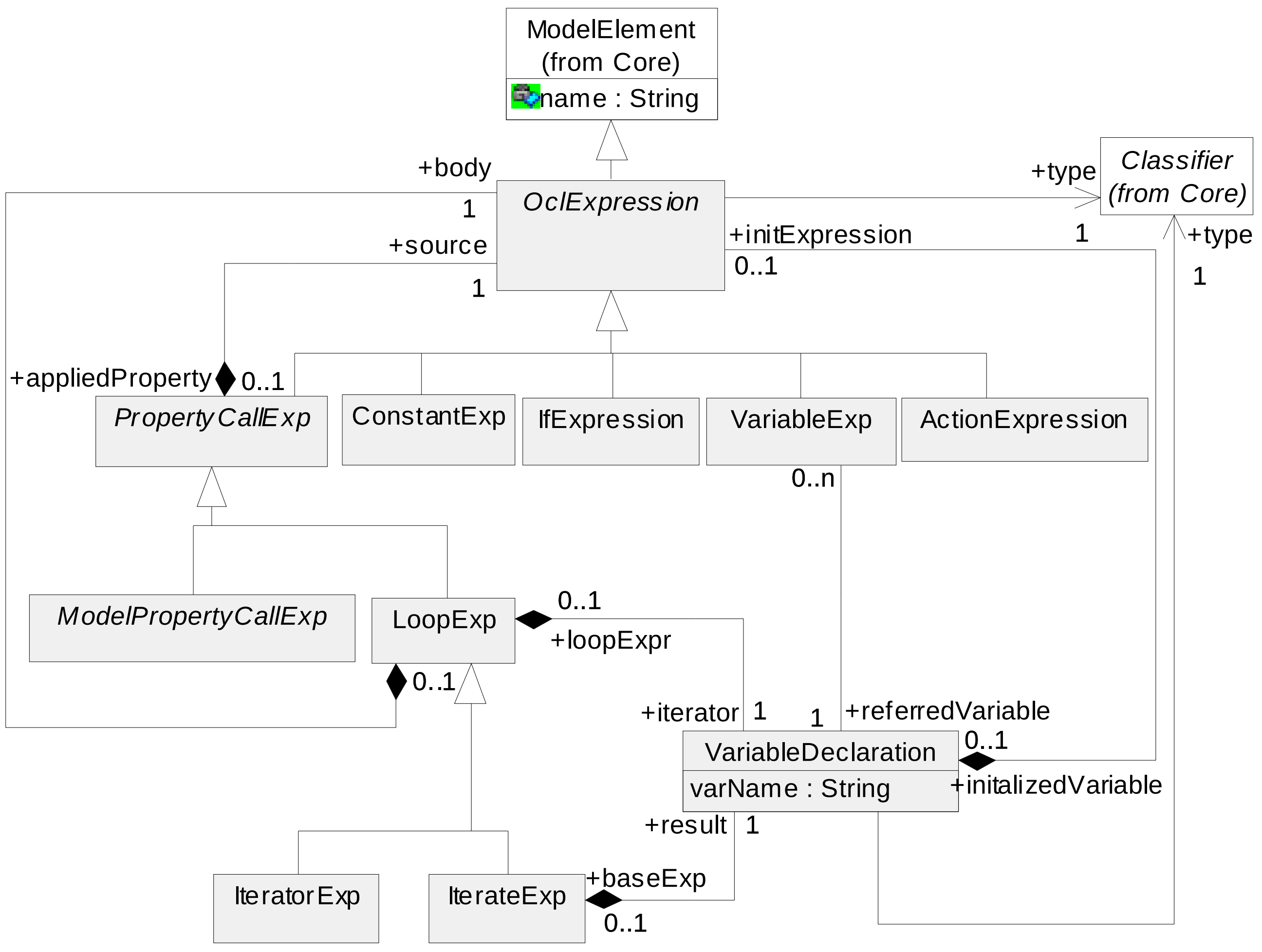


Figure 3-2 *The basic structure of the abstract syntax kernel metamodel for Expressions*

section we use the term 'property', which is a generalization of Feature, AssociationEnd and predefined iterating OCL collection operations.

A model propertycall expression generalizes all propertycalls that refer to Features or associations or associationends in the UML metamodel. In figure 3-3 on page 9 the various subtypes of model propertycall expression are defined.

Most of the remainder of the expressions package consists of a specification of the diffrent subclasses of *PropertyCallExp* and their specific structure. The model shows that an OCl expression always starts with a variable, on which a property is recusively applied.

ActionExpression

An action expression is defined in section (“ActionExpression”), but included in this diagram for completeness.

ConstantExp

A constant expression is an operation with no arguments producing a value. In general the result value is identical with the operation symbol. This includes things like the integer 1 or literal strings like 'this is a ConstantExp'.

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. All other collection expressions are defined in terms of the iterate expression.

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.

Associations

iterator	The *VariableDeclaration* that represents the iterator variable. This variable is bound to each 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 model property call 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 3-3 on page 9 the various subclasses of *ModelPropertyCallExp* are shown.

OclExpression

An OCL expression 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 OCL expression 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.

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 variable into the environment. the environment is not modeled as a separate metaclass, because it can be completely derived using derivation rules. Instead the additional operation *environment()* is defined which result in all visible modelelements.

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 3-3 on page 9.
initializedVariable	The variable of which the result of this expression is the initial value.

PropertyCallExp

A property call expression is an expression that refers to a property (i.e. operation, attribute, association end, predefined iterator for collections, ...). Its result value is the evaluation of the corresponding property.

Associations

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

VariableDeclaration

An 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 variable expression 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 variable declaration 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 by the three different subtypes, each of which is associated with its own type of *ModelElement*.

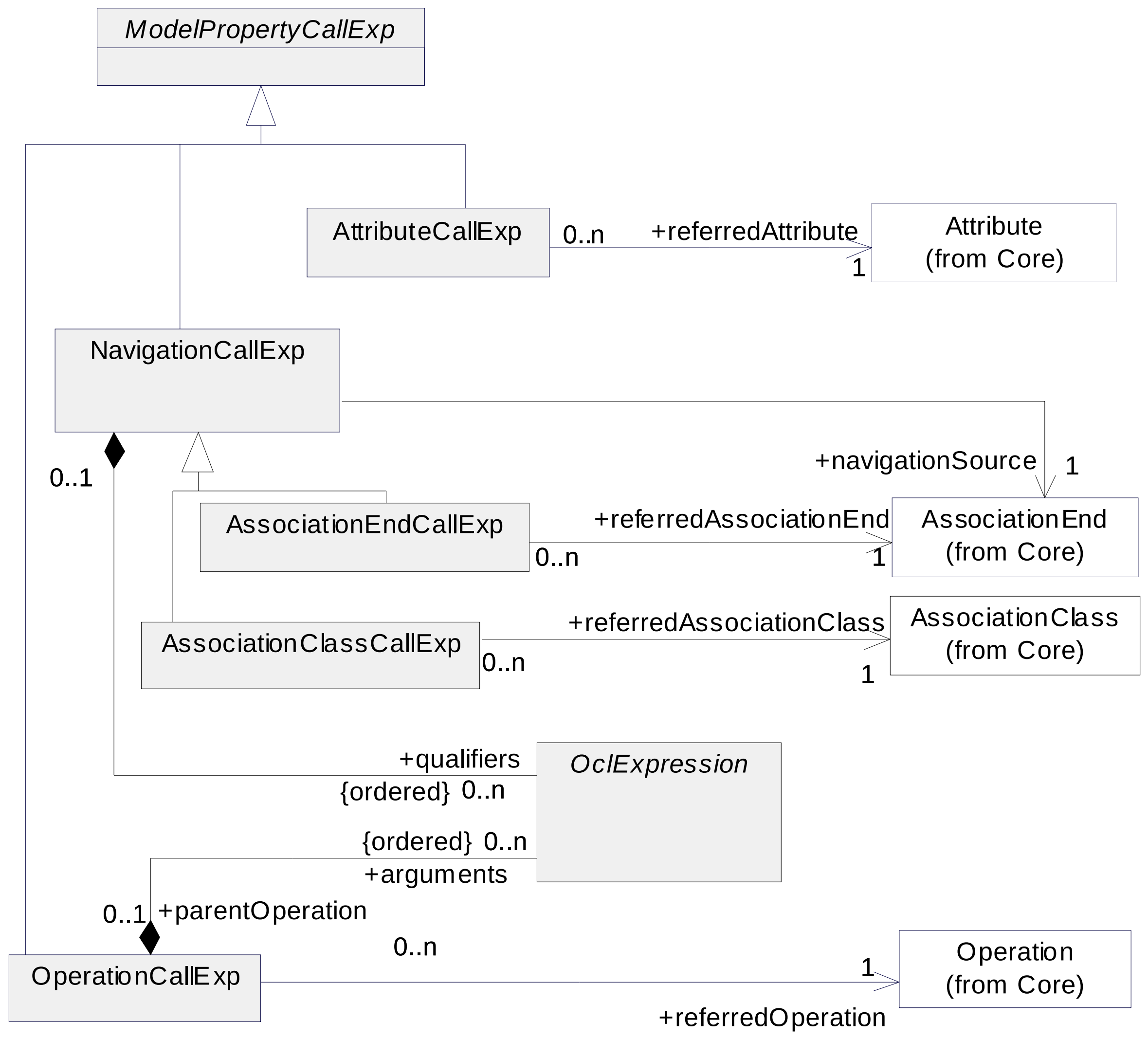


Figure 3-3 *Abstract syntax metamodel for ModelPropertyCallExp in the Expressios package*

AssociationEndCallExp

A 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

A 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. The expression refers to these target objects by the name of the ctarget 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 attribute call expression 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.

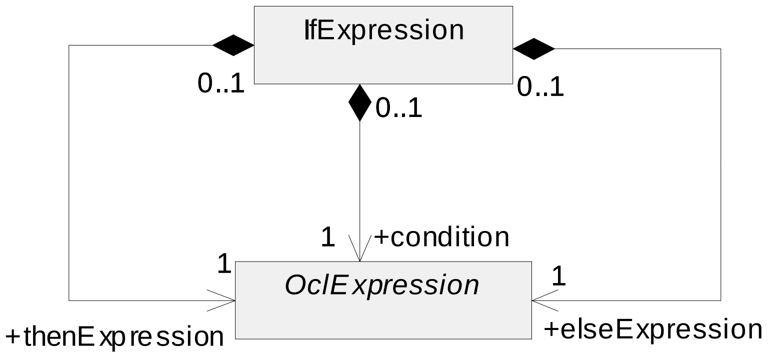


Figure 3-4 *Abstract syntax metamodel for action expression*

IfExpression

An if expression results in one of two alternative expressions depending on the evaulated value of a condition. Note that both the then and the else 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 would be left our.

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 Action Expressions

Action expressions are used to specify the fact that an object has will perform some action at a some moment in time.

ActionExpression

An action expression refers to an Action as defined in the Common Behavior section of the UML semantics. Whenever the *condition* evaluates to true, the object performs the associated action. If the *action* is a *SendAction*, then the *Operation* associated with the *CallAction* is called on all the target objects. This *CallAction* can both be synchronous and asynchronous, as described in the UML Semantics. If the action is a *SendAction*, then an instance of the *Signal* associated with the SendAction is sent to all *target* instances.

The moment at which the condition is evaluated depends on the place in the UML model where the action caluse is used. If the action clause is attached to a Classifier the meaning is that whenever the condition changes from false to true for an instance of the Clasifier, the action will be performed by that instance. If the action clause is attached to an operation, the condition is evaluated at postcondition time. If it evaluates to true, the action is performed by the object somewhere between precondition and postcondtion time. See [Kleppe2000] for a complete description and motivation of the action clause.

Associations

condition	The *OclExpression* that represents the boolean condition under which the actions are performed by the object.

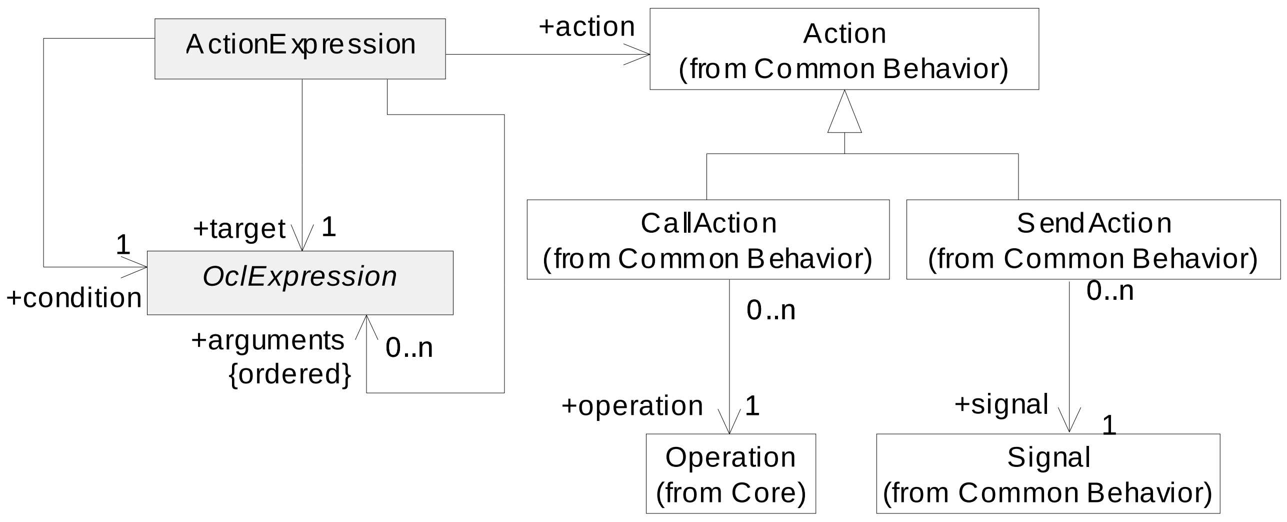


Figure 3-5 *Abstract syntax metamodel for action expression*

target	The *OclExpression* that represents the target instance or instances on which the action is perfomed. If the action is a *SendAction*, the target may be undefined. The signal is then send to instances, as is defined in the UML semantics under *SendAction*.
arguments	The *OclExpression* that represents the actual 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 isthe same as the order of the parameters of the *Operation* or the attributes of a *Signal*.
action	The *Action* that is performed by the object when the condition is true. The action can be a *CallAction*, denoting that an *Operation* has been called, or a *SendAction*, denoting that a *Signal* has been sent.

3.3.5 Constant Expressions

This section defines the different types of constant expressions OCL. It also refers to enumeration types and enumeration literals.

EnumConstantExp

An enumerationconstant expression represents a reference to an enumeration literal.

Associations

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

IntegerConstantExp

A integer constant denotes a value of the type Integer.

NumericConstantExp

A numeric constant denotes a value of either the type Integer or the type Real.

RealConstantExp

A real constant denotes a value of typeReal.

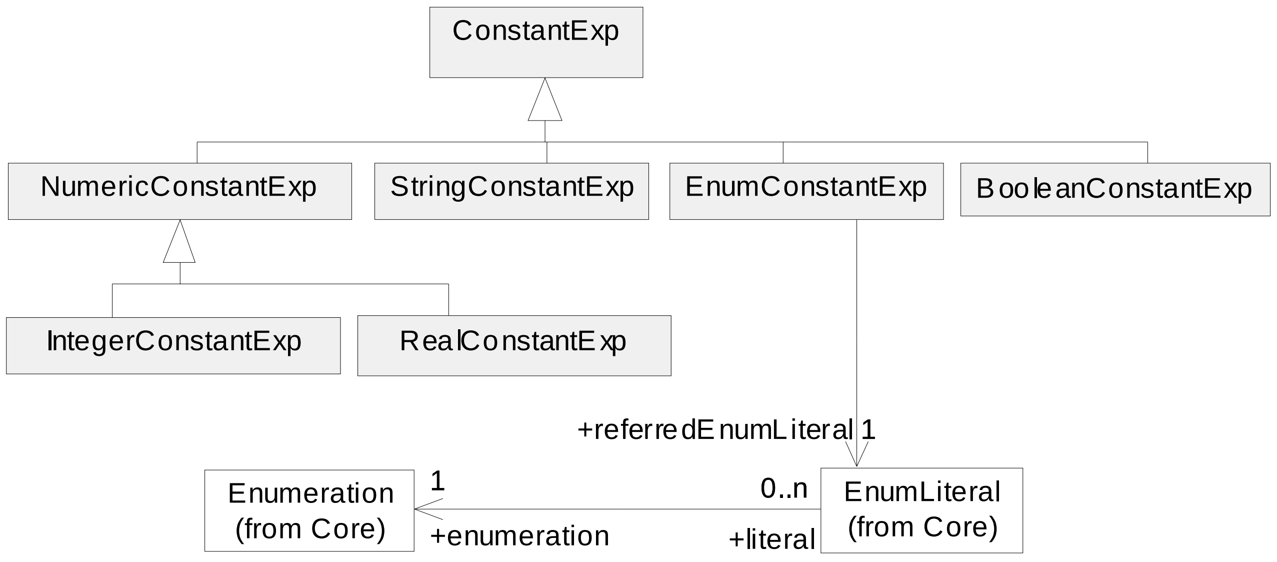


Figure 3-6 *Abstract syntax metamodel for constant expression*

StringConstantExp

A string denotes a value of the predefined type String.

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 LetExpression. 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 defined an additional operation in the UML Classifier, potentially with a special sterotype 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.

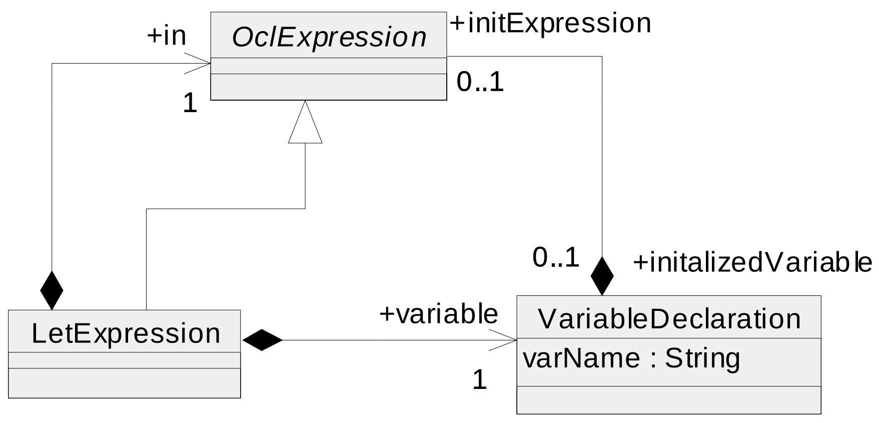


Figure 3-7 *Abstract syntax metamodel for let expression*

LetExpression

An Let expression is a special expression that defined a new variable with an initial value. A variable defined by a Let expression 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 expression in whose environment the defined variable is visible.

3.3.7 Operations with special source or argument

This section defines operations that have either as their source, or as their argment a ModelElement that is not an OclExpression. The reason for this is that a Classifier or a State do not inherit from OclExpression in the meta-model. .

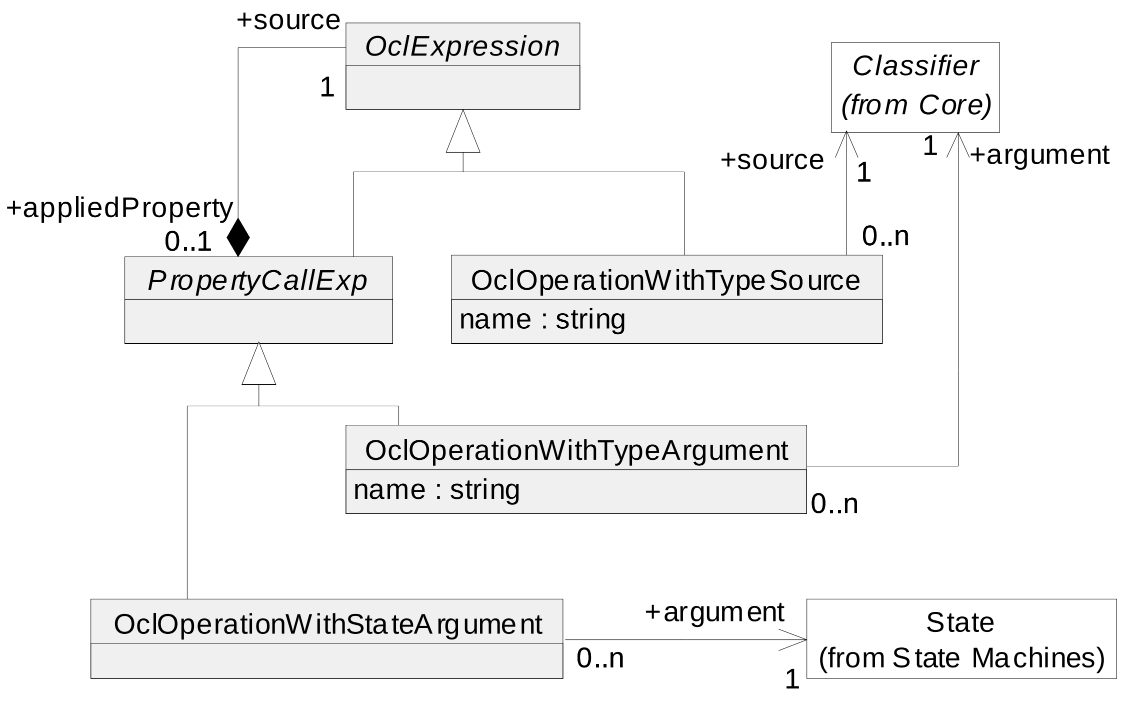


Figure 3-8 *Abstract syntax for operations that have non OclExpressions as their source or argument*

OclOperationWithTypeSource

An OclOperationWithTypeSource is a special operation that does not have an Ocl expression as a source, but a Classifier. This metaclass is used to model predefined operations like *allInstances()*, but also to represent classifier scoped attributes and operations.

Associations

source	The source is the Classifier that 'performs' this operation.

OclOperationWithTypeArgument

An OclOperationWithTypeArgument is a special operation that does not have an Ocl expression as a parameter, but a Classifier. This metaclass is used to model predefined operations like *oclIsTypeOf()* and *oclIsKindOf()*.

Associations

argument	The argument is the Classifier that is the argument of this operation.

OclOperationWithStateArgument

An OclOperationWithStateArgument is a special operation that does not have an Ocl expression as a parameter, but a State. This metaclass is used to model the predefined operation *oclInState()*.

Associations

argument	The argument is the State that is the argument for this operation.

3.3.8 Well-formedness Rules of the Expressions package

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

ActionExpression

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

```
context ActionExpression
inv: condition.type.isKindOf(Primitive) and condition.type.name = 'Boolean'
```

[2] The action must be a send action or a call action.

```
context ActionExpression
inv: action.oclIsTypeOf(CallAction) or action.oclIsTypeOf(SendAction)
```

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

```
context ActionExpression
inv: -- TBD see OperationCallExp for an identical constraint.
```

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

```
context ActionExpression
inv: -- TBD see OperationCallExp for an identical constraint
```

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

```
context ActionExpression
inv: action.oclIsTypeOf(CallAction) implies
       target.type.allOperations->includes(action.operation)
```

AttributeCallExp

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

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

BooleanConstantExp

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

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

Classifier

Additional Operations

The operation *conformsTo(c : Classifier) : Boolean* defines whether the self *Classifier* conforms to the argument *c*. The conformsTo() operation is defined in the section 3.2.1 (“Type Conformance”).

ConstantExpression

No additional well-formedness rules.

EnumConstantExp

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

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

IfExpression

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

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

IntegerConstantExp

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

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

IteratorExp

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

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

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

```
context IteratorExp
inv: self.iterator.initExpression->isEmpty()
```

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

```
context IteratorExp
inv: source.type.oclAsType(CollectionType).elementType.conformsTo(iterator.type)
```

[4] 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'
```

[5] The result type of the collect operation on a sequqnce type is q 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
```

[6] The ’select’ iterator has the same type as its source

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

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

```
context IteratorExp
inv: name = 'exists' or name = 'forAll' or name = 'select'
```

```
        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
```

LetExpression

No additional well-formedness rules.

LoopExp

No additional well-formedness rules.

ModelPropertyCallExp

No additional well-formedness rules.

NumericConstantExp

No additional well-formedness rules.

OclExpression

No additional well-formedness rules.

OclOperationWithStateArgument

No additional well-formedness rules.

OclOperationWithTypeArgument

No additional well-formedness rules.

OclOperationWithTypeSource

No additional well-formedness rules.

OperationCallExp

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

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

Alternative specification of the same constraint:

```
context OperationCallExp
inv: Sequence{1..arguments->size()}->forAll( i |
                    arguments->at(i).conformsTo(referredOperation.parameters->at(i).type)
```

PropertyCallExp

No additional well-formedness rules.

RealConstantExp

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

```
context RealConstantExp
inv: self.type.name = ’Real’
```

StringConstantExp

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

```
context StringConstantExp
inv: self.type.name = ’String’
```

VariableDeclaration

No additional well-formedness rule.

VariableExpression

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

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

4 Concrete Syntax

This section describes the concrete syntax of the OCL. This allows modelers to write down OCL expressions in a standardized way. An formal mapping from the concrete syntax to the abstract syntax from chapter 3 is also given. Although not required by the UML 2.0 for OCL RfP, section 4.4 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 1.3.2 (“Concrete syntax”) describes how we have developed the grammar and the motivation for this approach.

4.1 STRUCTURE OF THE CONCRETE SYNTAX

The concrete syntax of OCL is described using the EBNF formalism. This grammar is annotated with both synthesised and inherited attributes. It forms a full attribute grammar. For each metaclass in the abstract syntax metamodel there is a production rule. For some elements there might be a choice of multiple rules. The structure of the grammar and the naming of these nonterminals in the EBNF grammar is then easily mapped to the abstract syntax.

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 three 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)

A decision on which grammar production 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 an unambiguous parsing of *a.b()*. In this case the rules (in plain English) would be:

- if a is a defined variable in the current scope, a is a VariableExp.
- if not, check self and all iterator variables in scope. There must be exactly one of those whose type (a Classifier) has either:
  - an attribute with the name a, resulting in an AttributeCallExp
  - or an opposite association-end with the name a, resulting in an NavigationCallExp
- 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.

4.1.1 A Note to Tool Builders

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 a well-formed instance of the abstract syntax.

4.2 CONCRETE SYNTAX

The concrete syntax is described using a EBNF attribute grammar. Attribute derivations are described using OCL. There are three special annotations for each grammar production:

Synthesised attributes. Each production may have synthesised attributes attached to it. The value for these attributes is derived from other values and described using OCL. Synthesized attributes are attributes that are derived from the right hand side of the production and belong to the element at the left hand side. The 'type' attribute holds the type of an expression and is used for many productions.

Inherited attributes. Each production may have inherited attributes attached to it. The value for these attributes is derived from other values and described using OCL. Inherited attributes are attributes that are derived from the left hand side of the production and belong to elements at the right hand side. The main inherited attribute use is the 'env' attribute that defined the environment of each part of an expression.

Disambiguating rules. Each production has zero or more disambiguating rules. If one of the disambiguating rules is not met, the corresponding production rule in the grammar is invalid. The disambiguating rules are written in OCL, and use some definitions from the UML 1.4 semantics.

Start

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

```
Start               ::= OclExpression
```

Synthesized attributes

```
type = OclExpression.type
```

Inherited attributes

The environment of the OCL expression must be defined. below the self variable is added, and all the contents of the type of self. This defintion needs carefull consideration.

```
OclExpression.env = new Environment()
     ->including( new VariableDeclaration('self', TYPE) )
     ->union    ( type.allContents->collect(el |new NameBinding(el.name, el.type)) )
     --    if pre/post: add parameters to scope
     --    add anything else from the UML model ...
     -- TBD
```

OclExpression

An OclExpression has several production rules, one for each subclass of OclExpression.

```
[A] OclExpression ::= PropertyCallExp
[B] OclExpression ::= VariableExp
[C] OclExpression ::= ConstantExp
[D] OclExpression ::= LetExpression
[E] OclExpression ::= ActionExpression
[F] OclExpression ::= IfExpression
[G] OclExpression ::= OclOperationWithTypeSource
```

Synthesized attributes

```
[A] self.type = PropertyCallExp.type
[B] self.type = VariableExp.type
[C] self.type = ConstantExp.type
[D] self.type = LetExpression.type
[E] self.type = ActionExpression.type
[F] self.type = IfExpression.type
[G] self.type = OclOperationWithTypeSource.type
```

Inherited attributes

```
[A] PropertyCallExpr.env              = self.env
[B] VariableExpr.env                  = self.env
[C] ConstantExp.env                   = self.env
[D] LetExpression.env                 = self.env
[E] ActionExpression.env              = self.env
[F] IfExpression.env                  = self.env
[G] OclOperationWithTypeSource.env    = self.env
```

Disambiguating rules

The disambiguating rules are defined in the children.

VariableExpression

A variable expression is just a name.

```
VariableExpression ::= name
```

Synthesized attributes

```
self.referredVariable = env.lookup(name.value, VariableDeclaration)
self.type = referredVariable.type
```

Inherited attributes

```
-- none
```

Disambiguating rule

name must be a name of a visible VariableDeclaration in the current environment.

```
env->exists(el | el.name = name.value and el.oclIsKindOf(VariableDeclaration))
```

name

This rule results in an instance of String.

```
name                ::= <String>
```

Synthesized attributes

A name results in a String. 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 reasons for this is internationalization.

```
self.value = String
```

Inherited attributes

```
-- none
```

Disambiguating rules

```
-- none
```

ConstantExpression

Constant expressions are probably a special case that needs to be done explicitly for each type of constant we can have.

```
ConstantExpression ::= EnumConstantExp
```

Synthesized attributes

```
self.type = EnumConstantExp.type
```

Inherited attributes

```
EnumConstantExp.env = self.env
```

Disambiguating rules

```
-- none
```

EnumConstantExpression

Enumeration Constant expressions.

```
EnumConstantExp ::= name '::' name
```

Synthesized attributes

```
self.type = env.lookup(name[1].value, Classifier)
self.referredEnumLiteral = self.type.literal->select(l | l.name = name[2].value)->any()
```

Inherited attributes

```
-- none
```

Disambiguating rules

```
-- none
```

PropertyCallExpr

```
[A] PropertyCallExpr ::= ModelPropertyCallExp
[B] PropertyCallExpr ::= IteratorExp
[C] PropertyCallExpr ::= IterateExp
[D] PropertyCallExpr ::= OclOperationWithTypeArgument
[E] PropertyCallExpr ::= OclOperationWithStateArgument
```

Synthesized attributes

```
[A] self.type = ModelPropertyCall.type
[B] self.type = IteratorExp.type
[C] self.type = IterateExp.type
[D] self.type = OclOperationWithTypeArgument.type
[E] self.type = OclOperationWithStateArgument.type
```

Inherited attributes

```
[A] ModelPropertyCall.env              = self.env
[B] IteratorExp.env                    = self.env
[C] IterateExp.env                     = self.env
[D] OclOperationWithTypeArgument.env   = self.env
[E] OclOperationWithStateArgument.env = self.env
```

Disambiguating rules

The disambiguating rules are defined in the children.

IteratorExp

```
IteratorExp ::= source "->" name "(" iterator "|" body ")"
```

Synthesized attributes

The type of the iterator expression depends on its name.

```
self.type = if name.value = 'forAll'        then Boolean
            else if name.value = 'exists'   then Boolean
            else if name.value = 'select'   then source.type
            else if name.value = 'collect'  then source.collectionType()(body.type)
            else if name.value = 'isUnique' then Boolean
            else if name.value = 'sortedBy' then SequenceType(source.type.elementType)
            endif endif endif endif endif endif
```

Inherited attributes

Inside an iterator expression the body is evaluated with a new environment that includes the iterator variable. The calculation of the new environment needs to take into account the fact that both the name and the type of the iterator variable may be implicit.

```
body.env = self.env->including( new VariableDeclaration(
               if iterator.name->notEmpty then iterator.name else <<noname>> endif,
               if iterator.type->notEmpty then iterator.type else source.elementType
               endif))
iterator.env = env
```

Disambiguating rules

```
-- none
```

IterateExp

```
IterateExp ::= source "->" "iterate" "(" iterator ";" result "|" body ")"
```

Synthesized attributes

```
self.type = result.type
```

Inherited attributes

Inside an iterate expression the body is evaluated with a new environment that includes the iterator variable and the result variable. The calculation of the new environment needs to take into account the fact that both the name and the type of the iterator variable may be implicit.

```
body.env = self.env->including( new VariableDeclaration(
               if iterator.name->notEmpty then iterator.name else <<noname>> endif,
               if iterator.type->notEmpty then iterator.type else source.elementType
               endif))
iterator.env = env
result.env   = env
```

Source

```
source ::= OclExpression
```

Synthesized attributes

```
self.type = OclExpression.type
```

Inherited attributes

```
OclExpression.env = self.env
```

Body

```
body ::= OclExpression
```

Synthesized attributes

```
self.type = OclExpression.type
```

Inherited attributes

```
OclExpression.env = self.env
```

Iterator

```
iterator ::= VariableDeclaration?
```

Synthesized attributes

```
self.type = VariableDeclaration.type
self.name = VariableDeclaration.name
```

Inherited attributes

```
VariableDeclaration.env = self.env
```

Disambiguating rules

An iterator may not have an initExpression

```
self.initExpression->isEmpty()
```

Result

```
result ::= VariableDeclaration
```

Synthesized attributes

```
self.type = VariableDeclaration.type
self.name = VariableDeclaration.name
```

Inherited attributes

```
VariableDeclaration.env = self.env
```

Disambiguating rules

A result variable declaration must have a name and an initial value

```
self.type->notEmpty() and self.initExpression->notEmpty()
```

VariableDeclaration

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

```
VariableDeclaration::= varName (":" type)? ( "=" initExpression )?
```

Synthesised attributes

```
self.name = varName.name
```

Inherited attributes

```
initExpression.env = self.env
type.env           = self.env
```

Disambiguating rules

If the type and the initExpression are present, the type of the initExpression must conform to the declared type of the variable.

```
type->notEmpty() and initExpression->notEmpty()
                implies initExpression.type.conformsTo(type.type)
```

varName

```
varName ::= name
```

Synthesised attributes

```
self.name = name.value
```

Inherited attributes

```
-- none
```

Disambiguating rules

```
-- none
```

Type

```
type ::= name
```

Synthesised attributes

The value of this production the Classifier named ’name’.

```
self.type = env.lookup(name.value, Classifier)
```

Inherited attributes

```
-- none
```

Disambiguating rules

name must be a name of a Classifier in current environment

```
env->exists(el | el.name = name.value and el.oclIsKindOf(Classifier) )
```

initExpression

```
initExpression      ::= OclExpression
```

Synthesised attributes

```
self.type = OclExpression.type
```

Inherited attributes

```
OclExpression.env = self.env
```

Disambiguating rules

```
-- none
```

ModelPropertyExp

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

```
[A] ModelPropertyExp ::= OperationCallExp
[B] ModelPropertyExp ::= AttributeCallExp
[C] ModelPropertyExp ::= NavigationCallExp
```

Synthesised attributes

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

```
[A] self.type = OperationCallExp.type
[B] self.type = AttributeCallExp.type
[C] self.type = NavigationCallExp.type
```

Inherited attributes

```
[A] OperationCallExp.env  = self.env
[B] AttributeCallExp.env  = self.env
[C] NavigationCallExp.env = self.env
```

Disambiguating rules

These are defined in the alternatives

OperationCallExp

An operation call has three different forms. Syntax A is used if the operation is an infox operator. The source object cannot be implicit. Syntax B and C are used for collection operations and other operations respectively.

```
[A] OperationCallExp  ::= source name arguments
[B] OperationCallExp  ::= (source "->") name isMarkedPre? "(" arguments? ")"
[C] OperationCallExp  ::= (source ".")? name isMarkedPre? "(" arguments? ")"
```

Synthesised attributes

```
self.referredOperation = -- TBD
self.type = referredOperation.parameters>select(kind=ParameterDirectionKind::return).type
```

NB : This assumes exactly one return parameter, otherwise we get a tuple type !!

```
self.isMarkedPre = if isMarkedPre->notEmpty() then isMarkedPre.value else false endif
```

Inherited attributes

```
source.env    = self.env
arguments.env = self.env
```

Disambiguating rules

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

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

[B] The name of the referred Operation must be one of the names of the predefined collection operations.

```
Set{'select','collect','reject','forAll','exists','isUnque','size','isEmpty',
         'notEmpty','count','union','difference'}->includes(name.value)
```

[B] The source must be a collection.

```
source.type.isCollectionType()
```

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

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

[C] The source must not be a collection.

```
source->notEmpty() implies source.type.notCollectionType()
```

[C] The referred opperation name is the name of an Operation of the type of 'source' or if source is empty the name of an operation of self or any of the iterator variables in (nested) environment

```
-- TBD
```

[C] The type of arguments must match (see hasSameSignature() in UML semantics) The OCL has been written for Attribute and Navigation, Operation needs TBD.

```
-- TBD
```

AttributeCallExp

```
AttributeCallExp   ::= (source ".")? name isMarkedPre?
```

Synthesised attributes

The variable matchingNavigations is the set of all association ends that have the name 'name' and are defined in either self or an iterator variable in scope. Note that *allAttributes* is defined as an additional operation on *Classifier* in the UML 1.4 semantics.

```
def: Let matchingAttributes : Set(Attribute) =
     env.allIterators->select(it|it.type.allAttributes->includes(at|at.name = name.value))
                 ->union( env.lookup('self',VariableDeclaration).type.allAttributes
                                      ->includes(a | a.name = name.value))
self.referredAttribute = if source->notEmpty() then
                            source.type.allAttributes->select(a | a.name = name.value)
                         else
                            matchingAttributes.any()
                         endif
self.type = referredAttribute.type
self.isMarkedPre = if isMarkedPre->notEmpty() then isMarkedPre.value else false endif
```

Inherited attributes

```
source.env = self.env
```

Disambiguating rules

'name' is name of an Attribute of the type of source or if source is empty the name of exactly one attribute of 'self' or any of the iterator variables in (nested) scope. In OCL:

```
if source->notEmpty() then
   source.type.allAttributes->includes(a | a.name = name.value)
else
```

```
   matchingAttributes->size() = 1
endif
```

NavigationCallExp

```
NavigationCallExp  ::= (source ".")? name isMarkedPre?
```

Synthesised attributes

The variable matchingNavigations is the set of all association ends that have the name ’name’ and are defined in either self or an iterator variable in scope. Note that *allOppositeAssociationEnds* is defined as an additional operation on *Classifier* in the UML 1.4 semantics.

```
def: Let matchingNavigations : Set(AssociationEnd) =
                env.allIterators->select(it |
                  it.type.allOppositeAssociationEnds->includes(at | at.name = name.value))
                  ->union(
                  env.lookup('self', VariableDeclaration).type.allOppositeAssociationEnds
                              ->includes(a | a.name = name.value))
self.referredAssociationEnd =
               if source->notEmpty() then
                  source.type.allOppositeAssociationEnds->select(a|a.name = name.value)
               else
                  matchingNavigations.any()
               endif
self.type = Set(referredAssociationEnd.participant) or referredAssociationEnd.participant
            -- (The type depends on multiplicity)
self.isMarkedPre = if isMarkedPre->notEmpty() then isMarkedPre.value else false endif
```

Inherited attributes

```
source.env = self.env
```

Disambiguating rules

’name’ is name of AssociationEnd of the type of source or if source is empty the name of exactly one AssociationEnd of ’self’ or any of the iterator variables in (nested) scope

```
if source->notEmpty() then
   source.type.allOppositeAssociationEnds->includes(a | a.name= name.value)
else
   matchingNavigations->size() = 1
endif
```

isMarkedPre

```
isMarkedPre ::= "@" "pre"
```

Synthesised attributes

This production returns a boolean value, which is always equal to true.

```
self.value = true
```

Inherited attributes

```
-- none
```

Disambiguating rules

```
-- none
```

Arguments

```
arguments ::= OclExpression ( "," OclExpression )*
```

Synthesised attributes

The attribute ’types’ is a sequence that includes all types of the OclExpressions in the production in the order in which they appear.

```
self.types : Sequence(Classifier) = Set(OclExpression.type)
```

Inherited attributes

```
OclExpression.env = self.env
```

Disambiguating rules

```
-- none
```

OclOperationWithTypeArgument

```
OclOperationWithTypeArgument ::= source "." name "(" typeName ")"
```

Synthesised attributes

The type of the operation depends on its name.

```
self.type = if      name.value = 'oclIsTypeOf' then Boolean
            else if name.value = 'oclIsKindOf' then Boolean
            else if name.value = 'oclAsType'   then typeName.classifier
            endif endif enfif
self.argument = typeName.classifier
```

Inherited attributes

```
source.env   = self.env
typeName.env = self.env
```

Disambiguating rules

```
-- none
```

typeName

This is the argument for OclOperationWithTypeArguments operation. The name refers to a Clasifier.

```
typeName ::= name
```

Synthesised attributes

The type is the classifier named 'name' inm the current environment.

```
self.classifier = env->select(el |
                              el.name = name.value and el.oclIsKindOf(Classifier))->any()
```

Inherited attributes

```
-- none
```

Disambiguating rules

'name' must be the name of exactly one Classifier in scope

```
env->select(el | el.name = name.value and el.oclIsKindOf(Classifier) )->size() = 1
```

OclOperationWithStateArgument

```
OclOperationWithStateArgument ::= source "." name "(" stateName ")"
```

Synthesised attributes

The type of the operation depends on its name.

```
self.type = Boolean
self.argument = -- TBD the state in the statemachine attached to the 'type' of 'source'
                -- with the name stateName.name.
```

Inherited attributes

```
source.env = self.env
```

Disambiguating rules

```
-- none
```

stateName

This is the argument for OclOperationWithStateArgument operation. The name refers to a State.

```
stateName ::= name
```

Synthesised attributes

```
self.name = name.value
```

Inherited attributes

```
-- none
```

Disambiguating rules

```
-- none
```

LetExpression

```
LetExpression       ::= "Let" variable "in" in
```

Synthesised attributes

```
self.type = in.type
```

Inherited attributes

```
in.env = self.env->including(variable)
variable.env = self.env
```

Disambiguating rules

The variable name must bne unique in the current scope

```
env->forAll(el | el.name <> variable.name)
```

variable

```
variable            ::= VariableDeclaration
```

Synthesised attributes

```
self.type = VariableDeclaration.type
self.name = VariableDeclaration.name
```

Inherited attributes

```
VariableDeclaration.env = self.env
```

Disambiguating rules

A variable declaration inside a Let must have a declared type and an initial value.

```
self.type->notEmpty() and self.initExpression->notEmpty()
```

in

```
in                  ::= OclExpression
```

Synthesised attributes

```
self.type := OclExpression.type
```

Inherited attributes

```
OclExpression.env = self.env
```

Disambiguating rules

```
-- none
```

ActionExpression

The actionName must either be the name of a Signal, or the name of an Operation belonging to the target object(s).

```
ActionExpression ::= "when" condition "to" target "send"
                      actionName "(" arguments ")"
```

Synthesised attributes

```
-- Act
```

Inherited attributes

```
-- TBD
```

Disambiguating rules

```
-- none
```

IfExpression

```
IfExpression ::= "if" condition "then" thenExpression
                                  "else" elseExpression "endif"
```

Synthesised attributes

```
self.type := thenExpression.type
```

Inherited attributes

```
condition.env      = self.env
thenExpression.env = self.env
elseExpression.env = self.env
```

Disambiguating rules

```
-- none
```

condition

```
condition ::= OclExpression
```

Synthesised attributes

```
self.type := OclExpression.type
```

Inherited attributes

```
OclExpression.env = self.env
```

Disambiguating rules

```
-- none
```

thenExpression

```
thenExpression ::= OclExpression
```

Synthesised attributes

```
self.type := OclExpression.type
```

Inherited attributes

```
OclExpression.env = self.env
```

Disambiguating rules

```
-- none
```

elseExpression

```
elseExpression ::= OclExpression
```

Synthesised attributes

```
self.type := OclExpression.type
```

Inherited attributes

```
OclExpression.env = self.env
```

Disambiguating rules

```
-- none
```

target

```
target ::= OclExpression
```

Synthesised attributes

```
self.type := OclExpression.type
```

Inherited attributes

```
OclExpression.env = self.env
```

Disambiguating rules

```
-- none
```

actionName

```
actionName ::= Name
```

Synthesised attributes

```
self.name = name.value
```

Inherited attributes

```
-- none
```

Disambiguating rules

```
-- none
```

4.2.1 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.

4.2.2 Environment definition

The Enviroment is defined (informally) with the following operations.

```
context Environment::lookup(name, type : Classifier) : ModelElement
   pre :
   post: result = self->select(el|el.name=name and el.oclIsKindOf(type)).any()

context Environment::add(name, type : Classifier)
   pre : not self->exists(el|el.name=name and el.oclIsKindOf(type))
   post: self->exists(el|el.name=name and el.oclIsKindOf(type))
```

4.3 CONCRETE TO ABSTRACT SYNTAX MAPPING

The mapping from concrete to abstract syntax is described as part of the grammar. It is governed by the following rules:

- A production for a nonterminal with the name of a metaclass from the abstract syntax maps to an instance of this metaclass. In the grammar these nonterminals (when appearing at the left-hand side of a production rule) are marked with the color `magenta`.
- A production for a nonterminal that represents an association-end from the abstract syntax results in an instance of the corresponding metaclass. In the grammar these nonterminals (when appearing at the left-hand side of a production rule) are marked with the color `green`
- For any other nonterminal the resulting instance is explicitly mentioned in the grammar.
- A production for a nonterminal with the name of a metaclass (`magenta`) contains at the righthandside all the attributes and all the associationEnds [exception: next bullet] that are defined in the abstract syntax. The value of these attributes and association-ends become the value resulting from the productions (in `green`) for these attribute and association-end production rules.
- Exception to the previous rule: for each association, only one of the association-ends will appear in the grammar. The other end is defined as its reverse, by definition.

NOTE: This mapping can be described by adding a synthesized attribute to each production which gets the corresponding metaclass as its type. This has not been done yet, but allows the mapping to be fully formazied within the attribute grammar formalism.

4.4 ABSTRACT SYNTAX TO CONCRETE SYNTAX MAPPING

Comment – This section needs to be done

Basically use the grammar rules as production rules to produce the concrete syntax.

5 SEMANTICS

This section formally defines the syntax and semantics of OCL. Most of the material in this section is based on work presented in [Ric01]. This section is organized as follows. Section 5.1 defines the concept of object models. Object models provide information used as context for OCL expressions and constraints. Section 5.2 defines the type system of OCL and the set of standard operations. Finally, Section 5.3 defines the syntax and semantics of OCL expressions.

5.1 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 5.1.1 proceeds with a formal definition of the syntax of object models. The semantics of object models is defined in Section 5.1.2. This section also defines the notion of system states as snapshots of a running system.

5.1.1 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 5.1.1.7 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.

5.1.1.1 TYPES

Types are considered in depth in Section 5.2. 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*).

5.1.1.2 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 5.1 (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.

5.1.1.3 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 5.2 (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 $a : t_c \to t$ 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.

$$\forall t, t' \in T : (a : t_c \to t \in \text{ATT}_c \text{ and } a : t_c \to t' \in \text{ATT}_c) \implies t = t'$$

Attributes with the same name may, however, appear in different classes that are not related by generalization. Details are given in Section 5.1.1.6 where we discuss generalization. The set of attribute names and class names need not be disjoint.

5.1.1.4 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 5.3 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 5.3. 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 5.2.

DEFINITION 5.3 (OPERATIONS)
Let t and $t_1, \dots, t_n$ 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 $\omega : t_c \times t_1 \times \cdots \times t_n \to t$ 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.

5.1.1.5 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 5.4 (ASSOCIATIONS)
The set of associations is given by

i. a finite set of names $\textsc{Assoc} \subseteq \mathcal{N}$,

ii. a function $\mathrm{associates} : \begin{cases} \textsc{Assoc} \to \textsc{Class}^+ \\ as \mapsto \langle c_1, \ldots, c_n \rangle \text{ with } (n \geq 2) \end{cases}$.

$\square$

The function $\mathrm{associates}$ maps each association name $as \in \textsc{Assoc}$ 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 $\mathrm{associates}(sa) = \langle c, c \rangle$. The function $\mathrm{associates}$ 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 5.5 (ROLE NAMES)
Let $as \in \textsc{Assoc}$ be an association with $\mathrm{associates}(as) = \langle c_1, \ldots, c_n \rangle$. Role names for an association are defined by a function

$$\mathrm{roles} : \begin{cases} \textsc{Assoc} \to \mathcal{N}^+ \\ as \mapsto \langle r_1, \ldots, r_n \rangle \text{ with } (n \geq 2) \end{cases}$$

where all role names must be distinct, i.e.,

$$\forall i, j \in \{1, \ldots, n\} : i \neq j \implies r_i \neq r_j \ .$$

$\square$

The function $\mathrm{roles}(as) = \langle r_1, \ldots, r_n \rangle$ 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 5.6 (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 5.1.2.

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].

5.1.1.6 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 5.7 (GENERALIZATION HIERARCHY)
A generalization hierarchy $\prec$ is a partial order on the set of classes $\textsc{Class}$. □

Pairs in $\prec$ describe generalization relationships between two classes. For classes $c_1, c_2 \in \textsc{Class}$ 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.

$$\text{parents} : \begin{cases} \textsc{Class} \to \mathcal{P}(\textsc{Class}) \\ c \mapsto \{c' \mid c' \in \textsc{Class} \land c \prec c'\} \end{cases}$$

The full set of attributes of class c is the set $\textsc{Att}^*_c$ containing all inherited attributes and those that are defined directly in the class.

$$\textsc{Att}^*_c = \textsc{Att}_c \ \cup \bigcup_{c' \in \text{parents}(c)} \textsc{Att}_{c'}$$

We define the set of inherited user-defined operations analogously.

$$\textsc{Op}^*_c = \textsc{Op}_c \ \cup \bigcup_{c' \in \text{parents}(c)} \textsc{Op}_{c'}$$

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 5.8 (FULL DESCRIPTOR OF A CLASS)
The full descriptor of a class $c \in \textsc{Class}$ is a structure $\text{FD}_c = (\textsc{Att}^*_c, \textsc{Op}^*_c, \text{navends}^*(c))$ 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 $c \in \textsc{Class}$.

1. Attributes are defined in exactly one class.

$$\begin{aligned} &\forall (a : t_c \to t,\ a' : t_{c'} \to t' \in \textsc{Att}^*_c) : \\ &\quad (a = a' \implies t_c = t_{c'} \land t = t') \end{aligned} \qquad \textbf{(WF-1)}$$

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.

$$\forall(\omega : t_c \times t_1 \times \cdots \times t_n \to t,\ \omega : t_{c'} \times t_1 \times \cdots \times t_n \to t' \in \mathrm{OP}^*_c) : \\ (t_c = t_{c'}) \tag{WF-2}$$

3. Role names are defined in exactly one class.

$$\forall c_1, c_2 \in \mathrm{parents}(c) \cup \{c\} : \\ (c_1 \neq c_2 \implies \mathrm{navends}(c_1) \cap \mathrm{navends}(c_2) = \emptyset) \tag{WF-3}$$

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`.

$$\forall(a : t_c \to t \in \mathrm{ATT}^*_c) : \forall r \in \mathrm{navends}^*(c) : \\ (a \neq r) \tag{WF-4}$$

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()`.

5.1.1.7 FORMAL SYNTAX

We combine the components introduced in the previous section to formally define the syntax of object models.

DEFINITION 5.9 (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 5.1).

ii. ATT_c is a set of operation signatures for functions mapping an object of class c to an associated attribute value (Definition 5.2).

iii. OP_c is a set of signatures for user-defined operations of a class c (Definition 5.3).

iv. ASSOC is a set of association names (Definition 5.4).

   (a) associates is a function mapping each association name to a list of participating classes (Definition 5.4).
   (b) roles is a function assigning each end of an association a role name (Definition 5.5).
   (c) multiplicities is a function assigning each end of an association a multiplicity specification (Definition 5.6).

v. $\prec$ is a partial order on CLASS reflecting the generalization hierarchy of classes (Definitions 5.7 and 5.8).

□

5.1.2 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.

5.1.2.1 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 5.10 (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

$$\forall c_1, c_2 \in \text{CLASS} : c_1 \prec c_2 \implies I(c_1) \subseteq I(c_2) \enspace .$$

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.

5.1.2.2 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 5.11 (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 $l_{as} \in I_{\text{ASSOC}}(as)$. $\square$

5.1.2.3 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 5.12 (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):

$$\forall i \in \{1, \ldots, n\}, \forall l \in \sigma_{\text{ASSOC}}(as) :$$
$$|\{l' \mid l' \in \sigma_{\text{ASSOC}}(as) \wedge (\bar{\pi}_i(l') = \bar{\pi}_i(l))\}| \in \pi_i(\text{multiplicities}(as))$$

$\square$

5.2 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 5.2.1 defines the basic types *Integer*, *Real*, *Boolean* and *String*. Enumeration types are defined in Section 5.2.3. Section 5.2.4 introduces object types that correspond to classes in a model. Collection and tuple types are discussed in Section 5.2.5. The special types *OclAny* and *OclState* are considered in Section 5.2.6. Section 5.2.7 introduces subtype relationships forming a type hierarchy. All types and operations are finally summarized in a data signature defined in Section 5.2.8.

5.2.1 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 5.13 (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 5.14 (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.

5.2.1.1 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.

5.2.1.2 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 5.15 (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 $t \in T$. $\square$

Table 5.1 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 5.2.2, 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 5.1 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
$\omega : \mathit{Integer} \times \mathit{Integer} \to \mathit{Integer}$	$\omega \in \{+, -, *, \max, \min\}$
$\mathit{Integer} \times \mathit{Real} \to \mathit{Real}$	
$\mathit{Real} \times \mathit{Integer} \to \mathit{Real}$	
$\mathit{Real} \times \mathit{Real} \to \mathit{Real}$	
$\omega : \mathit{Integer} \times \mathit{Integer} \to \mathit{Integer}$	$\omega \in \{\mathrm{div}, \mathrm{mod}\}$
$/ : t_1 \times t_2 \to \mathit{Real}$	$t_1, t_2 \in \{\mathit{Integer}, \mathit{Real}\}$
$- : t \to t$	$t \in \{\mathit{Integer}, \mathit{Real}\}$
$\mathrm{abs} : t \to t$	
$\mathrm{floor} : t \to \mathit{Integer}$	
$\mathrm{round} : t \to \mathit{Integer}$	
$\omega : t_1 \times t_2 \to \mathit{Boolean}$	$\omega \in \{<, >, \leq, \geq\}$, $t_1, t_2 \in \{\mathit{Integer}, \mathit{Real}, \mathit{String}, \mathit{Boolean}\}$
$\omega : \mathit{Boolean} \times \mathit{Boolean} \to \mathit{Boolean}$	$\omega \in \{\mathrm{and}, \mathrm{or}, \mathrm{xor}, \mathrm{implies}\}$
$\mathrm{not} : \mathit{Boolean} \to \mathit{Boolean}$	
$\mathrm{size} : \mathit{String} \to \mathit{Integer}$	
$\mathrm{concat} : \mathit{String} \times \mathit{String} \to \mathit{String}$	
$\mathrm{toUpper} : \mathit{String} \to \mathit{String}$	
$\mathrm{toLower} : \mathit{String} \to \mathit{String}$	
$\mathrm{substring} : \mathit{String} \times \mathit{Integer} \times \mathit{Integer} \to \mathit{String}$	

Table 5.1: Schema for operations on basic types

5.2.1.3 SEMANTICS OF OPERATIONS

DEFINITION 5.16 (SEMANTICS OF OPERATIONS)
The semantics of an operation with signature $\omega : t_1 \times \cdots \times t_n \to t$ is a total function $I(\omega : t_1 \times \cdots \times t_n \to t) : I(t_1) \times \cdots \times I(t_n) \to I(t)$. □

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 5.1 analogously. The usual semantics of the boolean operations and, or, xor, implies, and not, is extended for dealing with undefined argument values. Table 5.2 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 5.2: Semantics of boolean operations

5.2.2 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). For each type $t \in T$, the constant operation undefined$_t$ $:\to t$ generates the undefined value $\perp$. The semantics is given by $I(\text{undefined}_t) = \perp$. The equality of values of the same type can be checked with the operation $=_t$: $t \times t \to$ *Boolean*. Furthermore, the semantics of $=_t$ defines undefined values to be equal. For two values $v_1, v_2 \in I(t)$, we have

$$I(=_t)(v_1, v_2) = \begin{cases} \text{true} & \text{if } v_1 = v_2 \text{, or } v_1 = \perp \text{ and } v_2 = \perp, \\ \text{false} & \text{otherwise.} \end{cases}$$

A test for inequality $\neq_t$: $t \times t \to$ *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 isDefined$_t$: $t \to$ *Boolean* and isUndefined$_t$: $t \to$ *Boolean*.

$$I(\text{isDefined}_t)(v) = I(\neq)(v, \perp)$$
$$I(\text{isUndefined}_t)(v) = I(=)(v, \perp)$$

5.2.3 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 5.17 (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 5.18 (SEMANTICS OF ENUMERATION TYPES)
The semantics of an enumeration type $t \in T_E$ is a function $I(t) = \text{literals}(t) \cup \{\perp\}$. □

5.2.3.1 OPERATIONS

There is only a small number of operations defined on enumeration types: the test for equality or inequality of two enumeration values, a test for the undefined value, and the generation of an undefined enumeration value. The syntax and semantics of these general operations was defined in Section 5.2.2 and applies to enumeration types as well.

In addition, the operation allInstances$_t$ $:\to$ *Set*(t) is defined for each $t \in T_E$ to return the set of all literals of the enumeration:

$$\forall t \in T_E : I(\text{allInstances}_t()) = \text{literals}(t)$$

5.2.4 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 5.19 (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 $c \in$ 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.

$$\text{typeOf} : \text{CLASS} \to T_C$$
$$\text{classOf} : T_C \to \text{CLASS}$$

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 5.10 on page 7.

DEFINITION 5.20 (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.

5.2.4.1 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

$$I(\text{allInstances}_{t_c} : \rightarrow \textit{Set}(t_c)) = \sigma_{\text{CLASS}}(c) \ .$$

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 5.21 (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.

$$I_{\text{ATT}}(a : t_c \rightarrow t)(\underline{c}) = \begin{cases} \sigma_{\text{ATT}}(a)(\underline{c}) & \text{if } \underline{c} \in \sigma_{\text{CLASS}}(c), \\ \perp & \text{otherwise.} \end{cases}$$

□

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 5.3 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 5.22 (SYNTAX OF NAVIGATION OPERATIONS)
Let $\mathcal{M}$ be a model

$$\mathcal{M} = (\textsc{Class}, \textsc{Att}_c, \textsc{Op}_c, \textsc{Assoc}, \mathrm{associates}, \mathrm{roles}, \mathrm{multiplicities}, \prec) \ .$$

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 5.1, 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 5.23 (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. $I({r_j}_{(as,r_i)} : t_{c_i} \rightarrow \mathit{Set}(t_{c_j}))(\underline{c}_i) = L(as)(\underline{c}_i)$.

□

5.2.5 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 5.2.7). 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*.

5.2.5.1 SYNTAX AND SEMANTICS

Since complex types are parameterized types, we define their syntax recursively by means of type expressions.

DEFINITION 5.24 (TYPE EXPRESSIONS)
Let $\hat{T}$ be a set of types and $l_1, \dots, l_n \in \mathcal{N}$ 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 $t_1, \dots, t_n \in T_{\text{Expr}}(\hat{T})$ then $\textit{Tuple}(l_1 : t_1, \dots, l_n : t_n) \in T_{\text{Expr}}(\hat{T})$.

□

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 $l_1, \dots, l_n$. 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 5.25 (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. $I(\textit{Tuple}(l_1 : t_1, \dots, l_n : t_n)) = I(t_1) \times \dots \times I(t_n) \cup \{\bot\}$.

$\square$

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.

5.2.5.2 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.

- $\text{mkSet}_t : t \times \cdots \times t \to \textit{Set}(t)$
- $\text{mkSequence}_t : t \times \cdots \times t \to \textit{Sequence}(t)$
- $\text{mkBag}_t : t \times \cdots \times t \to \textit{Bag}(t)$

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.

- $\text{mkTuple} : l_1 : t_1 \times \cdots \times l_n : t_n \to \textit{Tuple}(l_1 : t_1, \dots, l_n : t_n)$

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 5.3 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*	$\lvert C\rvert$
count : *Collection*(t) $\times t \rightarrow$ *Integer*	$\lvert C \cap \{v\}\rvert$
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) $\rightarrow t$	$\sum_{i=1}^{\lvert C\rvert} c_i$

Table 5.3: Operations for type *Collection*(t)

The operation schema in Table 5.3 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 5.3 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

$$I(\text{count} : \textit{Set}(t) \times t \rightarrow \textit{Integer})(s, v) = \begin{cases} 1 & \text{if } v \in s, \\ 0 & \text{if } v \notin s, \\ \perp & \text{if } s = \perp. \end{cases}$$

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$(\{\perp\}, \perp) = 1$ and count$(\{1\}, \perp) = 0$.

For bags (and very similar for sequences), the meaning of count is

$$\begin{aligned} &I(\text{count} : \textit{Bag}(t) \times t \rightarrow \textit{Integer})(\{\!\{v_1, \dots, v_n\}\!\}, v) \\ &\qquad = \begin{cases} 0 & \text{if } n = 0, \\ I(\text{count})(\{\!\{v_2, \dots, v_n\}\!\}, v) & \text{if } n > 0 \text{ and } v_1 \neq v, \\ I(\text{count})(\{\!\{v_2, \dots, v_n\}\!\}, v) + 1 & \text{if } n > 0 \text{ and } v_1 = v. \end{cases} \end{aligned}$$

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.

$$\begin{aligned} &I(\text{count} : \textit{Collection}(t) \times t \rightarrow \textit{Integer})(c, v) \\ &\qquad = \begin{cases} I(\text{count} : \textit{Set}(t) \times t \rightarrow \textit{Integer})(c, v) & \text{if } c \in I(\textit{Set}(t)), \\ I(\text{count} : \textit{Sequence}(t) \times t \rightarrow \textit{Integer})(c, v) & \text{if } c \in I(\textit{Sequence}(t)), \\ I(\text{count} : \textit{Bag}(t) \times t \rightarrow \textit{Integer})(c, v) & \text{if } c \in I(\textit{Bag}(t)), \\ \perp & \text{otherwise.} \end{cases} \end{aligned}$$

SET OPERATIONS

Operations on sets include the operations listed in Table 5.3. These are inherited from *Collection*(t). Operations which are specific to sets are shown in Table 5.4 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) × *Set*(t) → *Set*(t)	$S_1 \cup S_2$
union : *Set*(t) × *Bag*(t) → *Bag*(t)	$S \cup B$
intersection : *Set*(t) × *Set*(t) → *Set*(t)	$S_1 \cap S_2$
intersection : *Set*(t) × *Bag*(t) → *Set*(t)	$S \cap B$
− : *Set*(t) × *Set*(t) → *Set*(t)	$S_1 - S_2$
symmetricDifference : *Set*(t) × *Set*(t) → *Set*(t)	$(S_1 \cup S_2) - (S_1 \cap S_2)$
including : *Set*(t) × t → *Set*(t)	$S \cup \{v\}$
excluding : *Set*(t) × t → *Set*(t)	$S - \{v\}$
asSequence : *Set*(t) → *Sequence*(t)	
asBag : *Set*(t) → *Bag*(t)	

Table 5.4: 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 5.5. The operation asSequence is nondeterministic also for bags.

Signature	Semantics
union : *Bag*(t) × *Bag*(t) → *Bag*(t)	$B_1 \cup B_2$
union : *Bag*(t) × *Set*(t) → *Bag*(t)	$B \cup S$
intersection : *Bag*(t) × *Bag*(t) → *Bag*(t)	$B_1 \cap B_2$
intersection : *Bag*(t) × *Set*(t) → *Set*(t)	$B \cap S$
including : *Bag*(t) × t → *Bag*(t)	$B \cup \{\!\{v\}\!\}$
excluding : *Bag*(t) × t → *Bag*(t)	$B - \{\!\{v\}\!\}$
asSequence : *Bag*(t) → *Sequence*(t)	
asSet : *Bag*(t) → *Set*(t)	

Table 5.5: Operations for type *Bag*(t)

SEQUENCE OPERATIONS

Sequence operations are displayed in Table 5.6. 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 $\perp$ 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) $\times$ *Sequence*(t) $\rightarrow$ *Sequence*(t)	$S_1 \circ S_2$
append :	*Sequence*(t) $\times$ t $\rightarrow$ *Sequence*(t)	$S \circ \langle e \rangle$
prepend :	*Sequence*(t) $\times$ t $\rightarrow$ *Sequence*(t)	$\langle e \rangle \circ S$
subSequence :	*Sequence*(t) $\times$ *Integer* $\times$ *Integer* $\rightarrow$ *Sequence*(t)	$\pi_{i,j}(S)$
at :	*Sequence*(t) $\times$ *Integer* $\rightarrow$ t	$\pi_i(S)$
first :	*Sequence*(t) $\rightarrow$ t	$\pi_1(S)$
last :	*Sequence*(t) $\rightarrow$ t	$\pi_n(S)$
including :	*Sequence*(t) $\times$ t $\rightarrow$ *Sequence*(t)	$S \circ \langle e \rangle$
excluding :	*Sequence*(t) $\times$ t $\rightarrow$ *Sequence*(t)	$S - \langle e \rangle$
asSet :	*Sequence*(t) $\rightarrow$ *Set*(t)	
asBag :	*Sequence*(t) $\rightarrow$ *Bag*(t)	

Table 5.6: Operations for type *Sequence*(t)

FLATTENING OF COLLECTIONS

Type expressions as introduced in Definition 5.24 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 5.7 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 5.7: 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 5.3.1.2.

```
<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, $C_1 = \mathit{Bag}$, $C_2 = \mathit{Set}$, and $t = \mathit{Integer}$. 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 5.7) 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.

- $\mathrm{element}_{l_i} : \mathit{Tuple}(l_1 : t_1, \ldots, l_i : t_i, \ldots, l_n : t_n) \to t_i$
- $I(\mathrm{element}_{l_i} : \mathit{Tuple}(l_1 : t_1, \ldots, l_i : t_i, \ldots, l_n : t_n) \to t_i)(v_1, \ldots, v_i, \ldots, v_n) = v_i$

5.2.6 SPECIAL TYPES

Special types in OCL that do not fit into the categories discussed so far are *OclAny* and *OclState*.

- *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.

DEFINITION 5.26 (TYPE *OclAny*)
The set of special types is $T_S = \{\mathit{OclAny}\}$.

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 $I(\mathit{OclAny}) = \left(\bigcup_{t \in \hat{T}} I(t)\right) \cup \{\bot\}$. □

Operations on *OclAny* include equality (=) and inequality (<>) which already have been defined for all types in Section 5.2.2. 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 5.3.2.

5.2.7 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. 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 5.27 (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. *Set*(t) $\leq$ *Collection*(t),
Sequence(t) $\leq$ *Collection*(t), and
Bag(t) $\leq$ *Collection*(t).

v. 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).

vi. If $t'_i \leq t_i$ for $i = 1, \dots, n$ and $n \leq m$ then
Tuple($l_1 : t'_1, \dots, l_n : t'_n, \dots, l_m : t'_m$) $\leq$ *Tuple*($l_1 : t_1, \dots, l_n : t_n$).

vii. If $\text{classOf}(t'_c) \prec \text{classOf}(t_c)$ 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 5.3.1 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 $I(t')$:

$$\text{If } t' \leq t \text{ then } I(t') \subseteq I(t) \text{ for all types } t', t \in T.$$

5.2.8 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 5.28 (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. $T_\mathcal{M} = T_{\text{Expr}}(\hat{T})$,

ii. $\leq$ is a type hierarchy over $T_\mathcal{M}$,

iii. $\Omega_\mathcal{M} = \Omega_{T_{\text{Expr}}(\hat{T})} \cup \Omega_B \cup \Omega_E \cup \Omega_C \cup \Omega_S$.

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 $I(\omega) : I(t_1) \times \cdots \times I(t_n) \to I(t)$.

□

5.3 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 5.3.1 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 5.3.2 defines the meaning of operation specifications with pre- and postconditions.

5.3.1 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.

5.3.1.1 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 5.29 (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 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 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 $\mathbf{undefined} \in \text{Expr}_t$ 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 $e_1 \in \mathrm{Expr}_{\mathrm{Boolean}}$ and $e_2, e_3 \in \mathrm{Expr}_t$ then $\textbf{if } \boldsymbol{e_1} \textbf{ then } \boldsymbol{e_2} \textbf{ else } \boldsymbol{e_3} \textbf{ endif} \in \mathrm{Expr}_t$ and $\mathrm{free}(\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ endif}) := \mathrm{free}(e_1) \cup \mathrm{free}(e_2) \cup \mathrm{free}(e_3)$. 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 $e \in \mathrm{Expr}_t$ and $t' \le t$ or $t \le t'$ then $(\boldsymbol{e} \textbf{ asType } \boldsymbol{t'}) \in \mathrm{Expr}_{t'}$, $(\boldsymbol{e} \textbf{ isTypeOf } \boldsymbol{t'}) \in \mathrm{Expr}_{\mathrm{Boolean}}$, $(\boldsymbol{e} \textbf{ isKindOf } \boldsymbol{t'}) \in \mathrm{Expr}_{\mathrm{Boolean}}$ and $\mathrm{free}((e \text{ asType } t')) := \mathrm{free}(e)$, $\mathrm{free}((e \text{ isTypeOf } t')) := \mathrm{free}(e)$, $\mathrm{free}((e \text{ isKindOf } t')) := \mathrm{free}(e)$. This maps into some special instances of **OclOperationWithTypeArgument**.

vi. If $e_1 \in \mathrm{Expr}_{\mathit{Collection}(t_1)}$, $v_1 \in \mathrm{Var}_{t_1}, v_2 \in \mathrm{Var}_{t_2}$, and $e_2, e_3 \in \mathrm{Expr}_{t_2}$ then $\boldsymbol{e_1} \rightarrow \textbf{iterate}(\boldsymbol{v_1}; \boldsymbol{v_2} = \boldsymbol{e_2} \mid \boldsymbol{e_3}) \in \mathrm{Expr}_{t_2}$ and $\mathrm{free}(e_1 \rightarrow \mathrm{iterate}(v_1; v_2 = e_2 \mid e_3)) := (\mathrm{free}(e_1) \cup \mathrm{free}(e_2) \cup \mathrm{free}(e_3)) - \{v_1, v_2\}$. 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 $t' \le t$: if $e \in \mathrm{Expr}_{t'}$ then $e \in \mathrm{Expr}_t$.
□

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 $e_1.\omega(e_2, \dots, e_n)$. 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: $e_1 \rightarrow \omega(e_2, \dots, e_n)$. 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 5.3.1.3).

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.

5.3.1.2 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 5.30 (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 $e_1 \in \mathrm{Expr}_{\mathit{Sequence}(t_1)}$ then

$$I[\![e_1 \to \text{iterate}'(v_1 \mid e_3)]\!](\tau') = \begin{cases} I[\![v_2]\!](\tau') \\ \quad \text{if } I[\![e_1]\!](\tau') = \langle\rangle, \\ I[\![\text{mkSequence}_{t_1}(x_2,\dots,x_n) \to \text{iterate}'(v_1 \mid e_3)]\!](\tau'') \\ \quad \text{if } I[\![e_1]\!](\tau') = \langle x_1,\dots,x_n\rangle. \end{cases}$$

(b) If $e_1 \in \mathrm{Expr}_{\mathit{Set}(t_1)}$ then

$$I[\![e_1 \to \text{iterate}'(v_1 \mid e_3)]\!](\tau') = \begin{cases} I[\![v_2]\!](\tau') \\ \quad \text{if } I[\![e_1]\!](\tau') = \emptyset, \\ I[\![\text{mkSet}_{t_1}(x_2,\dots,x_n) \to \text{iterate}'(v_1 \mid e_3)]\!](\tau'') \\ \quad \text{if } I[\![e_1]\!](\tau') = \{x_1,\dots,x_n\}. \end{cases}$$

(c) If $e_1 \in \mathrm{Expr}_{\mathit{Bag}(t_1)}$ then

$$I[\![e_1 \to \text{iterate}'(v_1 \mid e_3)]\!](\tau') = \begin{cases} I[\![v_2]\!](\tau') \\ \quad \text{if } I[\![e_1]\!](\tau') = \emptyset, \\ I[\![\text{mkBag}_{t_1}(x_2,\dots,x_n) \to \text{iterate}'(v_1 \mid e_3)]\!](\tau'') \\ \quad \text{if } I[\![e_1]\!](\tau') = \{\!\{x_1,\dots,x_n\}\!\}. \end{cases}$$

□

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 5.2.4.

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 $\mathrm{O{\scriptstyle P}}_c$ are declared in a model specification. Let $\mathrm{oclexp} : \mathrm{O{\scriptstyle P}}_c \to \mathrm{Expr}$ be a partial function mapping object operations to OCL expressions. We define the semantics of an operation with an associated OCL expression as

$$I[\![\omega(p_1 : e_1,\dots,p_n : e_n)]\!](\tau) = I[\![\mathrm{oclexp}(\omega)]\!](\tau')$$

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 $\mathrm{oclexp}(\omega)$. 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 $\mathrm{oclexp}(\omega)$. 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 5.30, 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.

5.3.1.3 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].

$$
\begin{aligned}
&I[\![\, e_1 \rightarrow \text{exists}(v_1 \mid e_3)\,]\!](\tau) = \\
&\quad I[\![\, e_1 \rightarrow \text{iterate}(v1; v2 = \text{false} \mid v_2 \text{ or } e_3)\,]\!](\tau)
\end{aligned}
$$

$$
\begin{aligned}
&I[\![\, e_1 \rightarrow \text{forAll}(v_1 \mid e_3)\,]\!](\tau) = \\
&\quad I[\![\, e_1 \rightarrow \text{iterate}(v1; v2 = \text{true} \mid v_2 \text{ and } e_3)\,]\!](\tau)
\end{aligned}
$$

$$
\begin{aligned}
&I[\![\, e_1 \rightarrow \text{select}(v_1 \mid e_3)\,]\!](\tau) = \\
&\quad I[\![\, e_1 \rightarrow \text{iterate}(v1; v2 = e_1 \mid \\
&\qquad \text{if } e_3 \text{ then } v_2 \text{ else } v_2 \rightarrow \text{excluding}(v_1) \text{ endif})\,]\!](\tau)
\end{aligned}
$$

$$
\begin{aligned}
&I[\![\, e_1 \rightarrow \text{reject}(v_1 \mid e_3)\,]\!](\tau) = \\
&\quad I[\![\, e_1 \rightarrow \text{iterate}(v1; v2 = e_1 \mid \\
&\qquad \text{if } e_3 \text{ then } v_2 \rightarrow \text{excluding}(v_1) \text{ else } v_2 \text{ endif})\,]\!](\tau)
\end{aligned}
$$

$$
\begin{aligned}
&I[\![\, e_1 \rightarrow \text{collect}(v_1 \mid e_3)\,]\!](\tau) = \\
&\quad I[\![\, e_1 \rightarrow \text{iterate}(v1; v2 = \text{mkBag}_{\textit{type-of-}e_3}() \mid v_2 \rightarrow \text{including}(e_3))\,]\!](\tau)
\end{aligned}
$$

$$
I[\![\, e_1 \rightarrow \text{isUnique}(v_1 \mid e_3)\,]\!](\tau) =
$$

$$I[\![\, e_1 \rightarrow \text{iterate}(v1; v2 = \text{true} \mid v_2 \text{ and } e_1 \rightarrow \text{count}(v_1) = 1)\,]\!](\tau)$$

5.3.1.4 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 $v_1, \dots, v_n$ of types $C_1, \dots, C_n$ 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 $p_1, \dots, p_n$ 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 5.3.2.

5.3.1.5 INVARIANTS

An invariant is an expression with boolean result type and a set of (explicitly or implicitly declared) free variables $v_1 : C_1, \dots, v_n : C_n$ where $C_1, \dots, C_n$ 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.

```
$C_1$.allInstances->forAll($v_1$ : $C_1$ |
   ...
   $C_n$.allInstances->forAll($v_n$ : $C_n$ |
     <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.

5.3.2 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 $C :: \mathrm{op}(p_1 : T_1, \dots, p_n : T_n)$
  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 $\textsc{Params}_{\mathrm{op}} = \{p_1, \dots, p_n\}$. The UML model providing the definition of an operation signature also specifies the direction kind of each parameter. We use a function $kind : \textsc{Params}_{\mathrm{op}} \rightarrow \{\mathrm{in}, \mathrm{out}, \mathrm{inout}, \mathrm{return}\}$ 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 $C :: \mathtt{op}(p_1 : T_1, \dots, p_{n-1} : T_{n-1}) : T$ 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.

5.3.2.1 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 5.1 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 5.2 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

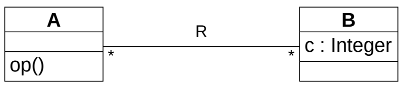
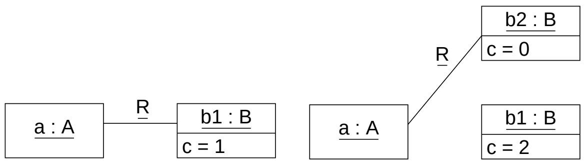


Figure 5.1: Example class diagram



(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 5.2: 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 5.2(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 5.2(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 = ⊥`
  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`

5.3.2.2 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 5.3.1. 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 5.29 for the syntax of OCL expressions to provide these additional features.

DEFINITION 5.31 (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 5.29 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 5.3.1.2). 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 5.32 (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 5.30 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. $I[\![\, \omega_{@\text{pre}}(e_1, \dots, e_n) \,]\!](\tau_{\text{pre}}, \tau_{\text{post}}) = I(\omega)(\tau_{\text{pre}})(I[\![\, e_1 \,]\!](\tau_{\text{pre}}, \tau_{\text{post}}), \dots, I[\![\, e_n \,]\!](\tau_{\text{pre}}, \tau_{\text{post}}))$

□

Standard expressions are evaluated as defined in Definition 5.30 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 5.30. Similarly, a postcondition Q satisfies a pair of pre- and post-environments, if the expression Q evaluates to true according to Definition 5.32:

$$\begin{aligned} \tau_{\text{pre}} \models P \quad &\text{iff} \quad I[\![\, P \,]\!](\tau_{\text{pre}}) = \text{true} \\ (\tau_{\text{pre}}, \tau_{\text{post}}) \models Q \quad &\text{iff} \quad I[\![\, Q \,]\!](\tau_{\text{pre}}, \tau_{\text{post}}) = \text{true} \end{aligned}$$

DEFINITION 5.33 (SEMANTICS OF OPERATION SPECIFICATIONS)
The semantics of an operation specification is a set $R \subseteq \text{Env} \times \text{Env}$ defined as

$$[\![\ \textbf{context}\ C :: \mathsf{op}(p_1 : T_1, \dots, p_n : T_n)\ \ \textbf{pre:}\ P\ \ \textbf{post:}\ Q\ ]\!] = R$$

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 5.34 (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 $f_S : \text{dom}(R) \to \text{im}(R)$ and $\text{graph}(f_S) \subseteq R$. $\square$

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.

[Ric01] M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. Dissertation, Universität Bremen, 2001. (submitted).

[Tho99] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 2nd edition, 1999.

6 The OCL Standard Library

This section describes the OCL Standard Library of predefined types and operations 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.

6.1 INTRODUCTION

The structure, syntax and semantics of the OCL is defined in the sections 3 (“Abstract Syntax”), 4 (“Concrete Syntax”) and 5 (“Semantics”). This section add another part to the OCL definition: a library of predefined types and operations. Any implementation of OCL must include this library. 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 standard OCL library includes several primitive types: Integer, Real, String and Boolean. These are mostly familiar from many other languages. The second part of the standard library consists of the collection types. They are the Bag, Set, Sequence and Collection, where Collection is an abstract type.

Note that 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).

6.2 OCLANY

6.2.1 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.

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*.

object = (object2 : OclAny) : Boolean

True if *object* is the same object as *object2*.

object <> (object2 : OclAny) : Boolean

True if *object* is a different object from *object2*.

```
post: result = not (object = object2)
```

object.oclIsNew() : Boolean

Can only be used in a postcondition.

Evaluates to true if the *object* is created during performing the operation. I.e. it didn't exist at precondition time.

6.3 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.

6.3.1 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.
Properties of Real, where the instance of Real is called *r*.

r + (r2 : Real) : Real

The value of the addition of *r* and *r2*.

r - (r2 : Real) : Real

The value of the subtraction of *r2* from *r*.

r * (r2 : Real) : Real

The value of the multiplication of *r* and *r2*.

- r : Real

The negative value of *r*.

r / (r2 : Real) : Real

The value of *r* divided by *r2*.

r.abs() : Real

The absolute value of *r*.

```
post: if r < 0 then result = - r else result = r endif
```

r.floor() : Integer

The largest integer which is less than or equal to *r*.

```
post: (result <= r) and (result + 1 > r)
```

r.round() : Integer

The integer which is closest to *r*. When there are two such integers, the largest one.

```
post: ((r - result) < r).abs() < 0.5) or ((r - result).abs() = 0.5 and (result > r))
```

r.max(r2 : Real) : Real

The maximum of *r* and *r2*.

```
post: if r >= r2 then result = r else result = r2 endif
```

r.min(r2 : Real) : Real

The minimum of *r* and *r2*.

```
post: if r <= r2 then result = r else result = r2 endif
```

r < (r2 : Real) : Boolean

True if *r1* is less than *r2*.

r > (r2 : Real) : Boolean

True if *r1* is greater than *r2*.

```
post: result = not (r <= r2)
```

r <= (r2 : Real) : Boolean

True if r1 is less than or equal to r2.

```
post: result = (r = r2) or (r < r2)
```

r >= (r2 : Real) : Boolean

True if r1 is greater than or equal to r2.

```
post: result = (r = r2) or (r > r2)
```

6.3.2 Integer

The standard type Integer represents the mathematical concept of integer. Properties of Integer, where the instance of Integer is called *i*.

- i : Integer

The negative value of *i*.

i + (i2 : Integer) : Integer

The value of the addition of *i* and *i2*.

i - (i2 : Integer) : Integer

The value of the subtraction of *i2* from *i*.

i * (i2 : Integer) : Integer

he value of the multiplication of *i* and *i2*.

i / (i2 : Integer) : Real

The value of *i* divided by *i2*.

i.abs() : Integer

The absolute value of *i*.

```
post: if i < 0 then result = - i else result = i endif
```

i.div(i2 : Integer) : Integer

The number of times that *i2* fits completely within *i*.

```
pre : i2 <> 0
post: if i / i2 >= 0 then result = (i / i2).floor() else result = -((-i/i2).floor()) endif
```

i.mod(i2 : Integer) : Integer

The result is *i* modulo *i2*.

```
post: result = i - (i.div(i2) * i2)
```

i.max(i2 : Integer) : Integer

The maximum of *i* an *i2*.

```
post: if i >= i2 then result = i else result = i2 endif
```

i.min(i2 : Integer) : Integer

The minimum of *i* an *i2*.

```
post: if i <= i2 then result = i else result = i2 endif
```

6.3.3 String

The standard type String represents strings, which can be both ASCII or Unicode. Properties of String, where the instance of String is called *string*.

string.size() : Integer

The number of characters in *string*.

string.concat(string2 : String) : String

The concatenation of *string* and *string2*.

```
post: result.size() = string.size() + string2.size()
post: result.substring(1, string.size() ) = string
post: result.substring(string.size() + 1, result.size() ) = string2
```

string.substring(lower : Integer, upper : Integer) : String

The sub-string of *string* starting at character number *lower*, up to and including character number *upper*.

6.3.4 Boolean

The standard type Boolean represents the common true/false values. Features of Boolean, the instance of Boolean is called *b*.

b or (b2 : Boolean) : Boolean

True if either *b* or *b2* is true.

b xor (b2 : Boolean) : Boolean

True if either *b* or *b2* is true, but not both.

```
post: (b or b2) and not (b = b2)
```

b and (b2 : Boolean) : Boolean

True if both *b1* and *b2* are true.

not b : Boolean

True if *b* is false.

```
post: if b then result = false else result = true endif
```

b implies (b2 : Boolean) : Boolean

True if *b* is false, or if *b* is true and *b2* is true.

```
post: (not b) or (b and b2)
```

6.4 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.

6.4.1 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.

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 5 (“Semantics”). In several cases the postcondtion refers to other collection operations, which in turn are defined in terms oof the *IterateExp* or *IteratorExp* constructs.

Properties of Collection, where the instance of Collection is called *collection*.

collection->size() : Integer

The number of elements in the collection *collection.*

```
post: result = collection->iterate(elem; acc : Integer = 0 | acc + 1)
```

collection->includes(object : T) : Boolean

True if *object* is an element of *collection*, false otherwise.

```
post: result = (collection->count(object) > 0)
```

collection->excludes(object : T) : Boolean

True if *object* is not an element of *collection*, false otherwise.

```
post: result = (collection->count(object) = 0)
```

collection->count(object : T) : Integer

The number of times that *object* occurs in the collection *collection.*

```
post: result = collection->iterate( elem; acc : Integer = 0 |
             if elem = object then acc + 1 else acc endif)
```

collection->includesAll(c2 : Collection(T)) : Boolean

Does *collection* contain all the elements of *c2* ?

```
post: result = c2->forAll(elem | collection->includes(elem))
```

collection->excludesAll(c2 : Collection(T)) : Boolean

Does *collection* contain none of the elements of *c2* ?

```
post: result = c2->forAll(elem | collection->excludes(elem))
```

collection->isEmpty() : Boolean

Is *collection* the empty collection?

```
post: result = ( collection->size() = 0 )
```

collection->notEmpty() : Boolean

Is *collection* not the empty collection?

```
post: result = ( collection->size() <> 0 )
```

collection->sum() : T

The addition of all elements in *collection*. 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 = collection->iterate( elem; acc : T = 0 | acc + elem )
```

6.4.2 Set

The Set is the mathematical set. It contains elements without duplicates. Operations of Set, the instance of Set is called *set*.

set->union(set2 : Set(T)) : Set(T)

The union of *set* and *set2.*

```
post: result->forAll(elem | set->includes(elem) or set2->includes(elem))
post: set   ->forAll(elem | result->includes(elem))
post: set2  ->forAll(elem | result->includes(elem))
```

set->union(bag : Bag(T)) : Bag(T)

The union of *set* and *bag.*

```
post: result->forAll(elem | result->count(elem) = set->count(elem) + bag->count(elem))
post: set->forAll(elem | result->includes(elem))
post: bag->forAll(elem | result->includes(elem))
```

set = (set2 : Set(T)) : Boolean

Evaluates to true if *set* and *set2* contain the same elements.

```
post: result = (set->forAll(elem | set2->includes(elem)) and
                                             set2->forAll(elem | set->includes(elem)) )
```

set->intersection(set2 : Set(T)) : Set(T)

The intersection of *set* and *set2* (i.e, the set of all elements that are in both *set* and *set2*).

```
post: result->forAll(elem | set->includes(elem) and set2->includes(elem))
post: set->forAll(elem | set2->includes(elem) = result->includes(elem))
post: set2->forAll(elem | set->includes(elem) = result->includes(elem))
```

set->intersection(bag : Bag(T)) : Set(T)

The intersection of *set* and *bag.*

```
post: result = set->intersection( bag->asSet )
```

set – (set2 : Set(T)) : Set(T)

The elements of *set*, which are not in *set2.*

```
post: result->forAll(elem | set->includes(elem) and set2->excludes(elem))
post: set->forAll(elem | result->includes(elem) = set2->excludes(elem))
```

set->including(object : T) : Set(T)

The set containing all elements of *set* plus *object*.

```
post: result->forAll(elem | set->includes(elem) or (elem = object))
post: set->forAll(elem | result->includes(elem))
post: result->includes(object)
```

set->excluding(object : T) : Set(T)

The set containing all elements of *set* without *object*.

```
post: result->forAll(elem | set->includes(elem) and (elem <> object))
post: set->forAll(elem | result->includes(elem) = (object <> elem))
```

```
post: result->excludes(object)
```

set->symmetricDifference(set2 : Set(T)) : Set(T)

The sets containing all the elements that are in *set* or *set2*, but not in both.

```
post: result->forAll(elem | set->includes(elem) xor set2->includes(elem))
post: set->forAll(elem | result->includes(elem) = set2->excludes(elem))
post: set2->forAll(elem | result->includes(elem) = set->excludes(elem))
```

set->count(object : T) : Integer

The number of occurrences of *object* in *set*.

```
post: result <= 1
```

set->flatten() : Set(T2)

If the element type is not a collection type this result in the same *set*. If the element type is a collection type, the result is the set containing all the elements of all the elements of *set*.

```
post: result = if self.type.elementType.oclIsKindOf(CollectionType) then
                  set->iterate(c; acc : Set() = Set{} |
                       acc->union(c->asSet() ) )
               else
                  set
               endif
```

set->asSet() : Set(T)

A Set identical to self. This operation exists for convenience reasons.

```
post: result = set
```

set->asSequence() : Sequence(T)

A Sequence that contains all the elements from *set*, in undefined order.

```
post: result->forAll(elem | set->includes(elem))
post: set->forAll(elem | result->count(elem) = 1)
```

set->asBag() : Bag(T)

The Bag that contains all the elements from *set.*

```
post: result->forAll(elem | set->includes(elem))
post: set->forAll(elem | result->count(elem) = 1)
```

6.4.3 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. Properties of Bag, where the instance of Bag is called *bag*.

bag = (bag2 : Bag(T)) : Boolean

True if *bag* and *bag2* contain the same elements, the same number of times.

```
post: result = (bag->forAll(elem | bag->count(elem) = bag2->count(elem)) and
bag2->forAll(elem | bag2->count(elem) = bag->count(elem)) )
```

bag->union(bag2 : Bag(T)) : Bag(T)

The union of *bag* and *bag2*.

```
post: result->forAll( elem | result->count(elem) = bag->count(elem) + bag2->count(elem))
post: bag   ->forAll( elem | result->count(elem) = bag->count(elem) + bag2->count(elem))
post: bag2  ->forAll( elem | result->count(elem) = bag->count(elem) + bag2->count(elem))
```

bag->union(set : Set(T)) : Bag(T)

The union of *bag* and *set*.

```
post: result->forAll(elem | result->count(elem) = bag->count(elem) + set->count(elem))
post: bag   ->forAll(elem | result->count(elem) = bag->count(elem) + set->count(elem))
post: set->forAll(elem | result->count(elem) = bag->count(elem) + set->count(elem))
```

bag->intersection(bag2 : Bag(T)) : Bag(T)

The intersection of *bag* and *bag2*.

```
post: result->forAll(elem |
      result->count(elem) = bag->count(elem).min(bag2->count(elem)) )
post: bag->forAll(elem |
      result->count(elem) = bag->count(elem).min(bag2->count(elem)) )
post: bag2->forAll(elem |
      result->count(elem) = bag->count(elem).min(bag2->count(elem)) )
```

bag->intersection(set : Set(T)) : Set(T)

The intersection of *bag* and *set.*

```
post: result->forAll(elem| result->count(elem) = bag->count(elem).min(set->count(elem)) )
post: bag->forAll(elem | result->count(elem) = bag->count(elem).min(set->count(elem)) )
post: set->forAll(elem | result->count(elem) = bag->count(elem).min(set->count(elem)) )
```

bag->including(object : T) : Bag(T)

The bag containing all elements of *bag* plus *object*.

```
post: result->forAll(elem |
        if elem = object then
           result->count(elem) = bag->count(elem) + 1
        else
           result->count(elem) = bag->count(elem)
        endif)
post: bag->forAll(elem |
        if elem = object then
           result->count(elem) = bag->count(elem) + 1
        else
           result->count(elem) = bag->count(elem)
        endif)
```

bag->excluding(object : T) : Bag(T)

The bag containing all elements of bag apart from all occurrences of *object*.

```
post: result->forAll(elem |
        if elem = object then
           result->count(elem) = 0
        else
           result->count(elem) = bag->count(elem)
        endif)
post: bag->forAll(elem |
        if elem = object then
```

```
            result->count(elem) = 0
         else
            result->count(elem) = bag->count(elem)
         endif)
```

bag->count(object : T) : Integer

The number of occurrences of *object* in *bag*.

bag->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 *bag*.

```
post: result = if self.type.elementType.oclIsKindOf(CollectionType) then
                  bag->iterate(c; acc : Bag() = Bag{} |
                       acc->union(c->asBag() ) )
               else
                  bag
               endif
```

bag->asBag() : Bag(T)

A Bag identical to self. This operation exists for convenience reasons.

```
post: result = bag
```

bag->asSequence() : Sequence(T)

A Sequence that contains all the elements from *bag*, in undefined order.

```
post: result->forAll(elem | bag->count(elem) = result->count(elem))
post: bag->forAll(elem | bag->count(elem) = result->count(elem))
```

bag->asSet() : Set(T)

The Set containing all the elements from *bag*, with duplicates removed.

```
post: result->forAll(elem | bag->includes(elem) )
post: bag->forAll(elem | result->includes(elem))
```

6.4.4 Sequence

A sequence is a collection where the elements are ordered. An element may be part of a sequence more than once. Operations of Sequence(T), where the instance of Sequence is called *sequence.*

sequence->count(object : T) : Integer

The number of occurrences of *object* in *sequence*.

sequence = (sequence2 : Sequence(T)) : Boolean

True if *sequence* contains the same elements as *sequence2* in the same order.

```
post: result = Sequence{1..sequence->size()}->forAll(index : Integer |
         sequence->at(index) = sequence2->at(index))
         and
         sequence->size() = sequence2->size()
```

sequence->union (sequence2 : Sequence(T)) : Sequence(T)

The sequence consisting of all elements in *sequence*, followed by all elements in *sequence2*.

```
post: result->size() = sequence->size() + sequence2->size()
post: Sequence{1..sequence->size()}->forAll(index : Integer |
       sequence->at(index) = result->at(index))
post: Sequence{1..sequence2->size()}->forAll(index : Integer |
                        sequence2->at(index) =  result->at(index + sequence->size() )))
```

sequence->flatten() : Sequence(T2)

If the element type is not a collection type this result in the same *sequence*. If the element type is a collection type, the result is the seuqnce containing all the elements of all the elements of *sequence.* The order of the elements is partial.

```
post: result = if self.type.elementType.oclIsKindOf(CollectionType) then
                  sequence->iterate(c; acc : Sequence() = Sequence{} |
                       acc->union(c->asSequence() ) )
               else
                  sequence
               endif
```

sequence->append (object: T) : Sequence(T)

The sequence of elements, consisting of all elements of *sequence*, followed by *object*.

```
post: result->size() = sequence->size() + 1
post: result->at(result->size() ) = object
post:   Sequence{1..sequence->size() }->forAll(index : Integer |
         result->at(index) = sequence ->at(index))
```

sequence->prepend(object : T) : Sequence(T)

The sequence consisting of *object*, followed by all elements in *sequence.*

```
post: result->size = sequence->size() + 1
post: result->at(1) = object
post:   Sequence{1..sequence->size()}->forAll(index : Integer |
        sequence->at(index) = result->at(index + 1))
```

sequence->subSequence(lower : Integer, upper : Integer) : Sequence(T)

The sub-sequence of *sequence* starting at number *lower*, up to and including element number *upper.*

```
pre : 1 <= lower
pre : lower <= upper
pre : upper <= sequence->size()
post: result->size() = upper -lower + 1
post: Sequence{lower..upper}->forAll( index |
         result->at(index - lower + 1) =
                          sequence->at(index))
```

sequence->at(i : Integer) : T

The *i-th* element of sequence.

```
pre : i >= 1 and i <= sequence->size()
```

sequence->first() : T

The first element in *sequence*.

```
post: result = sequence->at(1)
```

sequence->last() : T

The last element in *sequence.*

```
post: result = sequence->at(sequence->size() )
```

sequence->including(object : T) : Sequence(T)

The sequence containing all elements of *sequence* plus *object* added as the last element.

```
post: result = sequence.append(object)
```

sequence->excluding(object : T) : Sequence(T)

The sequence containing all elements of *sequence* apart from all occurrences of *object*.

The order of the remaining elements is not changed.

```
post:result->includes(object) = false
post: result->size() = sequence->size() - sequence->count(object)
post: result = sequence->iterate(elem; acc : Sequence(T)
     = Sequence{}|
         if elem = object then acc else acc->append(elem) endif )
```

sequence->asBag() : Bag(T)

The Bag containing all the elements from *sequence*, including duplicates.

```
post: result->forAll(elem | sequence->count(elem) = result->count(elem) )
post: sequence->forAll(elem | sequence->count(elem) = result->count(elem) )
```

sequence->asSequence() : Sequence(T)

The Sequence identical to the object itself. This operation exists for convenience reasons..

```
post: result = sequence
```

sequence->asSet() : Set(T)

The Set containing all the elements from *sequence*, with duplicated removed.

```
post: result->forAll(elem | sequence->includes(elem))
post: sequence->forAll(elem | result->includes(elem))
```

6.5 PREDEFINED OCLITERATOR LIBRARY

This section defines the standard OCL iterator expressions. In the abstract syntax these are all instances of OclIterator. 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 with the postcondition.

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 IterateExp
body	The body expression of the IterateExp
iterator	The iterator variable of the IterateExp
result	The result variable of the IteratorExp

6.5.1 Collection

exists

Results in true if *body* evaluates to true for at least one element in the *source* collection.

```
source->exists(iterator | body) =
                     source->iterate(iterator; 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(iterator | body ) =
                   source->iterate(iterator; 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(iterator | body) =
    source->collect(body)->forAll( iterator | source->collect(body)->count(iterator) = 1)
```

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(body) =
        ToBeDone--
```

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 Undefined.

```
source->any(iterator | body) =
       source->select(iterator | body)->asSequence()->first()
```

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
```

6.5.2 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)
```

reject

The subset of the *source* set for which *body* is false.

```
source->reject(iterator | body) =
        source->select(iterator | not body)
```

collect

The Bag of elements which results from applying *body* to every member of the *source* set. The result is flattened.

```
source->collect(iterator | body) =
        source->iterate(iterator; result : Bag(body.type) = Bag{} |
                        result->including(body->flatten() ) )
```

collectNested

The Bag of elements which results from applying *body* to every member of the *source* set.

```
source->collect(iterator | body) =
        source->iterate(iterator; result : Bag(body.type) = Bag{} |
                        result->including(body ) )
```

6.5.3 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)
```

reject

The sub-bag of the *source* bag for which *body* is false.

```
source->reject(iterator | body) =
        source->select(iterator | not body)
```

collect

The Bag of elements which results from applying *body* to every member of the *source* bag. The result is flattened.

```
source->collect(iterator | body) =
        source->iterate(iterator; result : Bag(body.type) = Bag{} |
                        result->including(body->flatten() ) )
```

collectNested

The Bag of elements which results from applying *body* to every member of the *source* bag.

```
source->collect(iterator | body) =
        source->iterate(iterator; result : Bag(body.type) = Bag{} |
                        result->including(body ) )
```

6.5.4 Sequence

The standard iterator expressions with source of type Sequence(T) are:

sequence->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)
```

reject

The subsequence of the *source* sequence for which *body* is false.

```
source->reject(iterator | body) =
        source->select(iterator | not body)
```

collect

The Sequence of elements which results from applying *body* to every member of the *source* sequence. The result is flattened.

```
source->collect(iterator | body) =
        source->iterate(iterator; result : Sequence(body.type) = Sequence{} |
                        result->append(body->flatten() ) )
```

collectNested

The Sequence of elements which results from applying *body* to every member of the *source* sequence.

```
source->collect(iterator | body) =
        source->iterate(iterator; result : Sequence(body.type) = Sequence{} |
                        result->append(body ) )
```

7

The Use of Ocl Expressions in the UML

This section describes the usage of OCL expressions in the UML.

7.1 INTRODUCTION

In principle, everywhere in the UML specification where the term *expression* is used, an OCL expression can be placed. The meaning of the value, which results from the evaluation of the OCL expression, depends on its placement within the UML model. In UML 1.4 OCL expressions could be used e.g. for invariants, preconditions and postconditons.

In this specification the structure of an expression, and its evaluation are separated from the usage of the expression. The section "Abstract Syntax" defines the structure of an expression, and the section "Semantics" defines the evaluation. For UML users (the modelers) this is not quite enough. They need to understand what the meaning and consequences are when an OCL expression is used at a certain place.

This section specifies a number of predefined places where OCL expressions can be used and their associated meaning. The modeler has to define her/his own meaning, if OCL is used at a place which is not defined in this section.

Three things need to be separated: the placement, the context, and the self instance of an ocl expression. The placement is the position where the ocl expression is used in the UML model. The context is the namespace in which the expression is evaluated. The self instance is the reference to the type of the object that evaluates the expression. For each predefined use of OCL in the next section these three things are stated explicitly. Each use also contains well-formedness rules, that define the exact structure of the place where the OCL expression is attached to the UML model.

7.2 STANDARD USES OF OCL EXPRESSIONS

7.2.1 Definition

Place:	The body of expression of UML metaclass Constraint, where the constraint has stereotype <,definition>> and is attached to a Classifier.
Context:	The Classifier to which the Constraint is attached.
Self:	The Classifier to which the Constraint is attached.

A definition constraint may only consist of one or more LetExpressions. The variable or function defined by the Let expression can be used in an identical way as an attribte 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.

Wellformedness rules

[1] The Constraint has the stereotype <<definition>>.

```
context OclExpression
inv: self.constraint.stereotype.name = 'definition'
```

[2] The Constraint is attached to only one model element.

```
context OclExpression
inv: self.constraint.constrainedElement->size() = 1
```

[3] The Constraint is attached to a Classifier.

```
context OclExpression
inv: self.constraint.constrainedElement.one(true).oclIsKindOf(Classifier)
```

7.2.2 Invariant

Place:	The body of expression of UML metaclass Constraint
Context:	The Classifier to which the Constraint is attached.
Self:	The Classifier to which the Constraint is attached.

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.

Wellformedness rules

[1] The type of the OCL expression must be Boolean.

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

[2] The Constraint has the stereotype <<invariant>>.

```
context OclExpression
inv: self.constraint.stereotype.name = 'invariant'
```

[3] The Constraint is attached to only one model element.

```
context OclExpression
inv: self.constraint.constrainedElement->size() = 1
```

[4] The Constraint is attached to a Classifier.

```
context OclExpression
inv: self.constraint.constrainedElement.one(true).oclIsKindOf(Classifier)
```

7.2.3 Precondition

Place:	The body of expression of UML metaclass Constraint
Context:	The Operation that the Constraint is attached to.
Self:	The Classifier that owns the Operation that the Constraint is attached to.

The precondition constraint 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.

Wellformedness rules

[1] The type of the OCL expression must be Boolean.

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

[2] The Constraint has the stereotype <<precondition>>.

```
context OclExpression
inv: self.constraint.stereotype.name = 'precondition'
```

[3] The Constraint is attached to only one model element.

```
context OclExpression
inv: self.constraint.constrainedElement->size() = 1
```

[4] The Constraint is attached to an Operation.

```
context OclExpression
inv: self.constraint.constrainedElement.one(true).oclIsKindOf(Operation)
```

7.2.4 Postcondition

Place:	The body of expression of UML metaclass Constraint
Context:	The Operation that the Constraint is attached to.
Self:	The Classifier that owns the Operation that the Constraint is attached to.

The postcondition constraint 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" 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.

Wellformedness rules

[1] The type of the OCL expression must be Boolean.

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

[2] The Constraint has the stereotype <<postcondition>>.

```
context OclExpression
inv: self.constraint.stereotype.name = 'postcondition'
```

[3] The Constraint is attached to only one model element.

```
context OclExpression
inv: self.constraint.constrainedElement->size() = 1
```

[4] The Constraint is attached to an Operation.

```
context OclExpression
inv: self.constraint.constrainedElement.one(true).oclIsKindOf(Operation)
```

7.2.5 Action invariant

Place:	The body of expression of UML metaclass Constraint
Context:	The Classifier that the Constraint is attached to.
Self:	The Classifier that the Constraint is attached to.

An action invariant consist of an OCL action expression. Whenever the condition evaluation changes from false to true, the specified action must be executed immediately by the instance for which the condition changed value.

Wellformedness rules

[1] The Constraint has the stereotype <<actionInvariant>>.

```
context OclExpression
inv: self.constraint.stereotype.name = ’actionInvariant’
```

[2] The Constraint is attached to only one model element.

```
context OclExpression
inv: self.constraint.constrainedElement->size() = 1
```

[3] The Constraint is attached to a Classifier.

```
context OclExpression
inv: self.constraint.constrainedElement.one(true).oclIsKindOf(Classifier)
```

7.2.6 Action postcondition

Place:	The body of expression of UML metaclass Constraint
Context:	The Operation that the Constraint is attached to.
Self:	The Classifier that owns the Operation that the Constraint is attached to.

An action postcondition consist of an OCL action expression. The condition is evaluated at postcondition time. If the condition evaluates to true, the instance that just executed the operation must have executed the specified action within the operation execution.

Within an OCL expression used in an action postcondition, the "@pre" 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.

Wellformedness rules

[1] The Constraint has the stereotype <<actionPostcondition>>.

```
context OclExpression
inv: self.constraint.stereotype.name = ’actionPostcondition’
```

[2] The Constraint is attached to only one model element.

```
context OclExpression
inv: self.constraint.constrainedElement->size() = 1
```

[3] The Constraint is attached to an Operation.

```
context OclExpression
inv: self.constraint.constrainedElement.one(true).oclIsKindOf(Operation)
```

7.2.7 Attribute initial value

Place:	The body of initialValue expression of UML metaclass Attribute
Context:	The Classifier that owns the Attribute that contains the expression.
Self:	The Classifier that owns the Attribute that contains the expression.

An OCL expression acting as the initial value of an attribute must conform to the defined type of the attribute. The OCL expression is eveluated at the creation time of the instance that owns the attribute for this created instance.

Wellformedness rules

[1] The type of the OCL expression must conform to the type of the attribute.

```
context OclExpression
inv: self.type.conformsTo(self.attribute.type)
```

7.2.8 Guard

Place:	The body of expression of the UML metaclass Guard
Context:	The guard that contains the expression. Specifically the parameters of the optional trigger event of the transition of the guard are visible within the OCL expression.
Self:	The Classifier that owns statemachine which contains the Guard.

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.

Wellformedness rules

[1] The type of the OCL expression must be Boolean.

```
context OclExpression
inv: self.type.name = ’Boolean’
```

[2] The statemachine in which the guard appears must be attached to a Classifier.

```
context OclExpression
inv: self.guard.transition.stateMachine->notEmpty() and
     self.guard.transition.stateMachine.context->notEmpty() and
     self.guard.transition.stateMachine.context.oclIsKindOf(Classifier)
```

7.3 CONCRETE SYNTAX

This section describes the concrete syntax for specifying the context of the different types of usage of OCL expressions.

Comment – This section will contain the syntax for the "context Classifier inv: ..." context declarations. This is to be done.

A

Semantics Described using UML

This appendix describes the semantics of the OCL using the UML itself to describe the semantic domain, and the mapping between semantic domain and abstract syntax. This UML-based description is added to the specification, as a courtesy to those who are not familiar with the formal notation used in chapter 5 (“Semantics”). It explains the same semantics 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].

A.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 evaluation 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 form the semantic domain.

This appendix describes the semantic domain in the form of a UML class diagram, containing 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 class diagram is used. In it, a set of so-called *evaluation* classes is defined. 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 class diagrams, resides on layer 1 of the OMG 4-layered architecture.

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 with a certain name space environment.

The UML-based semantics is divided into several packages. Figure REF shows how the packages relate to each other and to the packages from the abstract syntax.

- The *Values* package describes the semantic domain. It shows which values are predefined in OCL and which values are deduced from the UML models.
- The *Evaluations* package describes the evaluations of OCL expressions. The *Evaluations* package contains the rules that determine the result value for a given expression.
- The *Semantics* package describes the associations of the values and evaluations with elements from the abstract syntax. It is subdivided into two subpackages:
    - The *Semantics Type-Value* package contains the associations between the instances in the semantics domain and the types in the abstract syntax.

- The *Semantics Expression-Evaluation* package contains the associations between the evaluation classes with the expressions in the abstract syntax.

A.2 THE VALUES PACKAGE

OCL is an object language. Values 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 A-2 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 sub-type of Value called UndefinedValue, which is used to represent the undefined value for any Type in the abstract syntax.

Figure A-3 on page 4 show 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 (more generic) 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.

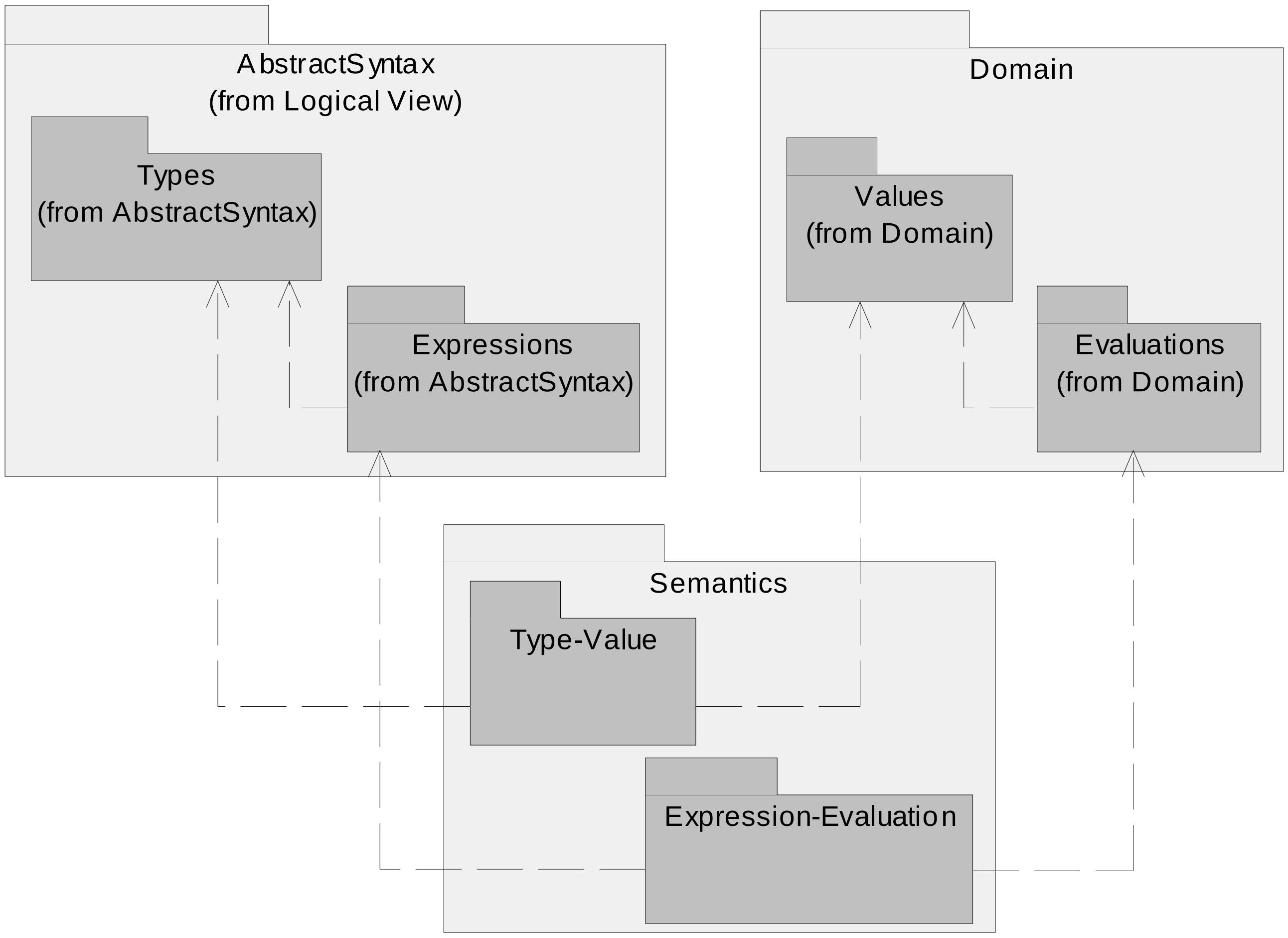


Figure A-1 *Overview of packages in the UML-based semantics*

A.2.1 Definitions of concepts for the Values package.

BagTypeValue

A bag type value is a collection value which is a multiset of elements where each element may occur multiple times in the bag. The elements are unordered. Part of a bag value are the values of its elements. In the metamodel, this is shown as an association from *CollectionValue* (a generalization of *BagTypeValue*) to *Element*.

CollectionValue

A collection value is a list of values. Part of every collection value are the values of its elements. In the metamodel, this 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. It is the generic superclass of all classes defined in this appendix, including Value and OCLExpEvaluation. 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 a name, an indexNr, and a value. The purpose of the name is to uniquely identify each component, when it is used as an element of a TupleValue. The purpose of the indexNr is to uniquely identify the position of each component, when it is used as an element of a SequenceValue.

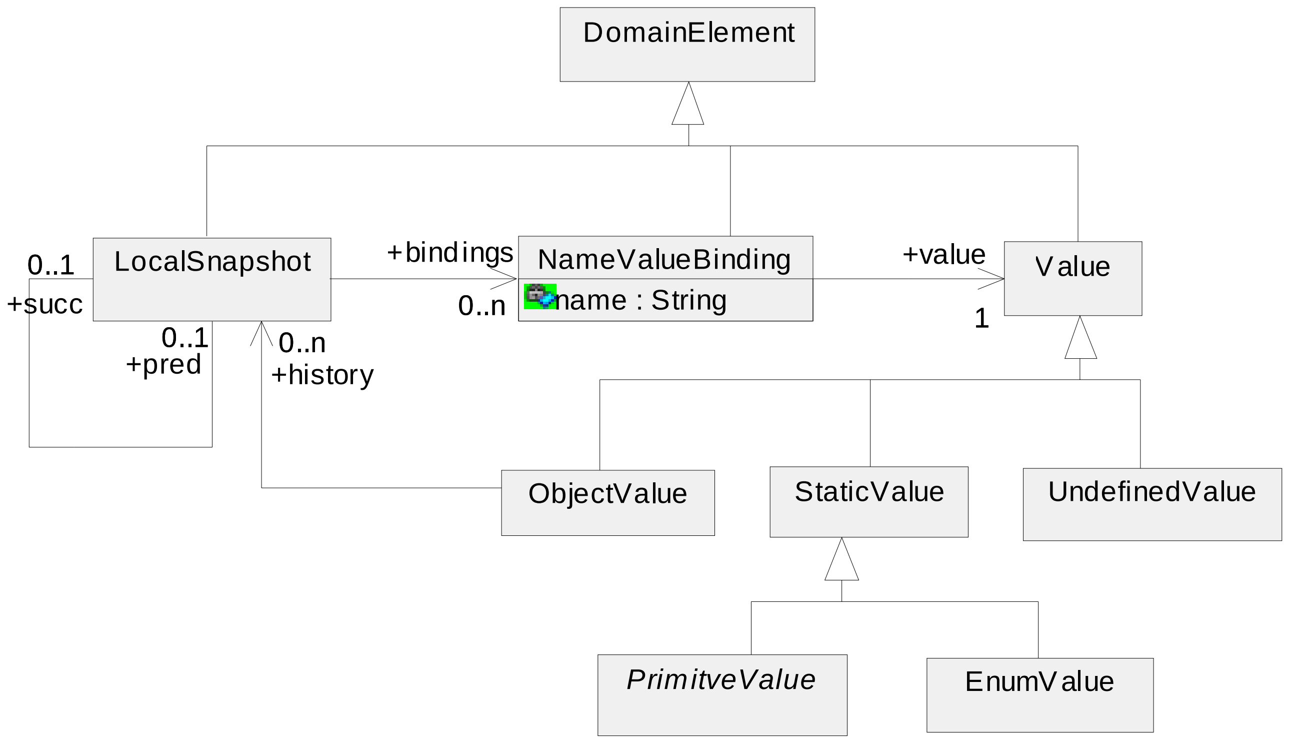


Figure A-2 *The kernel values in the semantic domain*

EnumerationValue

An enumeration value is a value taken from a set of user-specified named enumeration literals.

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, which is represented in the metamodel by the associations *pred*, and *succ*. An Object Value also holds a sequence of Actions (from the UML metamodel). This sequence of Actions will also change in time, therefore it is linked to a local snapshot. This is represented by an association in the metamodel called output-Queue.

Associations

bindings	The set of name value bindings that hold the changes in time of the subvalues of the associated object value.
outputQueue	The sequence of actions that the associated value at the certain point in time has send and are not yet put through to their targets.
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.

NameValueBinding

A name value binding is a domain element that binds a name to a value.

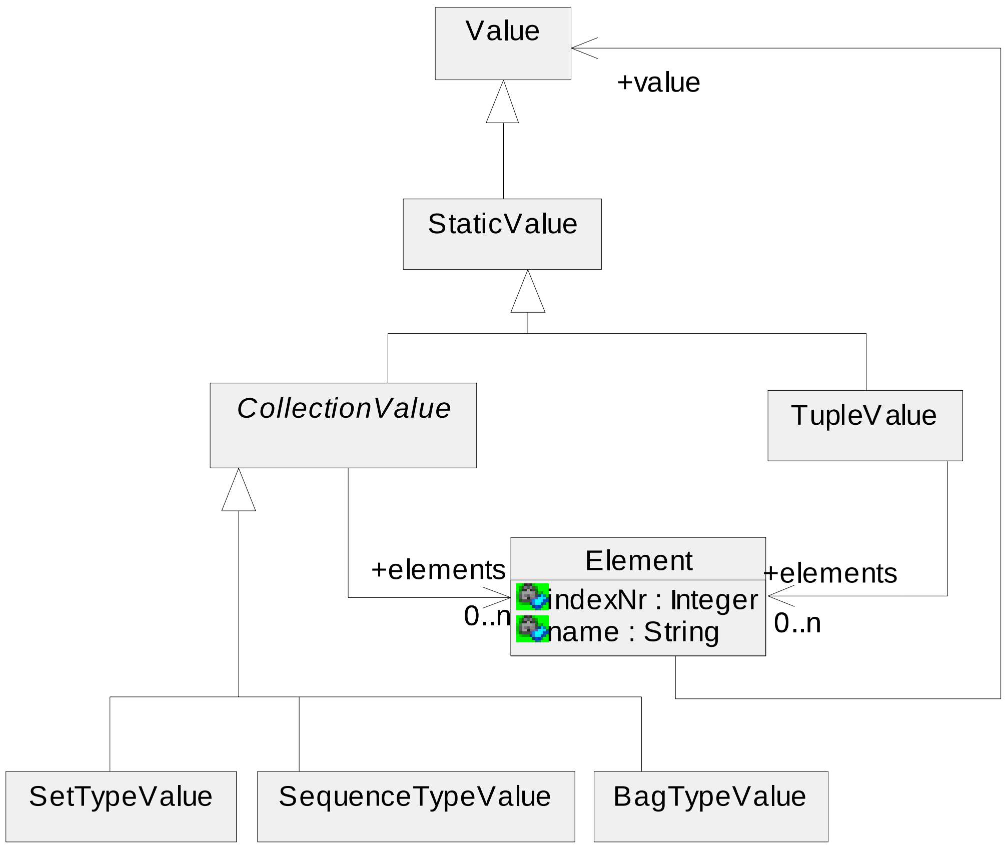


Figure A-3 *The collection values in the semantic domain*

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, and identification of these changes is necessary to describe the semantics of the action clause, 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.

PrimitiveValue

A primitive value is a predefined static value, without any relevant UML substructure (i.e., it has no UML 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. Part of a sequence value are the values of its elements. In the metamodel, this 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. Part of a set value are the values of its elements. In the metamodel, this 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 *Element*.

Associations

elements	The values of the elements in a collection.

UndefinedValue

An undefined value is a value that represent the ‘null’ value for any type.

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.

A.2.2 Well-formedness rules for the Values Package

BagTypeValue

No additional well-formedness rules.

CollectionValue

No additional well-formedness rule.

DomainElement

No additional well-formedness rule.

Element

No additional well-formedness rule.

EnumerationValue

No additional well-formedness rule.

LocalSnapshot

1. *All predecessors* is the collection of all snapshots before a snapshot. *All successors* is the collection of all snapshots after a snapshot.

```
context LocalSnapshot
def: Let allPredecessors : Set(LocalSnapshot) =
         if pred->notEmpty then
            pred->union(pred.allPredecessors)
         else
            Set {}
         endif
def: Let allSuccessors : Set(LocalSnapshot) =
         if succ->notEmpty then
            succ->union(succ.allSuccessors)
         else
            Set {}
        endif
```

NameValueBinding

No additional well-formedness rule.

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
```

2. The operation *getCurrentValueOf* results in the value that binds 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->one(name = n).value
```

PrimitiveValue

No additional well-formedness rule.

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 rule.

TupleValue

[1] All elements belonging to a tuple value have unique names.

```
self.element->isUnique(e : Element | e.name)
```

UndefinedValue

No additional well-formedness rule.

Value

No additional well-formedness rule.

A.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 A-4 on page 8 shows the core part of the Evaluations package. The basic structure in the package consists of the classes *OclEvaluation*, *PropertyCallExpEval* and *Variable-ExpEval*. An OclEvaluation always has a result value, and a name space that binds names to values. In figure A-5 on page 10 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.

A.3.1 Definitions of concepts for the Evaluations package

ActionExpEval

An action expression evaluation is defined in section A.3.4 ("Action Expression Evaluations"), but included in this diagram for completeness.

ConstantExpEval

A constant expression evaluation is an evaluation of a constant expression.

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.

IteratorExpEval

An *IteratorExp* is an expression evaluation which evaluates its *body* expression for each element of a collection.

Associations

bodyEvals	The *oclExpEvaluations* that represent the evaluation of the body expression for each element in the source collection.

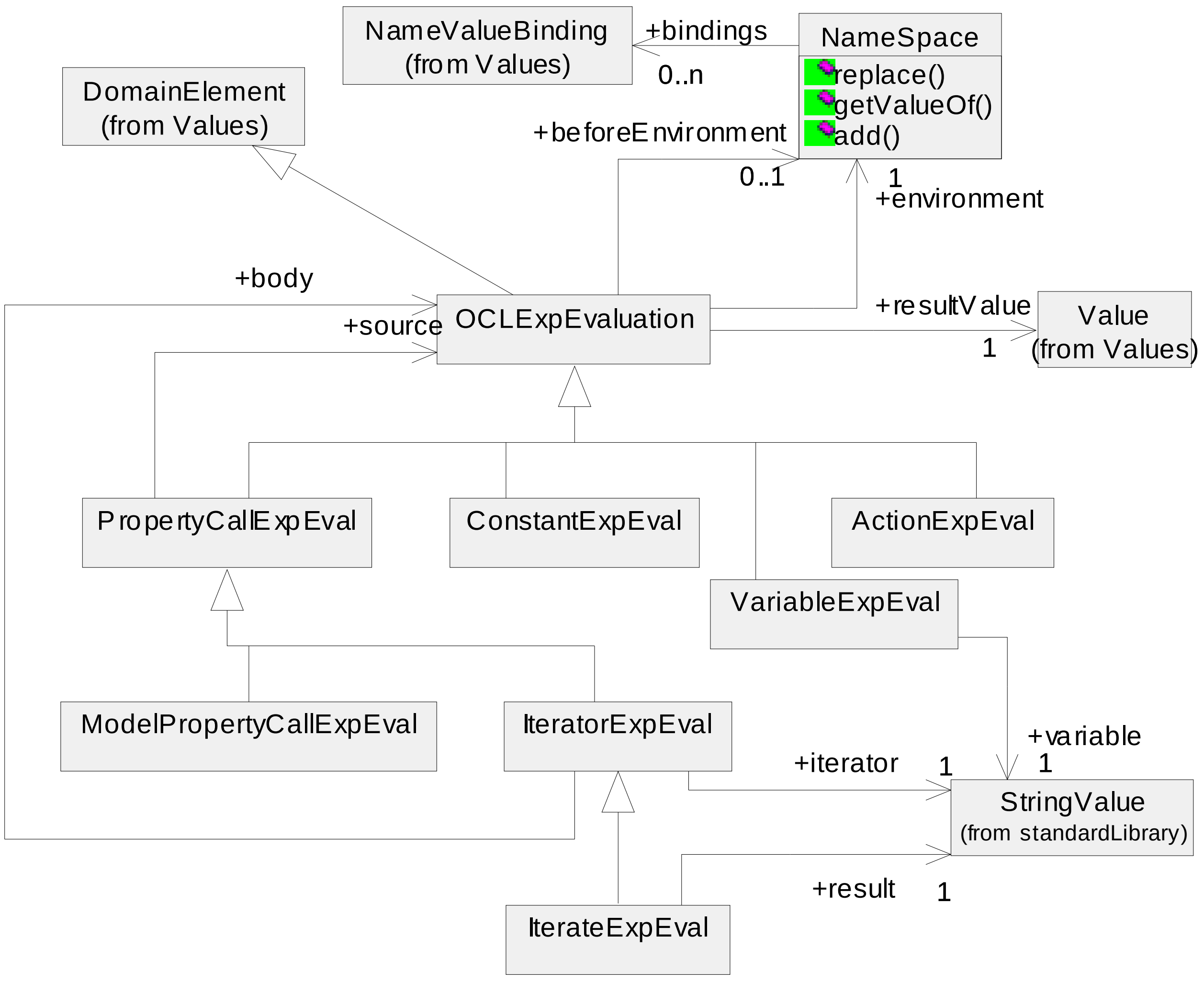


Figure A-4 *Domain model for ocl evaluations*

ModelPropertyCallExpEval

A model property call expression evaluation is an evaluation of a ModelPropertyCallExp. In A-5 on page 10 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.

NameSpace

A NameSpace is a set of NameValueBindings that form the environment in which an OCL expression is evaluated. A NameSpace has three operation which are defined in the well-formnedness rules (section A.3.8 (“Well-formedness Rules of the Evaluations package”).

Associations

bindings	The *NameValueBindings* that are the elements of this name space.

OclExpressionEval

An OCL evaluation is an evaluation or evaluation of an *OclExpression*. It has a result value, and it is associated with a set of name values 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 values bindings is used to evaluate any sub expression for which the operation *atPre* returns true, called *beforeEnvironment*.

Note that 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 that is the context for this evaluation, 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*.

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.

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 expression environment.

Associations

variable	The variable which value is the result of this evaluation.

A.3.2 Model PropertyCall Evaluations

The three subtypes of *ModelPropertyCallExpEval* are shown in figure A-5.

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 associationEnd name within the expression environment.

Associations

referredAssociationClass	The name of the *AssociationClass* to which the corresponding *AssociationClass-CallExp* 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.

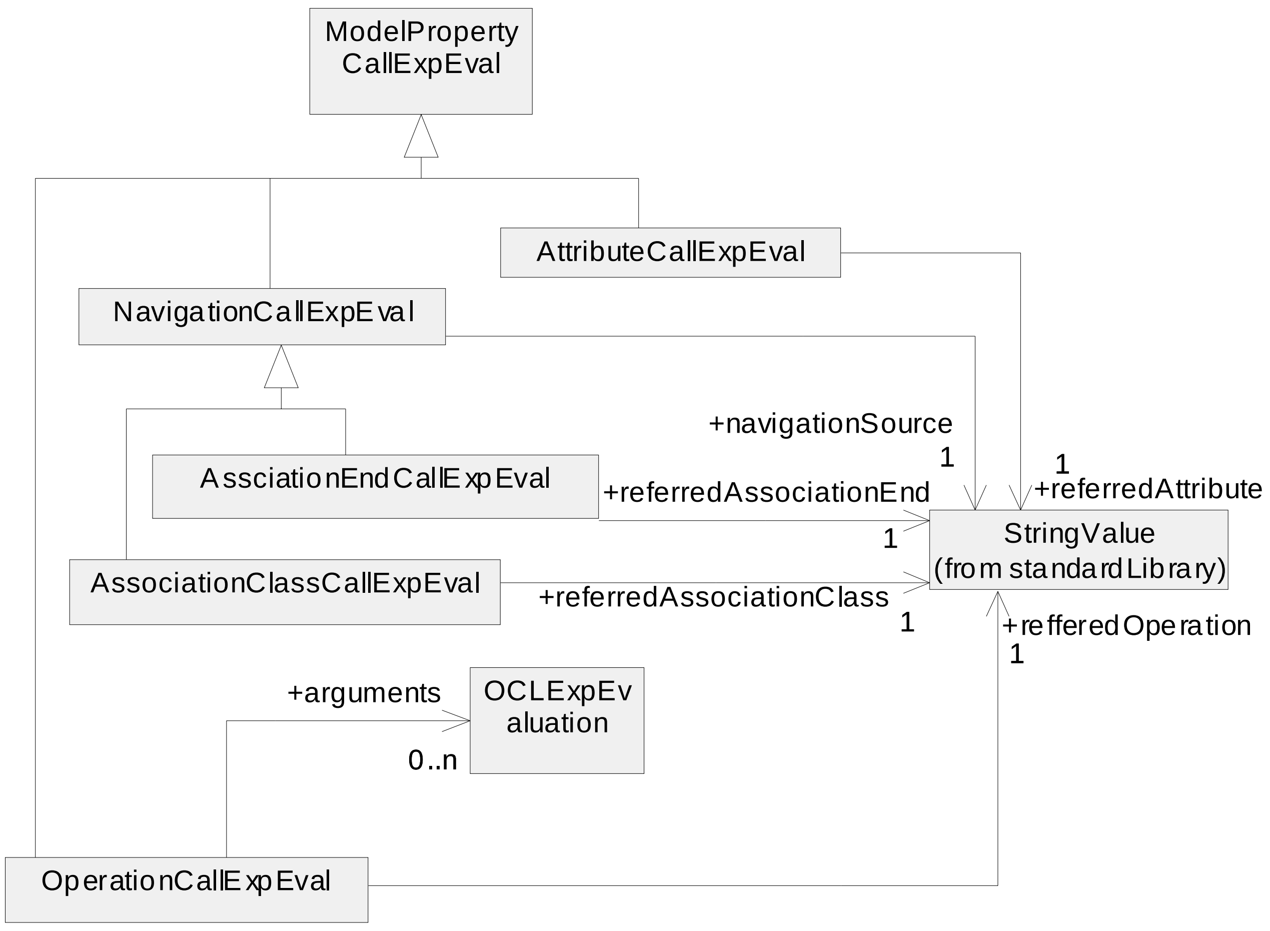


Figure A-5 *Domain model for ModelPropertyCallExpEval and subtypes*

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.

A.3.3 If Expression Evaluations

IfExpEval

An IfExpEval is an evaluation of an IfExpression.

Associations

condition	The OclExpEvaluation that evaluates the condition of the corresponding IfExpression.
thenExpression	The OclExpEvaluation that evaluates the thenExpression of the corresponding IfExpression.
elseExpression	The OclExpEvaluation that evaluates the elseExpression of the corresponding IfExpression.

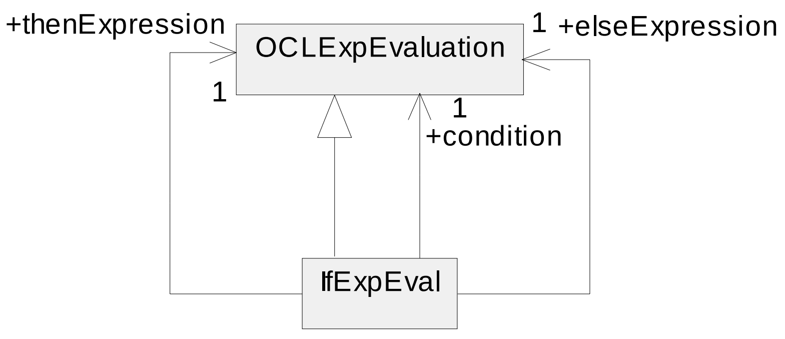


Figure A-6 *Domain model for action expression*

A.3.4 Action Expression Evaluations

Action expressions are used to specify the fact that an object has will perform some action at a some moment in time.

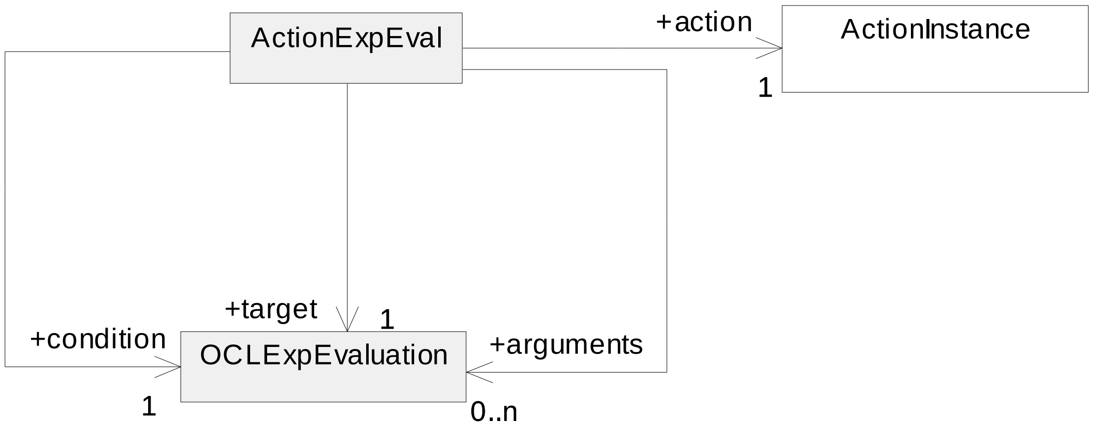


Figure A-7 *Domain model for action avaluation*

ActionExpEval

An action expression evaluation is an evaluation of an *ActionExpression.* As explained in [Kleppe2000] the only demand we can put on the action expression is that its associated action (operation call or signal) has been at some time between ‘now’ and a reference point in time in the output queue of the self instance. The ‘now’ time-point is the point in time at which this evaluation is performed. This point is represented by the *environment* link of the *ActionExpEval* (inherited from *OCLExpEvaluation*).

Note that the *Action* metaclass from the UML specification is a model element, or, in our terminology, an abstract syntax concept, therefore it can not be used in the definition of an *ActionExpEval*, which is an element of the semantic domain. Instead we must use another element of the semantic domain, which we call *ActionInstance*. Actually, *ActionInstance* should be a concept defined in the UML specification, but at the moment it is unclear wether we can substitute the *ActionExecution* concept from the Action Semantics [REF], or any other concept from the UML specification (e.g. Message, Stimulus, Signal) or Action Semantics for *ActionInstance*.

Associations

condition	The *OclExpEvaluation* that represents the evaluation of the boolean condition under which the actions are performed by the object.
target	The *OclExpEvaluation* that represents the evaluation of the target instance or instances on which the action is perfomed.
arguments	The *OclExpEvaluation* that represents the evaluation of the actual parameters to the *Operation* or *Signal*.
action	The *ActionInstance* that is send by the object when the condition is true. The action can be an instance of a *CallAction*, denoting that an *Operation* has been called, or an instance of a *SendAction*, denoting that a *Signal* has been sent.

A.3.5 Constant Expression Evaluations

This section defines the different types of constant expression evaluations in OCL. It is not a complete mirror image of the Constant Expressions in the abstract syntax, because there is no need to distinguish most of the sub-types of ConstantExp for the semantics. It does contain a metaclass for enumeration values, because there is a restriction that these come from the set of enumeration literals.

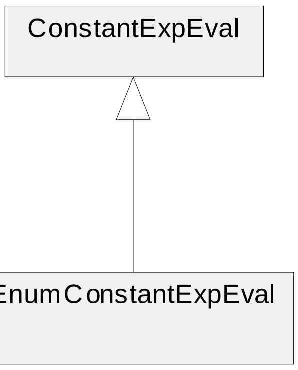


Figure A-8 *Domain model for constant expressions*

EnumConstantExpEval

An enumeration constant expression evaluation represents the evaluation of a reference to an enumeration literal.

Associations

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

A.3.6 Let expressions

LetExpEval

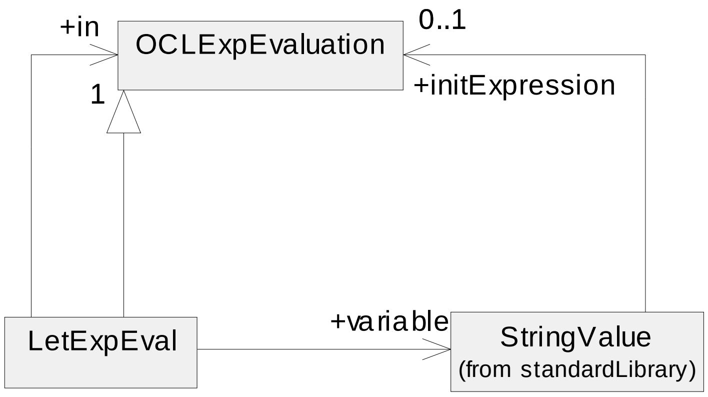


Figure A-9 *Domain model for let expression*

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

A.3.7 Operations with special source or argument

This section defines the evaluations for operations that have either as their source, or as their argment a ModelElement that is not an OclExpression, but a concept defined in the UML metamodel, e.g. Classifier, and State.

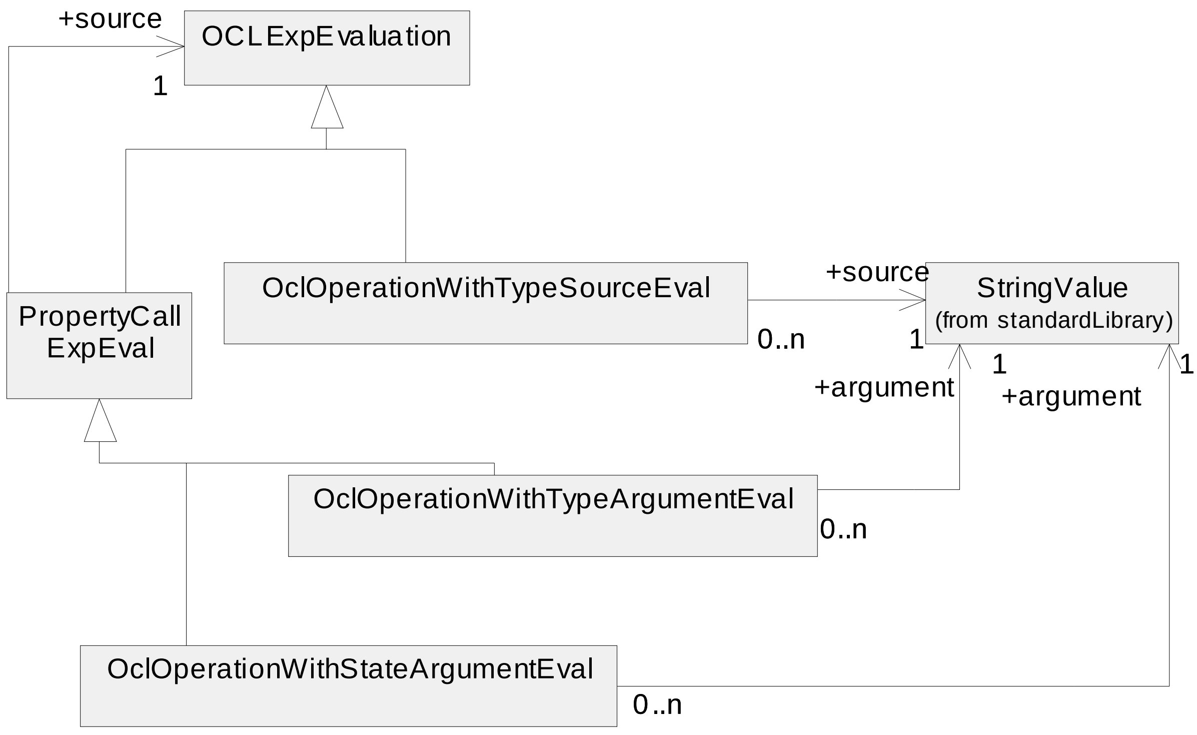

+source
1
OCLExpEvaluation
PropertyCall ExpEval
OclOperationWithTypeSourceEval
0..n
+source
1
StringValue (from standardLibrary)
1
+argument
1
+argument
1
OclOperationWithTypeArgumentEval
0..n
OclOperationWithStateArgumentEval
0..n

Figure A-10 *Domain model for predefined operations that have a non OclExpressions as their source or argument*

OclOperationWithTypeSourceEval

An OclOperationWithTypeSourceEval is an evaluation of an OclOperationWithTypeSource.

Associations

source	The source is the Classifier that ’performs’ this operation.

OclOperationWithTypeArgumentEval

An OclOperationWithTypeArgumentEval is an evaluation of an OclOperationWithTypeArgument.

Associations

argument	The argument is the Classifier that is the argiument of this operation.

OclOperationWithStateArgumentEval

An OclOperationWithStateArgument is an evaluation of an OclOperationWithStateArgument.

Associations

argument	The argument is the State that is the argument for this operation.

A.3.8 Well-formedness Rules of the Evaluations package

The metaclasses defined in the evaluations package have the following well-formednes rules. These rules state the the result value is defined. This defines the semantics of the OCL expressions.

Comment – The rules for building the name space environment of an ocl expression must still be added. These are equal to the rules given in the concrete syntax for the inherited attribute ’env’.

ActionExpEval

[1] The meaning of the action expression is that the action should have been in the output queue of the self instance at some time between ‘now’ and a reference point in time. ‘Now’ is represented by the environment of the expression, the reference point in time is represented by the before environment. The Lets in the invariant below are convenient to indicate the list of all snapshots between the ‘now’ and the reference time point.

```
context ActionExpEval inv:
Let start: LocalSnapshot = beforeEnvironment.getValueOf( ’self’ )->history->last() in
Let end: LocalSnapshot = environment.getValueOf( ’self’ )->history->last()  in
Let inBetween: Sequence( LocalSnapshot ) =
      start.allSuccessors->excluding( end.allSuccessors)->including( start ) in
resultValue = condition implies
                   inBetween->collect(outputQueue)->exists( q | q.includes( action ) )
```

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.

```
context AssociationClassCallExpEval inv:
resultValue =
   source.resultValue.getCurrentValueOf(referredAssociationClass.name)
```

AssociationEndCallExpEval

[1] The result value of an attribute call expression is the value bound to the name of the attribute to which it refers.

```
context  AssociationEndCallExpEval inv:
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 = source.resultValue.getCurrentValueOf(referredAttribute.name)
```

ConstantExpEval

[1] The result value of a constant expression is its symbol.

```
context ConstantExpEval inv:
resultvalue = symbol
```

EnumConstantExpEval

No extra well-formedness rule (follows its parent ConstantExpEval).

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
```

IterateExpEval

[1] The result value of an IterateExpEval is the result of the last of its body evaluations.

```
rule equal to that of its parent IteratorExpEval
```

[2] All sub evaluations have a different environment. The environment is the same environment as the one from its parent, plus the result variable which is bound to the result value of the last sub evaluation.

```
context IterateExpEval
inv: bodyEvals[1].environment = self.environment
      ->add( NameValueBinding( result.varName, null ))
inv: Sequence{2..source->size()}->forAll( i: Integer |
   bodyEvals[i].environment = self.environment
      ->replace( NameValueBinding( result.varName, bodyEvals[i-1].resultValue )))
```

IteratorExpEval

[1] The result value of an IteratorExpEval is the result of the last of its body evaluations.

```
context IteratorExpEval inv:
resultValue = bodyEvals->last().resultValue
```

[2] There is an OCLExpEvaluation (a sub evaluation) for every element of the source collection.

```
context IteratorExpEval
inv: bodyEvals->size() = source.value->size()
```

[3] All sub evaluations have a different environment. The environment is the same environment as the one from its parent, plus the iterator variable which is bound to the value of the next element in the source collection.

```
context IteratorExpEval
inv: bodyEvals[1].environment = self.environment

      ->add( NameValueBinding( iterator.varName, source->asSequence()->at[1] ))
inv: Sequence{2..source->size()}->forAll( i: Integer |
   bodyEvals[i].environment = self.environment
      ->including( NameValueBinding( iterator.varName, source->asSequence()->at[i] )))
```

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 ))
```

ModelPropertyCallExpEval

No additional well-formedness rules. Result value is determined by its subtypes.

NameSpace

[1] All names in a name space must be unique.

```
context NameSpace inv:
bindings->collect(name)->forAll( name: String | bindings->collect(name)->isUnique(name))
```

[2] The operation *getValueOf* results in the value that binds to the *name* parameter in the bindings of a name space. Note that the value may be the UndefinedValue.

```
context NameSpace::getValueOf(n: String): Value
pre: -- none
post: result = bindings->one(name = n).value
```

[3] The operation *replace* replaces the value of a name, by the value given in the *nvb* parameter.

```
context NameSpace::replace(nvb: NameValueBinding): NameSpace
pre: -- none
post: result.bindings = self.bindings
         ->excluding( self.bindings->any( name = nvb.name) )->including( nvb )
```

[4] The operation *add* adds the name and value indicated by the NameValueBinding given by the *nvb* parameter.

```
context NameSpace::add(nvb: NameValueBinding): NameSpace
pre: -- none
post: result.bindings = self.bindings->including( nvb )
```

NavigationCallExpEval

No additional well-formedness rules. Result value is determined by its subtypes.

OclExpressionEval

[1] The result value of an ocl expression is determine by its subtypes.

[2] Every OCLExpEvaluation has an environment in which at most one self instance is known.

```
context OCLExpEvaluation
inv: environment->select( name = 'self' )->size() = 1
```

OclOperationWithStateArgumentEval

[1] The result value of ...

```
context OclOperationWithStateArgumentEval inv:
resultValue =
```

To be done.

OclOperationWithTypeArgumentEval

[1]

```
context OclOperationWithTypeArgumentEval inv:
resultValue =
```

To be done

OclOperationWithTypeSourceEval

[1]

```
context OclOperationWithTypeSourceEval inv:
resultValue =
```

To be done

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 add a name value binding to its environment that binds a value to the name ’result’.

[1] The result of an operation call expression is equal to the result of the execution of the called operation.

```
context OperationCallEval inv:
resultValue = environment.getValueOf( ’result’ )
```

PropertyCallExpEval

No additional well-formedness rules. Result value is determined by its subtypes.

VariableExpEval

[1] The result of a VariableExpEval is the value bound to the name of the variable to which it refers.

```
context VariableExp inv:
resultValue = environment.getValueOf(referredVariable.varName)
```

A.4 THE SEMANTICS PACKAGE

The figures A-11 and A-12 show the associations between the abstract syntax concepts and the domain concepts defined in this appendix. 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 value.

Comment – The figures are not yet complete, but they show the ’general rule’ for associating abstract syntax and semantic domain.

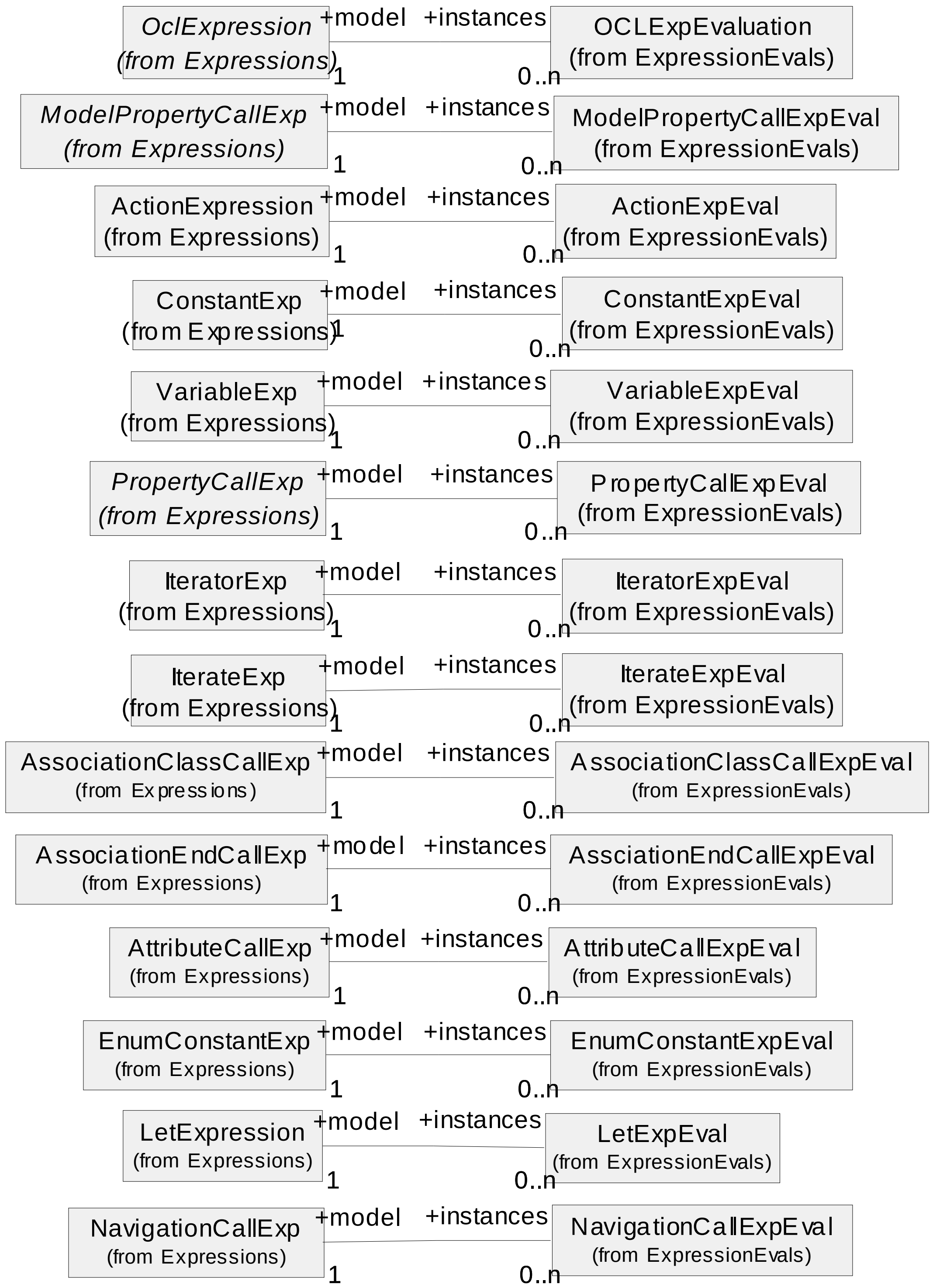

OclExpression (from Expressions)
+model
+instances
1
0..n
OCLExpEvaluation (from ExpressionEvals)
ModelPropertyCallExp (from Expressions)
ModelPropertyCallExpEval (from ExpressionEvals)
ActionExpression (from Expressions)
ActionExpEval (from ExpressionEvals)
ConstantExp (from Expressions)
ConstantExpEval (from ExpressionEvals)
VariableExp (from Expressions)
VariableExpEval (from ExpressionEvals)
PropertyCallExp (from Expressions)
PropertyCallExpEval (from ExpressionEvals)
IteratorExp (from Expressions)
IteratorExpEval (from ExpressionEvals)
IterateExp (from Expressions)
IterateExpEval (from ExpressionEvals)
AssociationClassCallExp (from Expressions)
AssociationClassCallExpEval (from ExpressionEvals)
AssociationEndCallExp (from Expressions)
AssciationEndCallExpEval (from ExpressionEvals)
AttributeCallExp (from Expressions)
AttributeCallExpEval (from ExpressionEvals)
EnumConstantExp (from Expressions)
EnumConstantExpEval (from ExpressionEvals)
LetExpression (from Expressions)
LetExpEval (from ExpressionEvals)
NavigationCallExp (from Expressions)
NavigationCallExpEval (from ExpressionEvals)

Figure A-11 *Associations between evaluations and abstract syntax concepts*

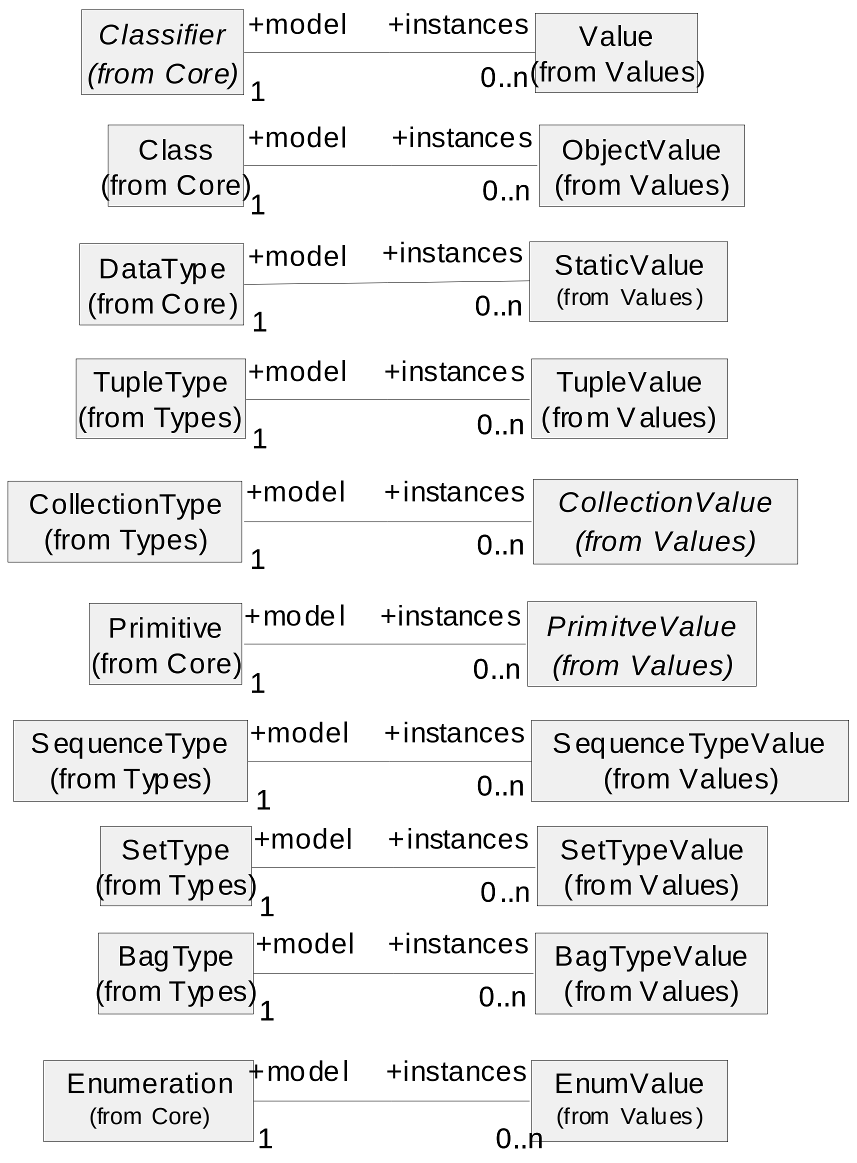


Figure A-12 *Associations between values and the types defined in the abstract syntax.*

A.4.1 Well-formedness rules for the Semantics.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 rule.

Element

No additional well-formedness rule.

EnumarationValue

No additional well-formedness rule.

ObjectValue

No additional well-formedness rule.

PrimitiveValue

No additional well-formedness rule.

SequenceTypeValue

No additional well-formedness rule.

SetTypeValue

No additional well-formedness rule.

StaticValue

No additional well-formedness rule.

TupleValue

[1] The elements in a tuple value must have a type that conforms to the type of the corresponding TupleParts.

```
context TupleValue inv:
elements->forAll( elem |
   Let correspondingPart: tuplePart =
      self.model.part->select( part | part.name = elem.name ) in
   elem.value.model.conformsTo( correspondingPart.type ) )
```

UndefinedValue

No additional well-formedness rule.

Value

[1] The additional operation isInstanceOf return 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 )
```

[2] The symbol, which is the result value, of an enumeration constant evaluation must be one of the literals of the enumeration type.

```
context EnumConstantExpEval
inv: model.type->contains( symbol )
```

A.4.2 Well-formedness rules for the Semantics.exp-eval Package

ActionExpEval

[1] The action instance of this evaluation must correspond with the Action in the action expression.
To be done.

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
```

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
```

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
```

ConstantExpEval

No additional well-formedness rule.

EnumConstantExpEval

[1] The result value of an EnumConstantExpEval must be equal to one of the literals defined in its type.

```
context EnumConstantExpEval inv:
model.literals->contains( 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
```

IterateExpEval

[1] To be done

IteratorExpEval

[1] All sub evaluations have the same model, which is the body of the associated IteratorExp.

```
context IteratorExpEval
inv: bodyEvals->forAll( model = self.model )
```

LetExpEval

[1] To be done

ModelPropertyCallExpEval

[1] To be done

NameSpace

[1] To be done

NavigationCallExpEval

[1] To be done

OclExpressionEval

[1] The result value of the evaluation of an ocl expression must be an instance of the type of that expression.

```
context OclExpEvaluation
inv: resultValue.isInstanceOf( model.type )
```

OclOperationWithStateArgumentEval

[1] To be done

OclOperationWithTypeArgumentEval

[1] To be done

OclOperationWithTypeSourceEval

[1] To be done

OperationCallExpEval

[1] To be done

PropertyCallExpEval

[1] To be done

VariableExpEval

[1] To be done

B Interchange Format

B.1 THIS APPENDIX IS INTENTIALLY LEFT BLANK.

This section contains the interchnage format for OCL. This XMI DTD should be generetaed 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,* to appear in the proceeding of the UML 2001 conference

Comment – This list of references is not complete.

Index