



Proceedings of the
Workshop on OCL and Textual Modelling
(OCL 2011)

Modeling the OCL Standard Library

Edward Willink

20 pages

Modeling the OCL Standard Library

Edward Willink¹

¹ ed_at_willink.me.uk, <http://www.eclipse.org/modeling>
Eclipse Modeling Project

Abstract: OCL is widely used by UML and other languages to constrain meta-models and perform evaluations on models. The OCL specification is the result of diligent but time-constrained human endeavor and so contains many inconsistencies, most of which are relatively easy to ignore as obvious typographical mistakes. However the need to ignore minor discrepancies undermines rigorous treatment of more significant issues. The minor issues can be substantially eliminated by auto-generating the specification. This paper provides early community visibility of proposed solutions to a variety of issues that arose while developing a model for the OCL Standard Library that forms the core of the OCL specification.

Keywords: OCL, meta-model, library, auto-generation, templates

1 Introduction

The Object Constraint Language (OCL) evolved, initially within the Unified Modeling Language (UML). As part of the UML 2.0[Obji] revision activities, OCL was separated out as a separate specification in recognition of OCL's utility in non-UML contexts. Unfortunately the UML Revision Task Force had insufficient resources to complete the revision of OCL 1.6[Objc] to align with UML 2.0. A partially revised OCL 2.0 draft[Objd] was all that was available to accompany UML 2.0.

When the QVT specification was developed, the utility of OCL was recognized and OCL 2.0[Objc] formed the basis for QVT 1.0[Objh]. The QVT Finalization Task Force also finalized the OCL 2.0 specification, but had insufficient resources to perform the very detailed proof reading and consistency checking for a specification involving so many cross-references.

The OCL 2.2[Objf] revision addressed a relatively small proportion of the outstanding issues.

The OCL 2.3[Objg] revision addressed a substantial inconsistency that arose when an undefined value evolved to a `null` or `invalid` value and a major under-specification of the concrete syntaxes of terminals. However the bulk of the inconsistencies remained.

1.1 Revision

The Object Management Group revision process for these specifications starts when an issue is raised against a specification; anyone can raise an issue. Each issue should be addressed by a Revision Task Force leading to a No-Change, Duplicate, Merged or Resolved response. In practice many issues have been Deferred through lack of time. Each issue that requires a change requires a written proposal to be prepared in which the issue is discussed and the associated editorial changes are clearly identified. Proposals are voted on by the Revision Task Force and in

due course a revised specification is prepared, approved by the Architecture Board and ultimately adopted by the OMG.

An attempt was made to pursue this process to resolve the incomplete change in alignment between UML 1.x's AssociationEnd and UML 2.x's Property class. The detailed changes proved too laborious. Now that good modeling technology tools are readily available, it is an anachronism that the OCL specification that underpins so many of the tools is maintained manually.

It was therefore decided to exploit models to capture as much of the specification as practical and then to auto-generate the specification from the models. Auto-generation should then ensure that typographic errors in technical content are almost non-existent, and should enable the detailed lists of editorial changes required by the OMG process to be auto-generated as well.

1.2 Models

The Essential OCL can be modeled using 5 models:

- the UML meta-model (UML Infrastructure)
- an OCL Standard Library Model (OCL Clause 11)
- an Abstract Syntax Meta-Model (OCL Clause 8)
- a Concrete Syntax to Abstract Syntax Mapping (OCL Clause 9)
- a Run-time Semantics Model (OCL Clause 10)

Complete OCL requires similar smaller models to flesh out Clause 12.

The models above are interdependent, with UML heavily reliant on OCL expressions, whose operators are defined by the OCL Standard Library. The OCL Standard Library has limited dependencies on the Abstract Syntax and Run-time Semantics, so defining the OCL Standard Library and consequently expression validity is the key to modeling the whole OCL specification.

Each version of an OCL 2.x specification states, in its Scope statement, that it is aligned with a corresponding UML 2.x specification. Sadly this statement is only an aspiration at present. Nonetheless we must proceed towards that goal and so clearly an OCL Standard Library Model should be describable by UML.

1.3 Paper

In this paper we discuss many of the issues that have arisen in building a UML-aligned meta-model for the OCL Standard Library and using that meta-model to define the Library. In a companion paper[[Will11](#)] we discuss issues arising from attempting to achieve UML-alignment more generally and establish the meaning of an OCL model as a combined instance of the merged UML and OCL meta-models.

It is hoped that by presenting the community with an early insight into changes that may be proposed for OCL 2.4, the community may be able to contribute constructively before, rather than after, the revised specification is adopted.

A prototype of the modeled OCL Standard Library may be found in the optional Examples and Editors of the Indigo release of Eclipse OCL[[MDT](#)], which is officially released in June 2011

and for which milestone builds have been available since December 2010. The prototype defines a Domain Specific Language to specify the OCL Standard Library, and uses Xtext[[TMF](#)] tooling to provide a rich editing capability for the DSL. This facilitates entry of OCL postconditions on library operations and validates that these postconditions are consistent with the library model.

The clarifications outlined below involve very few if any, actual changes to the user perception of OCL; they merely enable the specification to say what many users think it already says. Of course OCL implementations that have resolved ambiguities in alternative directions may see a change. However even changes in the specification need not impact compatibility, since with substantial parts of the OCL semantics migrating to the library model, it may be possible to encapsulate the semantics of a particular OCL tool version in a corresponding OCL Standard Library model and so preserve precisely those semantics for those users that require them.

In [Section 2](#) we discuss the basic properties of a library model, then in [Section 3](#) we show how a variety of OCL concepts can be captured by the library model. With a modeling capability established, in [Section 4](#) we identify aspects of the library that deserve revision and in [Section 5](#) we examine a number of operations that are difficult to model. Further minor revisions to enhance modeling are considered in [Section 6](#). Finally we conclude.

2 Library Utility

We start by identifying ways in which an OCL model of the OCL Standard Library can be useful.

2.1 Consumable

While auto-generation of the OCL specification may be a primary motivation for modeling the OCL Standard Library, provision of a consumable machine readable model is equally valuable. It avoids the need for OCL implementors to transcribe the specification, avoids associated transcription errors, and so avoids inconsistencies between alternate implementations. Availability of a model should encourage tools to use declarations in the model rather than hand coding to realize important tooling behaviors.

Definition of the library requires concepts that cannot be expressed in Essential OCL or Complete OCL, and so a Domain Specific Language with a new Concrete Syntax is used to define library elements. A simple example of part of the `Integer` definition is shown in [Figure 1](#).

The `Integer` type is defined to conform to the `Real` type and to be an instance of the `PrimitiveType` meta-type.

Four operations are defined, the first with a precedence labeled as `UNARY`, and the second with a precedence labeled as `ADDITIVE`. Precedence is discussed in [Section 3.1](#).

The first operation is unary minus. It takes no arguments and returns an `Integer`. It has a precedence and so may be used as a prefix operator.

The second operation is binary plus. It takes an `Integer` argument and returns an `Integer`. It has a precedence and so may be used as an infix operator.

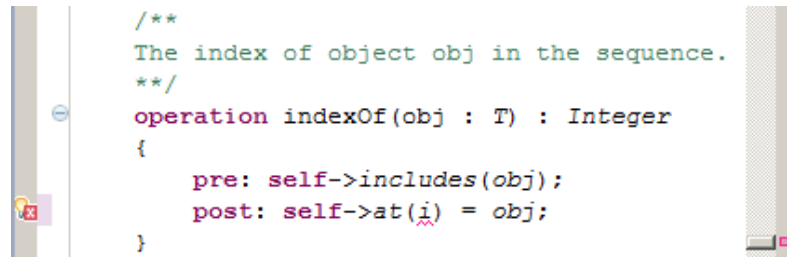
The third and fourth operations have simple names that do not require quotes; they have associated preconditions and postconditions expressed in OCL. These are verified by the editor tooling.

```

/**
The standard type Integer represents the mathematical concept of integer.
Note that UnlimitedNatural is a subclass of Integer, so for each parameter
of type Integer, you can use an unlimited natural as the actual parameter.
Integer is itself an instance of the metatype PrimitiveType (from UML).
**/
type Integer : PrimitiveType conformsTo Real {
  /**
  The negative value of self.
  **/
  operation "-"() : Integer precedence=UNARY;
  /**
  The value of the addition of self and i.
  **/
  operation "+"(i : Integer) : Integer precedence=ADDITIVE;
  /**
  The absolute value of self.
  **/
  operation abs() : Integer
  {
    post: if self < 0 then result = - self else result = self endif;
  }
  /**
  The number of times that i fits completely within self.
  **/
  operation div(i : Integer) : Integer
  {
    pre: i <> 0;
    post: if self / i >= 0
      then result = (self / i).floor()
      else result = -((-self/i).floor())
      endif;
  }
}

```

Figure 1: Simple Library Concrete Syntax Example.



```

/**
The index of object obj in the sequence.
**/
operation indexOf(obj : T) : Integer
{
    pre: self->includes(obj);
    post: self->at(i) = obj;
}

```

Figure 2: OrderedSet::indexOf with error.

```

type Boolean : PrimitiveType conformsTo OclAny {
    operation "="(object2 : OclAny) : Boolean precedence=EQUALITY
        => 'org.eclipse.ocl.examples.library.oclany.OclAnyEqualOperation';
    operation "<>"(object2 : OclAny) : Boolean precedence=EQUALITY
        => 'org.eclipse.ocl.examples.library.oclany.OclAnyNotEqualOperation';
    operation and(b : Boolean) : Boolean precedence=AND
        => 'org.eclipse.ocl.examples.library.logical.BooleanAndOperation';
    operation implies(b : Boolean) : Boolean precedence=IMPLIES
        => 'org.eclipse.ocl.examples.library.logical.BooleanImpliesOperation';
    operation not() : Boolean precedence=UNARY
        => 'org.eclipse.ocl.examples.library.logical.BooleanNotOperation';
    operation or(b : Boolean) : Boolean precedence=OR
        => 'org.eclipse.ocl.examples.library.logical.BooleanOrOperation';
    operation toString() : String
        => 'org.eclipse.ocl.examples.library.oclany.OclAnyToStringOperation';
    operation xor(b : Boolean) : Boolean precedence=XOR
        => 'org.eclipse.ocl.examples.library.logical.BooleanXorOperation';
}

```

Figure 3: ImplementationExample.

When applied to `OrderedSet::indexOf`, the typo involving `i` rather than `result` is detected as shown in Figure 2.

2.2 Implementable

If the OCL Standard Library is to be used by practical OCL tools, the library model must provide a body for every operation and iteration. This is difficult for the simplest operations whose body is just ‘obvious’ and potentially inefficient for more complicated bodies. OCL tools may wish to provide their own more optimum implementation. The DSL therefore allows an arbitrary string to be specified for use by the tooling. The usage for Eclipse OCL is shown in Figure 3. The string identifies the Java class that realizes the defined library feature.

A practical tool-independent OMG distribution might facilitate a simple automated conversion to tool-specific form by providing unique strings such as `org.omg.ocl.lib.Boolean.xor`.

2.3 Auto-generation

In order to support full specification auto-generation, all editorial content must be present in the model. Simplistically, this can be provided by embedding description in preceding comments, as in Figure 1. Providing sufficient richness to capture the limited usage of italics, bullets and figures currently in the specification is work in progress. A very limited subset of HTML markup is probably the solution since the Xtext tooling already supports embedded HTML.

Once the full description is present in the model, implementations may present the description to users in their documentation, help or hover text. Acceleo[M2T], which is the Eclipse implementation of the OMG MOFM2T[Objb] model-to-text transformation language, is being used to auto-generate Clause 11, the OCL Standard Library, from its OCL model.

2.4 Extensible

The term OCL Standard Library is something of a misnomer since the library has evolved incompatibly between OCL 2.x releases, and since it is extended by both QVT Operational and MOFM2T.

It is highly desirable to allow users or third parties to provide extended, alternate or additional libraries and so foster a growing collaborative community.

The library syntax therefore supports an import statement to allow a library to extend or exploit other libraries. Packages, types and features with identical hierarchical names (including parameter types for operations) are merged, provided no conflicts arise. This allows addition but not removal of

- packages and types to packages
- features, invariants and conformances to types
- preconditions, postconditions, bodies and implementations to operations or iterations
- initializations and derivations to properties

The merge uses the same yet-to-be-defined merge semantics as for Complete OCL.

3 Library Model

The UML-aligned OCL pivot meta-model is discussed in the companion paper[Wil11]. In this section we examine library concepts that require modeling, but for which there is either no UML or OCL representation, or for which the prevailing OCL representation is inconsistent with UML.

3.1 Precedence and Associativity

The precedence of `and`, `or` and `xor` was changed in the OCL 2.2 specification, so it is desirable to model precedence in order to allow either OCL 2.0 or 2.2 semantics to be used simply by selecting a corresponding OCL Standard Library Model. It is also desirable to allow languages

```
import 'OCL-2.3.oclstdlib'
library ocl
{
    precedence left:NAVIGATION left:UNARY left:MULTIPLICATIVE left:ADDITIVE
               left:RELATIONAL left:EQUALITY left:AND left:OR left:XOR left:IMPLIES;
    -- ...
}
```

Figure 4: Precedence Ordering.

that extend OCL freedom to redefine precedence and associativity to whatever is appropriate for the extended language.

Provision of a precedence label, as in Figure 1, specifies that a no argument operation can also be used as a prefix operator, and that a single argument operation can be used as an infix operator. Operators sharing a precedence label are equi-precedence, and precedences are ordered by a separate precedence ordering statement as shown in Figure 4. In the OCL 2.2 example, AND has higher precedence than OR so the expression `a and b or c and d` should be interpreted as `(a and b) or (c and d)`. Precedences orderings may be extended by merging non-conflicting orderings from imported libraries.

All operators are left associative in OCL, that is `a.b.c` is `(a.b).c` rather than `a.(b.c)`. The modeling allows right associativity to be specified.

3.2 Conformance

The conformance of Collection upon OclAny was changed in the OCL 2.2 specification, so this too should be model-able.

The `conformsTo` declaration in Figure 1 is similar to an `extends` or `implements` in Java; it may specify a comma-separated list of types to which a type conforms.

With conformance defined in the model, many conformance statements throughout the OCL specification can be eliminated as well as some definitions of a `conformsTo()` helper operation.

Much of the `conformsTo` relationship can be captured structurally, if OCL tooling loading a UML meta-model creates a `conformsTo` relationship for each generalisation relationship and adds a `conformsTo` Classifier relationship for every root class. However operational conformance is still needed to define:

- OclVoid conforms to all types (except OclInvalid).
- Collection conformance requires conformance of collection and element types.
- Tuple conformance requires part name match and type conformance.

3.3 Templates or Generics

UML defines templates which are necessary to define


```

type Set<T> : SetType conformsTo Collection<T> {
    -- ...
    operation union(s : Sequence<T>) : Sequence<T>;
    -- ...
}

```

Figure 5: Template Concrete Syntax Example.

```

type Collection<T> : CollectionType conformsTo OclAny {
    operation product<T2>(c2 : Collection<T2>) : Set<Tuple<first:T,second:T2>>
    => 'org.eclipse.oc1.examples.library.collection.CollectionProductOperation';
}

```

Figure 6: Template Operation Concrete Syntax Example.

- Collections as classes with a single template parameter
- Tuples which have many named template parameters
- the Collection::product operation which has an operation template parameter

Figure 5 shows an example type template.

Note that this use of templates for library definition purposes does not require OCL to support templates; it is full alignment of OCL with UML that requires it.

The usage of < > for templates is well-established in many languages such as C++, Java and even UML. Unfortunately OCL currently uses () to parameterize its Collection and Tuple types. This suggests that introduction of templates into OCL should use () for compatibility.

Eclipse OCL has successfully used () for templates in prototype tooling. Usage of this tooling indicates that () are acceptable but a little disconcerting for type templates. But for operation templates () are very confusing since an operation name may then be followed by one or two parenthesized clauses. The last clause is always for operation arguments or parameters. The first clause is optional and provides the template parameters or bindings. Eclipse OCL prototype tooling has therefore switched to using < >. Compatibility is preserved by allowing () following an explicit Collection name or the Tuple keyword.

It therefore seems appropriate for introduction of template functionality to OCL to align with UML and to use < >. The existing usage of () can be deprecated. < > are used throughout this paper, except where the template clause is obvious and can be omitted for clarity.

3.4 Collection::product(...)

Examination of the collection product operation reveals that it uses both type and operation template parameters. These are defined as shown in Figure 6.

The return type demonstrates the construction of a Set of Tuples whose element types are defined by the type template parameter T and the operation template parameter T2.

This provides an explicit declaration for T and T2, which are currently left to the reader's intuition.

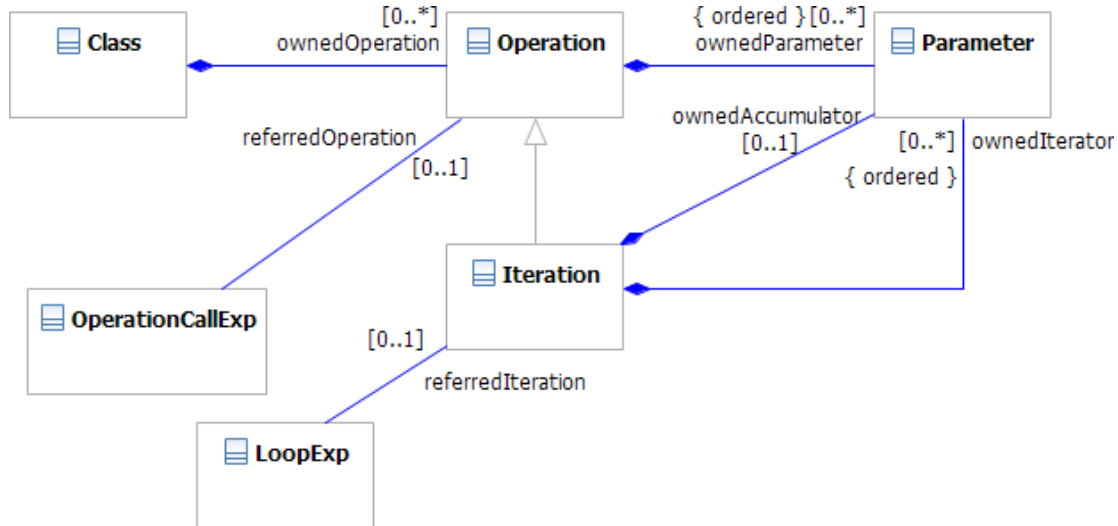


Figure 7: Iteration and LoopExp.referredIteration.

3.5 Iterators and Iterate

UML defines an **Operation** and associated **Parameters**. An iterator operation is similar to an operation but it also has **Iterators**, and the **iterate** operation has an **Accumulator** as well. Some form of UML extension is required to model iterator operations. Extending **Parameter** kinds from **IN**, **INOUT**, **OUT** and **RESULT** to accommodate **ITERATOR** and **ACCUMULATOR** was considered but found to be unhelpful in the context of references from **OperationCallExp** and **LoopExp**.

An **Iteration** class is therefore introduced, as shown in Figure 7. **Iteration** extends **Operation** with additional **ownedIterator** and **ownedAccumulator** properties; the **ownedParameter** is re-used for the iteration body¹. As an extension of **Operation**, iterations may be contained by the **ownedOperation** feature of a **Class** and so require no further changes. Introduction of a **LoopExp::referredIteration** analogous to **OperationCallExp::referredOperation** eliminates the need for iterations to be accessed by a vague name lookup at run-time.

Figure 8 shows that **iterate** and **iterator** operations can be defined in a very similar fashion to operations, re-using the semi-colon and vertical bar from the iteration call concrete syntax.

Iterations taking variable numbers of iterators are separately declared for each alternative.

3.6 Lambda Expressions

The equivalent iterator operation declarations to those in Figure 8 in the OCL specification variously omit the final body argument or resort to textual substitution of the italicized body expression within the iterator exposition. The required type signature is defined by commentary

¹ This is semantically consistent; a parameter is externally bound. However re-using **ownedParameter** for iterators and then introducing **ownedBody** rather than **ownedIterator** would be slightly simpler.

```

iteration forAll(i : T | body : Lambda T() : Boolean) : Boolean
    => 'org.eclipse.ocl.examples.library.iterator.ForAllIteration';
iteration forAll(i : T, j : T | body : Lambda T() : Boolean) : Boolean
    => 'org.eclipse.ocl.examples.library.iterator.ForAllIteration';
iteration isUnique(i : T | body : Lambda T() : OclAny) : Boolean
    => 'org.eclipse.ocl.examples.library.iterator.IsUniqueIteration';
iteration iterate<Tacc>(i : T; acc : Tacc | body : Lambda T() : Tacc) : Tacc
    => 'org.eclipse.ocl.examples.library.iterator.IterateIteration';

```

Figure 8: Iteration Concrete Syntax Example.

and sometimes well-formedness rules. On closer examination the body expression is clearly an implicit form of a Lambda Expression.

Following a casual query at the OCL Workshop in 2010, as to whether OCL should support Lambda Expressions, a discussion on the LinkedIn OCL Users Group[[Lin](#)] generally welcomed their introduction. The declarations in Figure 8 therefore use a Lambda Type to specify the constraints on the body expression. The syntax of a Lambda type draws on the Tuple and Operation signatures.

Lambda context-type (parameter-type-list) : context-type

An ExpressionInOcl is also similar to a Lambda Expression but there is one important difference. An ExpressionInOcl is self-contained with all variable references terminated by context, parameter or result variables. An Iteration body has access to the invoking environment, so the analysis resolves variable references, some of which may be implicit, to the appropriate variables which are then available for access while the iteration is evaluated.

The signatures provided above use the iterator type as the lambda expression context-type.

4 Library Content

Once an ability to define an OCL Library has been established, some problems with the content of the OCL Standard Library arise and can be addressed.

4.1 Covariant Operation Overloading

OCL specifies that OCL is aligned with UML, and UML specifies that the semantics of operation overloading is an implementation variation point. This unfortunately means that the utility of all overloaded operations is unclear in both OCL and UML. OCL must therefore define an operation overloading semantics. After discussion with a variety of concerned parties, invariant overloading semantics like Java seems appropriate. In Java, all overloads of `Object.equals(Object)` must have an `Object` argument.

For OCL, the most derived unique compatible operation signature can be statically determined when the OCL expression is parsed and persisted as the target of an `OperationCallExp::referredOperation` in the AST. When evaluated, the dynamic type of the expression source can be used to select the most derived visible operation with the same signature. Analysis of UML 2.4, identified a couple of places where covariant overloading had been assumed; these

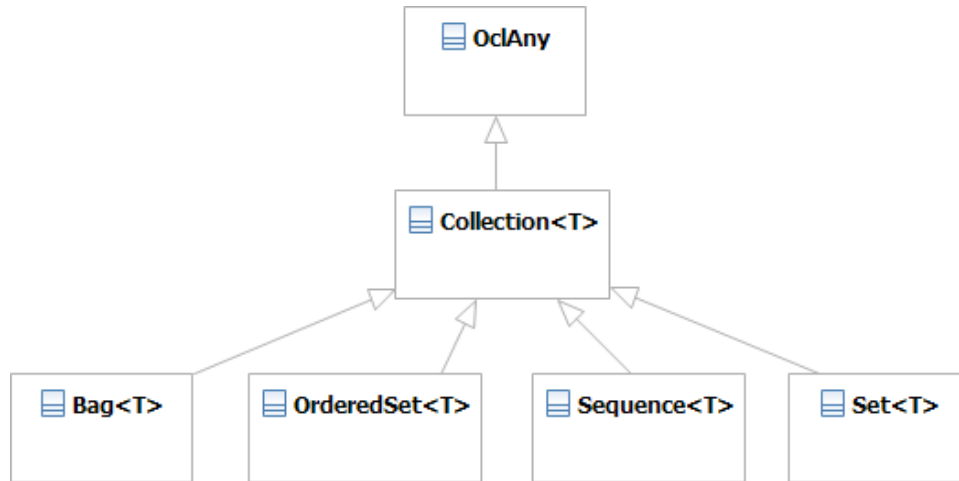


Figure 9: OCL 2.3 Collection Hierarchy.

usages were not significant and await correction in UML 2.5.

Covariant overloading allows a derived operation argument type to vary with the derived source type. This practice is widespread in the OCL standard library, where the equality operation `OclAny := (OclAny)` is overloaded by `Collection := (Collection)`.

If evaluation complexity is to be avoided, the library must be revised to eliminate covariant overloading.

4.2 Collection Hierarchy

The OCL 2.3 Collection hierarchy and conformance is shown in Figure 9².

There is limited declaration sharing between the Collection kinds and so when `OrderedSet` was introduced, it appears that its definition was cut and paste from `Sequence`. Unfortunately not all operations were reviewed to accommodate uniqueness of content, and insufficient `Set` operations were added.

These problems are discussed by Büttner[BGHK] who observes that `Set` is-a `Bag` and `OrderedSet` is-a `Sequence` and so proposes the refactoring shown in Figure 10.

This refactoring eliminates some duplication and inconsistency. For instance, `Set : asBag()` is doubly redundant; it can now be inherited, and the associated kind conversion is no longer needed.

Further redundancy arises once covariant overloads are removed. The OCL 2.3 specification defines each of:

```

Bag::union(Bag<T>) : Bag<T>,
Bag::union(Set<T>) : Bag<T>,
Set::union(Bag<T>) : Bag<T>,
Set::union(Set<T>) : Set<T>.
  
```

² using `<>` rather than a UML template to avoid tooling limitations

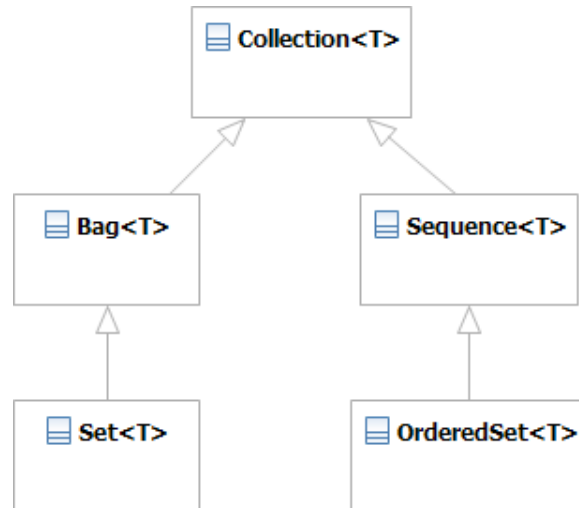


Figure 10: Collection Refactoring proposed by Buettner.

With the refactored hierarchy, the mixed Set/Bag signatures are redundant since Set conforms to Bag. We need only:

```

Bag<T>::union(Bag<T>) : Bag<T>
Set<T>::union(Set<T>) : Set<T>

```

Introduction of a further UniqueCollection as shown in Figure 11 provides further opportunities to eliminate redundant declarations.

The Set operations that are also applicable to OrderedSet may be placed in UniqueCollection so that Set and OrderedSet have consistent functionality. For instance `Set::-(Set) : Set` may be moved to `UniqueCollection::-(UniqueCollection) : Set`. This will avoid the need to convert OrderedSet to a Set in order to use a variety of Set operations.

4.3 Collection Operation Arguments

The current specification of a library operation such as `Bag::excluding(object : T)` leaves the semantics of T unclear. Is T the same or another type to the collection element type `Bag<T>`? Clause 11.6 has a preamble defining T, but does this apply to Clause 11.7 too?

A Java programmer will be surprised if the behavior does not support error-free ‘removal’ of inappropriate objects in the same way as `Collection<T>.remove(Object)`.

A user expecting very strong type checking may be disturbed if inappropriate types are silently ignored.

A modeled library requires the specification to explicitly choose between:

```

Bag<T1>::excluding(object : T1)
Bag<T1>::excluding(object : OclAny)
Bag<T1>::excluding<T2>(object : T2)

```

When dealing with null collection content, OCL provides a safe silent helpful behavior, so allowing the `excluding` operation to ignore inappropriate exclusions seems appropriate. This

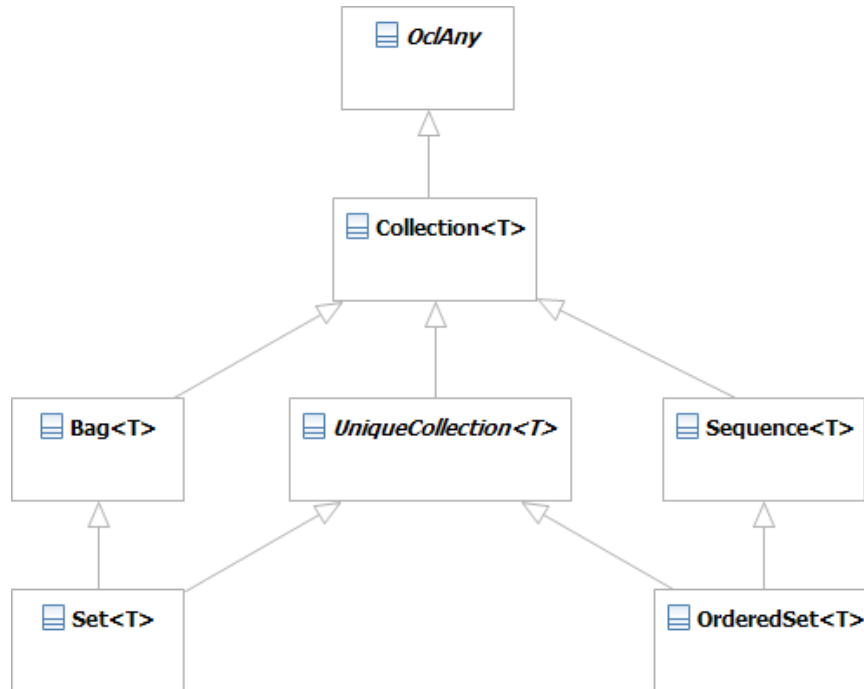


Figure 11: Further Collection Refactoring.

eliminates the first option. Introduction of the T2 template parameter is explicit but redundant so a Java-like `Bag<T>::excluding(object : OclAny)` seems the better clarification.

4.4 Modularity

The OCL specification provides support for concepts that have very specialized UML utility. Given OCL's increasing use in non-UML applications, it seems appropriate to eliminate the specialized concepts from OCL and migrate their specification to the relevant UML packages. In exchange, OCL should support merging of modules that extend a core behavior[CD10].

The import statement in the Library DSL supports migration of the `OclMessage` and associated declarations to perhaps `UmlMessage.oclstlib`.

The UML package merge defining the OCL pivot meta-model[Wil11] supports the merge of additional packages.

When Clause 9 has a complete exposition of the grammar for the concrete syntax, this too can support a merge of additional grammar constructs.

The most obvious concepts for elimination are messages and states. It is unclear whether further modularization as suggested by Akehurst[DZW08] isolating tuples, iterators, collections and primitives is useful. Perhaps even finer grain modularization could be useful so that either `Real` or `UnlimitedNatural` could be omitted.

5 Awkward Operations

Most of the library is easy to model using the concepts outlined above. However a few operations are more troublesome.

5.1 Implicit as-set: `OclAny::oclAsSet()`

The OCL specification variously specifies that, for a defined object, an `->` operation implicitly creates either a set or a bag containing that object. If the object is null an empty bag or set is created. If the object is invalid, no collection created, rather the invalid value propagates. This creation has to occur at run-time since, unlike the equivalent implicit collect shorthand, there is no way to reify the set/bag creation at compile-time. It is necessary to know whether the object is null or invalid in order to synthesize creation of the appropriate `CollectionLiteralExp` content.

Introduction of an `OclAny::oclAsSet()` operation allows the analysis to be performed statically and persisted in the AST. The complexities of null and invalid can then be resolved by dynamic dispatch selecting between:

```
OclAny::oclAsSet() – returns Set{self}
OclVoid::oclAsSet() – returns Set{}
OclInvalid::oclAsSet() – returns invalid
```

Alternate semantics may be configured by binding the `OclAny` and or derived operations to a different implementation.

The above declarations conveniently ignore the return type. Static type information is lost by a simple declaration such as `OclAny::oclAsSet() : Set<OclAny>`. This is resolved by introducing a reserved template parameter `OclSelf` to capture the statically determined context type.

The declaration `OclAny<OclSelf>::oclAsSet() : Set<OclSelf>` therefore allows 'aString'->forAll(...) to use `Set<String>` as the context type and consequently `String` rather than `OclAny` as the iterator type.

5.2 Reflection: `OclAny::oclType()`

The `oclType` operation is irregular in that its return is an expression whose value is a type rather than an object. The operation return is therefore at a different meta-level.

This operation was not present until OCL 2.2, since `Element::getMetaClass()` was thought to support reflection. Unfortunately `getMetaClass()` is defined by MOF which is not merged to form part of a UML meta-model.

In OCL 2.2, the specification is: `OclAny::oclType() : Classifier` with `Classifier` confusingly playing a role at two different meta-levels, even though EMOF has no `Classifier`.

Examination of the postcondition for `Sequence::first()` reveals a fully reflective usage of `oclType` to access the element type of the sequence:

```
if self.oclType().elementType.oclIsKindOf(CollectionType).
```

The `CollectionType::elementType` access requires `oclType` to return at least a `CollectionType`. An `oclAsType(CollectionType)` cast could be introduced to fix the above example, but it demonstrates that the derived type is statically determinate and that it is useful to

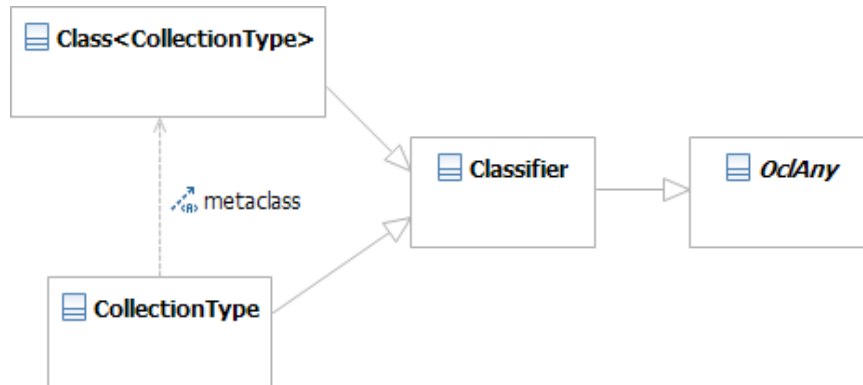


Figure 12: Class Conformance Relationships.

make that type available.

The templated Class type, shown in Figure 12, provides `Class<CollectionType>` as the meta-class of `CollectionType`. Both `CollectionType` and its meta-class conform to `Classifier` and `OclAny` since conformance of all types to `OclAny` occurs on both the model and meta-model level.

Use of the `OclSelf` template parameter in conjunction with the templated meta-class resolves the problem with the `oclAsSet` return type. A consistent specification is:

```
OclAny::oclType<OclSelf>() : Class<OclSelf>
```

The `Class<T>` is used to propagate static type information. `Class<T>` conforms to `Classifier` so a fully reflective tower is possible and it is meaningful to write:

```
self.oclType().oclType().oclType().ownedAttributes
```

5.3 Type-cast: `OclAny::oclAsType(...)`

The `oclAsType` operation is similarly irregular in that its argument is an expression whose value is a type rather than an object. The operation argument is therefore at a different meta-level.

In OCL 1.6, this is specified as: `OclAny::oclAsType(typename : OclType) : T` with `T` ill defined and `OclType` a mysterious Enumeration over an open set of all possible classifiers.

In OCL 2.0, this is declared as: `OclAny::oclAsType(typespec : OclType) : T` with `OclType` now a little-understood power-set.

In OCL 2.2, the declaration is: `OclAny::oclAsType(type : Classifier) : T` with `T` still ill-defined and the use `Classifier` at a different meta-level is unexplained. The descriptive text refers to ‘t’ rather than ‘type’.

In OCL 2.3, the declaration is unchanged but the typo in the descriptive text now refers to ‘T’ rather than ‘type’.

The underlying rationale seems to be to try and establish an identity for a meta-type without imposing the cost of reflective tower that would arise with a true meta-type. This was not a requirement in the OCL 2.0 Request For Proposals[Objg], and the usage of `oclType()` requires

reflection anyway, so it seems appropriate to provide reflection rather than an inconsistent facility that does not.

A consistent specification is: `OclAny::oclAsType<T>(type : Class<T>) : T`

This exploits the conformance of `T` to `Class<T>` to preserve the static type. It assumes that the model of the unspecified `TypeValue` class is specified to be a `Class<T>`.

5.4 OclAny::allInstances(...)

The `allInstances` operation is irregular in that it is the only class operation; all other operations are instance operations. This prompted OMG Issue 8937[OMG] to suggest that `::` rather than `.` should be used when invoking `allInstances` in the concrete syntax. However `allInstances` is not a `Classifier` class operation, rather it is an operation on the statically determined derived `Classifier` class. Thus all instances of `Person` are returned by the invocation `Person.allInstances()` which is handled by the `Classifier::allInstances()` operation. There is no such operation as `Person::allInstances()`.

A consistent specification for this is:

```
static Classifier::allInstances<OclSelf>() : Set<OclSelf>
```

Once again `OclSelf` solves the typing problem enabling the return type to use the statically determined derived `Classifier` type. The `'static'` keyword indicates that the source should be a type-valued expression. This ensures that `aPerson.allInstances()` can be diagnosed as an undefined operation.

5.5 Collection::collect(...)

The `collect()` iteration is potentially awkward since the return type is a flattened collection and consequently there is no unique mapping from input to output types.

In OCL 2.0, it might appear that a `Bag<T>` return value could have been produced by an input value corresponding to any of the `Bag<T>`, `Bag<Bag<T>>` or `Bag<Bag<Bag<T>>>` types, but the first level element types have no mutual conformance so such content is a type error. OCL 2.0 does not support mixed depth collections.

In OCL 2.2, `Collection` conforms to `OclAny`, and so a collection may contain both collection and non-collection values. However, for a heterogeneous collection, the existing type system cannot achieve greater precision than `Collection<OclAny>`, since `OclAny` is the only common type of a collection and a non-collection.

This lack of precision makes the declared return type of the `collect` operation and consequently an implicit `collect` almost unusable for heterogeneous collections. Fortunately this language limitation does not impact the `collect()` declaration and so we only need declarations that are valid for homogeneous collections.

```
Collection<T>::collect<T2>(i : T |
  body : Lambda T(): T2) : Collection<T2>
Collection<T>::collect<T2>(i : T |
  body : Lambda T(): Collection<T2>) : Bag<T2>
Bag<T>::collect<T2>(i : T |
```

```

type Comparable<T> {
  operation compareTo(object : T) : Boolean;
}

```

Figure 13: Comparable.

```

body : Lambda T(): T2) : Bag<T2>
Sequence<T>::collect<T2>(i : T |
  body : Lambda T(): T2) : Sequence<T2>
Sequence<T>::collect<T2>(i : T |
  body : Lambda T(): Sequence<T2>) : Sequence<T2>

```

Note that the final declaration for Sequence is not an overload. Introduction of modeled signatures highlights a gap in the specification as to what happens when unordered content is collected from an ordered collection. The lack of overload above gives an unambiguous interpretation; mixed ordering collect yields a Bag.

6 Further Modeling

The preceding sections have identified ways in which awkward declarations can be modeled. We now examine ways in which some well-formedness rules can be modeled more overtly.

6.1 Comparable

The `Collection::min` operation specifies “Elements must be of a type supporting the min operation. The min operation - supported by the elements - must take one parameter of type T and be both associative and commutative. UnlimitedNatural, Integer and Real fulfill this condition.”

The `sortedBy` iteration requires “The type of the body expression must have the < operation defined.”

These similar functionalities with informal and very different requirements can be modeled by taking inspiration from Java’s Comparable interface as shown in Figure 13. This enables the min operation to fully modeled as shown in Figure 14.

An additional pre-condition, not yet supported by prototype tooling, should permit the implicit `conformsTo Comparable` within the postcondition to be expressed explicitly:

```
pre: self.oclType().elementType.oclIsKindOf(Comparable)
```

6.2 Summable

A similar interface type may introduced to model the requirements of `Collection::sum()` as shown in Figure 15.

If this interface type is realized by String as well as Real, then string splicing can be achieved by `Sequence<String>::sum()`.

```

/**
The element with the minimum value of all elements in self.
Elements must conform to the Comparable type.
UnlimitedNatural, Integer and Real fulfill this condition.
**/
operation min() : T
{
    post: result = self->iterate(elem; acc : T = self->any() |
        if acc.oclAsType(Comparable).compareTo(elem) then acc else elem endif
    );
}

```

Figure 14: Collection::min.

```

type Summable<T> {
    operation add(object : T) : T;
    operation zero() : T;
}

```

Figure 15: Summable.

7 Impact

A number of new concepts have been introduced in this paper, but very few of them have any impact on OCL, merely on the ability to provide a modeled OCL specification.

From a user perspective, the main change is a recommendation to migrate towards `< >` rather than `()` so that OCL has explicit UML-aligned templates rather than just template-like collections.

The clearer declarations and semantics for the library will require implementers to review their endeavors to interpret earlier specifications.

- Elimination of covariant overloading
- Refactored Collection hierarchy
- Comparable and Summable interfaces
- oclAsSet

This will probably require some behavioral revision, which users can be protected from by providing an alternate OCL Standard Library with that a tool's traditional semantics.

Tools that offer OCL Standard Library customization will need to implement the library DSL and its associated interpretations of template and lambda types. Tools may choose to just re-use the standard UML-aligned OCL Standard Library Model, but this will still require support for the corollaries of UML alignment.

- Introduction of an Operation::precedence property

- Introduction of an Operation::implementation property
- Introduction of an Iteration class
- Introduction of a LoopExp::referredIteration property
- Use of template type specializations
- Use of template operation specializations
- Use of template lambda type specializations

8 Conclusions

Elimination of typographical errors from the OCL specification was an original motivation for introduction of an OCL Standard Library model. But as we have shown the benefits are much greater; the model is readable, extensible, interchangeable and toolable.

We have described prototype Eclipse OCL tooling for a Domain Specific language for the OCL Standard Library model. This tooling enables the official model and extended models to be produced easily and reliably.

We have followed through some of the problems raised by attempts to model the library and revised the library content slightly and introduced additional concepts to support the model.

We have shown how modeled declarations can detect errors and provide definitive semantics for gaps in the current specification.

Bibliography

- [BGHK] F. Büttner, M. Gogolla, L. Hamann, M. Kuhlmann. *On Better Understanding OCL Collections or An OCL Ordered Set is not an OCL Set*.
- [CD10] J. Chimiak-Opaka, B. Demuth. *A Feature Model for an IDE4OCL*. OCL 2010: Workshop on OCL and Textual Modelling, 2010.
- [DZW08] D.G.Akehurst, Z.Zschaler, W.G.J.Howells. *OCL: Modularising the Language*. Ocl4All: Modelling Systems with OCL, 2008.
- [Lin] Anonymous Functions.
<http://www.linkedin.com/groupItem?view=&gid=3007822&type=member&item=31810988>
- [M2T] Eclipse M2T/Accelleo Project.
<http://www.eclipse.org/Accelleo>
- [MDT] Eclipse MDT/OCL Project.
http://www.eclipse.org/projects/project_summary.php?projectid=modeling.mdt.ocl

- [Obja] Object Management Group. Version 2.3, omg document number: formal/2010-11-42 edition.
<http://www.omg.org/spec/OCL/2.3>
- [Objb] Object Management Group. MOF Model to Text Transformation Language. v1.0, omg document number: formal/2008-01-16 edition.
<http://www.omg.org/spec/MOFM2T/1.0>
- [Objc] Object Management Group. Object Constraint Language. Version 1.6, omg document number: ad/2003-01-07 edition.
<http://www.omg.org/cgi-bin/doc?ad/03-01-07.pdf>
- [Objd] Object Management Group. Object Constraint Language. Version 2.0 draft, omg document number: ptc/03-10-14 edition.
<http://www.omg.org/cgi-bin/doc?ptc/03-10-14>
- [Obje] Object Management Group. Object Constraint Language. Version 2.0, omg document number: formal/06-05-01, 2006 edition.
<http://www.omg.org/spec/OCL/2.0>
- [Objf] Object Management Group. Object Constraint Language. Version 2.2, omg document number: formal/2010-02-01 edition.
<http://www.omg.org/spec/OCL/2.2>
- [Objg] Object Management Group. Object Constraint Language version 2.0, Request for Proposals9. Version 2.0 rfp, omg document number: ad/2000-09-03 edition.
<http://www.omg.org/cgi-bin/doc?ad/2000-09-03>
- [Objh] Object Management Group. Query/View/Transformation Specification. Version 1.0, omg document number: formal/08-04-03, 2008 edition.
- [Obji] Object Management Group. Unified Modeling Language, Infrastructure. Version 2.03, omg document number: formal/2005-07-05 edition.
<http://www.omg.org/spec/UML/2.0/>
- [OMG] Notation for accessing class operations is inconsistent.
<http://www.omg.org/issues/ocl2-rtf#8937>
- [TMF] Eclipse TMF/Xtext Project.
<http://www.eclipse.org/Xtext>
- [Wil11] E. D. Willink. *Aligning OCL with UML*. submitted to TOOLS 2011, OCL Workshop, 2011.