Electronic Communications of the EASST
Volume 44 (2011)

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

Aligning OCL with UML

Edward Willink

20 pages

Guest Editors: Jordi Cabot, Robert Clariso, Martin Gogolla, Burkhart Wolff
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Aligning OCL with UML

Edward Willink[1]

[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. Unfortunately no OCL 2.x specification has ever been aligned with any UML 2.x specification. This lack of alignment makes some OCL compliance points such as XMI interchange unachievable. This paper describes how introduction of an OCL Pivot Meta-Model and clear exposition of the Values package may provide a solution to the alignment and a variety of other specification issues.

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

1 Introduction

The Object Constraint Language (OCL) evolved, initially within the Unified Modeling Language (UML) as the textual language for expressing constraints that could not be represented graphically. 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[Objb] to align with UML 2.0. A partially revised OCL 2.0 draft[Objc] 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[Objd] formed the basis for QVT 1.0[Objg]. 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.

Subsequent revisions [Obje],[Objf] have addressed a number of inconsistencies, but the major problems remain unaddressed.

Each version of OCL 2.x states in its Scope statement that it is aligned with the corresponding UML 2.x specification. Sadly this statement is only an aspiration at present. In this paper we examine the major misalignments and outline a proposal to resolve these and other minor misalignments.

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 UML-aligned OCL meta-models may be found in the optional Examples and Editors of the Indigo release of Eclipse OCL[MDT]. This is officially released in June 2011. Milestone builds have been available since December 2010. The prototype uses automated conversion starting with potentially standard UML meta-models for OCL and UML. From these a merged OCL Pivot Meta-Model and associated Eclipse Ecore[EMF] tooling are derived. Despite the use of Ecore tooling, with this approach the name of a meta-class is an OMG-compliant 'Class' rather than the Ecore 'EClass'.

The UML-alignment outlined below involves very few, if any, actual changes to the concrete syntax and semantics of OCL; the changes are intended to just make the specification say what many users think it already says. The changes to the abstract syntax are however quite significant.

In Section 2 we discuss the OCL specification and identify some problems in the way that it is used. Then in Section 3 we identify more technical problems. We propose an OCL Pivot Meta-Model as a solution to all these problems in Section 4. Finally we conclude.

2 Background

Although the OCL specification is partitioned very logically, it can appear that the specification contains more information than is necessary.

- Clause 7 provides a non-normative and readable overview of OCL
- Clause 8 specifies the Abstract Syntax, comprising the Types and Expressions classes that define the executable language,
- Clause 9 specifies the Concrete Syntax for the grammar and non-normative Concrete Syntax classes that define the readable language,
- Clause 10 specifies the Evaluation semantics using the Values and Evaluations classes to define the behavior,
- Clause 11 specifies the OCL Standard Library which provides the operations and iterations that make Types and their Values useful,
- Clause 12 specifies the Complete OCL language; an ability to define an independent OCL Document that complements a pre-existing model.
- Annex A provides a more formal but non-normative foundation for OCL semantics

A problem in understanding the specification, is that constraints that apply solely to the AS are found in Clause 8, constraints that affect construction of the AS are found in Clause 9, and constraints that affect execution of the AS are found in Clause 10, while the operations used in constraints are in Clause 11.

Much of Clause 10 on Evaluation seems strange, irrelevant and repetitious, but it serves an important and surprisingly practical purpose that we discuss in Section 4.2.

Comparison of Clause 12 on Complete OCL with the preceding clauses quickly reveals that Clause 12 is a bit thin; Clause 12 is still a preliminary draft, and it is in realizing Clause 12 that the major UML alignment issues arise. We discuss these in Section 3.1.

2.1 OCL Specification Compliance Points

The OCL specification defines three major compliance points, with additional minor options for evaluation.

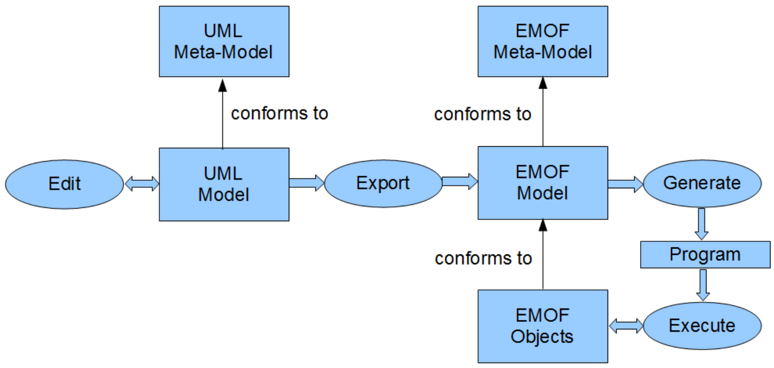

Figure 1: Basic Scenario for executable usage of UML.

2.1.1 Concrete Syntax

Interchange of concrete syntax between tools is moderately successful today, but is limited by ambiguities in the specification and consequent divergent misunderstandings by tool implementors. These difficulties should be significantly alleviated by a sound OCL Standard Library model as described in our companion paper[Wil].

2.1.2 XMI Interchange

XMI Interchange is important to allow the costs of parsing the Concrete Syntax to be isolated from execution costs. In the XMI representation, all syntax sugar is removed and references directly access target features using properties such as OperationCallExp::referredOperation. This requires the target operation to have a URI, which is only available for user models. The OCL Standard Library has no model and no policy for establishing URIs independent of a model, so XMI Compliance is impossible whenever a library operation is used. The lack of basic URIs is addressed by providing a model for the OCL Standard Library as described in our companion paper[Wil]. More serious URI problems arise with underspecification of Complete OCL and we address these in Section 3.1.

2.1.3 Evaluation semantics

The specification requires that tools evaluate in accordance with OCL semantics, which is a relatively modest requirement for basic arithmetic values, but becomes quite troublesome for null, invalid and very large or high precision values.

The specification provides no API (Application Programming Interface) by which a query can be invoked and, in practice, all OCL evaluation is encapsulated within some modeling environment. It is therefore not possible to detect whether an OCL tool uses the classes specified in the Values package. Models do not and so practical tools do not either. This has allowed some significant problems in the Values package to remain unreported. We address these in Section 4.2.

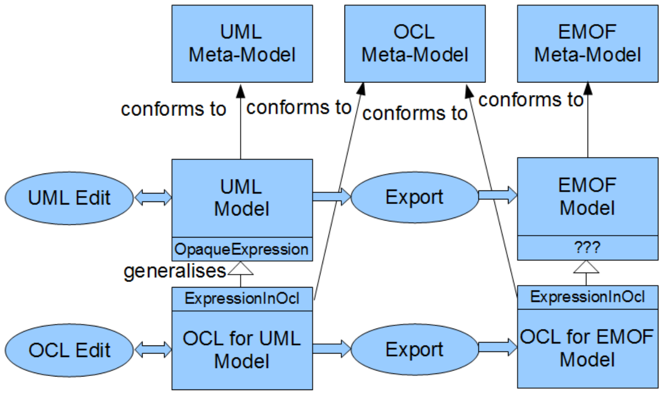

Figure 2: Basic OCL Integration Scenario.

2.2 OCL Usage

A standard way of using UML is shown in Figure 1. A UML model conforming to a UML meta-model is maintained by an editing activity. When appropriate, this model is exported to an EMOF (or Ecore) model conforming to a corresponding meta-model. The EMOF model contributes to a code generation activity that produces a program that can be executed to exploit EMOF objects that are instances of classes from the EMOF model.

The UML meta-model defines a rich suite of capabilities suitable for meta-modeling. The export to the EMOF meta-model reduces the capabilities to those necessary to support effective use of models at run-time. The export applies Profiles, merges Packages and eliminates Associations leaving just Properties at their ends.

When we extend models with OCL capabilities, we get the scenario shown in Figure 2 for developing a UML model that includes constraints. (We omit the unchanged model execution, in order to simplify the diagram.)

The OCL integration with UML is quite tidy with OCL providing an ExpressionInOcl class that extends UML's OpaqueExpression class. Distinct editors are usually required for the very different characteristics of graphical UML and textual OCL.

The corresponding EMOF integration is troublesome because EMOF has discarded too many UML concepts that OCL requires. We discuss the EMOF problems in Section 3.2. When we just treat UML/EMOF and OCL in combination, and then consider constraints, defined in the meta-model, for evaluation upon a model, we find the evaluation scenarios shown in Figure 3.

When we consider the desirable characteristics of the different ways that models are used we find:

- Definition of models requires the richness of UML
- Execution of models benefits from the slimmed down efficient characteristics of EMOF
- Definition of OCL constraints requires much of the richness of UML

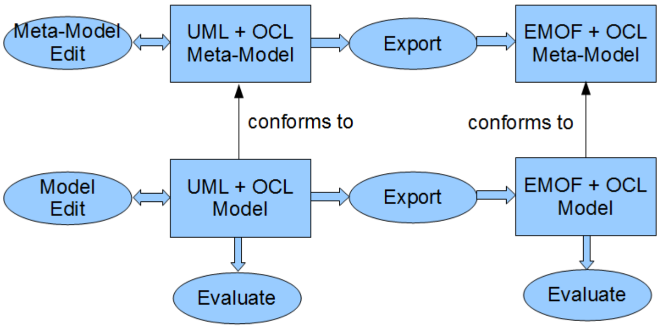

Figure 3: Evaluation for UML or EMOF.

- Evaluation of OCL constraints benefits from a slimmed down efficient representation

The last two OCL considerations pull in opposite directions; a rich OCL + UML environment for development and an efficient environment for evaluation. Neither is compatible with UML or EMOF characteristics. UML is too rich for efficient execution and EMOF is too limited for adequate model navigation.

The current OCL specification with its statement that OCL can be used with both UML and EMOF is unhelpful and unrealistic. The deficiencies for EMOF behavior are too great. Tool implementers are forced into pragmatic Bridge[AP] or Pivot[BD] solutions to support more than one of EMOF and UML.

We will therefore propose, in Section 4.1, that the specification define a combined UML and OCL Pivot Meta-Model that exhibits the more efficient characteristics of EMOF while retaining the relevant richness of UML.

3 Problems

We have just identified limitations in OCL meta-models with respect to some conflicting usage scenarios. We will now examine some more technical problems.

3.1 Complete OCL Problems

A Complete OCL document can complement a model and add features to it so that they can be used as if they were part of the complemented model. Library types may also be complemented, and so definition of an `OclAny::isPersistent()` operation may add an ability to evaluate constraints concerning the persistent storage associated with any model element.

The additional Abstract Syntax for Complete OCL in Clause 12 comprises just the ExpressionInOcl class. Unfortunately problems arise when we try to realize the first paragraph of Clause 12.5 which states:

“A definition constraint is a constraint that is linked to a Classifier. It may only consist of one or more LetExps. The variable or function defined by the Let expression can be used in

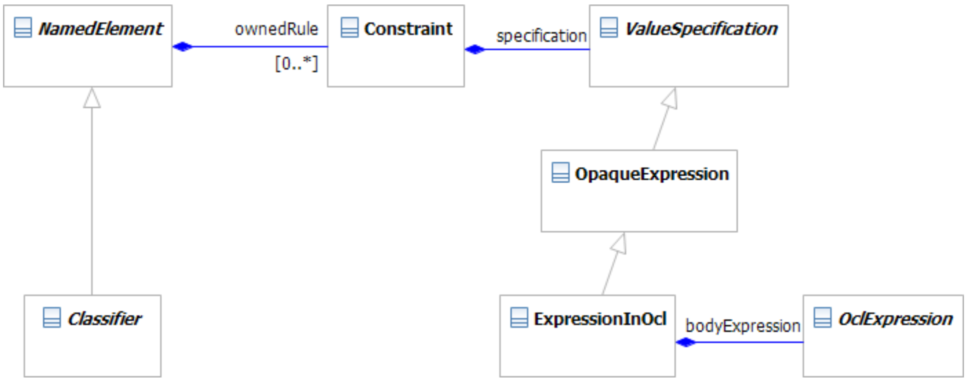

Figure 4: Specified relationship of a definition Constraint to a Class.

an identical way as an attribute or operation of the Classifier. Their visibility is equal to that of a public attribute or operation. The purpose of a definition constraint is to define reusable sub-expressions for use in other OCL expressions."

[We will ignore the references to LetExps and variables and functions that refer to an obsolete OCL 1.x concrete syntax and so makes the paragraph difficult to interpret for OCL 2.x.]

The subsequent description and figure show that the definition Constraint is realized by an ExpressionInOcl that is indirectly owned by the context classifier via a Constraint. The figure is redrawn as Figure 4 which corrects trivial UML misalignments.

The intent of "an identical way as an attribute or operation" is clear. A definition constraint provides the definition of a feature that is usable in an OCL expression, and so must be usable in both concrete and abstract syntax as if the defined feature formed part of the complemented model. In UML, an attribute is a Property and both Property and Operation are Features, so a definition Constraint should provide either a Property definition or an Operation definition.

Utility in the concrete syntax requires that the Feature definition can be looked up by hierarchical name in the OCL Environment. The specification of Environment lookup in Clause 9 can resolve Property and Operation model elements. Unfortunately the Inherited Attribute rules are missing from Clause 12 so it is unclear how a Property or Operation is resolvable from an object structure that does not include such a feature. It is also unclear how a definition Constraint is able to export a name into the Environment in the reverse direction to that for inherited attributes.

Utility in the abstract syntax means that a PropertyCallExp::referredProperty or an OperationCallExp::referredOperation is able to refer to the Feature definition. This requires the Feature definition to be either a Property or an Operation. None of the Constraint, ExpressionInOcl or OclExpression classes shown in Figure 4 satisfy this requirement.

These considerations all indicate that the abstract syntax must be revised so that a definition Constraint is realized by a Property or Operation as shown in Figure 5. With definition Constraints realized by features defined in models, the problem of installing the definition Constraint for lookup in the Environment is resolved; the defined features are installed in the same way as any other feature. The lookup results are features and so can be the target of references from the existing expression abstract syntax.

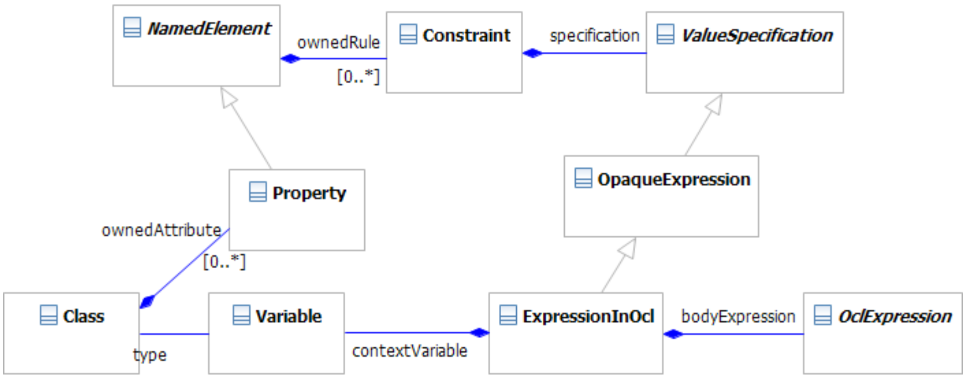

Figure 5: Property relationship of a Definition to a Classifier.

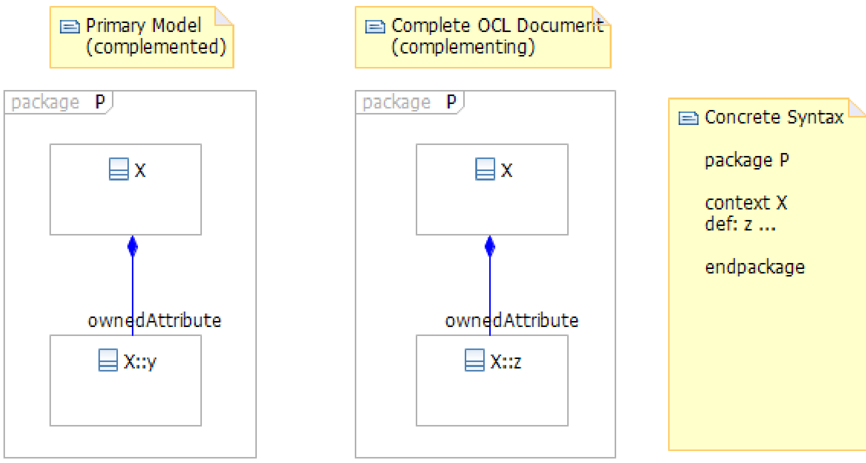

Figure 6: The Multiple Models problem for Complete OCL.

We have one more major problem to solve. What is the relationship between the Class, which Complete OCL complements with a definition, and the Class, for which the definition is an ownedAttribute or ownedOperation? We will examine this problem, shown in Figure 6[1], from a variety of perspectives.

3.1.1 Merged or Multiple models

The simplest solution is that the two Package P's and the two Class X's are combined to form merged Package P and Class X objects. However in an implementation, this means that the Package P and the Class X in the UML model are modified by usage of Complete OCL.

The alternative is to maintain some form of composite model or index in which multiple contributory models are treated as a coherent whole.

[1] Mis-drawn using class rather than attribute notation to highlight the distinct elements involved.

3.1.2 Model Maintenance

The merged model is a conventional model and so should be amenable to support by a UML modeling environment. However since the model is modified, it may be necessary to create a distinct model for each usage, since each usage may have a different combination of Complete OCL documents. The Complete OCL complements for one usage must not infect another usage.

The composite model comprises multiple unmodified conventional models, so these contributions can be shared between usages. However special functionality is required to allow the composite to behave coherently. This functionality will not be directly available from a UML modeling environment.

3.1.3 Implicit Access

Implicit access occurs through navigation of a property, or invocation of an operation, in an OCL expression.

For the merged model, this presents no new problems since the merged model is internally coherent.

For the composite model the OCL tooling must direct the implicit access to use the feature from the appropriate partial model.

3.1.4 Explicit Access

Explicit access occurs when reflection is used to access the properties. Since reflection is not consistently specified in OCL 2.3, there is discretion as to how this is specified in the future.

In Figure 6, should the reflective access to Class::ownedAttribute for X return both X::y and X::z hiding the distinct origins of the two features, or should there be a mechanism to obtain distinct ownedAttributes from each?

The merged model can only present the coherent view. Additional capabilities are needed to enable helper operations to provide partial returns.

The composite model naturally supports the disjoint view and helper operations can provide a merged view. Reflection can therefore allow expression access to see disjoint or coherent views.

3.1.5 URIs

When a reference to a complementing definition is persisted via XMI, a URI must be established for the definition so that the complementing definition can be reconstructed when the XMI is loaded.

The merged model will naturally provide URIs appropriate to the merged context, which solves the problem of providing a URI. But the merged context is unhelpful for reload, since the distinct identity of the Complete OCL document is obscured, unless some special form of URI is used to capture the distinct origin.

The composite model preserves the distinct model identities and so will naturally provide URIs that correspond to the relevant document.

3.1.6 Summary

Neither approach is entirely satisfactory. The merged model has significant problems with sharing, URIs and full reflection. The composite model has fewer problems but requires additional non-UML tooling to represent the composite. The first three perspectives identify implementation trade-offs that need not concern the specification. However the specification must make a choice on the reflective behavior and URIs for Complete OCL and so the specification reduces an implementation's freedom from all perspectives. In Section 4.1 we propose a solution that builds upon the composite model.

3.2 EMOF Problems

OCL is specified to be aligned with UML. Unfortunately this alignment collapses when UML is exported as EMOF (or Ecore) and so Clause 13 is required to explain what doesn't work and to introduce EMOF-specific adaptations to make anything work at all.

Some functionality, such as the lack of support for Messages and States, is excluded. This eliminates functionality that perhaps should not be in the specification at all.

Other exclusions such as the lack of support for Associations affect core modeling functionality and limit the ability to navigate models.

The whole of Complete OCL is excluded, since the absence of the ValueSpecification and Constraint classes is not rectified. Figure 2 shows the integration gap between OCL and EMOF.

EMOF omits Classifier and so the the OCL Standard Library cannot define operations for primitive types. A workaround convention is therefore suggested in Clause 13.2 bullet 6 whereby an accompanying class instance is provided for such types. This convention is not elaborated and the associated packaging and URI issues are not addressed.

3.3 Other Problems

The Complete OCL and EMOF problems are serious. We now identify more minor UML-alignment problems that we can also solve.

3.3.1 Iterator Operation

The iterate and iterator operations have no UML counterpart and so cannot be represented by a UML meta-model. The OCL meta-model provides a Types and an Expressions package but omits any reification of an Iterator operation and so the integration of an iteration with its context classifier is unspecified. As a result, all support for iterators requires built-in functionality, and indeed prior to OCL 2.3, the specification could be interpreted to require all names of iterators to be hard-wired into the OCL grammar. OCL 2.3 clarified the status of names so that any name can be used as an iterate or iterator operation.

In our Companion paper[Wil] we show how introduction of an Iteration class extending the Operation class is sufficient to allow the OCL Standard Library to be modeled.

3.3.2 OclAny conformance

OCL uses the conformsTo relationship between types to determine substitutability. This relationship is almost identical to UML generalization; the main difference being the definition that all UML classes conform to OclAny.

Direct realization of the above leads to some practical difficulties. Firstly the lookup of matching features must use one algorithm to traverse the generalization hierarchy, and another to extend on to OclAny. This irregularity becomes more of a concern when considering a UML operation such as `Classifier::conformsTo()` which is specified to traverse the generalization hierarchy and so requires that OclAny is part of the generalization hierarchy.

3.3.3 Reflection

The `OclAny::oclType()` operation was added to the OCL Standard Library when it was realized that the MOF `Element::getMetaClass()` operation was not accessible for UML meta-models, which do not merge MOF.

Since OCL mandates that all types conform to OclAny, use of `OclAny::oclType()` requires that all types at all meta-levels must conform to OclAny.

4 Solutions

We can accommodate the conflicting UML-alignment requirements in a variety of ways.

We could eliminate all non-UML facilities from OCL, but OCL without Iterations would not be of much utility, so this is untenable.

We could eliminate the statement that OCL is aligned with UML. This is pretty much unthinkable given UML's dependence on OCL.

We could revise UML so that it supports the facilities that OCL requires. This is possible in principle, but hardly desirable since it may incur political difficulties and further practical difficulties in mutual alignment.

In order to make OCL useful for EMOF meta-models, we could add the missing parts of UML such as Associations and OpaqueExpressions to OCL. This can solve some EMOF problems but creates inconsistencies whereby OCL provides additional classes solely for use in a simpler context. To offer Complete OCL and OCL Standard Library support for EMOF without requiring distinct EMOF and UML variants of OCL we would need to change EMOF.

4.1 OCL Pivot Meta-Model

In the following sections we will therefore pursue an alternative approach whereby we re-use the constructive nature of the UML specification to select those packages that are relevant and then merge these with additional OCL packages to create a new UML-derived OCL Pivot Meta-Model.

With the OCL Pivot Meta-Model UML-derived, large parts will automatically be UML-aligned. Since the UML and OCL meta-models are distinct, we can adjust any UML well-formedness rules that are not applicable to OCL .

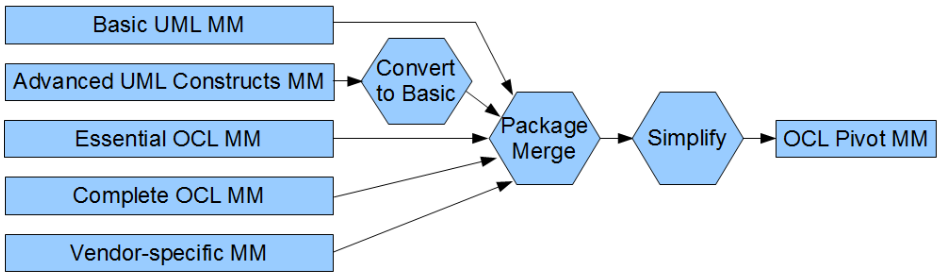

Figure 7: Meta-Model merge to produce the OCL Pivot Meta-Model.

As a pivot model, the OCL Pivot Meta-Model is neutral and so independent of UML or EMOF and so can be used in conjunction with a variety of alternate meta-models.

The OCL Pivot Meta-Model is a complete meta-model and follows the trend of ensuring that major meta-models are self describing. In UML 2.4, a UML::Class has a UML::Class as its meta-class. For the proposed UML-derived OCL Pivot Meta-Model, an OCL::Class has an OCL::Class as its meta-class.

4.1.1 Meta-Meta-Model Merge

The OCL Pivot Meta-Model is derived by the package merge shown in Figure 7. The contributions to the merge are:

Basic UML

The UML InfrastructureLibrary::Core::Basic package defines the Essential MOF. It provides efficient but inflexible representations of each class. For instance, subset properties are eliminated so that an Operation is found in Class::ownedOperation, but not in Class::feature or Namespace::member or Namespace::ownedMember or Element::ownedElement.

Additional UML Constructs

OCL Constraint integration requires the InfrastructureLibrary::Core::Abstractions::Constraints package.

Full type support requires the InfrastructureLibrary::AuxiliaryConstructs::Templates package.

OCL Message support requires the CallOperationAction and SendSignalAction classes that are defined by the UML::Actions::BasicActions package and the Signal class defined by the UML::CommonBehaviors::Communications package.

OCL State support requires State from the UML::StateMachines::BehaviorStateMachines package.

Unfortunately these packages were not intended to be merged into Basic, so they do not provide the same efficient representation. The Eclipse OCL prototype currently works around this problem by manual creation of 'Basic' equivalents.

It is possible that the UML simplification process[Objh] may eliminate the Basic package as a primary artifact, and exploit a QVT Operational transformation to derive it. It may be

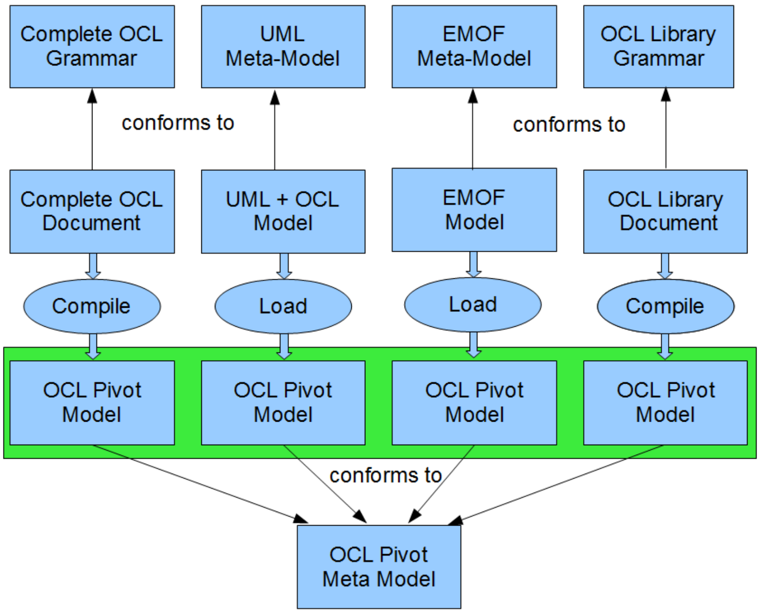

Figure 8: Composite Pivot Model derived from disparate sources.

appropriate to enhance this transformation to also derive basic variants of relevant Advanced Constructs packages.

Essential OCL
These are the packages defined by Clause 8 of the current OCL specification, with minor enhancements to align with UML.

Complete OCL
This is the single class package defined by Clause 12 of the current OCL specification, with some revision to align with UML.

Vendor-specific
The package merge is not constrained to the requirements of the OCL specification. Tool vendors may merge further packages to support visitor protocols, useful operations or transient caches.

Simplify
Following the package merge, the meta-model can be simplified to eliminate redundant classes such as Type, see Section 4.3.2, remove redundant generalizations and convert the residual generalizations to conformances.

4.1.2 Meta-Model Load

Before any evaluation on a user model can occur, its meta-model must be loaded. This currently presents challenges since users may use a variety of UML, CMOF, EMOF and Ecore dialects not all of which are supported by all tools. With a neutral pivot model the diverse sources are accommodated by a variety of compilation or loading activities as shown in Figure 8.

Introduction of the OCL Pivot Meta-Model requires the user model to be converted to, or at least interpreted in, a normalized form. The proposed OCL Pivot is similar to that advocated by Bräuer[BD] to make implementation of Dresden OCL more flexible. The OCL Pivot presented here is to make the OCL specification coherent.

Introduction of a meta-model load phase to the specification enables the following problems to be resolved:

- Diverse meta-model dialects can be intermixed
- Complete OCL documents can be represented as OCL Pivot Models
- OCL Standard Libraries can be fully represented as OCL Pivot Models
- UML generalization can be re-interpreted as OCL conformance
- OclAny can be inserted into the conformance hierarchy

With all OCL concepts consistently modeled, the OCL Pivot Meta-Model can be used to provide the URIs needed to solve the problem of XMI interchange.

With a normalized meta-model representation, limitations in OCL support such as navigating non-navigable associations are caused by limitations in the meta-model loader rather than in OCL. OCL is fully specified for such associations, but they are useable for EMOF only when the `org.omg.emof.oppositeRoleName` tag introduced in MOF 2.4[Obja] is exploited by both meta-model producer and consumer.

4.2 Primitive Types and Values

The representation of a value in OCL appears to be very similar to conventional languages, but is actually very different. We will therefore examine the issue carefully.

4.2.1 Primitive Values

UML provides a PrimitiveTypes package that defines the primitives, such as Integer or String, as a domain of values without specifying any representation or behavior. This vagueness is important to allow a UML model to specify the required behavior of a wide variety of alternate implementations without imposing a particular representation.

A primitive, as shown in Figure 9, can only exist within a suitable 'container', such as the NamedElement::name Property that binds a String to perform the role of naming its NamedElement. For a more general purpose role such as a string value, UML defines LiteralString which

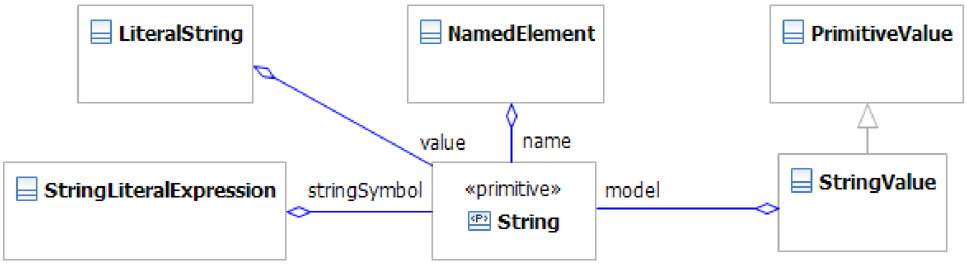

Figure 9: The String primitive attributes and their 'containers'.

subtypes the polymorphic ValueSpecification. Similarly, OCL defines a string role in an expression using StringLiteralExpression which subtypes the polymorphic OclExpression. The progression NamedElement::name, LiteralString::value to StringLiteralExpression::stringSymbol provides steadily richer roles, but does not specify any representation or behavior for use in that role.

It is the OCL Standard Library that specifies primitive and non-primitive behavior, and it is the OCL Values package that specifies an OCL representation for which that behavior applies.

In order to evaluate an operation such as String::toUpperCase() on a string, the string must be contained in a context that supports that operation evaluation. This is a StringValue in OCL. This is confusing to anyone familiar with almost any Object Oriented Language, since String is conventionally a class that provides a rich suite of behaviors. In UML and OCL, a primitive String has no associated representation or behavior. It is only as the model of a StringValue that behavior defined by the OCL Standard Library is usable.

This confusion is compounded by practical OCL implementations that may reuse the String type of their implementation language to realize the StringValue representation and behavior for OCL. This reuse can work very effectively for basic functionality, but is troublesome for precise functionality, since it is unlikely that a practical OO Language will have exactly the same semantics as OCL. For instance, consider the irregularity whereby the UnlimitedNatural for an unlimited value (plus-infinity) is invalid once the UnlimitedNatural is interpreted as either of Integer or Real to which UnlimitedNatural conforms.

It is also tempting to use the Collection capabilities of an implementation language to directly implement the OCL CollectionValue representations and behaviors. However considerable care is needed with `OrderedSet{Set{4.0}}->including{Set{4}}->size()` to ensure that the result is 1 rather than 2, since the encapsulated equality of 4.0 and 4 may not be respected by the language semantics.

4.2.2 Object Values

The benefits of the Value hierarchy imposing consistent semantics independent of the underlying implementation for primitive and collection values is equally applicable to object values.

OCL 2.3 specifies an ObjectValue class that maintains object history using a sequence of LocalSnapshot instances. The ObjectValue::getCurrentValueOf(String) method determines the prevailing value of an object property. This specification, right at the start of Clause 10, far

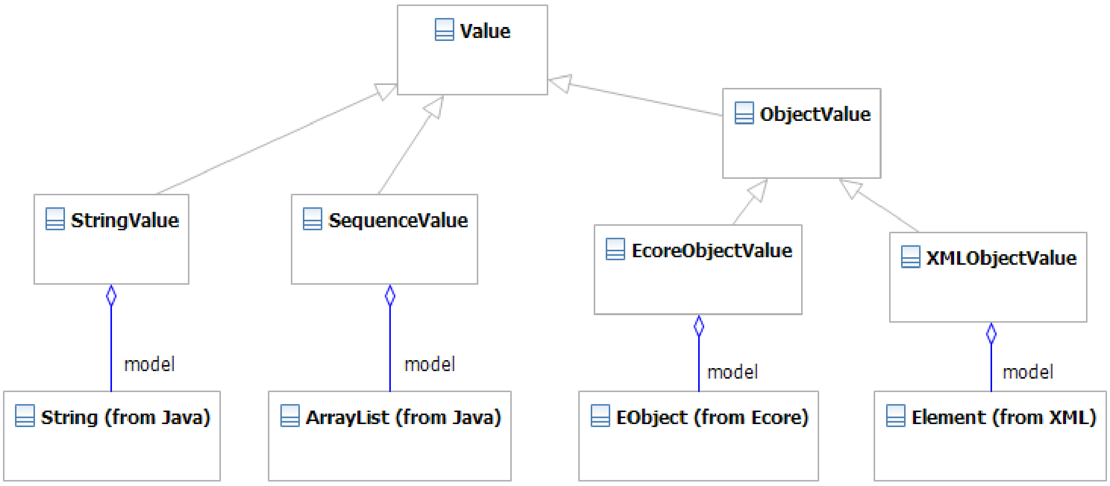

Figure 10: Practical simplified OCL Value hierarchy for a Java tool.

exceeds what is necessary for practical tooling and probably explains why practical tools have ignored the entire clause and realized values much more simply by direct use of implementation language types and modeling environment objects. And equally, since the clause is irrelevant to actual tools, the specification maintainers have failed to understand the clause and consequently Clause 10 has more inconsistencies than any other part of the specification.

Object history is useful to define the foundation for the semantics of pre- and post-conditions and of message histories, since the relationship between two system states must be specified. However, a simple cache of @pre expressions is all that is necessary to support pre- and post-conditions, and a selective trace of object activity can support OclMessage. No history at all is required by implementations that don't offer the @pre and message compliance points.

Discarding Clause 10 completely, and using model objects directly, loses the polymorphism and couples the OCL tooling to a particular model object representation. Wilke[WTW] recognized this limitation as the Model Instance Adaptation Variation Point and introduced Model Instance adapters to accommodate the different object representations of Java, Ecore, XML or Relational Data. It is ironic that this is what the OCL Specification already requires through Value polymorphism. While maintenance of history may be an excessive implementation burden, using a derived XMLObjectValue to mediate between the neutral OCL evaluation engine and the XML specific representation is eminently sensible and necessary to provide an ObjectValue as the object representation.

4.2.3 Summary

Adopting the approach described above, a Java-based tool, might choose to delegate StringValue and SequenceValue directly to Java's String and ArrayList classes as shown in Figure 10, while introducing an adapting layer of EcoreObjectValue or XMLObjectValue for specific Object representations.

Maintaining the separation between behavioral representation and implementation representation for primitive values as specified by Clause 10.2 has considerable advantages; the behavioral Value layer delegates to implementation types, but can impose OCL semantics consistently prior to delegation. The Value layer may therefore provide a behavioral inheritance hierarchy that matches the OCL primitive type hierarchy and so it does not matter whether the implementation language has that hierarchy or not.

Unfortunately Clause 10.2 is very deficient in supporting this view. There is a StringValue class that conforms to a PrimitiveValue class, but no BooleanValue, NumericValue or IntegerValue classes. Clause 10.2 does not define any features for the PrimitiveValue classes, so they appear to fail to fulfill their role of containing a primitive value. However in Clause 10.4, Figure 10.14 provides a consistent model property for primitive Values, although again unfortunately, Figure 10.14 as a whole exhibits a variety of meta-level confusions. Significant revision of Clause 10 is required to support the actual OCL (and UML) primitives.

The OCL specification already requires that all values are maintained by derived Value class instances within an evaluation. Once this aspect of the specification is implementable and realized by tools, interchange within sub-tools may be easier.

4.3 Details

In previous sections we have described significant changes that are required. In this section we identify a variety of comparatively small misalignments between UML and OCL and consider whether and how they might be resolved.

4.3.1 Primitive Types

UML 2.4 has moved the Primitive Types to a separate package to facilitate reuse and defined the previously missing Real type that OCL needed to use. OCL can therefore reuse this UML package.

4.3.2 Types

UML has distinct Type, Classifier and Class classes, but OCL allows features to be added to any type eliminating the major difference between the three UML classes. All three UML classes can therefore be merged into one. The main challenge is to decide which name to use in the merged OCL Pivot Meta-Model. UML has Package::ownedTypes and Class::superClasses and we want to avoid UML users needing to learn a new or inconsistent vocabulary for the OCL Pivot Meta-Model, so these names should persist. Type and Classifier all become Class reflecting the availability of Class functionality for all types.

Of course with all types uniform, the need for companion classes to support Complete OCL operations on data types is eliminated.

4.3.3 Iteration

With the OCL Pivot meta-model derived from UML by package merge, it is not necessary to modify UML in order to introduce an Iteration class. OCL can just define an Iterations package to contribute to the overall merge.

4.3.4 ExpressionInOcl

ExpressionInOcl is shown as deriving from OpaqueExpression in the first figure of Clause 12; this is aligned with UML. Unfortunately the remaining figures and editorial text all use derivation from Expression which is inappropriate but easily corrected.

4.3.5 Qualified Associations

The Concrete and Abstract syntax for qualified associations has never been quite right and has become less so as partial attempts have been made to align with UML evolution.

There is no abstract syntax for `self.associationEndName[qualifiers]` since PropertyCallExp does not support qualifiers.

There is no abstract syntax for `self.associationClassName[qualifiers]`, since AssociationClassCallExp does not support qualifiers.

There is no abstract or concrete syntax for the doubly qualified navigation that may arise with a recursive ternary association class, since AssociationClassCallExpCS does not support qualifiers.

Alignment with UML therefore requires a correct initial specification rather than maintenance.

4.3.6 AssociationEnd

All residual references to AssociationEnd must be revised to use Property.

4.3.7 Expression and OclExpression

UML provides for a homogeneous Expression tree in which nodes have a String name and an arbitrary number of operands.

OCL provides for a heterogeneous OclExpression tree in which nodes have node-specific content such as OperationCallExp::referredOperation.

Both forms of expression integrate with UML classes as the derived ValueSpecification of a Constraint.

Are two distinct trees appropriate? The OCL Abstract Syntax could be revised to extend Expression, but this would involve significant incompatibilities without any obvious benefit. OCL tools benefit from the richer Abstract Syntax, so stronger UML alignment of OclExpression does not seem appropriate.

4.3.8 LiteralSpecifications and LiteralExpressions

UML defines a LiteralString and OCL defines a StringLiteralExpression. These classes have very similar inheritance from TypedElement and could be merged, perhaps introducing a derived property to preserve whichever is obsoleted.

If merged, LiteralExpression would need to extend LiteralSpecification, allowing use of OCL literals more directly in UML Constraints without a wrapping ExpressionInOcl. If OclExpres-

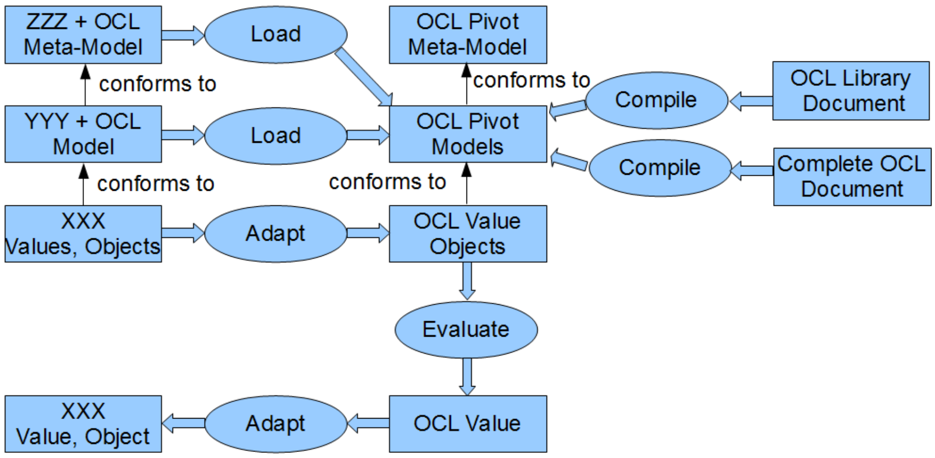

Figure 11: OCL Pivot and OCL Value Usage.

sion similarly extended ValueSpecification, then constant expressions, requiring no self, could also be used directly.

This is relatively minor tweaking, for which there is no obvious demand. However since it could easily accommodated by the UML to OCL merge, it is perhaps worth doing. But it doesn't work, the hybrid OCL::LiteralString and OCL::StringLiteralExpression would extend OCL::ValueSpecification rather than UML::ValueSpecification, and so a derived UML::Opaque-Expression wrapper would still be needed to permit use of OCL within UML.

The only UML-alignment that might be appropriate is a transformation of UML Literals into OCL LiteralExpressions when a UML model is loaded. OCL tooling for pivot models would then not need to handle two alternative forms of literal.

4.4 Practice

The way the OCL Pivot Meta-Models work in practice is shown in Figure 11.

Models upon which evaluations are to be performed are accessed via adapters between derived ObjectValues and the available XXX representation.

The corresponding models are loaded by transforming the YYY meta-model dialect to an OCL Pivot Model. If reflection is required, the meta-models are also transformed.

The models may be complemented by Complete OCL documents that are compiled from their concrete syntax representation, unless a compiled version is already available.

The OCL Library is similarly compiled unless already available.

Then with all values, models and meta-models in normalized form, an evaluation can be performed to produce a Value result which can be adapted back to the user's XXX representation.

Use of this normalization ensures that OCL can operate in the same way for all model and value representations. Use of, for instance, EMOF does not affect OCL. It becomes an EMOF rather than an OCL problem if loading from EMOF provides insufficient model information.

5 Related Work

This paper draws on previous academic work and implementations as the foundation for a proposal for a revised OCL specification.

A Bridge model was used by Kent OCL [AP] to provide a simple core meta-model. The Bridge subsequently mapped to UML1.x, Java and EMF by providing a distinct Bridge implementation for each target meta-model.

The Pivot model was used by Dresden OCL[BD] to provide variability for meta-model representations. A common pivot has adapters for each target meta-model. We take this further to resolve underspecification of Complete OCL and make the XMI compliance realizable. The Dresden pivot adopted an Ecore-like structure for generics with some dynamic typing. The proposal here is UML-aligned and so re-uses UML concepts for generics with static typing.

The Dresden team[WTW] introduced a kind of pivot and adapters for model representation variability. The proposal here clarifies the Values package to demonstrate that similar variability is already part of the specification. It could form the foundation for a Java binding for OCL and provide a neutral Value interchange API.

6 Conclusions

We have identified major problems in the OCL specification in regard to URIs for XMI Interchange and a coherent Abstract Syntax for Complete OCL.

A UML-derived OCL Pivot Meta-Model has been introduced to solve these problems and we have shown how this solves other problems such as UML-alignment, meta-model diversity, reflection, conformance modeling and OCL Standard Library modeling as well. The OCL Pivot Meta-Model decouples an OCL implementation from UML and EMOF and so facilitates meta-model diversity.

Examination of the OCL Values Package has revealed that it requires a similarly decoupling from model representations. This requirement should be clearly expressed in the specification so that value interchange between OCL tools is possible.

Many thanks to the anonymous referees for helpful comments.

Bibliography

[AP] D. Akehurst, O. Patrascoiu. OCL 2.0 - Implementing the Standard for Multiple Metamodels. In *OCL 2.0 - Industry standard or scientific playground?* UML 2003, San Francisco.

[BD] M. Bräuer, B. Demuth. Model-Level Integration of the OCL Standard Library Using a Pivot Model with Generics Support. In *Ocl4All: Modelling Systems with OCL*. Models 2007, Nashville.

[EMF] The Eclipse Modeling Framework Project.
http://www.eclipse.org/emf

[MDT] Eclipse MDT/OCL Project.
http://www.eclipse.org/projects/project_summary.php?projectid=modeling.mdt.ocl

[Obja] Object Management Group. Meta Object Facility (MOF) Core Specification. Version 2.4, omg document number: formal/2010-12-08 edition.
http://www.omg.org/spec/MOF/2.4

[Objb] Object Management Group. Object Constraint Language. Version 1.6, omg document number: formal/2010-02-01 edition.
http://www.omg.org/spec/OCL/1.6

[Objc] 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

[Objd] 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

[Obje] 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

[Objf] Object Management Group. Object Constraint Language. Version 2.3, omg document number: formal/2010-11-42 edition.
http://www.omg.org/spec/OCL/2.3

[Objg] Object Management Group. Query/View/Transformation Specification. Version 1.0, omg document number: formal/08-04-03, 2008 edition.

[Objh] Object Management Group. UML Specification Simplification Request For Proposal. Version 2.03, omg document number: ad/2009-12-10 edition.
http://www.omg.org/cgi-bin/doc?ad/2009-12-10

[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/

[Wil] E. D. Willink. Modeling the OCL Standard Library. In *OCL 2011, International Workshop on OCL and Textual Modelling*. TOOLS 2011, Zurich.

[WTW] C. Wilke, M. Thiele, C. Wende. Extending Variability for OCL Interpretation. In *OCL 2010: Workshop on OCL and Textual Modelling*. Modelsl 2010, Oslo.