UML/OCL – Detaching the Standard Library

D.H.Akehurst[1], W.G.J.Howells[1], K.D.McDonald-Maier[2]

[1] University of Kent
{D.H.Akehurst,W.G.J.Howells}@kent.ac.uk
[2] University of Essex
kdm@essex.ac.uk

Abstract. The Object Constraint Language (or variations of it) is increasingly being used as a text based navigation or expression language over Object-based modelling languages other than the original target of UML. The recent increase of Domain Specific Languages has in particular contributed to this process. As a consequence, it is useful to investigate the lengths to which an OCL like expression language can be made independent of the specifics of the underlying modelling language, concepts and implementation. This paper looks at the issue from the perspective of detaching the OCL specification from the standard library of types that is currently built into its definition.

1 Introduction

The Object Constraint Language (OCL) [3] was originally conceived and designed as a constraint language for use with the UML. Since then, its use has been extended to form a general expression and query language for use with MOF based modelling languages. Such languages need only to support a few basic concepts in order to facilitate the use of OCL for evaluating expressions over the language.

Based on the OCL standard, our works with the Kent OCL toolkit [2] have shown that it is possible to construct a "Bridge" to multiple modelling languages and / or multiple implementations of those languages. – UML, MOF, ECORE, MDR, Java..etc.

Our original bridge proposes a small set of interfaces that must be implemented in order to support the use of OCL with a new language. Some of these classes are required specifically to support the concepts in the OCL standard library, and thus the implementation is by necessity tied to that library. In addition, within the implementation of the OCL processor, the mapping from the OCL standard lib types to implementation types is fixed, e.g. an OCL *String* maps to a Java *String*.

When moving to different implementations of a model, one often discovers that the mapping of OCL standard library types onto similar types in the model implementation is different for different implementation techniques. For example, there may be a user defined String class that the OCL string should be mapped to. In our experience, we have found that it would be most useful to be able to replace the standard types (and methods available on those types) with alternatives, depending on the implementation of the model.

This would achieve three things:

1. The mapping to model implementation of basic types is simplified
2. It provides the '*user*' the option to extend, alter, or replace elements of the standard library.

3. It simplifies the implementation of an OCL compiler/interpreter.

The purpose of this paper is to examine the feasibility of defining an OCL-like language that would meet the requirement of being able to replace the standard library.

The paper is organised as follows: Section 2 looks at the basic requirements of a textual navigation language for object graphs. Section 3 discusses issues about mapping syntax and literal values onto a (potentially) user defined standard library. Section 4 presents a simple meta-model we have been experimenting with. Section 5 includes a discussion on the relationship between user models and the OCL-like language.

2 Basic requirements

Obviously we cannot, with OCL, provide a generic language for writing expressions over any language. However, if we make the not unreasonable assumption that the target languages will all be based on a notion of object-orientation, then we can assert that expressions in the language must be capable of navigating a path through a graph of objects and links. The following subsections introduce the main components we believe are necessary in such a language.

2.1 Navigation Paths

There are two basic requirements for providing a language for navigating such a graph.

1. A means to reference the stating point in a navigation path
2. A means to traverse a link.

A starting point is traditionally given by defining the 'type' of object that can be used as the starting point for the given expression. Further, there are traditionally two options for crossing a link to a new object, either via a property (attribute or association end in UML) or by an operation call. In some cases, the link is pre-defined as part of the initial graph (e.g. an association/link) or sometimes the link is dynamically constructed (e.g. as part of the execution of an operation or derived property). There is no relevant difference between qualified property and operation; the difference is really a syntactic one:

- object.property or object.property[qualifier]
- object.operation(argument)

From a navigation point of view, both of these result in a new 'node' (object) in the navigation path and could be seen as equivalent. However, with a closer look there are some differences that require us to treat them separately:

1. Due to various conventions, the implementation of operations and properties has tended to differ; e.g. the Java conventions of implementing properties with *accessor* and *mutator* methods, starting their name with '*get*' and '*set*'.
2. There is a semantic difference between an operation and a qualified derived property! An operation can modify the state of a model (i.e. modify the objects and links in the graph) whereas a derived property does not; from an OCL perspective, OCL is not supposed to alter the

state, and can thus only call operations that do not alter the state (e.g. marked with 'isQuery' in UML).

The second of these distinctions is not necessarily relevant if an extension of OCL is to be used as a textual means to describe behaviour, including state modification.

Our preference on these issues is to provide two mechanisms for navigation, operations and qualified properties (there may be no qualifier arguments); thus facilitating a differentiation between property and operation if necessary.

2.2 Iterator Expressions

One of the significant features of OCL is the "*iterator*" expressions. These are akin to higher order functions from functional programming languages. They are essential for the navigation over collections of objects. In general, these *iterator* operations take an argument, which is of the form of an OCL expression, and apply the expression to each element of the collection in order to calculate the result of the operation. In OCL, these *iterator* operations are currently built into the language, i.e. they are part of the language definition rather than operation defined on an object. This unfortunately means that users cannot define new *iterator* operations. In order to facilitate such definitions, it is necessary to introduce the concept of an *Expression* as an object type, so that it can be passed as an argument.

2.3 Alternative Paths

With a text expression, it is often necessary to define a set of alternative navigation paths, the choice of which to take being determined at runtime. Typically this facility is offered with concepts such as an "*if*" statement and/or a "*switch*" or "*select*" statement.

The OCL currently has the concept of an if statement; we see no reason not to extend this to the multiple path options offered by concepts such as the "switch" or "select" statements found in many programming languages. Such a concept could follow the same convention as the existing "*if*" statement in requiring a default option (the construct must always return a value), or assume that if a default option is not provided then an OCL "*null"* or "*invalid"'* value is returned.

This kind of multiple paths conditional statement is particularly useful when testing variables of an enumeration type, in addition to other situations, which using the current OCL must be formed using multiple nested "*if … then … else … endif*" statements.

2.4 Sub Expressions

The OCL concept of "*let … in …"* is essential to writing concise and readable complex expressions. They provide a means to define a number of sub-expressions that can be reused within the expression as a whole. This could be seen as syntactic sugar; however, its use can, in addition to improving readability, also improve the performance of evaluating the expression; hence we feel the concept should be included in the language definition.

3 Syntax and Literal Values

It is necessary to define a binding between literal values and types within the model. Usually this is built in to the language. However, we believe it is feasible to provide definitions/mappings at compile time.

An example mapping between literal values and types found in a Java implementation of a model could be given as shown in Table 1. This mapping would not, of course, provide the operations given by the standard OCL library. (The non-terminal names come from the grammar specification of the language.)

Literal non-terminal name	**Type**
stringLiteral	java.lang.String
integerLiteral	java.lang.Integer
realLiteral	java.lang.Double
booleanLiteral	java.lang.Boolean

Table 1

An alternative mapping shown in Table 2, could provide the standard OCL operations, but requires the model to be implemented in Java using the named types.

Literal non-terminal name	**Type**
stringLiteral	my.ocl.String
integerLiteral	my.ocl.Integer
realLiteral	my.ocl.Real
booleanLiteral	my.ocl.Boolean

Table 2

Operator symbols are also an issue; whether they are prefix, infix or postfix, they must to be mapped to appropriate operations on a type. This could be provided in a simple fashion by binding the operator symbols to operations names as illustrated in Table 3. The problem with this would be with operators such as '-', which has both a prefix and an infix meaning and hence would ideally be mapped to two different operation names – 'minus' and 'negate'.

Operator Symbol	**Operation Name**
+	plus
-	negate
-	minus
*	multiply
/	divide
and	and
or	or
%	modulus

Table 3

A more complex approach could be taken by binding the operator symbol and the operand types to operations on particular types, as indicated in Table 4. This approach

requires the mappings to explicitly reference the types on which the operators are applicable.

Types	Operator Symbol	Operation Name
{..Integer, ..Integer}	+	plus
{..Double, ..Double}	+	plus
{..String, ..String}	+	concatenate
{..Double, ..Double}	-	minus
{..Integer, ..Integer}	-	minus
{..Double }	-	negate
{..Integer }	-	negate
{..Boolean, ..Boolean }	and	and

Table 4

A third alternative would be to make a distinction in the mapping information between prefix, infix and postfix operators, but require the names of the implementing operation to be the same whatever type the operator is applied to.

Type	Operator Symbol	Operation Name
infix	+	plus
infix	-	minus
prefix	-	negate
prefix	not	not
infix	and	and

Table 5

The first option is nice and simple, but is too restrictive; the second option gives us extensive flexibility in mapping operators to operations, but in out opinion requires too much information to be specified. Our preference is for the third option, which gives sufficient flexibility without requiring the level of detail necessary with the second option.

4 A Simple Navigation Language Meta-Model

The meta-model for a language defined along these means could consist of two parts

1. Navigation Part: for defining navigation paths and expressions.
2. Bridge Part: for abstracting the connection between navigation paths and the object graph.

The meta-model of an OCL-like language that we have been using to experiment with the ideas discussed in this paper is shown below in the two figures.

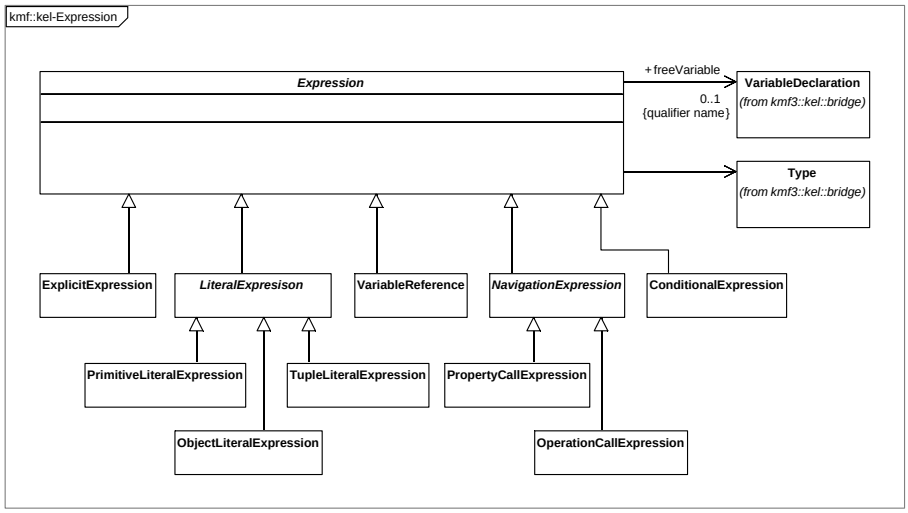

Figure 1 – Types of Expression

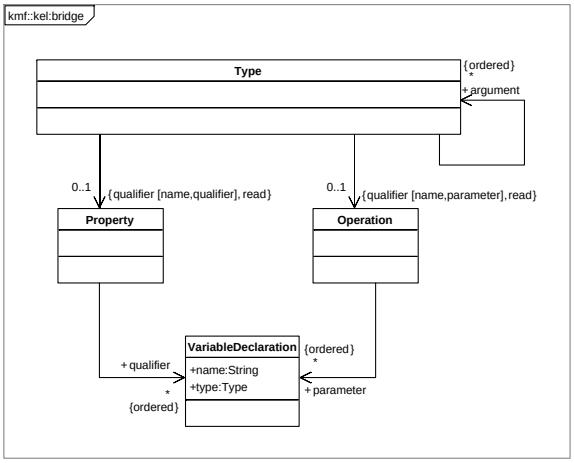

Figure 2 – Bridge Concepts

5 On Models

The OCL standard does not only provide a library of primitive and collection types and define a text based navigation language. It also alters the type hierarchy of the target execution model. That is to say, the OCL language makes an assertion that all types in the model extend types in the OCL standard library – e.g. OclAny and OclModelElement.

These types contain useful operations such as testing for equality or checking the type of an object. The assumption in OCL is that these operations are not provided by the model, and thus a common super type, that does provide these operations, is

required. This probably arose initially due to OCL being “added” to UML as an afterthought. We would argue that it is an unusual policy.

A more usual approach is to provide a common super type and ensure that every model type does extend it. For instance, common OO programming languages such as Java [4] and C# [1] have common super types of ‘java.lang.Object’ and ‘’System.Object’; all classes defined in these languages automatically extend the common super type, it is not added as an afterthought by the expression part of the language.

We propose that the same approach should be taken in the modelling world. All models must specify the type in the model that is the ‘common super type’ of the model; this type could be provided by a standard library, but may be replaced by an alternative.

The OCL standard library thus becomes a ‘model’ just like any other. However, this model will probably be included (imported) by specific domain models, providing a standard set of primitive and collection types and a standard common super type.

5.1 UML + OCL

The current situation of UML + OCL can be mimicked using the techniques discussed above as follows:

- The current OCL standard library is provided as a UML package and classes.
- A specific Domain Model includes the OCL standard library and defines the OclModelElement class as the root type for the model.
- The ‘new’ standard library will contain a type ‘*Expression’*.
- Iterator operations are defined on the collection classes, taking parameters of type *Expression*.

5.2 A New Standard Library

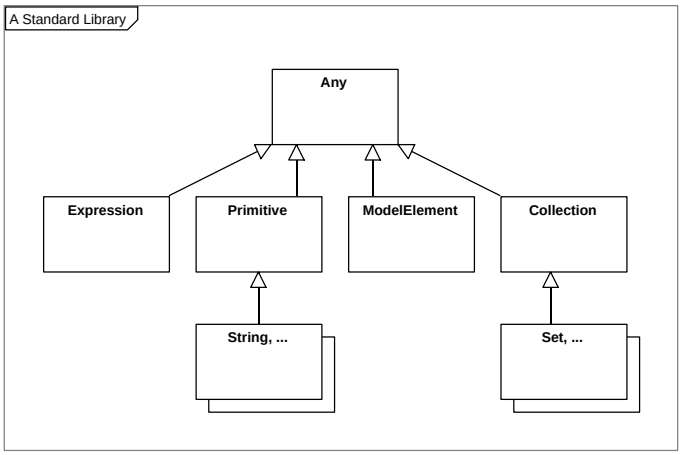

Figure 3 – A possible Standard Library

- **Expression**: An expression type enables us to pass expressions as parameters to operations.
- **Primitive**: A Primitive type is distinguished from other types as primitive objects can be constructed from string constants (e.g. by the OCL compiler).
- **ModelElement**: Within UML, a Model Element object embodies some notion of containment (i.e. composite/part structure indicated with black diamonds) and its type is given as a class in the model. This provides a candidate for the "*inheritance root*" of Models.
- **Collection**: A collection object does not require a notion of containment.

6 Conclusion

We have proposed an approach to "detaching" the OCL standard library from the navigation and expression part of the language. This moves the standard library (and types) to the same position as all other model elements, thus facilitating the extension or replacement of the standard types.

Our initial experiments based on the Kent OCL library have shown that this approach to providing OCL support has some merit; in particular we have found it very useful to be able to extend and replace the standard types.

An outcome of these experiments has been the production of a Java library that supports all the OCL iterator operations as operations on a collection classes; we have found that this library is very useful as a target for OCL code generation tasks, in addition to simply being a useful library for use within straightforward Java programmes.

We are currently experimenting with "bridging" the OCL-based navigation language to alternative object-based DSLs, alternative implementations of UML/MOF and with alternative libraries of primitive and collection types.

References

1. Microsoft: Visual C#. http://msdn.microsoft.com/vcsharp/
2. OCL-team: Kent OCL library. www.cs.kent.ac.uk/projects/ocl
3. OMG: UML 2.0 OCL Specification version 2.0. Object Management Group, pct/05-06-06 (June 2005)
4. Sun: Java Technology. http://java.sun.com/