Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 102 (2004) 21–41
www.elsevier.com/locate/entcs

OCL 2.0 - Implementing the Standard for Multiple Metamodels

David Akehurst[1]

Computing Laboratory
University of Kent
Canterbury, UK

Octavian Patrascoiu[2]

Computing Laboratory
University of Kent
Canterbury, UK

Abstract

OCL 2.0 is the newest version of the OMG's constraint language to accompany their suit of Object Oriented modelling languages. The use of OCL as an accompanying constraint and query language to modelling with these languages is essential. As tools are built to support the modelling languages, it is also necessary to implement the OCL. This paper reports our experience of implementing OCL based on the latest version of the OMG's OCL standard. We provide an efficient LALR grammar for parsing the language and describe an architecture that enables the language to be bridged to any OO modelling language. In addition we give feedback on problems and ambiguities discovered in the standard, with some suggested solutions.

Keywords: modelling, language, constraint, grammar, translator,compiler, interpreter, parser, bridge.

1 Introduction

This paper illustrates how we have implemented an executable version of OCL in such a manner that we can provide a bridge to a variety of OO metamodels. The prime motivation of this work has been to provide a tool to enable

[1] Email: D.H.Akehurst@kent.ac.uk
[2] Email: O.Patrascoiu@kent.ac.uk

1571-0661 © 2004 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2003.09.002

constraints to be checked over populations of a variety of models. Much of the original work was carried out as part of [1], with the latest versions and conformance to OCL version 2.0 being done under the Kent Modelling Framework (KMF) project [5] at the University of Kent, involving both the DSE4DS [3] and RWD [9] projects.

We propose a structure for the model of the OCL concepts that facilitates the use of OCL over a number of different metamodels. A carefully specified set of interfaces can be defined as a bridge, which enables a common library of OCL parser, analyzer, evaluator and code generator to be used in the context of a number of different metamodels. We have implemented bridges for three different metamodels, providing OCL for Java, for KMF and for the Eclipse Modelling Framework (EMF). Our KMF implementation will be updated to use UML 2.0 when it is finalised, the architecture proposed here facilitates an easy update path. This architecture provides a clean and well-defined division between the OCL model and the metamodel to which it is attached, whilst still providing the necessary linkage.

The experience of implementing this library has shown us where there exists ambiguity, errors, and missing parts of the OCL 2.0 specification; we highlight and discuss these issues, with some suggested options for fixing the problems.

We have produced an LALR grammar for the syntax, suitable for input to bottom up parser generators (CUP, YACC, BISON). Such grammars contain no look-ahead or backtracking. A previous version of the parser was based on a LL(k) grammar which was derived from the original OCL standard.

2 Implementation Structure

Our implementation follows the typical structure of a translator, consisting of 4 stages: lexical analysis, parsing, semantic analysis and either code generation or evaluation. This is shown in Figure 1.

The lexical analyser and parser generate an abstract syntax tree (AST) from the input text; syntactic errors are reported by these processes. The next process, Semantic Analysis, requires input of an AST and the user model to which the OCL expression is attached. Semantic analysis will generate static semantic errors. Finally, we provide two options for synthesis, either code generation or evaluation using an interpreter.

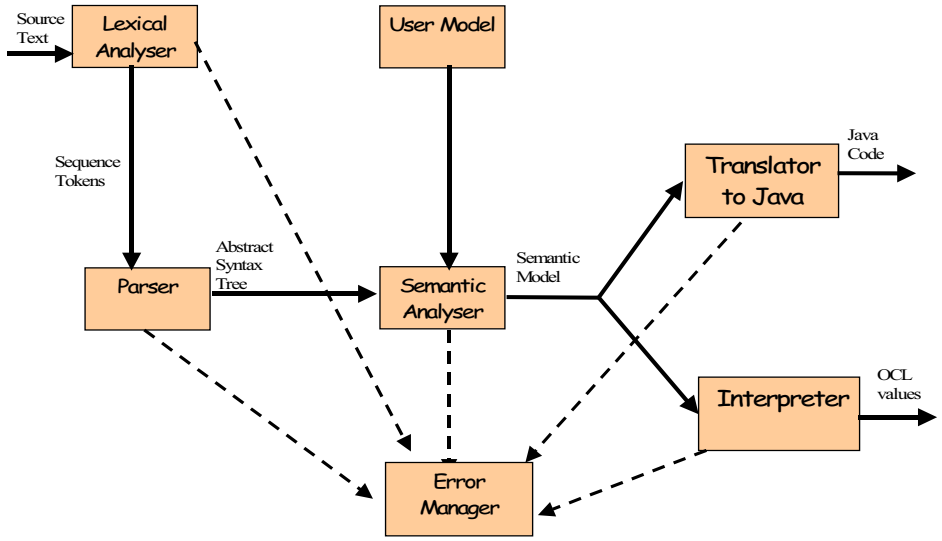

Figure 1 Implementation Structure

We refer to the output of the Semantic Analyser as a Semantic Model, as this model contains concepts relating to both the syntax of the expression and the concepts referring to elements from the user model. Each of these stages has involved different problems relating to the specification contained in the OCL standard; we discus each stage separately in the following sections.

3 Parsing

The OCL standard does not define a grammar suitable for input to a parser generator. The grammar specified in the standard is classified as an "Ambiguous" grammar. An unambiguous grammar is required if the language is to be implemented. The attributed grammar specified in the standard is not suitable as a parser specification as it contains ambiguities such as the rules for parsing:

$$A :: B :: C$$

Is this expression referencing an enumeration literal, or a path name to a type? The grammar in the standard distinguishes by using contextual information, which is not available during a purely syntax based analysis (such as parsing). The disambiguating rules depend on information from the environment, i.e. semantic information from the user model and context of the expression.

However, there is no syntactic difference.

Our grammar is an equivalent grammar to that defined, however we have had to make changes to enable deterministic parsing. This has generally involved providing a common syntactic construct for the terms differentiated by disambiguating rules (E.g. enumeration literals and path names or the different types of property call).

Appendix A contains the EBNF for an LALR grammar we have used in our implementation. Our grammar is distinct from the one defined in the standard in that it is unambiguous. The most noticeable difference is the manner in which we define the rules for the *OclExpression* non-terminal.

In every programming language operators have an associated precedence. The compiler uses this attribute to decide in which order are the operators evaluated. In order to specify the precedence of the operators there are two choices:

- The grammar can be structured on several levels using extra non-terminals, or
- In the case of LR grammars the precedence can be specified using directives.

We decided to use the second option because it will give us a smaller and faster parser. This happens because the first approach generates grammars with more non-terminal symbols and hence more rules. For example, if we consider an arithmetic expression with + and * the grammar build according to 1) is

$$E ::= TX$$
$$X ::= \lambda \mid {}' +' \, TX$$
$$T ::= FY$$
$$Y ::= \lambda \mid {}' *' \, FY$$

Using the second approach the following grammar can be defined:

$$E ::= E' *' \, E$$
$$E ::= E' +' \, E$$

Thus, instead of breaking down the non-terminal for *OclExpression*, using extra levels, we specify a rule for the use of each operator and include a definition of operators precedence.

The following subsections discuss issues we have discovered with the grammar specification contained in the standard.

3.1 Parsing Types as Arguments

One issue we have discovered is how to parse an expressions of the form :

$$expr.oclAsTypeOf(Type)$$

If the Type is a collection or tuple type then the parser will fail as types are not considered to be valid expressions in their own right. Types that are referenced as path names are parsed ok and can be disambiguated during semantic analysis.

A suggested solution is to extend the definition of literal expressions to include the syntax for collection and tuple types.

3.2 Iterator and Accumulator Variables

Another issue with the grammar is the syntactic construct for iterator and accumulator variables. These variables have the same syntax and are used in iterator and iterate expressions. According to [7] an expression like this

$$Set\{1,2,3\}->select(x:Integer, y:Integer \mid x+y=3)$$

is an iterator expression, and

$$Set\{1,2,3,4,5,6\}->iterate(e:Integer; acc:Integer=0 \mid acc+e)$$

is an iterate expression. The first expression contains two iterator variables. The second contains an iterator variable and an accumulator variable.

The grammar of the language is defined such that the $'|'$ symbol, which provides the syntactic information that a parsed name is an iterator variable rather than an expression, comes after the definition of the iterator variable. In itself this is not a problem, the problem is caused by the fact that multiple iterator variables can be defined, separated by commas, with an optional type definition and (syntactically) optional init expression (although the init expressions are not allowed from a semantic perspective).

These aspects make it hard for rules to be written that correctly parse the language; we have solved the problem by separately listing the variation in number and style of iterator variable definitions. This would be made simpler if an alternative separator were to be used. A possible alternative could be a semicolon; such a separator is used within *iterate* expressions, hence it would not be inconsistent in the iterator expressions. Without making a change along these lines the language can not include facility for multiple iterator variables - more than two - unless the options for all possible numbers are separately listed. It may also be advantageous to distinguish between a syntactic variable declaration construct that may have an init expression and one that may not.

4 Semantic Analysis

The parser generates an Abstract Syntax Tree - i.e. a model of the text expression entered. This is of a form where there is a direct association between rules in the grammar and nodes in the tree. The AST is purely an abstract representation of the syntax, modelled as a tree.

Before we can interpret the meaning of the syntax, we must provide a semantic context for the expression. This context involves two parts: the UML (or other) user model over which the expression is to be interpreted; and the entry point into that model - i.e. the type of the *self* variable.

It is the job of the analyser to map the AST onto a model of the expression that contains semantic information relating to the context model and to report 'static semantic' errors e.g. those relating to type inconsistence.

The model defined in the OCL standard and named, incorrectly, Abstract Syntax Model (ASM), is such a model. We feel that this model is misleadingly named; it does not contain purely Abstract Syntax information, it contains a mix of syntax nodes and semantic nodes. A semantic node is a node containing information relating to the user model and context of the expression. We suggest that an alternative name be used. As the model contains semantic information referencing the user defined context model, we refer to this as the OCL Semantic Model. (Not to be confused with a model of the semantics, which is something else not addressed in this paper.)

Our analyser performs two jobs: it maps string based path names onto types, properties and methods in the context model and maps OCL specific operations onto the appropriate semantic model constructs. The mapping to semantic model constructs is performed in accordance with the disambiguating rules defined in the standard and the mapping to user model elements is carried out by the operations defined on the Environment class, also defined in the standard.

4.1 Semantic Model

The semantic model (or ASM) defined in the standard can be divided into three sets of classes:

(i) Those that define the OCL concepts.

(ii) Those that refer to concepts from the UML metamodel.

(iii) Those that define the type system for the standard library.

The concepts from (i) we further divide into those relating to the context of an expression and those dealing with concepts in an expression.

The classes from (ii) are distinguished in the standard by the definition that they come from various packages in the UML metamodel and they are additionally coloured white as opposed to grey. We redefine these classes to be members of a single package named *bridge*. They keep the same names as before, but should be considered to map to the classes form the UML model, rather than directly being classes from the UML model.

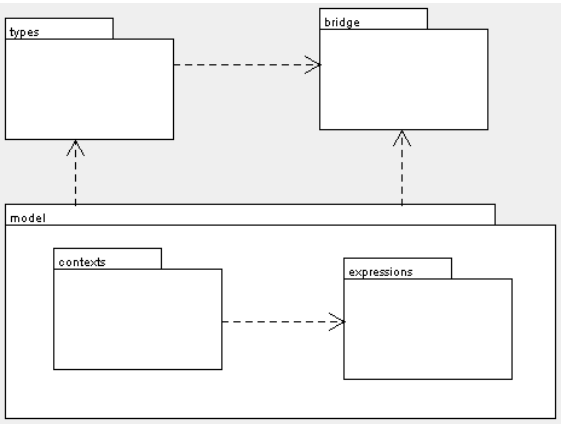

Figure 2 Overview of OCL Semantic Model

The classes from (iii) define the type system for the standard library; they define the types contained in the library and the operations available on those types. We pull these classes out into a separate package as they do not form part of the Semantic Model, although they are required by it and do form part of the defined language semantics of OCL. These sets of classes we divide into packages as illustrated in Figure 2.

To map our OCL library onto different models, it is necessary to provide different implementations of the bridge classes; many of which can be extensions of provided common implementations.

In the following subsections we show the content of the *bridge*, *contexts*, and *type* packages as these are additions or variations to the classes described in the standard. The *expressions* package contains the classes as defined in the standard, excepting those relating to *ModelPropertyCall* expressions, which we have also altered and show in a following subsection.

4.2 Bridge

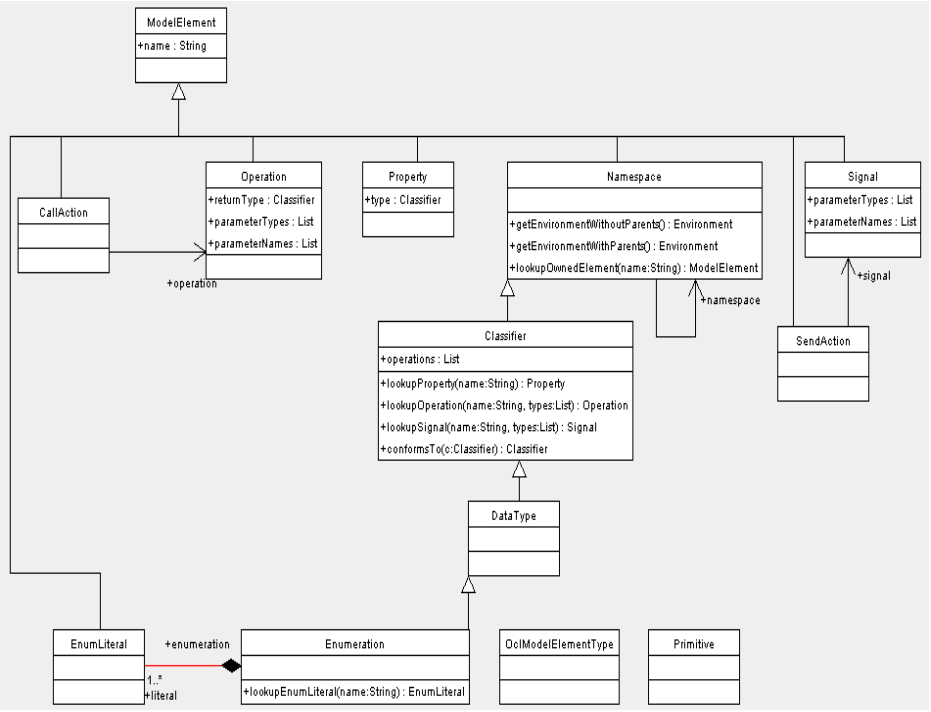

Figure 3 Bridge Classes

The classes in the bridge package (Figure 3) are those that must be supported by any model over which it is wished to interpret OCL expressions. These classes collectively provide the contextual information that enables an OCL expression to be evaluated. They easily map to classes from the UML 1.X metamodel as that is the model for which OCL was originally designed. However, we have successfully mapped the classes to the metamodel for Java and to the ECore Metamodel associated with IBM's Eclipse Modelling Framework. We see no problems mapping the classes to the UML 2.0 metamodel or MOF metamodels as and when their specifications are finalised.

The operations and properties on the classes are those used within the disambiguating rules and the definition of the operations on the *Environment* class included in the OCL 2.0 standard.

4.3 Types

There are two versions of OCL Type model included in the standard; one that forms part of the definition of the standard library and one that forms part of the Semantic Model. These two type models are not entirely consistent. We have merged the information from the two models to provide something consistent Figure 4.

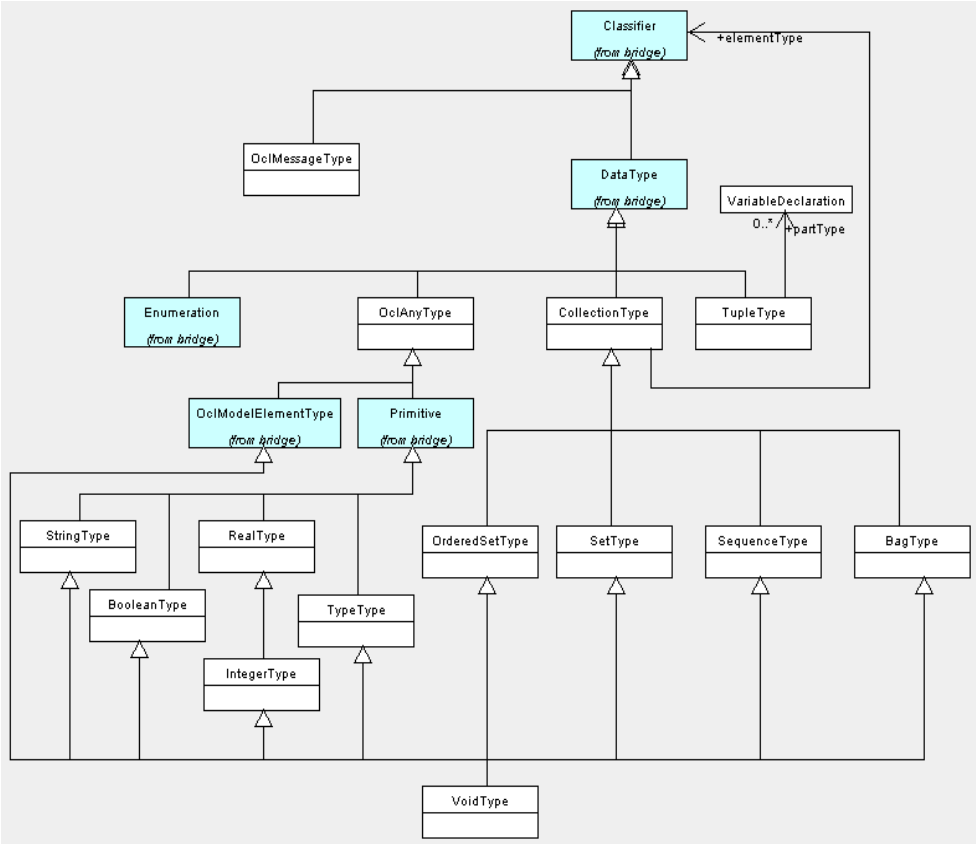

Figure 4 OCL Types

The objects defined in the standard library mirror the type hierarchy defined here.

The main changes are:

- The inclusion of a type for *OclType* objects.

- A change to hierarchy structure to provide consistency between this model and the type hierarchy of the standard library objects.
- Addition *OclAnyType*.
- Addition of *TuplePart* to *TupleType*.

According to OCL 2.0 proposal collection and tuple types are not considered to be subtypes of *OclAny*. This means that operations specific to *OclAny* cannot be applied to instances of tuple and collection types. We do not find any reason why collection and tuple types cannot be considered subtypes of *OclAny* and in fact we find that it be necessary that they are if we are to enable them to be type cast. For example, consider a *Set* of type *Animal* that we know to contain only objects of type *Dog*, we may wish to perform a cast on the *Set*, as shown in the following expression:

$$Set\{rover, fido, fluffy\}.oclAsType(Set(Dog))$$

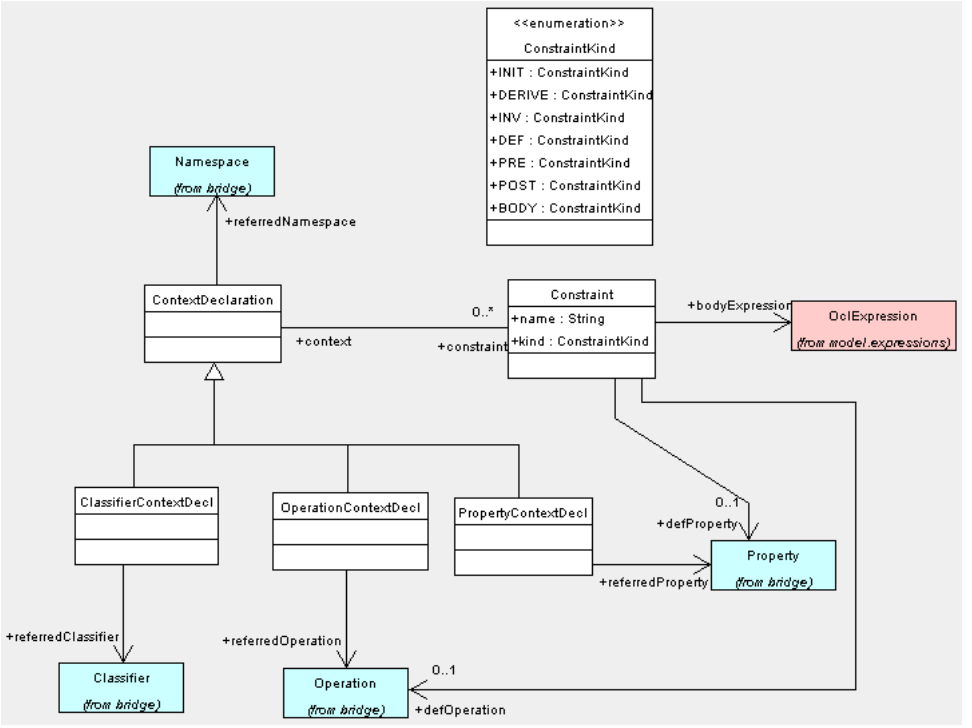

Figure 5 OCL Expression Contexts

Such an expression is not currently accepted syntactically (see above) or semantically, as the *OclAny* operations cannot be used on collections. Considering tuple and collection types as subtypes of *OclAny* will increase the expressiveness and the usability of OCL. If such a feature is not available in OCL, the user will have to use other syntactical constructions in order to obtain the same effect (e.g. iterate over the above collection and cast each element).

4.4 Context

A concrete syntax for context definitions is given in the standard, but a semantic model for such contexts is not provided. The model in Figure 5 is that used by our implementation.

4.5 Expressions

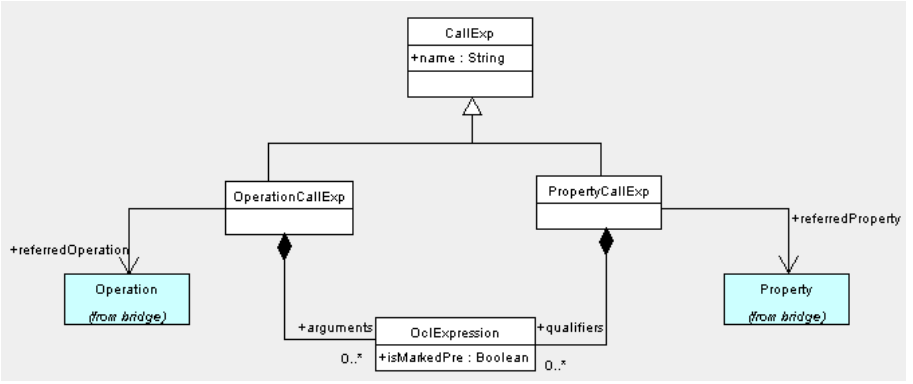

Figure 6 Model Call Expressions

The model for expressions is used as defined in the standard except for the classes surrounding *ModelPropertyCall* (Figure 6). We see no need from an OCL semantics perspective to distinguish between attributes and association ends, we hence combine these into a single class *PropertyCall*. This class covers also static attributes, so the expressiveness is not reduced. In addition the class *ModelPropertyCall* is defined as a super type for Operations and the new class *PropertyCall*; an operation call is not a property call thus we feel it should derive directly from *CallExp* and hence the class *ModelPropertyCall* becomes redundant.

There is one other comment regarding classes in the expressions package; Let expressions are syntactically defined as a sequence of statements, but in the semantic model are defined as expressions nested inside each other. We feel that either approach is acceptable, but that they should be consistent.

4.6 Environment

The specification (in the standard) of the *Environment* class is missing a few things that are used or referred to elsewhere in the standard; some are missing altogether and some are missing from the class diagram:

- The association from an environment to its parent.
- The operations *lookupImplicitSourceForOperation*, *lookupPathName*, and *addEnvironment*

We show a more complete specification in Figure 7. We also add a convenience method *addVariableDeclaration*; although not necessary as *addElement* can be used to add a *VariableDeclaration*, this operation avoids the need to construct the *VariableDeclaration* before adding it to the environment. The specification of the *Environment* operations uses various methods on the *bridge* classes; we have added these operations to the classes, as shown in the previous section about the *bridge* classes.

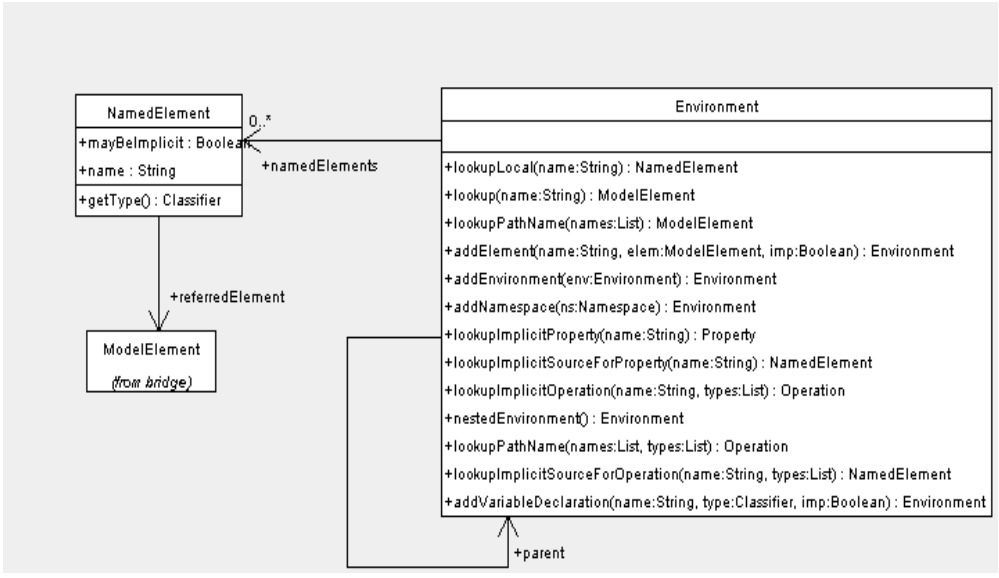

Figure 7 Specification of the Environment Class

5 Synthesis

The semantics of OCL seem to be well defined and we have had few issues regarding the implementation of the evaluation and code generation processes. Both processes are implemented as visitors over the semantic model.

6 OCL Standard Library

Our implementation of the OCL standard library is built on top of the basic types found in the *java.lang* and *java.util* packages. We have had few issues regarding implementation of the standard library classes.

When comparing collections (implementation of '=') one must not use the equals method provided on the *java.util* collection classes as this does not give the correct results regarding the comparison of nested collections.

The implementation of the *OclAny* operations is dependant on the implementation of the model that supports the bridge package. Our implementation provides facility to adapt the evaluation of the *OclAny* operations as applied to *OclModelElements* depending on the specific bridge implementation.

There is an issue regarding the implementation of the *OclVoid* class (and *undefined* object); the type hierarchy states that the class extends (in addition to others) all the collection types. Unfortunately, some of the collection types have methods with same signature but different return types.

We have no satisfactory solution to this at present, and our implementation returns java 'null' values if the semantics require an undefined value. These null values are mapped to undefined values by the evaluator if necessary.

There is some ambiguity regarding *OclType*. The standard states in section 3 that the class has been removed and yet it occurs within the definition of the standard library classes and operations.

We have opted to include the *OclType* class as it is necessary for operations such as *oclAsType*. We have also found that throughout the standard a number of operations are used on OCL types that do not occur within the definition of the standard library. For example, the operation *tail* on Sequence objects.

7 Bridge Implementations

The main purpose of the bridge classes is to provide linkage between the OCL expressions and the model over which the expression should be evaluated; it provides type information from the user model. Consequently, depending on the metamodel that we use to implement the bridge, the implementation of the bridge classes will vary, as will the issues involved.

The following subsections discuss the issues relative to each of our three bridge implementations. Each of these bridge implementations provides support for the *Enumeration*, *Namespace*, *Operation* and *Property* classes. The implementation of the other bridge classes is common to each of these three, and we suspect common to most bridge implementations.

7.1 OCL for KMF

KMF version 2.0 is based on the UML1.4 metamodel. KMF uses a UML 1.4 XMI file to build a model implementation; it is this implementation that we wish to use as the user model for our OCL expressions. In order to get the correct type information, irrespective of the model implementation details, the KMF bridge implementation gets all of its information from the same XMI file used to store the model information and generate the Java code which implements the model.

The file is use to populate an implementation of the UML 1.4 metamodel, which is used as the underlying implementation of the bridge classes.

7.2 OCL for EMF

The Eclipse Modelling Framework (EMF) is IBM's version of a similar tool to KMF, to quote the overview of EMF:

> "EMF is a Java framework and code generation facility for building tools and other applications based on a structured model. For those of you that have bought into the idea of object-oriented modeling, EMF helps you rapidly turn your models into efficient, correct, and easily customizable Java code".

EMF code generation is based on a metamodel called ECore (Figure 8); as you can see, there are similarities between this and the UML metamodel. The java code generated by EMF carries with it all the information from the defined model (unlike KMF), i.e. it is possible to access an instance of an ECore class from each object instantiating a user model class. Thus the implementation of the bridge classes is achieved by forwarding calls to the appropriate ECore classes.

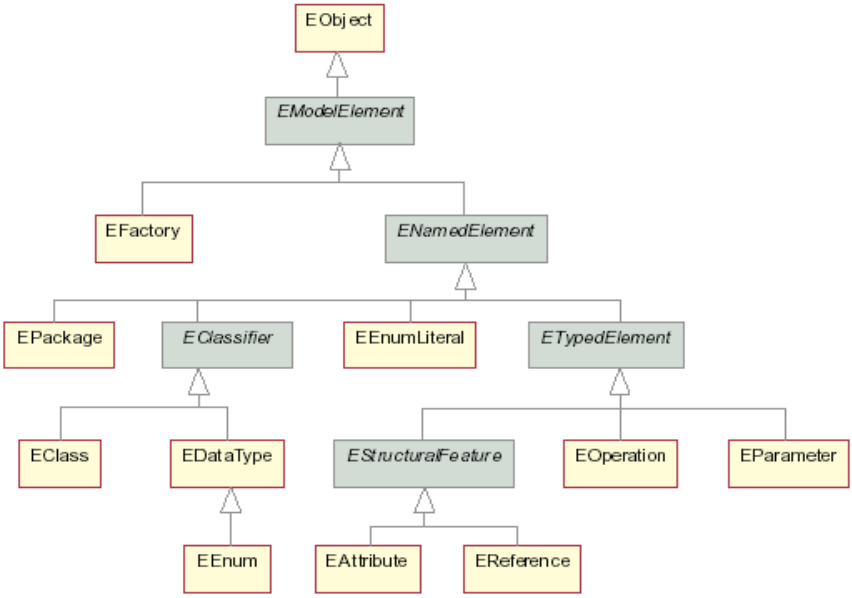

Figure 8 ECore model (taken from EMF overview)

The similarities between the ECore model and the UML metamodel mean that there are no difficulties in providing a bridge implementation. The only issue is the use of collection classes. EMF makes use of an EList implementation and extension of *java.util.List* for all types of collection. This class has an *isUnique* property to enable distinction between collections with *Set* like properties and those without. There is no distinct difference made between *Sequences* and *Bags* or between *Sets* and *OrderedSets* - all collections are ordered; however, this has not proved to cause problems in building the bridge, but it must be born in mind that one will always get a *Sequence* or *OrderedSet* when getting collection properties from a user model.

7.3 OCL for Java

The most problematic bridge implementation is the one for Java. Java does not provide an explicit mechanism for creating enumerations; it does not provide typed collections classes; and its notion of a package does not match the UML package concept. The reflective capabilities of java have proved essential to forming our bridge implementation.

Enumerations

We identify an enumeration in one of two ways. Either by looking up the enumeration in a pre instantiated list of enumerations, or by testing if the class extends *java.util.Enumeration*. This is a slight misuse of the *java.util.Enumeration* class, but it provides a nice solution to the problem. Such enumerations are assumed to be implemented with each enumeration literal being a static member of the enumeration class and an instance of that class.

Namespaces

The problem with a namespace is that java packages are separately identified by their full package name. Although appearing to support the notion of sub-packages, the java reflection features do not hold this sub-package relationship. Hence, to lookup an owned element of a namespace by name, we first try and find a java class with the element name plus full path name of the current namespace; if that fails, we assume the name is a sub-namespace, create the appropriate sub-namespace object, and return the sub-namespace. This is not necessarily the best approach, but seems to work in most situations.

Operations

We simply use reflection to get the java signature of an operation and convert this to the correct representation as a bridge class.

Properties

We assume standard java get/set methods are implemented for each property. The bridge implementation simply capitalises the name of the property, adds a "get" prefix, and use the same reflexive process as for an operation with no arguments.

Typed Collections

To construct the correct OCL typed collection type for property types and operation return types, it is necessary to get extra information about the type of the collection. Java collections do not carry this information. We provide two options; one is to pre-instantiate a list mapping properties and operation names to java classes that are the collection element types; or when a property or operation has a collection as its return type, a static final field can be added that is named with the name of the property/operation + "*_elementType*" and whose type is the element type of the collection. Reflection operations are used to look up this field when needed.

8 Related Work

There are many CASE tools supporting drawing of UML diagrams and features like code generation and reverse engineering. However, support for OCL and transformation and mappings between models is rarely found in these tools. There are several tasks that a CASE tool should offer in order to provide support for OCL. For example, syntax analysis of OCL construction and a precise mechanism for reporting syntactical errors, help in writing syntactically correct OCL statements. The next step could be a semantic analyser, which should report as many errors as possible in order to help the user to develop solid OCL code. If the tool offers both an interpreter and a compiler, the user has the possibility to choose the best approach in order to obtain a high quality software.

Probably the first available tool for OCL was a parser developed by the OCL authors at IBM, now maintained at Klasse Objecten. The parser uses the grammar described in [6]. Another toolset was developed at TU Dresden [4]. A part of this tool has been integrated with the open source CASE tool Argo [2]. [10] contains a description of an OCL interpreter. It is based partly on a OCL meta-model describing the abstract syntax of OCL. [8] provides also a good implementation for OCL.

9 Conclusion

We have been experimenting with implementations of the OCL since it was first added to the UML. It is our opinion that the language is invaluable as part of the OMG modelling environment however we feel that it is imperative that the language be implemented as part of the standardization process in order to avoid the ambiguities and inconsistencies we have discovered.

Our experience has illustrated many areas in which the standard requires improvement and we have provided ideas to address some of these improvements. In particular we suggest the need for a reference implementation of language in order to improve the definitions included in the standard.

9.1 Unsupported Concepts

Our implementation currently does not fully support the following constructs:

- hasSent and message Operators ('^' and '^^')
- contexts, other than inv:
- OclState, OclMessage types
- @pre references

References

[1] Akehurst D. H., "Model Translation: A UML-based specification technique and active implementation approach", PhD. thesis, Department of Computing, University of Kent at Canterbury, Canterbury, 2000.

[2] ArgoUML, A UML design tool with cognitive support, URL: http://www.argouml.org.

[3] DSE4DS-team, Design Support for Distributed Systems (DSE4DS), URL: http://www.cs.kent.ac.uk/projects/dse4ds/index.html.

[4] Demuth B., H. Hussman, F. Finger, *Modular architecture for a toolset supporting OCL*. In Evans A., S. Kent, and B. Selic, UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings volume **1939** (2000)of LNCS, pages 440-450, Springer 2000.

[5] KMF-team, Kent Modelling Framework (KMF), URL: http://www.cs.kent.ac.uk/projects/kmf.

[6] Object Constraint Language Specification. In OMG Unified Modelling Language Specification, Version 1.3, June 1999 [19], chapter 7.

[7] Object Management Group, The Unified Modelling Language 2.0 - Object Constraint Language 2.0 Proposal, URL: http://www.omg.org.

[8] Object Contraint Language Evaluator, Research Laboratory for Informatics, University Babes-Bolyai, Cluj, Romania.

[9] RWD-team, Reasoning with Diagrams (RWD), URL: http://www.cs.kent.ac.uk/projects/rwd.

[10] Gogolla M., M., Richters M., *A metamodel for OCL*. In France R. and Rumpe B. editors. UML'99 - The Unified Modeling Language. Beyond the standard. Second International Conference, Fort Collins, CO, USA, October 28-30, 1999, Proceedings, volume **1723** (1999) of LNCS, pages 156-171, Springer 1999.

Annex A

packageDeclaration ::= 'package' pathname contextDeclList 'endpackage'
packageDeclaration ::= contextDeclList
contextDeclList ::= contextDeclaration*
contextDeclaration ::= propertyContextDecl
contextDeclaration ::= classifierContextDecl
contextDeclaration ::= operationContextDecl
propertyContextDecl ::= 'context' pathname simpleName ':' type
initOrDerValue+
initOrDerValue ::= 'init' ':' oclExpression
initOrDerValue ::= 'derive' ':' oclExpression
classifierContextDecl ::= 'context' pathname invOrDef+
invOrDef ::= 'inv' [simpleName] ':' oclExpression
invOrDef ::= 'def' [simpleName] ':' defExpression
defExpression ::= simpleName ':' type '=' oclExpression
defExpression ::= operation '=' oclExpression
operationContextDecl ::= 'context' operation prePostOrBodyDecl+

prePostOrBodyDecl ::= 'pre' [simpleName] ':' oclExpression
prePostOrBodyDecl ::= 'post' [simpleName] ':' oclExpression
prePostOrBodyDecl ::= 'body' [simpleName] ':' oclExpression
operation ::= pathName '(' [variableDeclarationList] ')' [':' type]
variableDeclarationList ::= variableDeclaration (',' variableDeclaration)*
variableDeclaration ::= simpleName [':' type] ['=' oclExpression]
type ::= pathname
type ::= collectionType
type ::= tupleType
collectionType ::= collectionKind '(' type ')'
tupleType ::= 'TupleType' '(' variableDeclarationList ')'
oclExpression ::= literalExp
oclExpression ::= '(' oclExpression ')'
oclExpression ::= pathName ['@' 'pre']
oclExpression ::= oclExpression '[' argumentList ']' ['@' 'pre']
oclExpression ::= oclExpression '.' simpleName ['@' 'pre']
oclExpression ::= oclExpression '− >' simpleName
oclExpression ::= oclExpression '(' ')'
oclExpression ::= oclExpression '(' oclExpression ')'
oclExpression ::= oclExpression '(' oclExpression ',' argumentList ')'
oclExpression ::= oclExpression '(' variableDeclaration '|' oclExpression ')'
oclExpression ::= oclExpression '(' oclExpression ',' variableDeclaration '|'
oclExpression ')'
oclExpression ::= oclExpression '(' oclExpression ':' type ','
variableDeclaration '|' oclExpression ')'
oclExpression ::= oclExpression '− >' 'iterate' '(' variableDeclaration [';'
variableDeclaration] '|' oclExpression ')'
oclExpression ::= 'not' oclExpression
oclExpression ::= '-' oclExpression
oclExpression ::= oclExpression '*' oclExpression
oclExpression ::= oclExpression '/' oclExpression
oclExpression ::= oclExpression 'div' oclExpression
oclExpression ::= oclExpression 'mod' oclExpression
oclExpression ::= oclExpression '+' oclExpression
oclExpression ::= oclExpression '-' oclExpression
oclExpression ::= 'if' oclExpression 'then' oclExpression 'else' oclExpression
'endif'
oclExpression ::= oclExpression '<' oclExpression
oclExpression ::= oclExpression '>' oclExpression
oclExpression ::= oclExpression '<=' oclExpression
oclExpression ::= oclExpression '>=' oclExpression

oclExpression ::= oclExpression '=' oclExpression
oclExpression ::= oclExpression '<>' oclExpression
oclExpression ::= oclExpression 'and' oclExpression
oclExpression ::= oclExpression 'or' oclExpression
oclExpression ::= oclExpression 'xor' oclExpression
oclExpression ::= oclExpression 'implies' oclExpression
oclExpression ::= 'let' variableDeclarationList 'in' oclExpression
oclExpression ::= oclExpression '^^ ' simpleName '('
[oclMessageArgumentList] ')'
oclExpression ::= oclExpression '^ ' simpleName '('
[oclMessageArgumentList] ')'
argumentList ::= oclExpression (',' oclExpression)*
oclMessageArgumentList ::= oclMessageArgument (',' oclMessageArgument
)*
oclMessageArgument ::= oclExpression
oclMessageArgument ::= '?' [':' type]
literalExp ::= collectionLiteralExp
literalExp ::= tupleLiteralExp
literalExp ::= primitiveLiteralExp
collectionLiteralExp ::= collectionKind '{' collectionLiteralParts '}'
collectionLiteralExp ::= collectionKind '{' '}'
collectionKind ::= 'Set' | 'Bag' | 'Sequence' | 'Collection' | 'OrderedSet'
collectionLiteralParts ::= collectionLiteralPart (',' collectionLiteralPart)*
collectionLiteralPart ::= oclExpression | collectionRange
collectionRange ::= oclExpression '..' oclExpression
tupleLiteralExp ::= 'Tuple' '' variableDeclarationList ''
primitiveLiteralExp ::= integer | real | string | 'true' | 'false'
pathname ::= simpleName | pathName '::' simpleName
integer ::= [0-9]+
real ::= integer[.]integer[eE][+-]?integer | integer[eE][+-]?integer |
integer[.]integer
string ::= ['][']*[']
simpleName ::= [a-zA-Z_][a-zA-Z0-9_]*

Operator Precedence

All operations are left associative and defined to have the following precedence (week to strong). These precedence's are required to remove the ambiguity in parsing the oclExpression non-terminal.

- '.'
- '^' and '^^'

- 'implies'
- 'and', 'or', and 'xor'
- $'='and'<>'$
- $'<','>', and'<=','>='$
- 'if', 'then', 'else', and 'endif'
- '+' and '-' (binary minus)
- '*', '/', 'div', and 'mod'
- 'not' and '-' (unary minus)
- $'.', and'->'$
- '@'
- '(' ')' and '[' ']'
- ':'