Computer Science at Kent

OCL 2.0: Implementing the Standard

David Akehurst, Peter Linington,
Octavian Patrascoiu

Technical Report No. 12-03
November 2003

Published by the Computing Laboratory,
University of Kent, Canterbury, Kent CT2 7NF, UK

OCL: Implementing the Standard

David Akehurst

Peter Linington

Octavian Patrascoiu

{D.H.Akehurst, P.F.Linington, O.Patrascoiu}@kent.ac.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 for 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, UML models and MDA [17] techniques supported by the Kent Modelling Framework (KMF) [12], developed at the University of Kent. We provide an efficient LALR grammar for parsing the language and describe an architecture that enables the language to be bridged to any other modelling framework or tool. We also provide both syntactic and semantic models, which were used as inputs for KMFStudio [12] in order to generate Java code. In addition we give feedback on problems and ambiguities discovered in the standard, with some suggested solutions.

1 Introduction/Motivation

The Object Constraint Language (OCL) [16] is a textual language, especially designed for use in the context of diagrammatic languages such as the UML. OCL was added to UML, as it turned out a visual diagram-based language is limited in its expressiveness. For instance, although the UML is powerful and covers many important situations, it is often not sufficient to describe certain important constraints. Using natural language on the one hand introduces ambiguities, due to freedom of interpretation and on the other hand there are no tools capable to cope with its complexity.

Hence, the Object Constraint Language was introduced as a textual add-on to the diagrams to cover the above aims. OCL is deeply connected to UML diagrams, as it is used as a textual addendum within the diagrams, e.g. to define pre- and post- conditions, invariants, or transition guards, but it also uses the elements defined in the UML diagram, such as classes, methods, and attributes.

The prime motivation of this work has been to provide support for OCL constraint checking over populations from a variety of models. This work has been done under the Kent Modelling Framework [12] project at the University of Kent, involving both the RWD [18] and DSE4DS [6] projects. Integration with the Eclipse framework was also supported by a grant from IBM.

2 Implementation Structure

The task of a translator is to transform a program written in one language, called the source language, into an equivalent program written in another language, called the target language. In order to achieve this goal, a translator must first determine and understand the structure and meaning of the source program, and then generate the equivalent representation using concepts from the target language. The first phase is called analysis and the second synthesis. Afterwards, the resulting program is executed on the target virtual machine, either executing the generated code or interpreting the generated representation.

The OCL implementation presented in this paper follows this typical structure of a language processor, consisting of an initial analysis phase, providing afterwards two options for execution: code generation or interpretation. This structure is described in Figure 1.

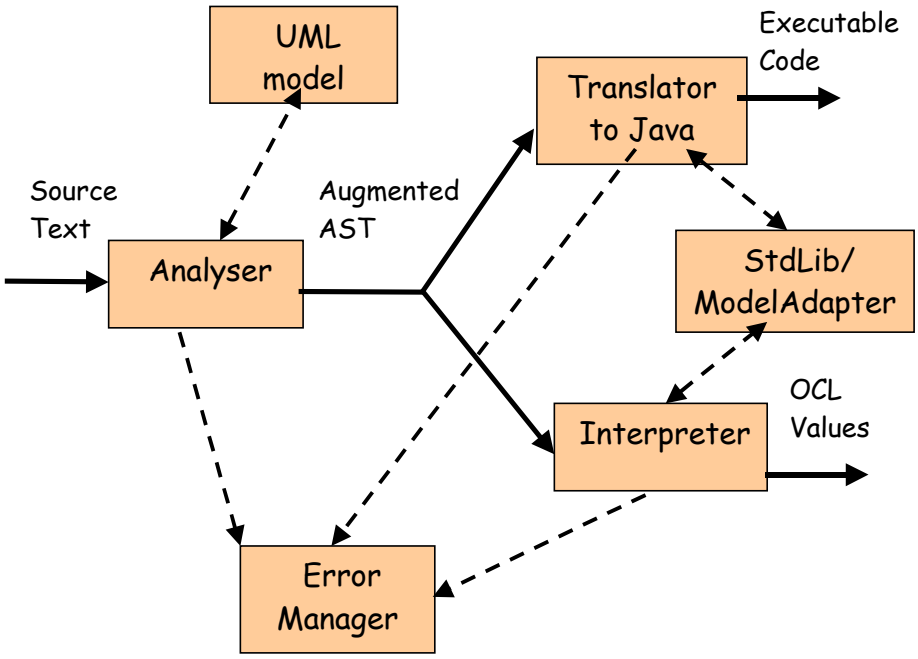


Figure 1 Implementation Structure

The analyser generates an internal description of the semantics of the input text, according to the information contained in the UML model. We chose to represent the semantics of the input using a classic augmented Abstract Syntax Tree (AST). The Translator to Java and the Interpreter use the internal representation of the semantics in order to perform the corresponding task: generation of executable code and evaluation. Framework and modelling tools details are implemented using a model description, a bridge, and an implementation adapter. This approach increases the portability of the implementation.

Each of these stages has involved different problems relating to the specification contained in the OCL standard. We discuss each stage separately in the following sections.

3 Analysis

In order to analysis the input text, a translator must perform the following three steps:

1) Lexical Analysis: the input program is broken into basic symbols or tokens
2) Syntax Analysis: construct the phrase structure of the program
3) Semantic Analysis: compute the meaning of the program

Our implementation follows this approach. The structure of our analyser, including the dataflow and the dependencies between the modules, is presented in Figure 2.

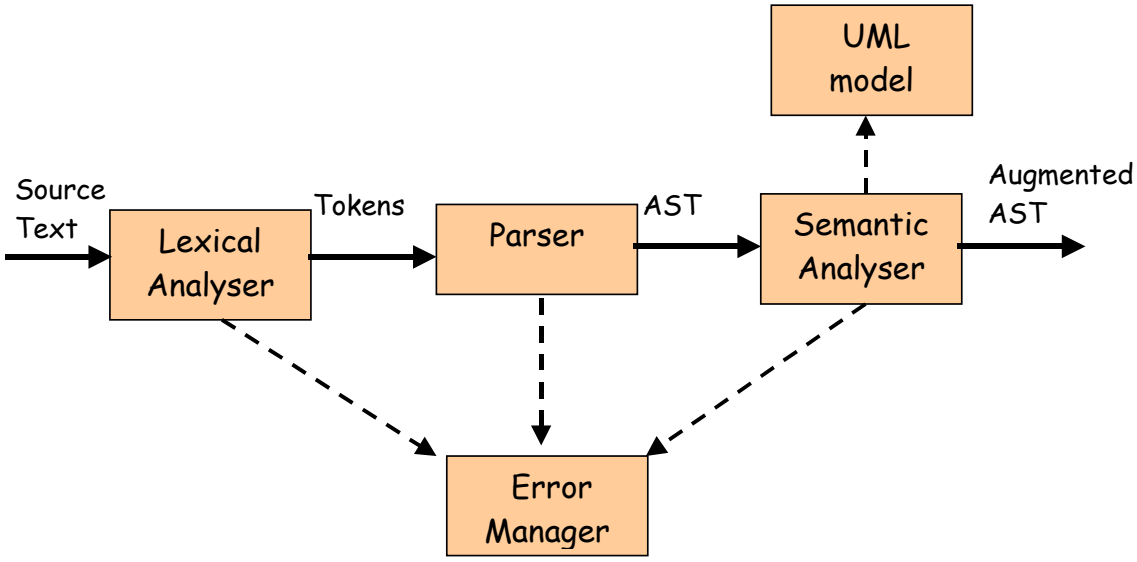


Figure 2 Structure of the analyser

3.1 Lexical Analysis

The lexical analyser transforms the source program, seen as a sequence of characters, into a sequence of symbols with a semantic meaning. These symbols, together with their encoding, form the intermediate language generated by the lexical analyser.

In order to separate the lexical analyses from the rest of the translator, the grammar G of the OCL language was partitioned on two levels: the first level contains the OCL grammars, and the second level contains the sub-grammars G_1, …, G_n, associated to each token. The purpose of this partitioning operation is to allow the OCL grammar G to describe the language using basic symbols. The language L(G) is obtained by replacing the terminal symbols of grammar G with strings from $L(G_1)$, …, $L(G_n)$.

According to [20], the construction of a lexical analyser should be done following the steps:

1) Define the basic symbols of OCL, partition the grammar of OCL and determine the rules associated to the basic symbols.

2) Decide how the basic symbols will be recognized, extracted and represented.
3) Decide how to report the lexical errors.
4) Design and implement the finite state machine for each basic symbol, and then design the overall finite state transducer.
5) Implement the lexical analyser.

Defining the basic symbols

After performing steps 1 and 2, we have obtained the grammars for the basic tokens: *name*, *integer*, *string*, *keyword*, *specialSign*, and *comment.* The description of these grammars, according with the rules from Annex A, is given below:

name → letter alpha* .

integer → digit+ .

real → integer '.' integer .

real → integer ['e' | 'E']['+' | '-'] integer

real → integer '.' integer ['e' | 'E']['+' | '-'] integer .

keyword →

'package' | 'endpackage' | 'context' | 'init' | 'derive' | 'inv' | 'def' | 'pre' | 'post' | 'body' |
'implies' | 'and' | 'or' | 'not' | 'true' | 'false' | 'xor' |
'div' | 'mod' |
'Collection' | 'Bag' | 'Set' | 'Sequence' | 'OrderedSet' | 'TupleType' | 'Type' |
'if' | 'then' | 'else' | 'endif' |
'let' | 'in' | 'iterate' .

specialSign →

'..' | '::' | ':' |
'(' | ')' | '[' | ']' | '{' | '}' |
',' | ';' | '|' | '@' | '?' |
'=' | '<>' | '<=' | '>=' | '<' | '>' |
'+' | '-' | '*' | '/' | '.' | '->' |
'^^' | '^' .

comment → lineComment | paragraphComment .

lineComment → '-''-' (any character except a line terminator)* lineTerminator .

paragraphComment → '/*' (any character sequence except the sequence */) '*/' .

lineTerminator →'\r' | '\n' | '\r\n' | '\n\r' .

whiteSpace → '\t' | '\f' | '\n' | '\r' .

letter → 'a' | … | 'z' | 'A' | … | 'Z' | '_' .

digit → '0' | … | '9' .

alpha → letter | digit .

Beside the above grammar we have to specify the set of allowed characters. We consider that the allowed set of characters in OCL is the Unicode set.

Error handling

Every time a lexical error occurs, the lexical analyser will pass a message to the Error Manager, which is responsible for reporting errors (see Figure 2). The error manager considers the following cases:

- Input contains an illegal character
- A lexical rules is broken

If the input string contains an illegal character, the lexical analyser removes the illegal character from the input and invokes the error manager in order to report an error and returns a bad token to the context. If a lexical rule is broken, the lexical analyser invokes the error manager in order to report an error and returns a bad token to the context.

Lexical Analyser Construction

Usually there are two possibilities to implement the lexical analyser:

- Write the code by hand
- Use a lexical analyser generator

Our OCL implementation uses a lexical analyser generator. Usually the lexical analyser is either standalone or integrated into a parser. For reasons that we will explain latter, we decide to use JFlex, a lexical analyzer generator for Java. The input for JFlex is specified in Annex B.

3.2 Syntax Analysis

Firstly, a syntax analyser must recognize whether the input program belongs to the language of a grammar. Secondly, it must represent the input program in order to provide accurate information to later phases, like the semantic analysis.

In order to build a syntax analyser for a given language, one must follow the steps:

- Write an ambiguous grammar for language, usually a LL(1) or LALR(1) grammar.
- Build the corresponding parser, manually or by using a parser generator.
- Choose the intermediate language to represent the structure of the input
- Design the semantic actions to build the internal representation of a given source program

Choosing the grammar type and the parser generator

We decided to use a LALR(1) grammar for several reasons:

- The weakness of LL(k) parsing techniques is that they must predict which production to use, having seen only the first k tokens of the right-hand side.

- As LR(k) parsing technique is more powerful, it is able to postpone the decision until it has seen input tokens corresponding to the entire right-hand side of the production in question (and k more input tokens beyond).
- Any reasonable programming language has a LALR(1) grammar, and there are many parser-generator tools available for LALR(1) grammar.
- For this reason LALR(1) has become a standard for programming languages and for automatic parser generators e.g. Flex/CUP and SableCC.

The LR grammar of OCL, which uses OCL tokens as terminal symbols, is described below according to the rules from Annex A:

```
packageDeclaration → 'package' pathName contextDeclList 'endpackage'.
packageDeclaration → contextDeclList .
contextDeclList → contextDeclaration* .
contextDeclaration → propertyContextDecl .
contextDeclaration → classifierContextDecl .
contextDeclaration → operationContextDecl .
propertycontext'Decl → 'context' pathName simpleName ':' type initOrDerValue+ .
initOrDerValue → 'init' ':' oclExpression | 'derive' ':' oclExpression
classifiercontext'Decl → 'context' pathName invOrDef+ .
invOrDef → 'inv' [simpleName] ':' oclExpression .
invOrDef → 'def' [simpleName] ':' defExpression .
defExpression → simpleName ':' type '=' oclExpression .
defExpression → operation '=' oclExpression .
operation'context'Decl → 'context' operation prePostOrBodyDecl+ .
prePostOrBodyDecl → 'pre' [simpleName] ':' oclExpression .
prePostOrBodyDecl → 'post' [simpleName] ':' oclExpression .
prePostOrBodyDecl → 'body' [simpleName] ':' oclExpression .
operation → pathName '(' [variableDeclarationList] ')' [':' type]
variableDeclarationList → variableDeclarationList (',' variableDeclaration)* .
variableDeclaration → simpleName [':' type] ['=' oclExpression]
type → pathName | collectionType | tupleType .
collectionType → collectionKind '(' type ')' .
tupleType → 'TupleType' '(' variableDeclarationList ')' .
oclExpression →
   literalExp |
   '(' oclExpression ')' |
   pathName isMarkedPre  |
   oclExpression DOT simpleName isMarkedPre  |
   oclExpression '->' simpleName  |
   oclExpression '(' ')'  |
   oclExpression '(' oclExpression ')'  |
   oclExpression '(' oclExpression ',' argumentList ')'  |
   oclExpression '(' variableDeclaration '|' oclExpression ')'  |
   oclExpression '(' oclExpression ',' variableDeclaration '|' oclExpression ')'  |
   oclExpression '(' oclExpression ':' type ',' variableDeclaration '|' oclExpression ')'  |
   oclExpression '[' argumentList ']' isMarkedPre  |
   oclExpression '->' 'iterate' '(' variableDeclaration [';' variableDeclaration] '|'
oclExpression ')'  |
   'not' oclExpression  |
   '-' oclExpression  |
   oclExpression '*' oclExpression  |
   oclExpression '/' oclExpression  |
   oclExpression 'div' oclExpression  |
   oclExpression 'mod' 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 'and' oclExpression  |
   oclExpression 'or' oclExpression  |
   oclExpression 'xor' oclExpression  |
   oclExpression 'implies' oclExpression  |
   'let' variableDeclarationList 'in' oclExpression  |
   oclExpression '^^' simpleName '(' [oclMessageArgumentList] ')'  |
   oclExpression '^' simpleName '(' [oclMessageArgumentList] ')' .
argumentList → oclExpression (',' oclExpression)* .
oclMessageArgumentList → oclMessageArgument (',' oclMessageArgument)* .
oclMessageArgument → oclExpression | '?' [':' type] .
isMarkedPre → ['@' 'pre'] .
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 .
primitiveLiteralExp → real .
primitiveLiteralExp → string .
primitiveLiteralExp → 'true' .
primitiveLiteralExp → 'false' .
pathName → simpleName .
pathName → pathName '::' simpleName .
```

Rationale for Using ASTs

As programming languages become more and more complex, one-pass compilers, which do not produce any intermediate form of the compiled code, are becoming very rare. This is also true in the area of program understanding and software engineering [19]. This has led to a variety of intermediate representations of the syntactic structure of source code.

[13] contains a survey of common program representations, such as Abstract Syntax Trees (AST), Directed Acyclic Graphs (DAG), Prolog rules, code and concepts object, and control and data flow graphs. Among these, the AST is the representation that is most advantageous, and is most widely used by software developers. The reason for the popularity of the AST as a program representation scheme is its ability to capture all the necessary information for performing any operations with respect to data flow and control flow source code properties.

To justify the AST representation, we need to distinguish between a parse tree, or derivation tree, and an AST. A parser explicitly or implicitly uses a parse tree to represent the structure of the source program [2]. An AST is a compressed representation of the parse tree where the operators appear as nodes, and operands of the operator are the children of the node representing the operator.

In Annex C we present the syntax model of OCL using UML diagrams.

Designing the Semantic Actions

Formal mechanisms used to describe the translation from one language to another are based on the following basic idea: the information contained in a syntax construction is specified by using attributes attached to grammar symbols, and the values for these attributes are computed using semantic rules attached to grammar rules.

There are two notations to attach the semantic rules to grammar rules: Syntax Driven Definitions (SDD) and Syntax Driven Translation Scheme (SDTS). The definitions are high-level specifications of the translation. They hide a lot of the implementation details and do not specify the order in which the semantic rules should be executed. The translation schemes specify the order in which the semantic rules should be evaluated, therefore contain more details about the implementation.

The STDS, which has been used to translate an OCL string into instances of elements from the OCL syntax model, is described in Annex D. The translation scheme was designed so that it can be used as input for CUP, a parser generator for Java that works with JFlex.

3.3 Semantic Analysis

The semantic analysis phase of our implementation connects variable definitions to their uses, checks that each expression has a correct type, and translates the abstract syntax into a simpler representation suitable for generating machine code or interpretation. This phase is characterized by the maintenance of symbol tables (also called an environment), mapping identifiers to their types and locations. As the declarations of variables, type expressions, and expressions are processed, these identifiers are bound to *meanings* in the symbol tables. When *uses* of identifiers are found, they are looked up in the symbol tables.

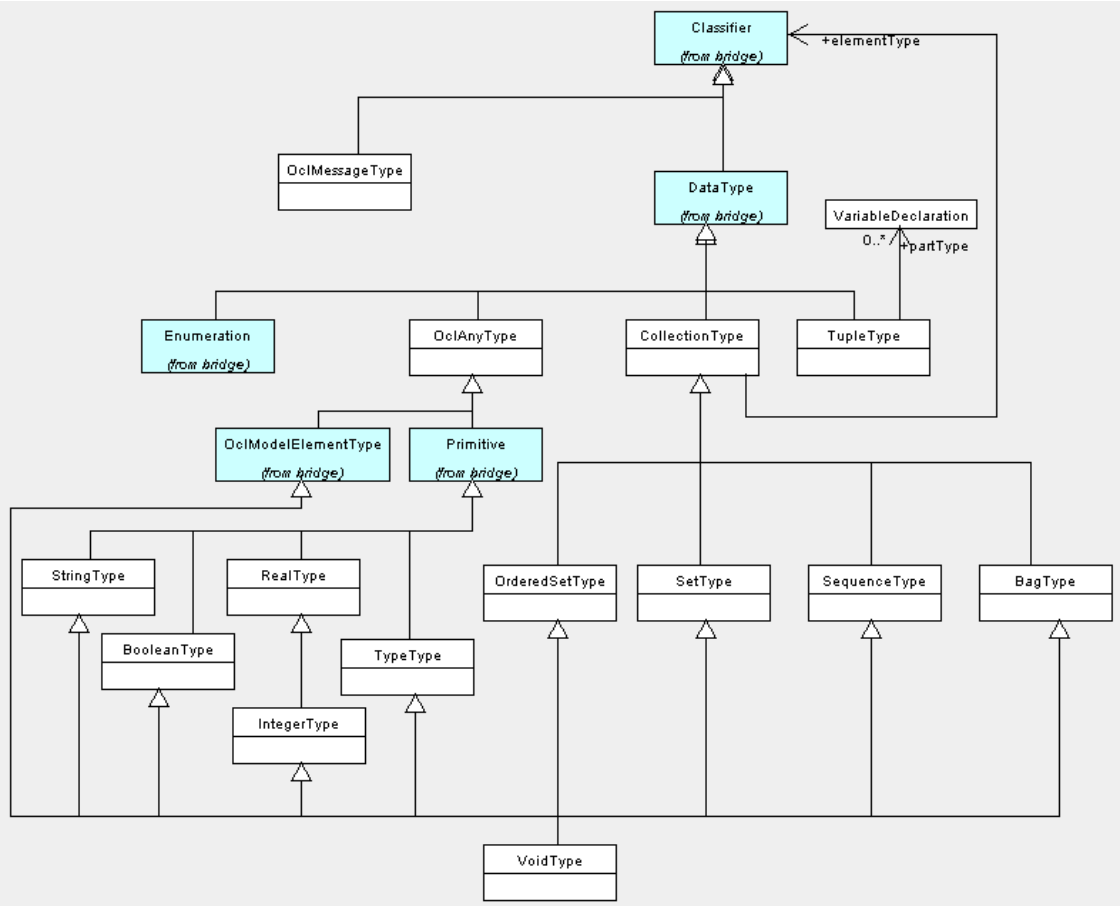


Figure 3 OCL Types

Types

OCL is a typed language and the basic value types are organized in a type hierarchy. This hierarchy determines the relation between different types from the OCL's type system. In OCL, a number of basic types are predefined and available to the modeller at all times. These predefined value types are independent of any object model and part of the definition of OCL. The basic types are Boolean, Integer, Real and String. OCL also contains collection types: Collection(T), Bag(T), OrderedSet(T), Sequence(T), and Set(T) where T is a type. The OCL type hierarchy that we used is specified in Figure 3. Differences between this hierarchy and the one from the OCL standard specification are explored in a following section.

Bridge

Due to the fact that the OCL language is tied to UML, the OCL model specified in [16] can be divided into two sets of elements:

1) Those that define the OCL concepts.
2) Those that refer to concepts from the UML metamodel.

The concepts from 1) we further divide into those relating to OCL's static semantics (e.g. OCL types) and those dealing with OCL's dynamic semantics (e.g. OCL values).

The classes from 2) are distinguished in the standard [16] by the fact that they come from various packages in the UML metamodel and they are coloured white as opposed to grey.

We choose to implement the interaction between the semantic analyser and the UML model by using a bridge to separate the abstractions and the implementations of the model dependent actions. We redefine classes from 2) 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 from the UML model, rather than directly being classes from the UML model.

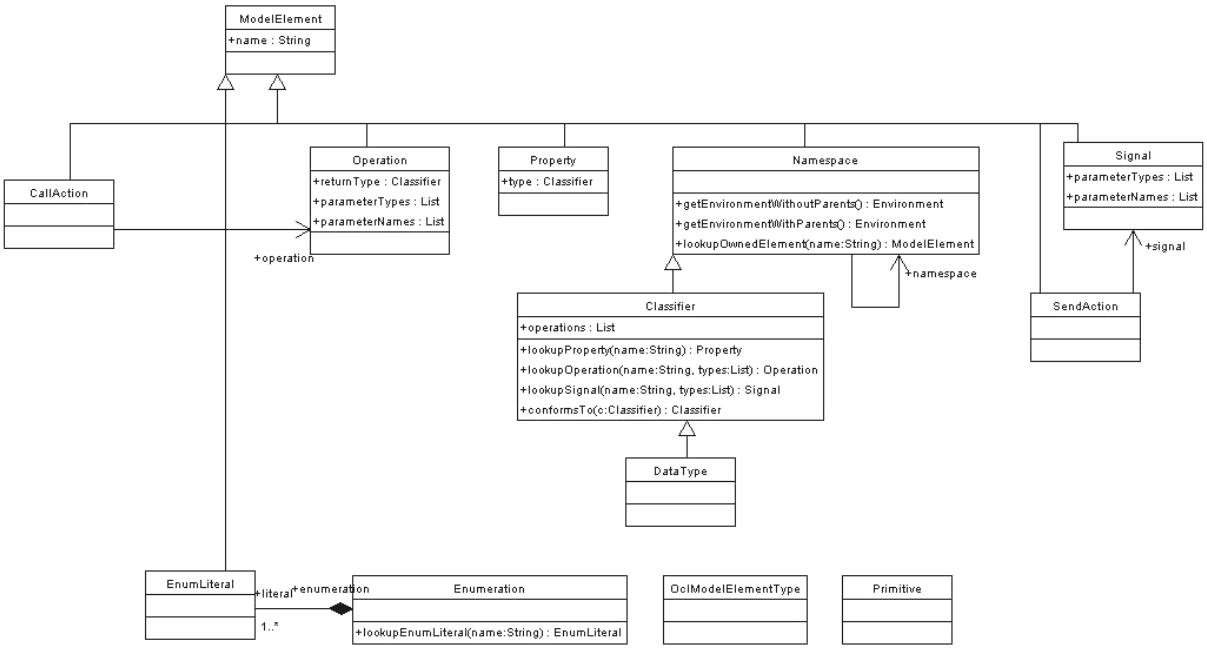


Figure 4 Bridge

This approach offers the following advantages:

- The implementation of an abstraction can be configured or changed at run-time.
- The abstraction and implementation hierarchies can be extended independently.
- The clients can be shielded from implementation details.

As a result, the portability of the OCL processor to different modelling frameworks and tools has increased.

The classes in the bridge package (Figure 4) 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 [11]. 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.

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.

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.

OCL for EMF

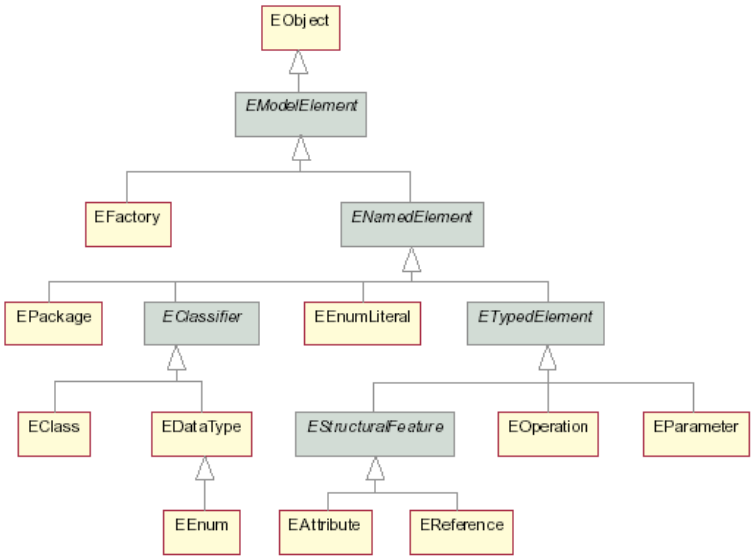


Figure 5 ECore model (taken from EMF overview)

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 5); 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.

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 a 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.

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 collection 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 reflective 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; another is to add a static final field named with the name of the property/operation + "_elementType" whose type is the element type of the collection, when a property or operation has a collection as its return type. Reflection operations are used to look up this field when needed.

Type-checking

According to [16] the type conformance rules for types in the class diagram are simple:

- Type conformance is reflexive: Type1 conforms to Type1
- Type conformance is antisymmetric: if Type1 conforms to Type2 and Type2 conforms to Type1 then Type1 is identical to Type2
- Type conformance is transitive: if Type1 conforms to Type2, and Type2 conforms to Type3, then Type1 conforms to Type3

- Each type conforms to each of its supertypes
- Integer conforms to Real
- The types Set(T), Bag(T), OrderedSet(T), Sequence(T) and Set(T) are all subtypes of Collection(T)
- Collection(Type1) conforms to Collection(Type2), when Type1 conforms to Type2

In order to implement the type-checking process, the abstract syntax tree is augmented with new attributes to allow the computation of expression types and type conformance checking:

- **Synthesized attributes**. Each node from the abstract syntax tree has one inherited attribute called *type*, which holds the type of that expression.
- **Inherited attributes**. Each production rule has one inherited attribute called *env* (short for environment), which holds a list of names that are visible from the expression. All names are references to elements in the model. In fact, *env* is a name space environment for the expression or expression part denoted by the current node.

Type checking can be done in several ways. We decided to use the visitor pattern over the abstract syntax tree, which was built by the previous phase. The type-checking visitor scans bottom-up the augmented tree and computes the *type* attribute for each node using predefined rules and the *env* attribute.

This approach increases the values for the following software quality attributes:

- **Understandability**: better understanding of the logical concepts and the way they apply;
- **Learnability**: faster learning of the application
- **Analysability**: reduces the effort required to diagnose the failures and crash causes; reduces the effort to identify the parts that must be changed in case of a failure
- **Changeability**: reducing the effort required to modify the source code
- **Testability**: reducing the effort to validate the software
- **Reusability**: increases the reusability of the code

4 Synthesis

This sections contains the descriptions of the standard library, code generation and interpretation.

4.1 Standard Library

In order to support the evaluation of OCL expressions, we provided a library, which contains OCL values: strings, numbers, model elements, enumerations etc. These classes were extracted from [13] selecting all the features that are related to the OCL's dynamic semantics.

4.2 Code Generation and Interpretation

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.

As an example, for the OCL expression

```
context library::Library inv: self.books->asSequence()->first().author
```

the following Java code is generated or interpreted using reflection:

```
try {
        // Call property 'books'
        OclSet t17 = StdLibAdapterImpl.INSTANCE.Set(self.getBooks());
        // Call operation 'asSequence'
        OclSequence t16 = (OclSequence)t17.asSequence();
        // Call operation 'first'
        library.Book t15 = (library.Book)t16.first();
        // Call property 'author'
        library.Author t14 = (library.Author)t15.getAuthor();
        // return result
        if (t14 != null) return t14;
        else return StdLibAdapterImpl.INSTANCE.Undefined();
} catch (Exception e) {
        return StdLibAdapterImpl.INSTANCE.Undefined();
}
```

5 OCL Issues

Annex E contains some observation regarding the way OCL was designed and specified in [16]. The observations are both at a syntactic and a semantic level.

6 Related Work

There are many CASE tools supporting the drawing of UML diagrams and features like code generation and reversing 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 provide 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 analyzer, which should report as many errors as possible in order to help the user to develop solid OCL code. If the tool provides both an interpreter and a compiler, the user has the possibility to choose the best approach in order to obtain 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 [16]. Another toolset was developed at TU Dresden [7]. A part of this tool has been integrated with the open source CASE Tool Argo [4]. [10] contains a description of an OCL interpreter. It is based partly on an OCL meta-model describing the abstract syntax of OCL. [15] also provides a good implementation for OCL.

7 Conclusions

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.

7.1 Unsupported Concepts

Our implementation currently does not fully support the following constructs

- ^ and ^^ hasSent and message Operators
- contexts, other than inv:
- OclState, OclMessage types

@pre

References

[1] Aho A. V., J. Ullman, The Theory of Parsing, Translation and Compiling, Prentice-Hall, 1972.

[2] Aho A. V., Sethi R., Ullman J. D., Compilers, Principles, Techniques, and Tools, Addison-Wesley, 1986.

[3] Appel A. V., Modern Compiler Construction Implementation in Java, second edition, Cambridge University Press, 2002.

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

[5] Cooper J. W., Java Design Patterns. A Tutorial, Addison-Wesley, 2000.

[6] Design Support for Distributed Systems http://www.cs.kent.ac.uk/projects/dse4ds/index.html

[7] 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.

[8] Erdogmus H., O. Tanir, Advances in Software Engineering. Comprehension, Evaluation, and Evolution, Springer Verlag, 2002.

[9] Gamma E., Helm R., Johnson R., Vlissides J., Design Patterns, Addison-Wesley, 1995.

[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.

[11] International Business Machines Eclipse Modelling Framework http://www.eclipse.org/emf

[12] Kent Modelling Framework, http://www.cs.kent.ac.uk/projects/kmf

[13] Kontogiannis K., Program Representation and Behavioural Matching for Localizing Similar Code Fragments, In Proceedings of CASCON'93, Toronto, Ontario, Canada, October.

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

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

[16] Object Management Group, Object Constraint Language Specification Revised Submission, Version 1.6, January 6, 2003, OMG document ad/2003-01-07.

[17] Object Management Group, Model Driven Architecture (MDA) http://www.omg.org/mda

[18] Reasoning with Diagrams, http://www.cs.kent.ac.uk/projects/rwd

[19] Tilley S., D. Smith, Coming Attractions in Program Understanding, Technical Report CMU/SEI-96-TR-019 ESC-TR-96-019, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, December.

[20] Waite W., Goos G., Compiler Construction, Springer-Verlag, 1984.

Annex A. Grammar Specification Rules

Grammar specification is done using the following rules:

1) Left hand-side and right hand-side are separated by symbol $\rightarrow$.
2) Each production ends with a dot.
3) Terminal symbols are written using capital letter or delimited by apostrophes.
4) The following shortcuts are permitted:

SHORTCUT	MEANING
$x \rightarrow \alpha (\beta) \gamma .$	$x \rightarrow \alpha Y \gamma . Y \rightarrow \beta .$
$x \rightarrow \alpha [\beta] \gamma .$	$x \rightarrow \alpha \gamma \mid \alpha (\beta) \gamma .$
$x \rightarrow \alpha u + \gamma .$	$x \rightarrow \alpha Y \gamma . Y \rightarrow u \mid u Y .$
$x \rightarrow \alpha u * \gamma .$	$x \rightarrow \alpha Y \gamma . Y \rightarrow u \mid u Y \mid \lambda .$
$x \rightarrow \alpha \mid\mid a.$	$x \rightarrow \alpha (a \alpha) * .$

where α, β and γ strings over the language alphabet, *Y* is a symbol which does not appear elsewhere in the specification, *u* is either a unique symbol or an expression delimited by parentheses, and *a* is a terminal symbol.

Annex B. Flex Input

```
// Usercode Section
package uk.ac.kent.cs.ocl20.syntax.parser;
import java_cup.runtime.*;
import uk.ac.kent.cs.kmf.util.*;

%%
// Options Sections
%unicode
%cup
%line
%column

// Declarations Section
%{
  // Debug flag
  public static boolean lexDebug = false;
  protected void debug(int type) {
    if (lexDebug) {
       log.reportMessage(
          yyline+":"+yycolumn+" Token "+type+" '"+yytext()+"'");
    }
  }
  // Output log
  protected ILog log;
  public void setLog(ILog log) {
    this.log = log;
  }

  // Create a new Symbol with information about the current token
    protected Symbol symbol(int type) {
      debug(type);
      return new Symbol(type, yyline, yycolumn, new String(yytext()));
    }
    protected Symbol symbol(int type, Object value) {
      debug(type);
      return new Symbol(type, yyline, yycolumn, value);
    }
%}

%eofval{
  return symbol(sym.EOF);
%eofval}


// Macro declarations
lineTerminator     = \r|\n|\r\n|\n\r
whiteSpace         = [ \t\f\n\r]
comment        = {paragraphComment} | {lineComment}
paragraphComment  = "/*" ~"*/"
lineComment      = "--" ~{lineTerminator}

lowerCase      = [a-z]
upperCase      = [A-Z]
digit        = [0-9]
letter        = {lowerCase} | {upperCase} | [_]
alpha        = {letter} | {digit}
integer        = {digit}+
real        = {integer}"\."{integer}[eE][+-]?{integer} |
                           {integer}[eE][+-]?{integer} | {integer}"\."{integer}
string        = "'" ~"'"
simpleName      = {letter}{alpha}*

%%
// Lexical Rules Section

<YYINITIAL> {
  {whiteSpace}    { /* just skip what was found, do nothing */ }
  {comment}      { /* just skip what was found, do nothing */ }
```

```
".."          { return symbol(sym.DOT_DOT); }
"::"          { return symbol(sym.COLON_COLON); }
"^^"          { return symbol(sym.UP_UP); }
"."           { return symbol(sym.DOT); }
":"           { return symbol(sym.COLON); }
"^"           { return symbol(sym.UP); }

"("           { return symbol(sym.LEFT_PAR); }
"["           { return symbol(sym.LEFT_BRK); }
"{"           { return symbol(sym.LEFT_BRA); }
")"           { return symbol(sym.RIGHT_PAR); }
"]"           { return symbol(sym.RIGHT_BRK); }
"}"           { return symbol(sym.RIGHT_BRA); }

","           { return symbol(sym.COMMA); }
";"           { return symbol(sym.SEMICOLON); }
"|"           { return symbol(sym.BAR); }
"@"           { return symbol(sym.AT); }
"?"           { return symbol(sym.QUESTION); }

"="           { return symbol(sym.EQ); }
"<>"          { return symbol(sym.NE); }

"<="          { return symbol(sym.LE); }
">="          { return symbol(sym.GE); }
"<"           { return symbol(sym.LT); }
">"           { return symbol(sym.GT); }

"+"           { return symbol(sym.PLUS); }
"->"          { return symbol(sym.MINUS_GT); }
"-"           { return symbol(sym.MINUS); }

"*"           { return symbol(sym.TIMES); }
"/"           { return symbol(sym.DIVIDE); }

"package"       { return symbol(sym.PACKAGE); }
"endpackage"    { return symbol(sym.ENDPACKAGE); }
"context"       { return symbol(sym.CONTEXT); }
"init"          { return symbol(sym.INIT); }
"derive"        { return symbol(sym.DERIVE); }
"inv"           { return symbol(sym.INV); }
"def"           { return symbol(sym.DEF); }
"pre"           { return symbol(sym.PRE); }
"post"          { return symbol(sym.POST); }
"body"          { return symbol(sym.BODY); }

"Set"           { return symbol(sym.SET); }
"Bag"           { return symbol(sym.BAG); }
"Sequence"      { return symbol(sym.SEQUENCE); }
"Collection"    { return symbol(sym.COLLECTION); }
"OrderedSet"    { return symbol(sym.ORDERED_SET); }
"TupleType"     { return symbol(sym.TUPLE_TYPE); }
"Tuple"         { return symbol(sym.TUPLE); }

"if"            { return symbol(sym.IF); }
"then"          { return symbol(sym.THEN); }
"else"          { return symbol(sym.ELSE); }
"endif"         { return symbol(sym.ENDIF); }

"let"          { return symbol(sym.LET); }
"in"           { return symbol(sym.IN); }
"iterate"      { return symbol(sym.ITERATE); }

"implies"      { return symbol(sym.IMPLIES); }
"and"          { return symbol(sym.AND); }
"or"           { return symbol(sym.OR); }
"xor"          { return symbol(sym.XOR); }
"not"          { return symbol(sym.NOT); }
"true"         { return symbol(sym.TRUE); }
"false"        { return symbol(sym.FALSE); }

"div"          { return symbol(sym.INT_DIVIDE); }
"mod"          { return symbol(sym.INT_MOD); }
```

```
  {real}         { return symbol(sym.REAL); }
  {integer}      { return symbol(sym.INTEGER); }

  {simpleName}   { return symbol(sym.SIMPLE_NAME); }
  {string}       { return symbol(sym.STRING); }
}

[^]              { log.reportError("Illegal character '"+yytext()+"'");
                   return symbol(sym.BAD);}
```

Annex C. OCL Syntax Model

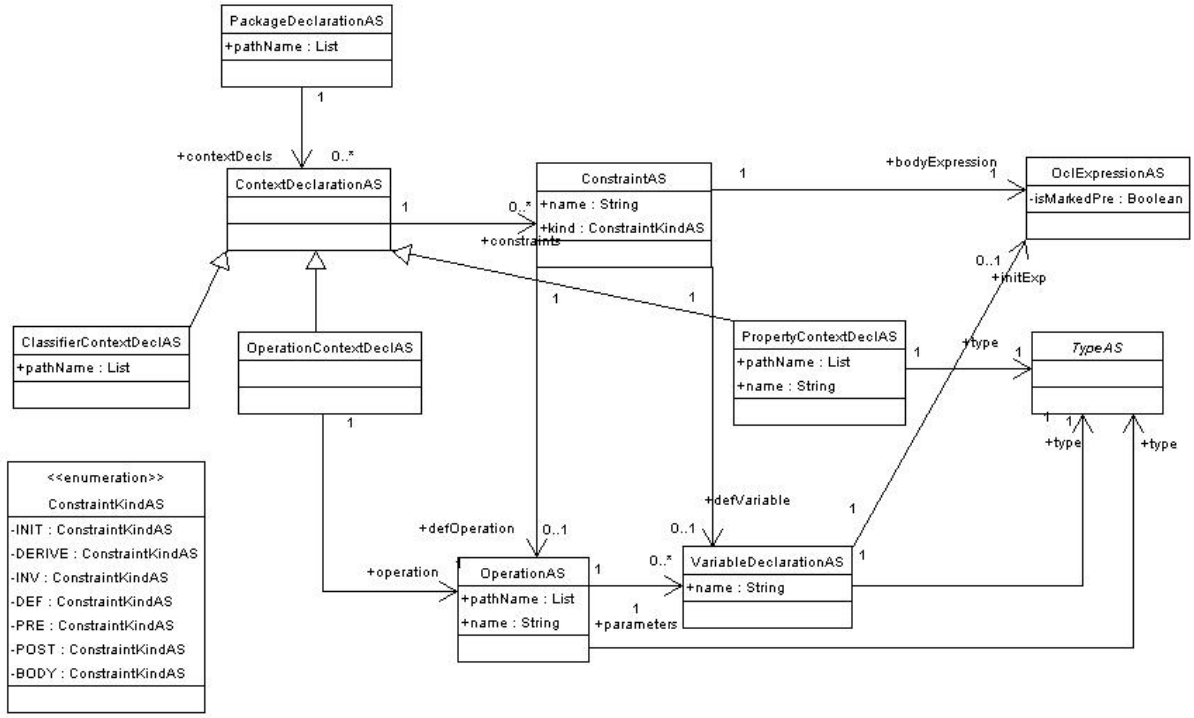


Figure 6 Syntax contexts

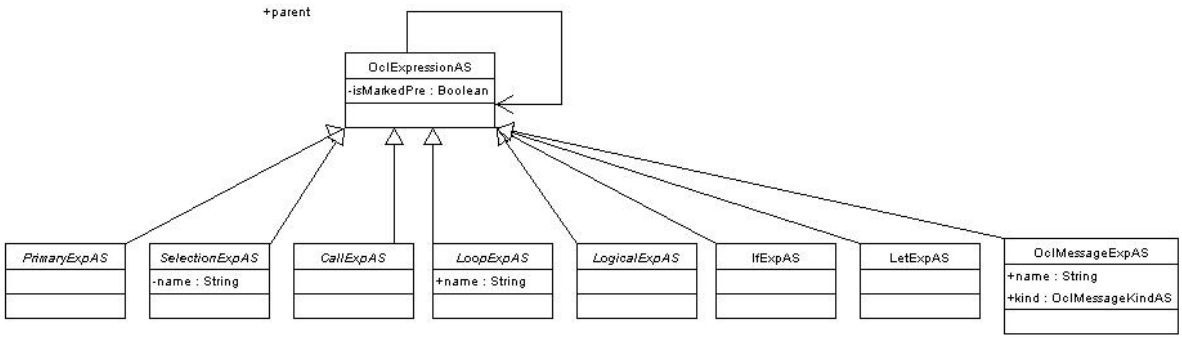


Figure 7 Expressions

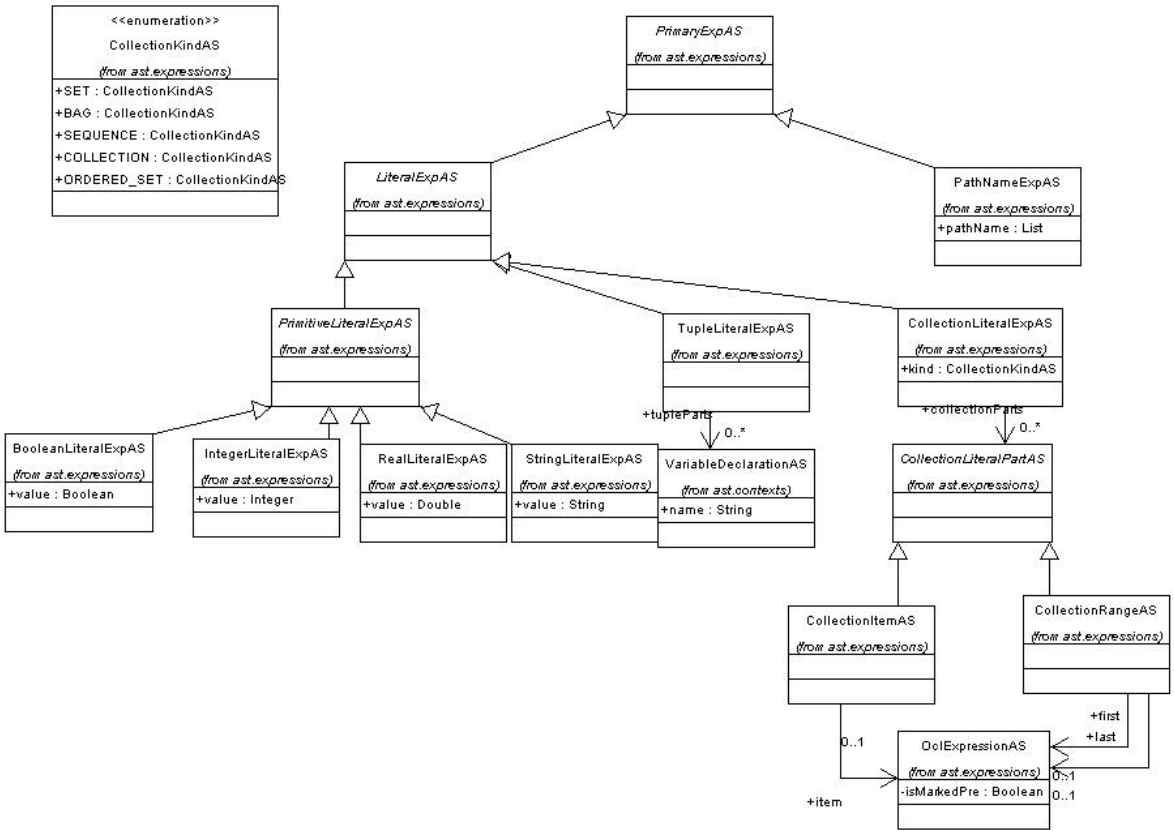


Figure 8 Primary Expressions

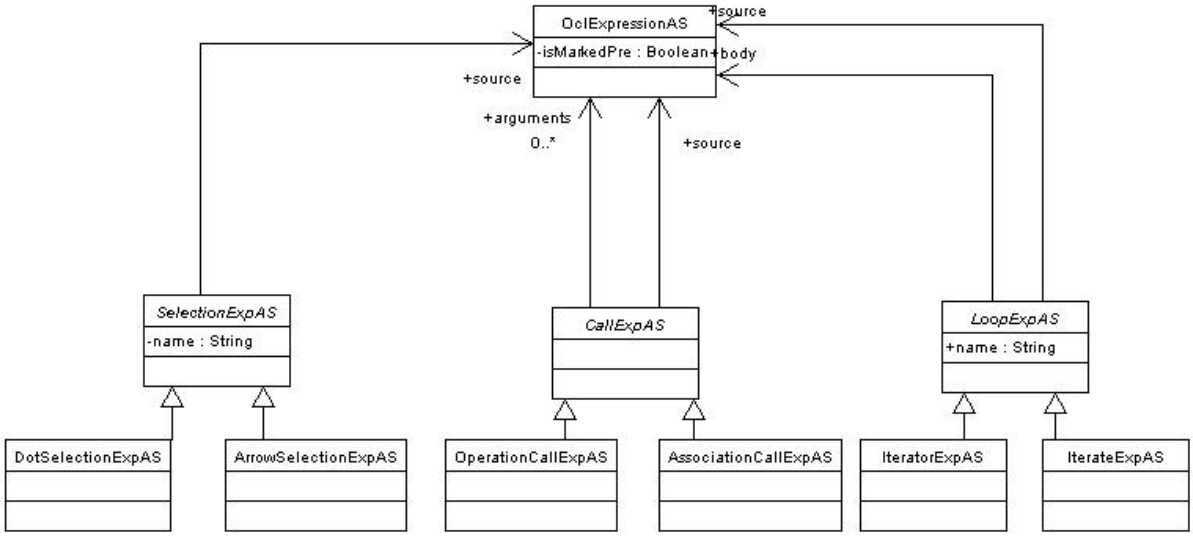


Figure 9 Selection, Call, and Loop Expressions

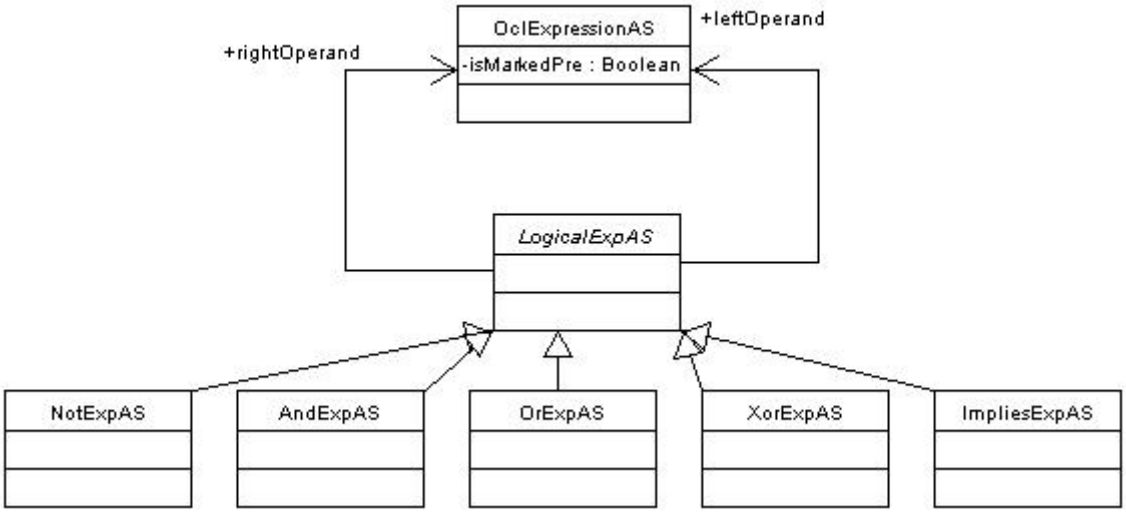


Figure 10 Logical Expressions

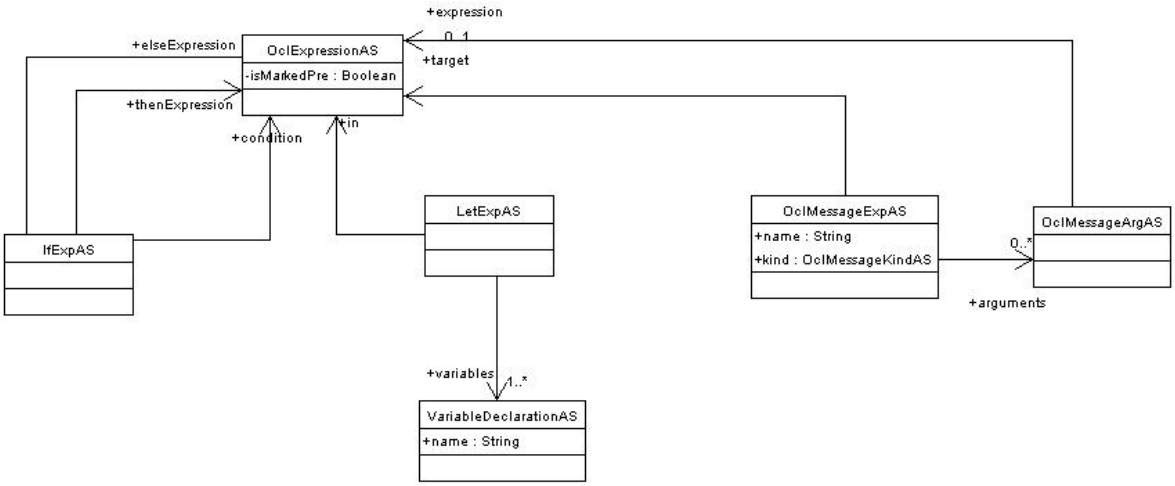


Figure 11 If, Let, Message Expressions

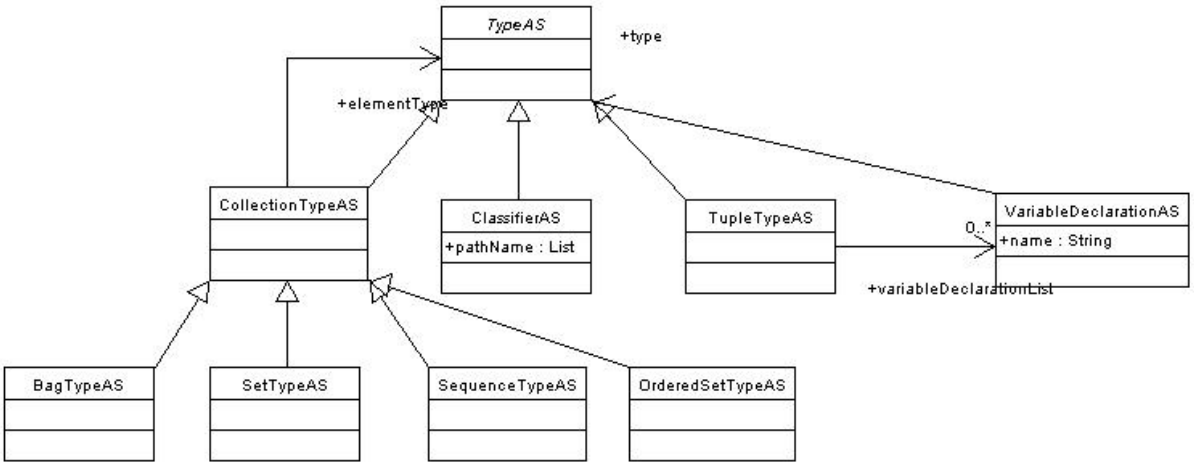


Figure 12 Types

Annex D. SDTS to Build the AST

This section contains the description of the SDTS which is being use to build ASTs from OCL inputs. The SDTS is described using the rules from Annex A and the following extra rules:

5) Semantic actions are identified using % followed by a number.
6) Semantic rule identifiers are local to a production.
7) The place in which a semantic action is invoked is described by inserting the rule in the right hand side of a production.
8) The body of a semantic action is described after the production, using Java code.

```
packageDeclaration →  PACKAGE pathName:path contextDeclList:contextDecls ENDPACKAGE %1
                      |
                      contextDeclList:contextDecls %2 .
%1 = {
// Create a PackageDeclaration
        RESULT = factory.buildPackageDeclaration(path, contextDecls);
}
%2 = {
        // Create a PackageDeclaration
        RESULT = factory.buildPackageDeclaration(new Vector(), contextDecls);
}

contextDeclList → % 1
                  |
                  contextDeclList:list contextDeclaration:contextDecl %2.
%1 = {
        // Create a LIST
        RESULT = new Vector();
}
%2 = {
        // Add element to list
        RESULT = list;
        RESULT.add(contextDecl);
:}

contextDeclaration →  propertyContextDecl:contextDecl %1
                      |
                      classifierContextDecl:contextDecl %2
                      |
                      operationContextDecl:contextDecl %3 .
%1 = {
        // Copy rule
        RESULT = contextDecl;
}
%2 = {
// Copy rule
        RESULT = contextDecl;
:}
%3 = {:
        // Copy rule
        RESULT = contextDecl;
:}

propertyContextDecl → CONTEXT pathName:path simpleName:name COLON type:type
initOrDerValue:constraints %1 .
%1 = {:
        // Create PropertyContextDecl
        RESULT = factory.buildPropertyContextDeclaration(path, name, type, constraints);
:}

initOrDerValue    →   INIT COLON oclExpression:exp %1
```

```
                    |
                    DERIVE COLON oclExpression:exp %2
                    |
                    initOrDerValue:list INIT COLON oclExpression:exp %3
                    |
                    initOrDerValue:list DERIVE COLON oclExpression:exp %4 .
%1 = {:
        // Create a LIST and add constraint
        RESULT = new Vector();
        RESULT.add(factory.buildConstraint(ConstraintKindAS$Class.INIT, "", exp, null));
:}
%2 = {:
        // Create a LIST and add constraint
        RESULT = new Vector();
        RESULT.add(factory.buildConstraint(ConstraintKindAS$Class.DERIVE, "", exp, null));
:}
%3 = {:
        // Add constraint to list
        RESULT = list;
        RESULT.add(factory.buildConstraint(ConstraintKindAS$Class.INIT, "", exp, null));
:}
%4 = {:
        // Add constraint to list
        RESULT = list;
        RESULT.add(factory.buildConstraint(ConstraintKindAS$Class.DERIVE, "", exp, null));
:}

classifierContextDecl → CONTEXT:loc pathName:path invOrDef:constraints %1 .
%1 : {:
        // Create a ClassifierContext
        RESULT = factory.buildClassifierContextDeclaration(path, constraints, new Symbol(0,
        locleft, locright));
:}

invOrDef   →  INV simpleName:name COLON oclExpression:exp %1
              |
              INV COLON oclExpression:exp %2
              |
              DEF simpleName:name COLON defExpression:exp %3
              |
              DEF COLON defExpression:exp %4
              |
              invOrDef:list INV simpleName:name COLON oclExpression:exp %5
              |
              invOrDef:list INV COLON oclExpression:exp %6
              |
              invOrDef:list DEF simpleName:name COLON defExpression:exp %7
              |
              invOrDef:list DEF COLON defExpression:exp %8 .
%1 = {:
        // Create a LIST and add constraint
        RESULT = new Vector();
        RESULT.add(factory.buildConstraint(ConstraintKindAS$Class.INV, name, exp, null));
:}
%2 = {:
        // Create a LIST and add constraint
        RESULT = new Vector();
        RESULT.add(factory.buildConstraint(ConstraintKindAS$Class.INV, "", exp, null));
:}
%3 = {:
        // Create a LIST and add constraint
        RESULT = new Vector();
        RESULT.add(factory.buildConstraint(ConstraintKindAS$Class.DEF, name,
        (OclExpressionAS)exp[1], exp[0]));
:}
%4 = {:
        // Create a LIST and add constraint
        RESULT = new Vector();
        RESULT.add(factory.buildConstraint(ConstraintKindAS$Class.DEF, "",
        (OclExpressionAS)exp[1], exp[0]));
:}
%5 = {:
        // Add constraint to list
        RESULT = list;
        RESULT.add(factory.buildConstraint(ConstraintKindAS$Class.INV, name, exp, null));
```

```
:}
%6 = {:
        // Add constraint to list
        RESULT = list;
        RESULT.add(factory.buildConstraint(ConstraintKindAS$Class.INV, "", exp, null));
:}
%7 = {:
        // Add constraint to list
        RESULT = list;
        RESULT.add(factory.buildConstraint(ConstraintKindAS$Class.DEF, name,
        (OclExpressionAS)exp[1], exp[0]));
:}
%8 = {:
        // Add constraint to list
        RESULT = list;
        RESULT.add(factory.buildConstraint(ConstraintKindAS$Class.DEF, "",
        (OclExpressionAS)exp[1], exp[0]));
:}

defExpression  →      simpleName:name COLON type:type EQ oclExpression:exp %1
                      |
                      operation:oper EQ oclExpression:exp %2 .
%1 = {:
        // Create a container
        VariableDeclarationAS var = new VariableDeclarationAS$Class();
        var.setName(name);
        var.setType(type);
        var.setInitExp(exp);
        RESULT = new Object[] {var, exp};
:}
%2 = {:
        // Create a container
        RESULT = new Object[] {oper, exp};
:}

operationContextDecl → CONTEXT operation:oper prePostOrBodyDecl:list %1.
%1 = {:
        // Create OperationContextDecl
        RESULT = factory.buildOperationContextDeclaration(oper, list);
:}

prePostOrBodyDecl  →  PRE simpleName:name COLON oclExpression:exp %1
                      |
                      PRE COLON oclExpression:exp %2
                      |
                      POST simpleName:name COLON oclExpression:exp %3
                      |
                      POST COLON oclExpression:exp %4
                      |
                      BODY simpleName:name COLON oclExpression:exp %5
                      |
                      BODY COLON oclExpression:exp %6
                      |
                      prePostOrBodyDecl:list PRE simpleName:name COLON oclExpression:exp %7
                      |
                      prePostOrBodyDecl:list PRE COLON oclExpression:exp %8
                      |
                      prePostOrBodyDecl:list POST simpleName:name COLON oclExpression:exp %9
                      |
                      prePostOrBodyDecl:list POST COLON oclExpression:exp %10
                      |
                      prePostOrBodyDecl:list BODY simpleName:name COLON oclExpression:exp %11
                      |
                      prePostOrBodyDecl:list BODY COLON oclExpression:exp %12.
%1 = {:
        // Create a LIST and add constraint
        // Create a constraint
        ConstraintAS cons = factory.buildConstraint(ConstraintKindAS$Class.PRE, name, exp,
        null);
        // Create a list
        RESULT = new Vector();
        RESULT.add(cons);
:}
%2 = {:
        // Create a LIST and add constraint
```

```
        RESULT = new Vector();
RESULT.add(factory.buildConstraint(ConstraintKindAS$Class.PRE, "", exp, null));
:}
%3 = {:
        // Create a LIST and add constraint
        RESULT = new Vector();
        RESULT.add(factory.buildConstraint(ConstraintKindAS$Class.POST, name, exp, null));
:}
%4 = {:
        // Create a LIST and add constraint
        RESULT = new Vector();
        RESULT.add(factory.buildConstraint(ConstraintKindAS$Class.POST, "", exp, null));
:}
%5 = {:
        // Create a LIST and add constraint
        RESULT = new Vector();
        RESULT.add(factory.buildConstraint(ConstraintKindAS$Class.BODY, name, exp, null));
:}
%6 = {:
        // Create a LIST and add constraint
        RESULT = new Vector();
        RESULT.add(factory.buildConstraint(ConstraintKindAS$Class.BODY, "", exp, null));
:}
%7 = {:
        // Add constraint to list
        RESULT = list;
        RESULT.add(factory.buildConstraint(ConstraintKindAS$Class.PRE, name, exp, null));
:}
%8 = {:
        // Create a constraint
        RESULT = list;
        RESULT.add(factory.buildConstraint(ConstraintKindAS$Class.PRE, "", exp, null));
:}
%9 = {:
        // Create a constraint
        RESULT = list;
        RESULT.add(factory.buildConstraint(ConstraintKindAS$Class.POST, name, exp, null));
:}
%10 = {:
        // Create a constraint
        RESULT = list;
        RESULT.add(factory.buildConstraint(ConstraintKindAS$Class.POST, "", exp, null));
:}
%11 = {:
        // Create a constraint
        RESULT = list;
        RESULT.add(factory.buildConstraint(ConstraintKindAS$Class.BODY, name, exp, null));
:}
%12 = {:
        // Create a constraint
        RESULT = list;
        RESULT.add(factory.buildConstraint(ConstraintKindAS$Class.BODY, "", exp, null));
:}

operation  →  pathName:path COLON_COLON simpleName:name
              LEFT_PAR variableDeclarationList:params RIGHT_PAR COLON type:type %1
              |
              pathName:path COLON_COLON simpleName:name
              LEFT_PAR variableDeclarationList:params RIGHT_PAR %2
              |
              pathName:path COLON_COLON simpleName:name
              LEFT_PAR RIGHT_PAR COLON type:type %3
              |
              pathName:path COLON_COLON simpleName:name LEFT_PAR RIGHT_PAR %4
              |
              simpleName:name
              LEFT_PAR variableDeclarationList:params RIGHT_PAR COLON type:type %5
              |
              simpleName:name LEFT_PAR variableDeclarationList:params RIGHT_PAR %6
              |
              simpleName:name LEFT_PAR RIGHT_PAR COLON type:type %7
              |
              simpleName:name LEFT_PAR RIGHT_PAR %8.
%1 = {:
        // Create an Operation
```

```
        RESULT = factory.buildOperation(path, name, params, type);
:}
%2 = {:
        // Create an Operation
        RESULT = factory.buildOperation(path, name, params, null);
:}
%3 = {:
        // Create an Operation
        RESULT = factory.buildOperation(path, name, new Vector(), type);
:}
%4 = {:
        // Create an Operation
        RESULT = factory.buildOperation(path, name, new Vector(), null);
:}
%5 = {:
        // Create an Operation
        RESULT = factory.buildOperation(new Vector(), name, params, type);
:}
%6 = {:
        // Create an Operationn
        RESULT = factory.buildOperation(new Vector(), name, params, null);
:}
%7 = {:
        // Create an Operationn
        RESULT = factory.buildOperation(new Vector(), name, new Vector(), type);
:}
%8 = {:
        // Create an Operationn
        RESULT = factory.buildOperation(new Vector(), name, new Vector(), null);
:}

variableDeclarationList → variableDeclaration:var %1
                          |
                          variableDeclarationList:varList COMMA variableDeclaration:var %2.
%1 = {:
        // Create a List
        RESULT = new Vector();
        RESULT.add(var);
:}
%2 = {:
        // Append 'var' to 'varList'
        RESULT = varList;
        RESULT.add(var);
:}

variableDeclaration → simpleName:name COLON type:type EQ oclExpression:init %1
                      |
                      simpleName:name COLON type:type %2
                      |
                      simpleName:name EQ oclExpression:init %3
                      |
                      simpleName:name %4.
%1 = {:
        // Create a VariableDeclaration
        RESULT = factory.buildVariableDeclaration(name, type, init);
:}
%2 = {:
        // Create a VariableDeclaration
        RESULT = factory.buildVariableDeclaration(name, type, null);
:}
%3 = {:
        // Create a VariableDeclaration
        RESULT = factory.buildVariableDeclaration(name, null, init);
:}
%4 = {:
        // Create a VariableDeclaration
        RESULT = factory.buildVariableDeclaration(name, null, null);
:}

type → pathName:path %1
       |
       collectionType:type %2
       |
       tupleType:type %3 .
%1 = {:
```

```
        // Create PathNameType
        RESULT = factory.buildPathNameType(path);
:}
%2 = {:
        // Copy rule
        RESULT = type;
:}
%3 = {:
        // Copy rule
        RESULT = type;
:}
                                                        ;
collectionType → collectionKind:kind LEFT_PAR type:elementType RIGHT_PAR %1 .
%1 = {:
        // Create CollectionType
        RESULT = factory.buildCollectionType(kind, elementType);
:}
                                                        ;
tupleType → TUPLE_TYPE LEFT_PAR variableDeclarationList:varList RIGHT_PAR %2
%1 = {:
        // Create TupleType
        RESULT = factory.buildTupleType(varList);
:}

oclExpression   →     literalExp:exp %1
                      |
                      LEFT_PAR oclExpression:exp RIGHT_PAR %2
                      |
                      pathName:path isMarkedPre:isMarkedPre %3
                      |
                      oclExpression:exp DOT simpleName:simpleName isMarkedPre:isMarkedPre %4
                      |
                      oclExpression:exp MINUS_GT simpleName:simpleName %5
                      |
                      oclExpression:exp LEFT_PAR RIGHT_PAR %6
                      |
                      oclExpression:exp LEFT_PAR oclExpression:arg RIGHT_PAR %7
                      |
                      oclExpression:exp
                      LEFT_PAR oclExpression:arg1 COMMA argumentList:list RIGHT_PAR %8
                      |
                      oclExpression:exp1
                      LEFT_PAR oclExpression:nameExp COMMA variableDeclaration:var2
                      BAR oclExpression:exp2 RIGHT_PAR %9
                      |
                      oclExpression:exp1
                      LEFT_PAR oclExpression:nameExp COLON type:type COMMA
                      variableDeclaration:var2 BAR oclExpression:exp2 RIGHT_PAR %10
                      |
                      oclExpression:exp1
                      LEFT_PAR variableDeclaration:var1 BAR oclExpression:exp2 RIGHT_PAR %11
                      |
                      oclExpression:exp
                      LEFT_BRK argumentList:arguments RIGHT_BRK isMarkedPre:isMarkedPre %12
                      |
                      oclExpression:exp1 MINUS_GT ITERATE
                      LEFT_PAR variableDeclaration:var1 SEMICOLON variableDeclaration:var2
                      BAR oclExpression:exp2 RIGHT_PAR %13
                      |
                      oclExpression:exp1 MINUS_GT ITERATE
                      LEFT_PAR variableDeclaration:var2 BAR oclExpression:exp2 RIGHT_PAR %14
                      |
                      NOT oclExpression:opd %15
                      |
                      MINUS oclExpression:opd %16
                      %prec UMINUS
                      |
                      oclExpression:left TIMES oclExpression:right %17
                      |
                      oclExpression:left DIVIDE oclExpression:right %18
                      |
                      oclExpression:left INT_DIVIDE oclExpression:right %19
                      |
                      oclExpression:left INT_MOD oclExpression:right %20
                      |
```

```
                        oclExpression:left PLUS oclExpression:right %21
                        |
                        oclExpression:left MINUS oclExpression:right %22
                        |
                        IF oclExpression:condition
                        THEN oclExpression:thenExp ELSE oclExpression:elseExp ENDIF %23
                        |
                        oclExpression:left LT oclExpression:right %24
                        |
                        oclExpression:left GT oclExpression:right %25
                        |
                        oclExpression:left LE oclExpression:right %26
                        |
                        oclExpression:left GE oclExpression:right %27
                        |
                        oclExpression:left EQ oclExpression:right %28
                        |
                        oclExpression:left NE oclExpression:right %29
                        |
                        oclExpression:left AND oclExpression:right %30
                        |
                        oclExpression:left OR oclExpression:right %31
                        |
                        oclExpression:left XOR oclExpression:right %32
                        |
                        oclExpression:left IMPLIES oclExpression:right %33
                        |
                        LET variableDeclarationList:variables IN oclExpression:exp %34
                        |
                        oclExpression:target UP_UP simpleName:name
                        LEFT_PAR oclMessageArgumentList:arguments RIGHT_PAR %35
                        |
                        oclExpression:target UP_UP simpleName:name LEFT_PAR RIGHT_PAR %36
                        |
                        oclExpression:target UP simpleName:name
                        LEFT_PAR oclMessageArgumentList:arguments RIGHT_PAR %37
                        |
                        oclExpression:target UP simpleName:name LEFT_PAR RIGHT_PAR %38.
%1 = {:
       // Literal expression without enumLiteralExp
       // Copy rule
       RESULT = exp;
:}
%2 = {:
       // Copy rule
       RESULT = exp;
:}
%3 = {:
       // Create PathNameExp
       RESULT = factory.buildPathNameExp(path, isMarkedPre);
:}
%4 = {:
       // Create DotSelectionExp
       RESULT = factory.buildDotSelectionExp(exp, simpleName, isMarkedPre);
:}
%5 = {:
       // Create ArrowSelectionExp
       RESULT = factory.buildArrowSelectionExp(exp, simpleName);
:}
%6 = {:
       // Create OperationCallExp
       RESULT = factory.buildOperationCallExp(exp, new Vector());
:}
%7 = {:
       // Create OperationCallExp
       List args = new Vector();
       args.add(arg);
       RESULT = factory.buildOperationCallExp(exp, args);
:}
%8 = {:
       // Create OperationCallExp
       List args = new Vector();
       args.add(arg1);
       args.addAll(list);
       RESULT = factory.buildOperationCallExp(exp, args);
```

```
:}
%9 = {:
        // Create first variable - check the name
        VariableDeclarationAS var1 = makeVariableDeclaration(nameExp, null, null, nameExpleft,
        nameExpright);
        // Create IteratorCallExp
        RESULT = factory.buildIteratorCallExp(exp1, var1, var2, exp2);
:}
%10 = {:
        // Create first variable - check the name
        VariableDeclarationAS var1 = makeVariableDeclaration(nameExp, type, null, nameExpleft,
        nameExpright);
        // Create IteratorCallExp
        RESULT = factory.buildIteratorCallExp(exp1, var1, var2, exp2);
:}
%11 = {:
        // Create IteratorCallExp
        RESULT = factory.buildIteratorCallExp(exp1, var1, null, exp2);
:}
%12 = {:
        // Create AssociationCallExp
        RESULT = factory.buildAssociationCallExp(exp, arguments, isMarkedPre);
:}
%13 = {:
        // Create IterateExp
        RESULT = factory.buildIterateExp(exp1, var1, var2, exp2);
:}
%14 = {:
        // Create IterateExp
        RESULT = factory.buildIterateExp(exp1, null, var2, exp2);
:}
%15 = {:
        // Create NotExp
        RESULT = factory.buildLogicalExp(sym.NOT, opd, null);
:}
%16 = {:
        // Create an OperationCallExp
        RESULT = factory.buildOperationCallExp("-", opd, null);
:}
%17 = {:
        // Create an OperationCallExp
        RESULT = factory.buildOperationCallExp("*", left, right);
:}
%18 = {:
        // Create an OperationCallExp
        RESULT = factory.buildOperationCallExp("/", left, right);
:}
%19 = {:
        // Create an OperationCallExp
        RESULT = factory.buildOperationCallExp("div", left, right);
:}
%20 = {:
        // Create an OperationCallExp
        RESULT = factory.buildOperationCallExp("mod", left, right);
:}
%21 = {:
        // Create an OperationCallExp
        RESULT = factory.buildOperationCallExp("+", left, right);
:}
%22 = {:
        // Create an OperationCallExp
        RESULT = factory.buildOperationCallExp("-", left, right);
:}
%23 = {:
        // Create IfExp
        RESULT = factory.buildIfExp(condition, thenExp, elseExp);
:}
%24 = {:
        // Create an OperationCallExp
        RESULT = factory.buildOperationCallExp("<", left, right);
:}
%25 = {:
        // Create an OperationCallExp
        RESULT = factory.buildOperationCallExp(">", left, right);
:}
```

```
%26 = {:
        // Create an OperationCallExp
        RESULT = factory.buildOperationCallExp("<=", left, right);
:}
%27 = {:
        // Create an OperationCallExp
        RESULT = factory.buildOperationCallExp(">=", left, right);
:}
%28 = {:
        // Create an OperationCallExp
        RESULT = factory.buildOperationCallExp("=", left, right);
:}
%29 = {:
        // Create an OperationCallExp
        RESULT = factory.buildOperationCallExp("<>", left, right);
:}
%30 = {:
        // Create AndExp
        RESULT = factory.buildLogicalExp(sym.AND, left, right);
:}
%31 = {:
        // Create OrExp
        RESULT = factory.buildLogicalExp(sym.OR, left, right);
:}
%32 = {:
        // Create OrExp
        RESULT = factory.buildLogicalExp(sym.XOR, left, right);
:}
%33 = {:
        // Create ImpliesExp
        RESULT = factory.buildLogicalExp(sym.IMPLIES, left, right);
:}
%34 = {:
        // Create LetExp
        RESULT = factory.buildLetExp(variables, exp);
:}
%35 = {:
        // Create OclMessageExpAS
        RESULT = factory.buildOclMessageExp(OclMessageKindAS$Class.UP_UP, target, name,
        arguments);
:}
%36 = {:
        // Create OclMessageExpAS
        RESULT = factory.buildOclMessageExp(OclMessageKindAS$Class.UP_UP, target, name, new
        Vector());
:}
%37 = {:
        // Create OclMessageExp
        RESULT = factory.buildOclMessageExp(OclMessageKindAS$Class.UP, target, name,
        arguments);
:}
%38 = {:
        // Create OclMessageExp
        RESULT = factory.buildOclMessageExp(OclMessageKindAS$Class.UP, target, name, new
        Vector());
:}

argumentList → oclExpression:arg %1
               |
               argumentList:argList COMMA oclExpression:arg %2 .
%1 = {:
        // Create a List
        List seq = new Vector();
        seq.add(arg);
        RESULT = seq;
:}
%2 = {:
        // Append 'arg' to 'argList'
        RESULT = argList;
        argList.add(arg);
:}

oclMessageArgumentList → oclMessageArgument:arg %1
                       |
                       oclMessageArgumentList:argList COMMA oclMessageArgument:arg %2 .
```

```
%1 = {:
        // Create List
        List seq = new Vector();
        seq.add(arg);
        RESULT = seq;
:}
%2 = {:
        // Append 'arg' to 'argList'
        RESULT = argList;
        argList.add(arg);
:}

oclMessageArgument →  QUESTION %1
                      |
                      QUESTION COLON type:type %2
                      |
                      oclExpression:exp %3 .
%1 = {:
        // Create OclMessageArg
        OclMessageArgAS arg = new OclMessageArgAS$Class();
        RESULT = arg;
:}
%2 = {:
        // Create OclMessageArg
        OclMessageArgAS arg = new OclMessageArgAS$Class();
        arg.setType(type);
        RESULT = arg;
:}
%3 = {:
        // Create OclMessageArg
        RESULT = factory.buildOclMessageArg(exp);
:}

isMarkedPre → %1
              |
              AT PRE %2 .
%1 = {:
        RESULT = new Boolean(false);
:}
%2 = {:
        RESULT = new Boolean(true);
:}

literalExp →  collectionLiteralExp:exp %1
              |
              tupleLiteralExp:exp %2
              |
              primitiveLiteralExp:exp %3 .
%1 = {:
        RESULT = exp;
:}
%2 = {:
        RESULT = exp;
:}
%3 = {:
        RESULT = exp;
:}

collectionLiteralExp → collectionKind:kind LEFT_BRA collectionLiteralParts:parts RIGHT_BRA%1
                       |
                       collectionKind:kind LEFT_BRA RIGHT_BRA %2 .
%1 = {:
        // Create CollectionLiteralExp
        RESULT = factory.buildCollectionLiteralExp(kind, parts);
:}
%2 = {:
// Create CollectionLiteralExp
        RESULT = factory.buildCollectionLiteralExp(kind, new Vector());
:}

collectionKind   →    SET %1
                      |
                      BAG %2
                      |
```

```
                        SEQUENCE %3
                        |
                        COLLECTION %4
                        |
                        ORDERED_SET %5 .
%1 = {:
        // Set kind to SET
        RESULT = CollectionKindAS$Class.SET;
:}
%2 = {:
        // Set kind to BAG
        RESULT = CollectionKindAS$Class.BAG;
:}
%3 = {:
        // Set kind to SEQUENCE
        RESULT = CollectionKindAS$Class.SEQUENCE;
:}
%4 = {:
        // Set kind to COLLECTION
        RESULT = CollectionKindAS$Class.COLLECTION;
:}
%5 = {:
        // Set kind to ORDERED_SET
        RESULT = CollectionKindAS$Class.ORDERED_SET;
:}

collectionLiteralParts → collectionLiteralPart:colPart %1
                         |
                         collectionLiteralParts:seq COMMA collectionLiteralPart:colPart %2 .
%1 = {:
        // Create a List
        List seq = new Vector();
        seq.add(colPart);
        RESULT = seq;
:}
%2 = {:
        // Add collPart to seq
        RESULT = seq;
        seq.add(colPart);
:}

collectionLiteralPart → oclExpression:exp %1
                        |
                        collectionRange:range %2 .
%1 = {:
        // Create CollectionItem
        RESULT = factory.buildCollectionItem(exp);
:}
%2 = {:
        // Copy rule
        RESULT = range;
:}

collectionRange → oclExpression:first DOT_DOT oclExpression:last %1 .
%1 = {:
        // Create CollectionRange
        RESULT = factory.buildCollectionRange(first, last);
:}

tupleLiteralExp → TUPLE LEFT_BRA variableDeclarationList:seq RIGHT_BRA %1 .
%1 = {:
        // Create TupleLiteralExp
        RESULT = factory.buildTupleLiteralExp(seq);
:}

primitiveLiteralExp → INTEGER:value %1
                      |
                      REAL:value %2
                      |
                      STRING:value %3
                      |
                      TRUE:value %4
                      |
                      FALSE:value %5.
```

```
%1 = {:
        // Create IntegerLiteralExp
        RESULT = factory.buildIntegerLiteralExp(value);
:}
%2 = {:
        // Create RealLiteralExp
        RESULT = factory.buildRealLiteralExp(value);
:}
%3 = {:
        // Create StringLiteralExp
        RESULT = factory.buildStringLiteralExp(value);
:}
%4 = {:
        // Create BooleanLiteralExp
        RESULT = factory.buildBooleanLiteralExp(value);
:}
%5 = {:
        // Create BooleanLiteralExp
        RESULT = factory.buildBooleanLiteralExp(value);
:}


pathname → simpleName:name %1
            |
            pathName:path COLON_COLON simpleName:name %2 .
%1 = {:
        // Create a
        List seq = new Vector();
        seq.add(name);
        RESULT = seq;
:}
%2 = {:
        // Add name to path
        RESULT = path;
        path.add(name);
:}

simpleName → SIMPLE_NAME:value %1 .
%1 = {:
        RESULT = value;
:}
```

Annex E. OCL issues

Issue: General section to define OCL concepts

Description: The specification should contain an introductory section containing definitions of the terms used in the specification and other notations that are used (e.g. well-formed expression, ill-formed expression, behaviour, undefined-behaviour etc.). Rationale: This will avoid ambiguities and provide a better specification of the OCL (see specifications for C++, Java, and C#).

Issue: Virtual machine

Description: The OCL 2.0 specification should be behaviour-oriented and not implementation-oriented (see section 4.3). Rationale: The idea of using OCL to describe itself is interesting from the research point of view, but unfortunately OCL is not a suitable metalanguage to define the meaning of other textual languages. We think that the best thing to do is to define a virtual machine and to describe the behaviour of the virtual machine using natural language. This technique was successfully used for languages like C, C++, Java, C#, and Prolog. We see no reasons why such a technique would fail for OCL. After all, OCL is less complex than modern programming language like C++, Java, or C#. A proper description and implementation of the OCL virtual machine will create all the conditions to have a language that is platform/tool independent.

Issue: Set of characters

Description: The OCL 2.0 specification should describe the set of characters allowed in the OCL constructions (e.g. Unicode or ASCII). Rationale: This will help implementers to solve an ambiguity and to produce portable implementations. Unicode will be in our opinion the best choice.

Issue: Unspecified syntax and semantics for Integer, Real, and String

Description: The specification does not describes the syntax of integer, real or string literals. Also, it does not contain the description of the allowed set of values. Rationale: Specifying the syntax and the semantics of basic types will increase the portability of OCL programs. In order to describe the semantics of basic types, the specification should describe the set of values, the allowed operations, and the standard used to perform the allowed operations. We think that, in order to optimize the computational process, it will be also useful to allow different types of integers and reals, like Integer(16), Integer(32), Integer(64), Real(32), and Real(64).

Issue: Keywords

Description: OCL 2.0 uses keywords (e.g. and, or, xor, implies etc.) that cannot be used elsewhere.

Rationale: This means that these names cannot be used to identify properties, classes, or packages. There are two options to solve this problem: either UML 2.0 specifies the names that cannot be used to denote model elements or the OCL concept of keywords has to be revised.

Issue: Comments

Description: OCL 2.0 comments start with --

Rationale: This means that an expression like –-4 cannot be interpreted as an arithmetic expression without inserting at least one space between the first – and the second -. I think that this problem can be resolved if the OCL comments start with // instead of --.

Issue: Operator precedence

Description: Section 4.3.2 does not specify precedence for operators like div, mod, ^^, or ^.

Rationale: In order to provide a platform-independent implementation the operator precedence must be very precise. We think that logical operators should be organized on different levels of precedence:

‘not’

‘and’

‘or’

‘xor’

‘implies’

Issue: Grammar of OCL

Description: The grammar presented in 4.3, which is our opinion a semantic grammar, is not suitable to describe the syntax of OCL.

Rationale: Introducing non-terminals like primary-expression, selection-expression, and operation-call-expression will solve all the problems and will reduce the number of ambiguities. Hence, the grammar contained in the specification will suffer fewer changes in order to be used to design and implement a deterministic parser. This is the case of the specifications for C, C++, Java, C#, and Prolog.

Issue: Abstract syntax tree

Description: Some of the elements presented in 3.3.10 (e.g. EnumLiteralExp, children of ModelPropertyCallExp) cannot be constructed without using semantic information (e.g. the type of the expression determines if a name denotes an attribute, an association end, or an operation).

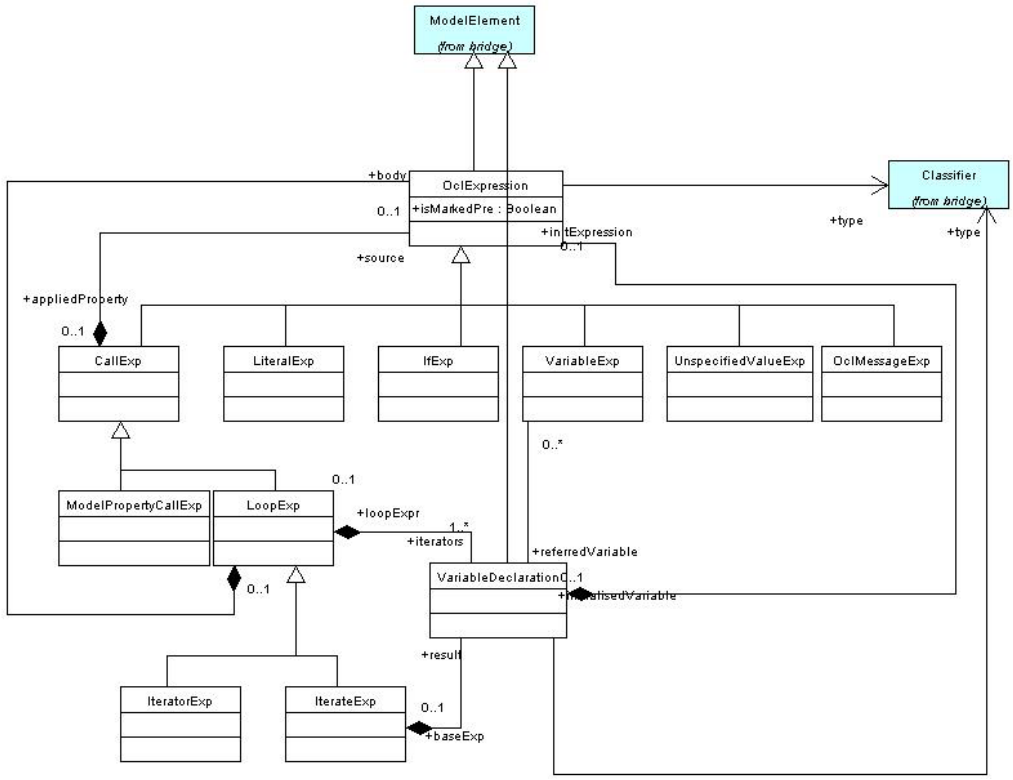
Rationale: Usually a parser produces an AST. The semantic analyser augments the AST by computing for each node from AST the values of the attached attributes. The semantic analysis also checks if there are static semantics errors and reports them. Using other terms in the AST and hence other non-

terminals in 4.5 (e.g. dot-selection-expression, arrow-selection-expression, call-expression etc.) will solve this problem.

Issue: Attributes and Association Ends versus Properties

Description: The submission uses the terms of Attributes and Association Ends, which are no longer used in UML 2.0.

Rationale: In order to align OCL 2.0 and UML 2.0 specifications we think that the expression package should look like:



We also think that the OCL grammar should be rewritten accordingly.

Issue: oclIUndefined() versus isEmpty()

Description: OCL offers two choices to test if a value is undefined or not: isEmpty and oclIsUndefined.

Rationale: Most of the modern programming languages contain null values. The best OCL mapping for null value is the undefined value. Using isEmpty to test if a value is null/undefined is confusing:

* the result of property->isEmpty() must be true if the value of the property is null/undefined

* the result of Set{1/0}->isEmpty() must be false

because the expression property->isEmpty() is converted according to the OCL specification to Set{property}->empty()

These situations are a source of errors and confusion at the implementation level. We think that isEmpty() should be used only to test if a collection is empty or not; the null/undefined values should be tested using ocIslUndefined. This operation should be also valid on collections. This approach will also work nice and clear for nested collections. On the other hand we don’t think that () should not be optional, if the called operation

has no arguments. This is feature specific to old languages like TAL and Pascal, while in modern languages like C, C++ the meaning of f and f() is different.

Issue: OclType

Description: OclType should disappear from the OCL type hierarchy.

Rationale: OclType should be only present in the standard library to support values for the type expression used in functions like oclAsType(), oclIsKindOf(), and oclIsTypeOf().

Issue: OclModelElement

Description: The object OclModelElement object should be removed from the standard library, while OclModelElementType should remain in OCL type hierarchy.

Rationale: It implies a useless level of wrapping for the model objects.

Issue: Syntax of Operation Call, Iterator, and Iterate Expressions

Description: Syntax for the above constructions is extremely ambiguous and it might involve backtracking.

Rationale: According to OCL specification

* self.f(x, y)
* Set{1,2,3}->select(x, y| x+y = 3)
* Set{1,2,3,4,5,6}->iterate(x; acc:Integer=0 | acc + x)

describes an operation call, an iterator, and an iterate expression.

In order to make the distinction between an iterator call and an operation call we need in this case a three token lookahead, starting from x. The problem gets even more complicated if we consider that an argument for an operation call can be an expression.

In order to solve this problem, which is a potential source of problems for the implementation (error-prone, inefficiency etc), we think that these OCL constructs should contain some extra syntax markers. There are several choices:

* change the comma marker from iterator calls to something else, maybe a semicolon

* add a syntax marker to an iterator name

* do not allow the default types

Each of the above choices will allow to a deterministic parser to deal with the enumerated problems more efficiently. We would prefer the third because it will solved the above problem and because we do not agree with textual language in which variables are given a default type according to the context in which they are used, especially if these languages are designed for industrial use. The same problems were in the previous versions of C standard, which allowed implicit type *int* for variables in constructions, as in

```
      x;
```

The latest C standard states that variables with default type are not allowed.

Issue: Parsing Tuple Types and Collection Types as Arguments

Description: One issue we have discovered is about expressions of the form: expr.oclAsTypeOf(Type) The OCL standard does not support Type as a collection or tuple type.

Rationale: We think that the syntax should be extended to support collection and tuple types. This will make the language more symmetric and increase the expressiveness of the language.

Issue: OclAny operations of tuples and collections

Description: The OCL specification does not allow operations like = or <> to be performed tuple values. It also forbids operations like oclIsKindOf and oclIsTypeOf on collections.

Rationale: Add such operations to tuple and collection types signatures directly or by inheritance, will make the language more powerful (e.g. a set of dogs can be conveted to a set of animals).

Issue: Signature of Environment

Description: 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*, *addEnvironment*

Rationale: We show a more complete specification below. 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.