Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 102 (2004) 43–61
www.elsevier.com/locate/entcs

A Metamodel-Based OCL-Compiler for UML and MOF

Sten Loecher, Stefan Ocke[1,2]

Department of Computer Science
Dresden University of Technology
Dresden, Germany

Abstract

After becoming part of the UML standard, OCL has been applied successfully in various domains. As a result, requirements to be met by future versions of OCL have evolved. A key requirement is to provide means for the formal integration of OCL and UML. Therefore, the latest proposal for a revised specification of OCL contains a metamodel defining the language concepts and semantics. Based on this metamodel, we are currently redesigning the Dresden OCL Toolkit. Apart from basic functionality for processing OCL expressions, the toolkit is designed to support the evaluation of well-formedness rules defined on both models and metamodels. In this paper, we introduce the revised architecture of the toolkit. The alignment of OCL with UML and MOF is discussed and several concepts for the explicit modelling of dependencies between these metamodels are presented. Finally, we prove the adequacy of our design by presenting a code generator for the evaluation of well-formedness rules.

Keywords: OCL, MOF, UML, metamodelling, compiler

1 Introduction

In the past few years, the Object Constraint Language (OCL)[12] has evolved from being merely an extension of the Unified Modeling Language (UML)[7] to representing an integral part of it. The original purpose of OCL was to precisely express constraints over object models in terms of preconditions,

[1] Email: `Sten.Loecher@inf.tu.dresden.de`
[2] Email: `so3@wh2.tu-dresden.de`
[3] This work was supported by grants of the Deutsche Forschungsgemeinschaft, Graduiertenkolleg 191.

doi:10.1016/j.entcs.2003.09.003

postconditions, or invariants. Users soon realized the potential for alternative applications. The usage in according scenarios resulted in a number of requirements to be met by future versions of the specification language, such as the incorporation of concepts like message expressions and tuple types.

The latest response to the UML 2.0 OCL request for proposal [1] therefore contains a completely reworked specification of the OCL, which defines it as a general query language that can be used everywhere in UML models to express desired properties. In contrast to earlier versions of the OCL, this response contains a definition of the OCL concepts and semantics by means of a metamodel compliant to the Meta Object Facility (MOF)[8]. The provision of the OCL metamodel is motivated by the prospective integration of OCL with the UML metamodel as investigated for example in [11]. It also serves as a basis for efficient development of tool support.

Our current work aims at redesigning the Dresden OCL Toolkit [4] to support OCL as defined by the metamodel in [1]. On the one hand, the toolkit is designed to provide basic functionality for processing OCL expressions, such as parsing, type checking, and code generation. On the other hand, the toolkit design is driven by the requirement to support both the evaluation of well-formedness rules (WFR) defined on metamodels and constraints on normal models. As a first step we have developed an appropriate OCL compiler infrastructure that is based on current specifications of the UML and MOF (version 1.4 of both). A major part of our work concerned the alignment of the OCL metamodel to the metamodels of UML and MOF. The alignment was necessary because of dependencies defined between OCL and UML metamodel and the requirement to support WFRs on metamodels, which are actually MOF models. Solutions to several problems have been elaborated and a common interface of OCL to UML and MOF has been developed. We think that the results presented in this paper can be useful for the prospective alignment of OCL and UML version 2.0. To demonstrate the adequacy of our design, a code generator has been developed to generate Java expressions by which well-formedness rules over metamodels can be evaluated.

After a discussion of related work and a number of preliminary issues in Sections 2 and 3, the general architecture of the OCL compiler is introduced in Section 4. Section 5 discusses the alignment of OCL to UML and MOF. The code generation is presented in Section 6. We conclude by summarizing the results of our work and give a short outlook about our future work in section 7.

2 Related Work

Based on version 1.3 of the OCL specification, our group has already developed the Dresden OCL Toolkit [4,5,3,13]. It features a modularly structured OCL compiler that provides a set of interfaces for easy enhancement and integration into different software engineering environments. With the toolkit it is possible to process preconditions, postconditions, and invariants on UML models. Code generators have been developed for Java and the Structured Query Language (SQL) and the usage of the toolkit within several projects, partly in conjunction with partners in industry, has led to extensive tests and eventually to a solid foundation for OCL support. In our current work, we use artifacts from this earlier toolset, e.g. an OCL Basis Library.

[10] describes the USE tool, which can evaluate OCL expressions over UML models. It provides functionality to describe models on a textual basis and associated OCL expressions and to create instances, so-called snapshots, of them. The OCL expressions are evaluated based on these snapshots. Basically, these models can also be metamodels and therefore can be subject to OCL expression evaluation. Differences between the OCL expression evaluation in [10] and our tool concern the used metamodels and the kind of OCL expression evaluation. Whereas USE is based on a small core of the UML and a precursor of the OCL metamodel, our tool is based on proposed and accepted standards of the Object Management Group (OMG). Furthermore, in contrast to our toolkit, USE does not generate code for the evaluation of OCL expressions but uses interpretation.

3 Preliminaries

This section discusses some preliminary issues concerning terms and technologies used throughout this paper. The discussion is centered around the MOF metadata architecture, which is introduced first. Terms like XML Metadata Interchange (XMI) and Java Metadata Interface (JMI) are explained subsequently.

The MOF, defined by the OMG, aims at providing a framework for the management of metadata. It comprises a layered metadata architecture and a meta-metamodel. Figure 1 illustrates the metadata architecture by an example containing metamodels for representing UML diagrams and OCL expressions. The layers are numbered consecutively from M0 to M3. The data on each layer is described by the model one layer up, e.g. actual data and objects are modeled by UML models which are themselves instances of the UML metamodel. The UML metamodel itself is described in terms of the

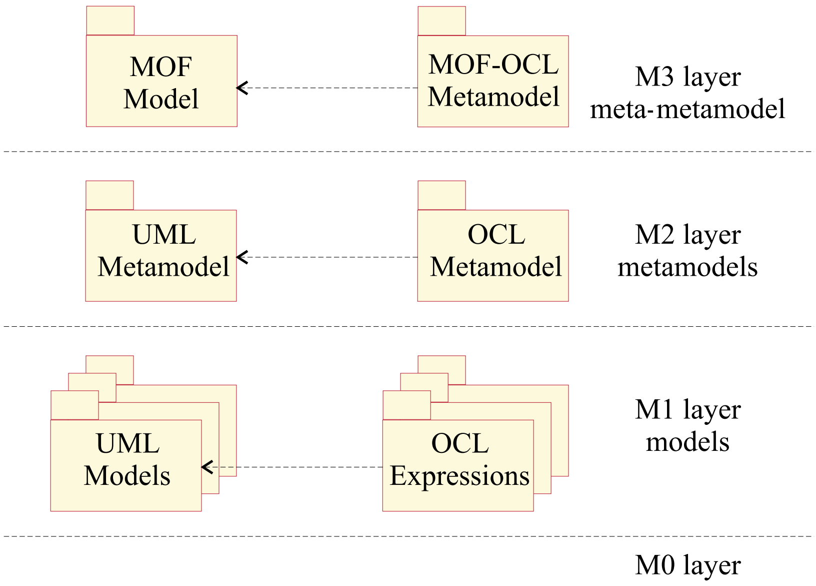

Fig. 1. MOF Metadata Architecture.

MOF meta-metamodel. This meta-metamodel, or MOF model, is object-oriented and includes an essential set of modeling constructs aligned with a specific subset of UML object modeling constructs. The MOF model on its part is self-describing, i.e., it is formally defined using its own metamodelling constructs.

OCL is defined by a metamodel with dependencies to elements of the UML metamodel and allows to write OCL expressions for UML models, which reside on layer M1. Since metamodels on M2 layer are instances of the MOF model but OCL is rather designed to write expressions for models on M1 layer, the specification of WFRs by OCL would not be possible without having an adapted OCL metamodel for MOF. Section 5 describes how to adapt the original OCL metamodel in order to be used for the specification of WFRs on M2 layer.

The MOF metadata architecture provides the foundation for the definition of XMI, which describes a standard for the exchange of models among software engineering tools. XMI comprises DTD[4] and document production rules. The DTD production rules are used to describe the generation of DTDs from metamodels (M2) whereas the document production rules describe the generation of XML documents from actual models (M1). The XML documents containing the UML metamodel and MOF model for example are provided by

[4] Document Type Definition

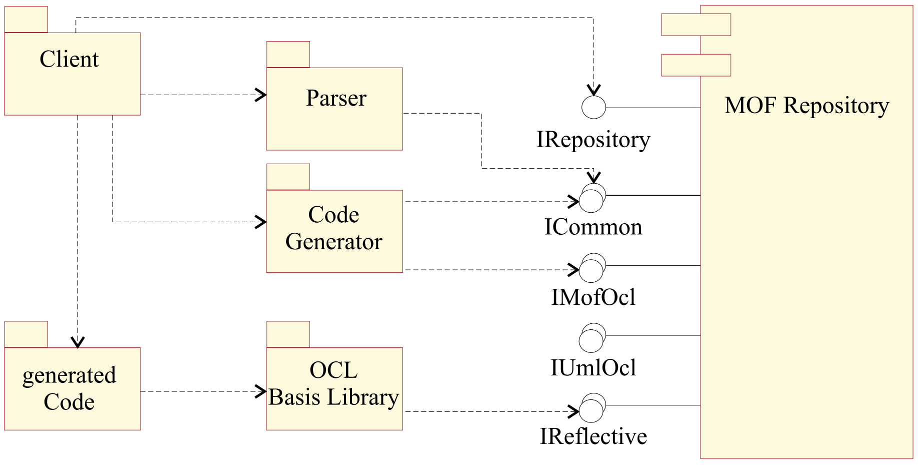

Fig. 2. Compiler Architecture.

the OMG.

XMI enables the exchange of data between tools. For tools to share data located in a common repository, JMI and the MOF-IDL mapping provide the technology to generate interfaces from arbitrary metamodels (M2) which are then used to access according instances (M1) or to perform necessary operations on them. Whereas JMI describes the production of Java interfaces and is not part of the OMG standards, the MOF-IDL mapping is part of the MOF specification and describes the production of CORBA-IDL interfaces.

4 Compiler Architecture

The general architecture of the compiler, illustrated by Figure 2, is determined by the requirement to develop a toolkit based on standardized metamodels. Therefore, the primary component of the architecture is a MOF repository. The purpose of the repository is to manage models and metamodels, to generate particular interfaces for accessing these models, and to provide implementations for the interfaces according to given specifications. The interfaces provided by the MOF repository can be classified with respect to their functionality[5]:

IRepository comprises operations for loading and saving models and metamodels via XMI and operations for generation of JMI interfaces from metamodels. Since the standardization of `IRepository` is not currently part of MOF, the interface is at the moment of a rather proprietary nature.

[5] Sets of interfaces are illustrated by double circles in Figure 2.

IMofOcl and IUmlOcl are sets of JMI interfaces generated from the MOF 1.4 and the UML 1.4 metamodel, respectively. The metamodels are appropriately integrated with the specific OCL metamodel, namely MOF-OCL and UML-OCL. Section 5 provides detailed information about the individual metamodels and the subject of integration.

ICommon is a set of JMI interfaces that enable access to instances of the OCL metamodel in a transparent way, independent of MOF or UML. They abstract common concepts of `IMofOcl` and `IUmlOcl`. Details about the common interface and its relations to UML and MOF are described in section 5.1.

IReflective is a set of reflective JMI interfaces that enable to access models without the generated metamodel-specific interfaces. These interfaces are used during the evaluation of WFRs, for example. Section 6 provides more information on this particular subject.

The interfaces explained so far are used by several components to accomplish standard tasks like parsing or more specialized tasks like code generation. A parser basically transforms OCL expressions, stated for a metamodel or UML model, from concrete syntax into abstract syntax. That is, it creates instances of the MOF-specific or UML-specific OCL metamodel in the repository. An example code generator for the evaluation of WFRs is discussed in more detail in section 6. It has been developed to test the compiler architecture and to prove its adequacy. We would like to emphasize that the parser uses `ICommon` exclusively. That is, the same parser can be used to work with UML and MOF models. Besides that, the code generator is to a large extent decoupled from the MOF metamodel. Only in case of MOF-specific information, it has to fall back on `IMofOcl`.

Figure 2 also shows a client to illustrate the use of the toolkit. This client could be, for example, a transformation engine according to the upcoming MOF 2 QVT standard [9] that uses OCL as query language for models. After loading a metamodel into the repository via `IRepository`, the parser is used to create OCL metamodel instances from OCL expressions. The code generator is used afterwards to generate appropriate code to evaluate the OCL expressions for arbitrary instances of the metamodel loaded at the beginning. The instances themselves can also be created by using `IRepository`.

To realize this architecture, we had to make a decision on what MOF repository to use for our toolset. After an evaluation and comparison of several MOF repository implementations we decided to use the NetBeans Metadata Repository [6]. This product is part of the NetBeans Integrated Development Environment, but can be used as a standalone tool as well.

5 Alignment of OCL to MOF and UML

This section discusses the alignment of OCL to MOF and UML. After a discussion of the differences between MOF and UML core, we introduce an abstract interface that allows to access instances of the OCL metamodel independently of the UML-specific or MOF-specific metamodels. We then present an adapter-based solution for the mapping between MOF and OCL datatypes. Finally, we show how the WFRs of OCL can be decoupled from the internal structure of the UML.

5.1 Common OCL Metamodel

As mentioned previously, OCL is expected to be integrated with the common core of MOF version 2 and UML version 2 in the long run. The current version of the OCL submission [1] is based on UML version 1.4. Due to our requirement to enable the evaluation of WFRs on metamodels, the current OCL metamodel had to be integrated with MOF 1.4 as well. Because of differences between MOF 1.4 and the core of UML 1.4 it was necessary to introduce an abstraction layer that comprises common concepts of MOF and UML that are used by the OCL metamodel. In the following, the differences between MOF and UML are described and the structure of the common OCL metamodel that uses the MOF/UML abstraction is explained.

5.1.1 Differences between MOF and UML Core

Chapter 8 of the current OCL submission provides a list of all classes of the UML metamodel that are used by the OCL metamodel. By looking for their equivalents in MOF 1.4, three cases can be distinguished:

(1) There is an *adequate class in MOF* that may be named differently. For example, the OCL metamodel uses the UML metaclass `Classifier` as supertype for all classes of the `Types` package. This way, properties of `Classifier` - like the ability of having operations - can be reused by the OCL types. In MOF on the other hand, there is a metaclass called `Classifier` as well but instances are not allowed to have operations. Instead, only instances of metaclass `Class` may have operations. That is, `Class` is the right choice for an equivalent of `Classifier` in MOF.

(2) There is a *similar concept in MOF*, but it is not modeled by a separate class. There is for example an Association between `EnumLiteralExp` in the OCL metamodel and `EnumerationLiteral` in the UML metamodel. MOF knows the concept of enumerations as well, but there is no separate class for enumeration literals. Rather, enumeration literals are stated by

the multivalued attribute `labels` in the metaclass `EnumerationType`.

(3) There is *no comparable concept in MOF*. For instance, the UML metaclass `AssociationClass` is referenced by the OCL metaclass `AssociationClassCallExp`. In MOF, association classes are not supported in any way.

As explained in the following section, problems (1) and (3) are solved by the introduction of the MOF/UML abstraction layer, while (2) requires the introduction of adapters for MOF.

Whereas the metaclasses of MOF and UML core are generally quite similar, the associations between them are very different. Since the WFRs of the OCL metamodel and the rules for the abstract syntax mapping navigate along such internal associations of the UML metamodel, they cannot be directly applied to MOF. Therefore, Section 5.3 introduces additional operations in the MOF/UML abstraction layer that encapsulate internal navigations through the UML metamodel.

5.1.2 Structure of the Common OCL Metamodel

The package structure of the common OCL metamodel is illustrated by Figure 3. `Common-OCL` is the central element of the abstraction. It can be accessed through the `ICommon` interfaces, which are generated from this package respectively from the contained classes by the MOF repository. `Common-OCL` comprises `CommonModel` on the one hand and the `Expressions` and `Types` packages on the other hand. The package `CommonModel` provides abstractions for the UML metaclasses that are used by the `Expressions` and `Types` packages of the OCL metamodel. The latter do not exclusively exist within `Common-OCL` but also within the individual OCL metamodels aligned to MOF and UML, which are contained within `MOF-OCL` and `UML-OCL`. Classes from the individual OCL metamodels inherit from the according abstract classes in `Common-OCL` and have relations to respective classes of the MOF or UML metamodel. The reader should be aware that instances of the OCL metamodel are either instances of `MOF-OCL` or `UML-OCL`.

The package `UML-OCL` contains the OCL metamodel defined in [1]. It has dependencies on the UML metamodel due to association and generalization relationships. The packages `UML` and `UML-OCL` serve as the basis for generating `IUmlOcl`. These interfaces are accessed, for example, by a code generator for OCL expressions in UML models.

Figure 4 gives an example of the relations between `Common-OCL` and `UML-OCL`: the class `CollectionType` from the `Types` package. To reuse the

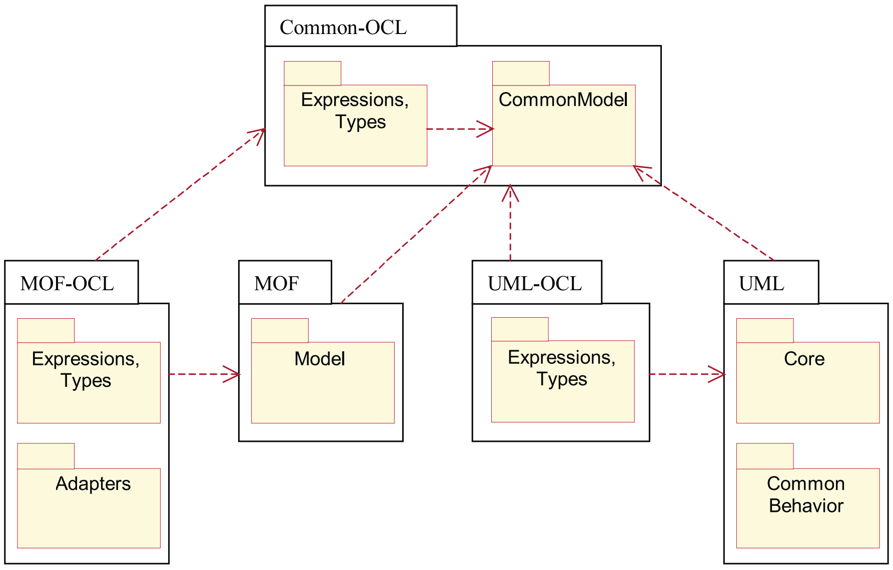

Fig. 3. Packages of Common OCL Metamodel.

association between `Classifier` and `Operation` in the UML metamodel[6], `CollectionType` in UML-OCL inherits from `UML::Core::Classifier`. In `CommonModel`, there is no association between `Classifier` and `Operation`, because an according association would not model the reuse of the already existing association in `UML` properly. Instead, the operations are made accessible by defining the additional operation `lookupOperation()` for `Classifier`.

For the case of MOF, which is not illustrated in Figure 4, `MOF-OCL::Types:CollectionType` inherits from `MOF::Model::Class`, because `MOF::Model::Classifier` cannot have operations.

5.2 Adapters for MOF and Datatype Mapping

MOF defines its own datatype schema, providing metaclasses for constructing primitive types, structured types, collection types, and enumeration types. With the exception of primitive types like Integer and String, all types are user-defined. In contrast to the datatypes defined in the OCL Standard Library, the MOF datatypes cannot have any operations, because they do not inherit from `MOF::Model::Class`. Consider, for example, a metamodel containing a class `A` that has an attribute `x` of type Integer. To represent a simple OCL constraint like

[6] In Figure 4, this relation is shown simplified.

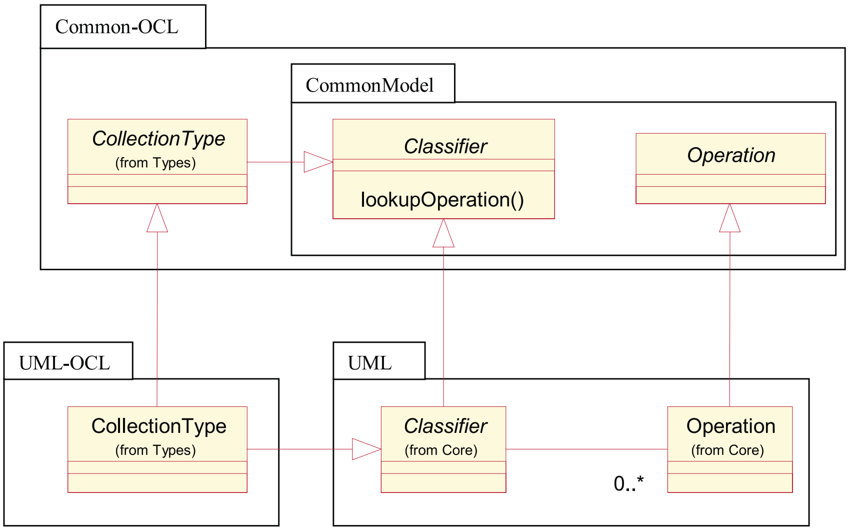

Fig. 4. CollectionType and Classifier in Common-OCL and UML-OCL.

```
context A inv: x > 0
```

in abstract syntax, `x` has to be treated as an attribute of the primitive OCL datatype Integer. When applied to a MOF model, the constraint cannot be modeled, because `x` has to be treated as having the primitive MOF datatype Integer. That is, the operation call to "greater than" is not available. Thus, a mapping between MOF and OCL datatypes is necessary.

Figure 5 depicts the mapping for primitive types as implemented in the toolkit. As shown on the left side, MOF classes do not need to be mapped and can be used as OCL types. However, instances of `MOF::Model::DataType` inherit from `Classifier` and therefore lack the ability to possess any operations. Therefore, the adapter `AdDataType` is introduced whose instances represent OCL equivalents of MOF datatypes. `AdDataType` inherits from `Class` and has therefore the ability of possessing Operations.

With the help of `AdDataType` and the association `TypeMapping`, the mapping for an arbitrary MOF classifier can be described by the following operation:

```
context MOF::Model::Classifier
def: toOclType() : MOF::Model::Class
     = if self.oclIsKindOf(MOF::Model::Class) then
          self
       else if self.oclIsKindOf(MOF::Model::DataType) then
               self.oclAsType(MOF::Model::DataType).oclType
            else
               -- Associations are not mapped to OCL types
```

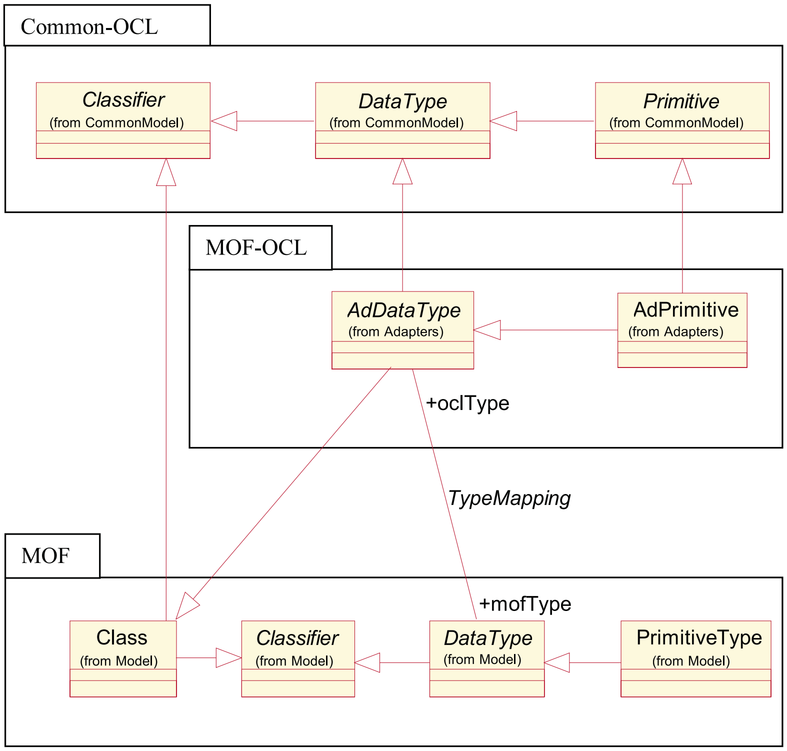

Fig. 5. Type Mapping for PrimitiveType.

```
                OclUndefined
             endif
        endif
```

Furthermore, the (non-injective) mapping for primitive types is specified by the following constraint:

```
context MOF::Model::PrimitiveType
inv: self.oclType.oclIsTypeOf(MOF-OCL::Adapters::AdPrimitive)
inv: self.name = 'Boolean' implies oclType.name = 'Boolean'
inv: self.name = 'Integer' implies oclType.name = 'Integer'
inv: self.name = 'Long' implies oclType.name = 'Integer'
inv: self.name = 'Float' implies oclType.name = 'Real'
inv: self.name = 'Double' implies oclType.name = 'Real'
inv: self.name = 'String' implies oclType.name = 'String'
```

The mappings for the remaining MOF datatypes are modeled in a similar way. The following table lists the relations between MOF datatype metaclasses and the corresponding classes in Common-OCL:

MOF datatype metaclass	**OCL datatype metaclass**
PrimitiveType	Primitive
EnumerationType	Enumeration
CollectionType	CollectionType
StructureType	TupleType

While it was necessary for primitive types to introduce an adapter (`AdPrimitive`), this is not the case for `CollectionType` and `TupleType`. Instead, the according classes from `MOF-OCL::Types` are used directly. They inherit from their counterparts in `Common-OCL` as well as from `AdDataType`. For mapping of collection and structure types, the types of the elements or fields need to be mapped as well. This is done by using the previously defined operation `toOclType()`.

As already mentioned in Section 5.1.1, `EnumerationType` needs special treatment, because the OCL metamodel expects enumeration literals to be modeled by a separate class whereas MOF models literals as the multivalued attribute `labels` in `EnumerationType`. The mapping is realized by introducing two adapter classes, namely `AdEnumeration` and `AdEnumerationLiteral`. The following constraint describes the relation between an `EnumerationType` and the associated adapters:

```
context MOF::Model::EnumerationType
inv: oclType.oclIsTypeOf(MOF::Adapters::AdEnumeration) and
     oclType.name = name and
     oclType.oclAsType(MOF::Adapters::AdEnumeration).literal.name
     = labels->asBag()
```

For UML it is necessary to map datatypes as well. Unlike MOF, UML does not predefine any datatypes for UML models (it should be noted that there is a difference between datatypes used in UML models and datatypes used to define the UML metamodel, the latter are contained within the package `UML:Core::DataType`). For example, some models may contain a primitive type named "Integer" whereas other models possibly call it "int". In both cases, it is expected to be considered as the primitive OCL type Integer when referring to it in OCL expressions. Therefore, the operation `toOclType()` is defined for `UML::Core::Classifier` as well. While it has no effect on classes, instances of `UML:Core::DataType` are mapped to OCL primitive types, as long as they have a conventional name (like "int" or "Integer"). Datatypes with unknown names will result in `OclUndefined`.

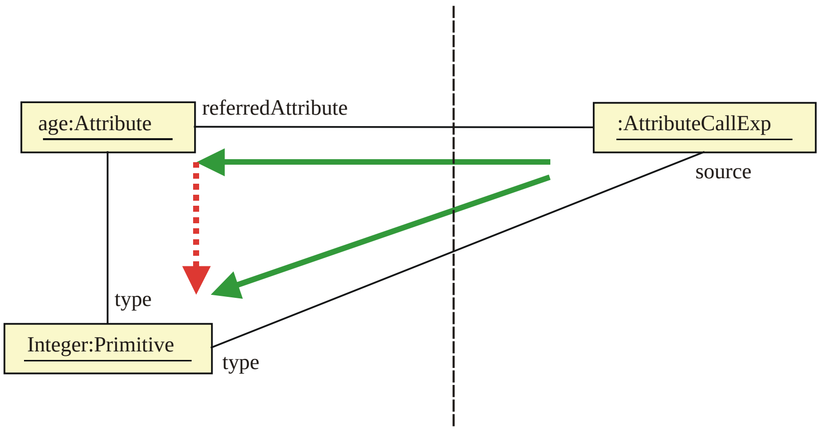

Fig. 6. WFR for AttributeCallExp.

5.3 Well-formedness Rules

The well-formedness rules of the OCL metamodel navigate along internal associations of the UML metamodel and access attributes defined in UML metaclasses. Due to these dependencies with UML, the WFRs cannot be applied directly to MOF-OCL or Common-OCL.To decouple the WFRs from the internal structure of the UML metamodel, we introduce additional operations for the `CommonModel` classes. Note, that this kind of decoupling is useful to ease the alignment with UML 2.0 as well. As long as there are only changes in the UML metamodel, but not in OCL, the WFRs do not need an update. Only the helper operations have to be aligned, because they essentially give a complete definition of the interface between OCL and UML.

The concept is illustrated by the following example WFR:

```
context AttributeCallExp
inv: self.type = self.referredAttribute.type
```

As illustrated in figure 6, the WFR navigates along two associations between UML and OCL metamodel and along one internal association of the UML metamodel. To avoid the latter, the helper operation `getOclType()` is defined for `Attribute` on `Common-OCL` level and used in the WFR instead of the association. It has distinct specifications for UML and MOF respectively:

```
context UML::Core::Attribute
def: getOclType() : OCL::CommonModel::Classifier
     = self.type.toOclType()
```

```
context MOF::Model::Attribute
def: getOclType() : OCL::CommonModel::Classifier
     = self.type.toOclType()
```

The definitions are different, because in the first case `self.type` means a navigation from `UML::Core::Attribute` to `UML::Core::Classifier` but in the second case from `MOF::Model::Attribute` to `MOF::Model::Classifier`. As described in the previous chapter, the type of the attribute is mapped to the according OCL type by the `toOclType()` operation.

6 Code Generation

To prove the adequacy of the compiler architecture and design, a code generator to support the evaluation of WFRs has been developed. For this purpose, we use an existing OCL Standard Library previously developed for the Dresden OCL Toolkit. After a discussion of the standard library adaptation in section 6.1, the transformation of OCL metamodel instances into code is described in section 6.2.

6.1 OCL Basis Library

In [2] a Java implementation of the OCL Standard Library, called OCL Basis Library in this paper, has been presented. Primarily designed to support run-time OCL constraint checking in Java programs, it uses Java reflection to access model information. Since we aim at generating code for the evaluation of OCL expression over metamodels, an implementation of the library that supports model access via JMI reflection is required. Figure 7 depicts the main artifacts of the new library. The actual implementation of OCL types like `Integer`, `Boolean`, or `Set` has been reused to a large extent. To improve the design and increase the reusability of the library, an interface for flexible replacement of the actual model access implementation has been developed. According classes have been introduced, for example `OclModelObject` and `OclEnumLiteral`, which provide template methods to be implemented by actual model access classes. `OclModelObject` provides, for example, methods for getting attribute values or invoking operations of objects from the model. For model access through JMI it is implemented by `JmiModelObject`.

The following code fragment for the example invariant

```
context A inv: x>0
```

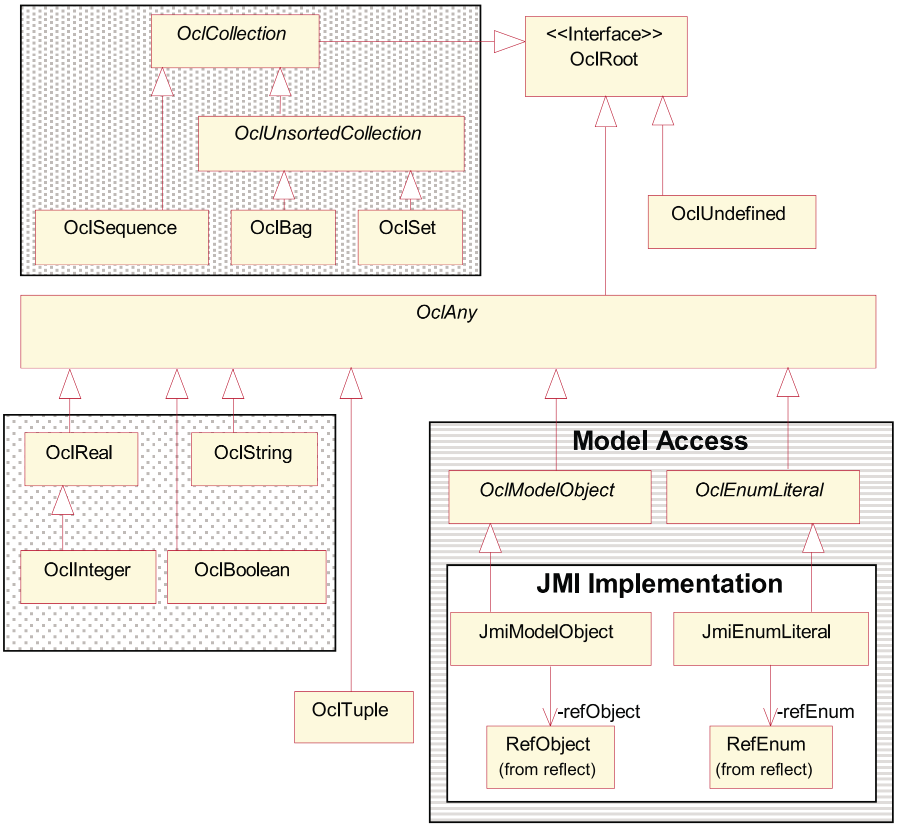

Fig. 7. OCL Basis Library with reflective JMI Model Access.

shows an example usage of the redesigned OCL Basis Library:

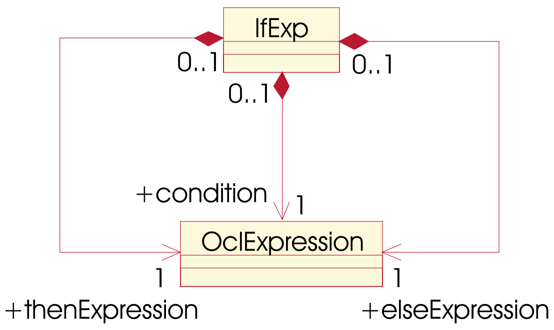

Fig. 8. Definition of the `IfExp`.

```
public OclBoolean evaluate_example_constraint(OclModelObject self){
  OclPrimitiveType intType = OclPrimitiveType.getOclInteger();
  OclInteger x = Ocl.toOclInteger(self.getFeature(intType, "x"));
  OclInteger zero = new OclInteger(0);
  return x.isGreaterThan(zero);
}
```

The transformation of OCL expressions into such code fragments is described in the following section.

6.2 Specification of Code Generation

The code generation described here is a transformation that has instances of the OCL metamodel as input and Java code as output. We specify the transformation by using OCL in a similar way as the abstract syntax mapping in the OCL submission is stated.

In the following, the specification of code generation is demonstrated by the quite simple transformation rule for `IfExp`, which is defined as shown in Figure 8:

```
context IfExp
def: appendJavaCode(env: Env) : Env =
let javaOclType : String = self.type.mapToJavaOcl(env, true) in
let withCond : Env = self.condition.appendJavaCode(env) in
let withThen : Env = self.thenExpression.appendJavaCode(withCond) in
let withElse : Env = self.elseExpression.appendJavaCode(withThen) in
let withId : Env = withElse.createId(self) in
withId.appendLine('final $1 $2 = $3;',
        Sequence{javaOclType, withId.getId(self),
          Cast::withCast(javaOclType,'$1.ifThenElse($2,$3)',
              Sequence{withId.getId(condition),
                       withId.getId(thenExpression),
                       withId.getId(elseExpression)})})
```

The code generation for all subclasses of `OclExpression` is specified as an operation `appendJavaCode()`, which expects an argument of type `Env`. This environment, see figure 9, contains the current state of the code generation. That is, it holds the code generated so far as well as all identifiers created for sub-expressions and variables. The operation `appendJavaCode()` results in a

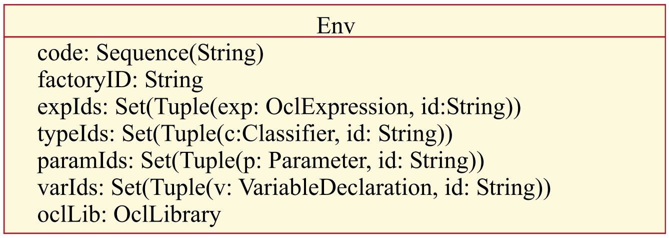
Env
code: Sequence(String)
factoryID: String
expIds: Set(Tuple(exp: OclExpression, id:String))
typeIds: Set(Tuple(c:Classifier, id: String))
paramIds: Set(Tuple(p: Parameter, id: String))
varIds: Set(Tuple(v: VariableDeclaration, id: String))
oclLib: OclLibrary

Fig. 9. Environment for Code Generation.

new environment containing the code generated by the specified transformation rule. The let-expressions are used

- to determine the `IfExp` type and the corresponding class name within the OCL Basis Library (`javaOclType`);
- to get the code for the three sub-expressions of `IfExp`, namely `withCond`, `withThen`, and `withElse`;
- and to create an identifier for the expression (`withID`), which is added to the environment.

Note, that all operations on environment are without side effects. They do not alter the state of the environment but yield a new environment, that is a ”clone” of the original one with some changes.

The operation `appendLine()` adds a line of code to the environment. The template parameters stated in the code string, such as `$1`, are replaced by the arguments given as a sequence.

The generated code uses the method `ifThenElse()`, which is defined in `OclBoolean`. This method has the return type `OclRoot`, the common superinterface of all classes in the OCL Basis Library. Thus, it is necessary to insert a type cast into the code, which yields the proper type of the `IfExp`. The operation `getId()` results in the identifier for the `IfExp` that has been created before by `createId()`.

7 Summary and Outlook

In this paper, we have introduced the general architecture of a metamodel-based OCL compiler that is based on a MOF repository. The key requirements of the OCL compiler are: to be based on standardized metamodels and to support the evaluation of well-formedness rules on them.

For this, we had to align the OCL metamodel, defined by the latest response to the UML 2.0 OCL request for proposal, with current versions of the UML and MOF. We introduced several concepts for the alignment, namely a common interface for MOF and UML to OCL, adapters at meta-level, datatype mapping between OCL and MOF, and the decoupling of WFRs from the respective metamodels by appropriate operations. We think that the proposed concepts could be useful for the upcoming alignment of OCL with UML version 2.0. To prove the adequacy of our approach, a code generator to support the evaluation of WFRs has been presented.

We can already present a prototype of the metamodel-based OCL compiler that comprises the MOF repository implementation and the presented code generator by which we can demonstrate the code generation for selected WFRs. At the moment, a parser module is under development based on the current proposal for the OCL specification. We are investigating, to which extent a parser can be generated automatically from the provided specification.

For concepts to be applied in practice, efficient tool support is essential. From our point of view, the latest proposal for the OCL specification is a major step towards moving OCL from scientific playground to industrial practice. It provides the right starting point for the efficient development of OCL tool support.

Acknowledgement

The authors would like to thank Steffen Zschaler for his valuable remarks and suggestions to this paper.

References

[1] Boldsoft, Rational Software Corporation, IONA, and Adaptive Ltd. Response to the UML 2.0 OCL RfP (OMG Document ad/2003-01-07), revised submission,version 1.6, January 6 2003.

[2] Frank Finger. Java-Implementierung der OCL-Basisbibliothek. http://www-st.inf.tu-dresden.de/ocl/ff3/beleg.pdf, July 1999.

[3] Frank Finger. Design and Implementation of a Modular OCL Compiler. http://www-st.inf.tu-dresden.de/ocl/ff3/diplom.pdf, March 2000.

[4] Heinrich Hussmann, Birgit Demuth, and Frank Finger. Modular Architecture for a Toolset Supporting OCL. In *<<UML>> 2000, The Unified Modeling Language, 3rd International Conference, York, UK. LNCS 1939*. Springer, 2000.

[5] Heinrich Hussmann, Birgit Demuth, and Sten Loecher. OCL as Specification Language for Business Rules in Database Applications. In *<<UML>> 2001, The Unified Modeling Language, 4th International Conference, Toronto, Canada. LNCS 2185*. Springer, October 2001.

[6] Sun Microsystems. Metadata repository. http://mdr.netbeans.org.

[7] Object Management Group (OMG). Unified Modelling Language (UML) specification, version 1.4, September 2001.

[8] Object Management Group (OMG). Meta Object Facility (MOF) specification, version 1.4, April 2002.

[9] Object Management Group (OMG). Request for proposal: MOF 2.0 query / views / transformations., October 2002.

[10] Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, 2001.

[11] Mark Richters and Martin Gogolla. A Metamodel for OCL. In *<<UML>>'99, The Unified Modeling Language, 2nd International Conference, Fort Collins, Colorado, USA. LNCS 1723*. Springer, 1999.

[12] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

[13] Ralf Wiebicke. Utility Support for Checking OCL Business Rules in Java Programs. http://rw7.de/ralf/diplom00/intro.html, December 2000.