Definition of OCL 2.0 Operational Semantics by means of a Parameterized Algebraic Specification [♦]

Artur Boronat, Isidro Ramos, José Á. Carsí

Department of Information Systems and Computation
Technical University of Valencia
C/Camí de Vera, s/n
46022 Valencia-Spain
{aboronat | iramos | pcarsi}@dsic.upv.es

Abstract. This paper presents an algebraic specification of the operational semantics of part of the OCL 2.0 standard. This specification is used in a model management tool that provides support for model-driven tasks within the Eclipse platform. The algebraic specification of OCL has been developed in Maude by using its parameterization mechanism, providing a suitable modularization and making reuse easier. In this work, OCL can be used in the Eclipse Modeling Framework to represent models in an algebraic setting and to perform queries or constraints over software artefacts that can be represented as models.

Keywords: OCL queries, Eclipse Modeling Framework, algebraic specification, Model-Driven Engineering.

1 Introduction

Model-Driven Development is a field in Software Engineering that, for several years, has been representing software artefacts as models in order to improve productivity, quality, and economic incomes. Models provide a more abstract description of a software artefact than the final code of the application. A model can be built by defining concepts and relationships. The set of primitives that permit the definition of these elements constitute what is called the metamodel of the model. The emergence of important model-driven initiatives such as the Model-Driven Architecture [1], supported by OMG, and the Software Factories [2], supported by Microsoft, ensures a model-driven technology stock for the near future.

In the MDA context, the standard Meta-Object Facilities (MOF) [3] provides a way to define metamodels. The standard proposal Query/Views/Transformations (QVT) [4] will provide support for both transformations and queries. While model transformation technology is being developed [5-7], the Object Constraint Language (OCL) remains as the best choice for queries.

[♦] This work was supported by the Spanish Government under the National Program for Research, Development and Innovation, DYNAMICA Project TIC 2003-07804-C05-01.

OCL [8] is a textual language that is defined as a standard “add-on” to the UML standard. It is used to define constraints and queries on UML models, allowing the definition of more precise and more useful models. It can also be used to provide support for metamodeling (MOF-based and Domain Specific Metamodeling), model transformation, Aspect-Oriented Modeling, extensions for behaviour specifications, support for model testing and simulation, ontology development and validation for the Semantic Web [9]. Despite its many advantages, while there is wide acceptance for UML design in CASE tools, OCL lacks a well-suited technological support.

In this paper, we present an algebraic specification of generic OCL queries, by using Maude [10], that can be used in a MOF-like industrial tool. The OCL algebraic specification permits the study of formal features and their proof, and the execution of OCL expressions as well. This algebraic specification has been developed in the framework MOMENT (MOdel manageMENT) [11], which provides a set of generic operators to deal with models. The MOMENT operators use OCL queries to perform model queries and transformations.

The structure of the paper is as follows: Section 2 provides an example; Section 3 describes the algebraic specification of OCL, indicating the support for basic data types and collection types, and the support for collection operations; Section 4 presents the integration of the algebraic specification of OCL within an industrial modelling framework, indicating the automatic support for representing software artefacts as algebraic specifications that can be queried; Section 5 presents some related works; Section 6 provides some conclusions and ongoing work.

2 The Rdbms Metamodel

The Meta-Object Facility standard (MOF) provides a metadata management framework and a set of metadata services to enable the development and interoperability of model and metadata-driven systems. The main achievement of this standard is the definition of a common terminology in the Model-Driven Architecture initiative, which can be used conceptually in other model-driven approaches.

As an example, in this paper, we have taken the Rdbms metamodel that has been provided in the QVT standard as case study. This metamodel provides some primitives to define relational schemas as models. The metamodel can be considered a UML model, as shown in Figure 1. Thus, the OCL-like specification that is presented can also be used for queries over any software artefact that might be defined following the MOF conceptual framework: metamodels, regular models, and instances of models.

OCL invariants permit a more precise definition of a UML model by adding constraints. In this case, we use OCL invariants to make the definition of the Rdbms metamodel more precise. For instance, to define foreign keys, we must ensure that the number of columns that participate in a foreign key must be the same as the number of columns that participate in the corresponding referred primary key. In addition, the type of the columns that participate in a foreign key must be equal to each corresponding column (by order) of the referred primary key. This constraint is expressed by means of the following invariant:

```
context ForeignKey:
inv: if (self.column->size() = self.refersTo.column->size()) then
        self.column->forAll(c:Column |
            self.refersTo.column-> at(self.column->indexOf(c)).type = c.type
        )
    else false
    endif
```

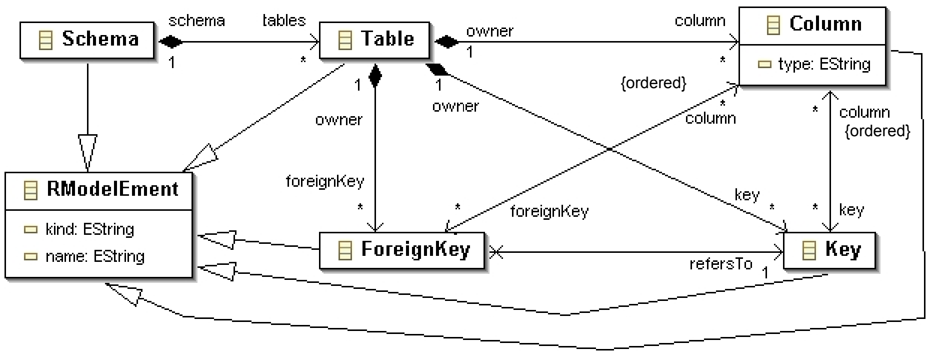

Fig. 1. Rdbms metamodel

3 Algebraic Specification of Generic[1] OCL Queries

In this section, we describe the parameterized algebraic specification of OCL that permits the query of either metamodels or UML models.

3.1 Functional Modules in Maude

In Maude, functional modules describe data types and operations on them by means of membership equational theories. Mathematically, such a theory can be described as a pair $(\Sigma, E \cup A)$, where: Σ is the signature that specifies the type structure (sorts, subsorts, kinds, and overloaded operators); E is the collection of equations and memberships declared in the functional module; and A is the collection of equational attributes (associativity, commutativity, and so on) declared for the different operators. Computation is the form of equational deduction in which equations are used from left to right as simplification rules, with the rules being Church-Rosser and terminating.

A signature in membership equational logic (MEL) is a triple $\Sigma = (K, S, OP)$, where: K represents a set of kinds; $S= \{S_k\}_{k\in K}$ represents a pairwise disjoint K-kinded

[1] In this work, OCL genericity refers to the possibility of reusing the OCL specification for any software artifact that can be represented as a model, including metamodels.

family of sets of sorts; and $OP = \{op_{k_1 \ldots k_n, k}\}_{(k_1 \ldots k_n, k) \in K^* \times K^+}$ represents a set of many-kinded operations. For a specific operation: op is the operation symbol; $k_i \in K^+$ (i=1..n) are the argument sorts; and k is the range sort.

3.2 Algebraic Specification of OCL Types

Types in OCL are divided in basic data types, collection types, and user-defined types. In this section, the algebraic support for the first two kinds of types is presented.

3.2.1 Basic Data Types

In OCL, there are four basic data types [12] that have a direct correspondence to Maude basic data types. In Table 1, we show the correspondences between OCL 2.0 and the Maude data type system and their corresponding primitives. In the table, when the operations have different symbols in OCL and Maude, we indicate the Maude symbol in brackets.

OCL 2.0	**Maude**	**Common operators**
Boolean	Bool	or, and, xor, not, = (==), <> (=/=), implies, if-then-else-endif (if-then-else-fi)
Integer	Int	= (==), <> (=/=), <, <=, >, >=, +, -, *, / (quo), mod (rem), abs (abs), max, min
Real	Float	/, round (ceiling), floor
String	String	concat (+), size (length), substring(substr), = (==), <> (=/=)

Table 1. OCL and Maude data type correspondences

There are also differences in invoking the operations in OCL and in Maude. For example, to invoke the *size* operator over a string *a,* we use dot notation: *a.size()*; while in Maude, we use the prefix notation: *length(a)*.

The Maude data type system is richer than the OCL data type system, and provides more types such as support for identifiers and rational numbers, and related operations. Thus, Maude operations can also be used in OCL queries.

3.2.2 Collection Types

OCL provides four specific collection types that are defined as follows:

– A Set is a collection that contains instances of a valid OCL type, where order is not relevant and duplicate elements are not allowed.
– An OrderedSet is a set whose elements are ordered.
– A Bag is a collection that may contain duplicate elements. Elements in a bag are not ordered.
– A Sequence is a bag whose elements are ordered.

The OCL support has been specified generically in the parameterized module *OCL-SUPPORT(X :: TRIV)*, where the formal parameter *X* has the trivial theory as type. The trivial theory only contains a sort *Elt* (referred to as *X$Elt* in the OCL specification) that represents the sort of elements that can be contained in an OCL collection. This sort represents the *OCLAny* type of the standard OCL specification. The *OCL-SUPPORT(X::TRIV)* module imports the basic data types and provides the constructors that are needed to define collections of elements. It provides collection operators as well.

To take into account the uniqueness and order features of an OCL collection, we introduce two intermediate sorts and their constructors (shown in Table 2): *Magma{X}* and *OrderedMagma{X}*. Basically, we define the sort *Magma{X}* as the sort of the term that represents a set of elements that are not ordered by means of the association and the commutativity attributes. Instead, the constructor of the sort *OrderedMagma{X}* does not have the commutativity property, producing terms that represent ordered concatenations of elements.

```
1. sorts Magma{X} OrderedMagma{X} .
2. subsort X$Elt < Magma{X} .
3. subsort X$Elt < OrderedMagma{X} .
4. op _,_ : Magma{X} Magma{X} -> Magma{X} [assoc comm ctor] .
5. op _::_ : OrderedMagma{X} OrderedMagma{X} -> OrderedMagma{X} [assoc ctor] .
```

Table 2. Specification of groups of elements

Terms of the sort *Magma{X}* are used to define sets (line 4) and bags (line 8), while terms of the sort *OrderedMagma{X}* are used in ordered sets (line 6) and sequences (line 10). In Table 3, we show the Maude code that specifies the four types of collections. In our specification, collections of collections are allowed by indicating that one collection can be an element of another collection (line 2). The sort *Collection{X}* can be considered as an abstract concept on the grounds that there is no specific constructor for it. Each collection has a constant constructor that defines an empty collection (lines 5,7, 9, 11).

```
1. sorts Collection{X} Set{X} OrderedSet{X} Bag{X} Sequence{X} .
2. subsort Collection{X} < X$Elt .
3. subsorts Set{X} OrderedSet{X} Bag{X} Sequence{X} < Collection{X} .
4. op Set{_} : Magma{X} -> Set{X} [ctor] .
5. op empty-set : -> Set{X} [ctor] .
6. op OrderedSet{_} : OrderedMagma{X} -> OrderedSet{X} [ctor] .
7. op empty-orderedset : -> OrderedSet{X} [ctor] .
8. op Bag{_} : Magma{X} -> Bag{X} [ctor] .
9. op empty-bag : -> Bag{X} [ctor] .
10. op Sequence{_} : OrderedMagma{X} -> Sequence{X} [ctor] .
11. op empty-sequence : -> Sequence{X} [ctor] .
```

Table 3. Specification of OCL collection types

In this specification, the uniqueness property of both the collection Set and the collection OrderedSet is checked in the operations that join two collections: *union*, *intersection* and *including* for Set, and *union*, *append*, *prepend*, *insertAt* and *including* for OrderedSet.

A view has been defined for each Maude simple data type in order to deal with collections of simple data types. For instance, to deal with collections of integers, the following view is defined: *view INT from TRIV to INT is sort Elt to Int . endv*

This view is used to instantiate the OCL-SUPPORT(X) module as OCL-SUPPORT(INT). This way, the following example is a valid collection of integers:
OrderedSet{ Set{1, 2, 3} :: Bag{1, 2, 3, 3} :: Sequence{3 :: 3 :: 2 :: 1}}

3.3 OCL Collection Operations

Two kinds of operations on collection types can be distinguished in OCL 2.0: regular operations and loop operations or iterators. Regular operations provide common functionality over collections. Loop operations or iterators permit looping over the elements in a collection performing a specific action.

3.3.1 Regular Collection Operations

To specify the common operations, a classification of the operations has been performed taking into account the return type of each operation (shown in Table 4). Each kind of operation is identified by a sort. We define the OCL operations as constant constructors of one of these sorts. For instance, the *size* operation, which returns an integer, is defined as follows: *op size : -> IntFun{X} [ctor]* .

	Return type	Collection operator symbols					Iterator symbols
		Collection	Set	OrderedSet	Bag	Sequence	Collection
Fun{X}	Collection	union, flatten, including, excluding, iterate	--, inter-secti-on	--, insertAt, append, prepend	intersection	insertAt, append, prepend	select, reject, any, sortedBy, collect, collectNested, iterate
EltFun{X}	Element			first, last, at		first, last, at	
BoolFun{X}	Boolean value	includes, includesAll, excludes, excludesAll, isEmtpy, notEmpty					one, forAll, forAll2[2], exists, isUnique
IntFun{X}	Integer value	count, size, sum, product		indexOf		indexOf	

Table 4. OCL collection operations that have been specified

To make these operations work, two more elements are needed. In the signature, a skeleton operator is needed to use operations in an OCL-like way. In the axiomatic presentation, axioms are needed to define the operational semantics of each operation.

The skeleton of an operation defines its syntactical structure and can be reused for several operations. For instance, to invoke the size operator in an OCL-like way, we provide the following operator: op _->_ : Collection{X} IntFun{X} -> Int . ; where the first argument is a collection, and the second argument is the corresponding operator symbol. To invoke the *size* operator over a set A, we proceed as follows: *A -> size* . The axiomatic definition of the operations defines the use of operation terms in the skeleton constructor. For instance, the operational semantics of the operation *size* is axiomatically defined for the sort *Set{X}* by using recursion as follows:

```
var N : X$Elt . var M : Magma{X} .
eq Set{ N , M } -> size = (Set{ M } -> size) + 1 .
eq Set{ N } -> size = 1 .
eq empty-set -> size = 0 .
```

[2] The forAll2 operation has been included to provide support when two iterators are being used in the forAll operation.

However, the same skeleton operator is used to define the size operator for the sort *Sequence{X}*:

```
eq Sequence{ N :: OM } -> size = (Sequence{ OM } -> size) + 1 .
eq Sequence{ N } -> size = 1 .
eq empty-sequence -> size = 0 .
```

Nonetheless, we cannot use the *count* symbol (representing the count operator) as a term in the above skeleton constructor because it is not defined in the OCL axiomatic presentation. This is due to the fact that the count operator has an argument.

3.4.2 Loop Operations or Iterators

An iterator permits the loop over the elements of a collection. Every loop operation has an OCL expression as parameter. This is called the body, or body parameter, of the operation. As a guiding example, we use a standard OCL expression that permits obtaining the odd numbers from a set of integers:

Set{1,2,3,4,5,6} -> select(i | i.mod(2) <> 0)

In this expression, *select* is the iterator operation while the expression *(i | i.mod(2) <> 0)* is the body. A body expression may have arguments to process external information. In the OCL algebraic specification, these arguments are considered by means of the sorts *Parameter* and *ParameterList*. *Parameter* has a polymorphic constructor that allows the use of any Maude data type:

op ?_ : Universal -> Parameter [ctor poly (1)] .

This constructor allows defining the following examples as valid parameters: *? 1, ? 1.0, ? 'a, ? "a", ? true.* To obtain the actual value that is enclosed in a parameter, the following casting operators can be used: *asCollection, asElt, asInt, asFloat, asString, asBool, asQid.* For example, the following expression obtains the corresponding integer value: *red (? 1 :: asInt) .*

ParameterList is a list of *Parameter* terms. The *Parameter* sort is subsort of the *ParameterList* sort indicating that a parameter list can be constituted by a single parameter. There are three constructors for lists (shown in Table 5): the *Parameter* constructor; the *empty-params* is the constant that constitutes an empty list; and the concatenation operator, which builds a list by means of blank spaces, in which the *empty-params* constant has the identity property. Thus, *? 1 ? "a"* is a valid parameter list.

```
subsort Parameter < ParameterList .
op empty-params : -> ParameterList [ctor] .
op      : Parameter ParameterList -> ParameterList [ctor id: empty-params] .
```

Table 5. Constructors for the the ParameterList sort.

Both iterator operations and body expressions are considered in the algebraic specification separately. This separation is needed to simulate higher-order functions in Maude by considering body functions as terms that can be passed as arguments to iterator operations. More precisely, the body of an iterator is associated to a symbol, which is defined as a term of a Body{X}. Thus, the iterator specification is defined independently of the specification of the body. To specify iterators and body operations, the following elements are included in the specification:

– Body sorts. Several sorts of body functions can be defined depending on their resulting type, as in collection operations. The sorts that can be used are the following: Body{X} (for functions that return a collection or an element); BoolBody{X} (for boolean functions); RatBody{X} (for functions that return a rational or an integer value); FloatBody{X} (for functions that return a float value); StringBody{X} (for functions that return a string value); and IdBody{X} (for functions that return a Qid value).

- Body symbols. These are the names of the functions. They are not included in the OCL specification since they must be defined by the user depending on the desired functionality. For instance, to define a body function, called *isOdd,* that returns an boolean value, we use the following constructor:
  op isOdd : -> BoolBody{RAT} [ctor] .
- Body skeleton constructor. To provide the way body functions must be defined, skeleton constructors are provided for each body sort in the following way:
  op _::_`(_;_`) : BodyElt{X} BoolBody{X} ParameterList Collection{X} -> Bool .
  In all of them, the first argument is a term that represents either a magma of elements or an ordered magma, the second is the corresponding body symbol, the third one is a variant list of parameters that can be empty, and the last one is the whole initial collection to which the first argument belongs.
- Body axiomatic definition. This constitutes the specification of the body function out of the iterator operation. This makes a difference between the standard OCL specification and our OCL algebraic specification. To define a body function, the axioms must be provided by the user in Maude notation. For instance, to define a boolean function that determines whether an integer number is odd or not using the previously defined *isOdd* symbol, we define the following axiom:

```
var intN : Int . var intCol : Collection{RAT} . var PL : ParameterList .
eq intN :: isOdd ( PL ; intCol ) = ((intN rem 2) =/= 0) .
```

  Where intN is an integer variable, intCol is a collection of rational numbers (also integer numbers), and PL is the list of parameters, which is ignored in this case.
- Sorts of iterators. These are the same sorts of the operations that classify collection operations by their return type.
- Iterator symbols. The symbol of the iterators corresponds to the name of the iterators that are defined in the specification. The iterators that are considered in our specification are presented in Table 4, indicating their corresponding sorts.
- Iterator skeleton constructors. They permit both the invocation of the iterator in an OCL-like way and the definition of iterators as higher-order functions that have a body operation as first argument. For instance, the skeleton constructor for the *select* iterator is defined as follows:

```
op _->_`(_;_;_`) : Collection{X} Fun{X} BoolBody{X} ParameterList Collection{X} ->
Collection{X} .
```

  Where the first argument is the collection to be looped, the second argument is an iterator symbol, the third argument is the body operation, the fourth argument is a list of arguments for the body operation, and the fifth argument is the proper collection that is looped. The fifth argument is needed due to the free side-effect that characterizes Maude and is useful when the collection must be navigated in the body operation. When the iterator is processed, if this argument is not added, the recursion mechanism consumes the elements of the collection, and queries over the whole collection would not be complete.

To invoke the select iterator over a set of integers with the body isOdd we use:

```
Set{1, 2, 3, 4, 5, 6} -> select(isOdd ; empty-params ; empty-set) .
```

– Iterator axiomatic specification. It defines the operational semantics of iterator operations independently of body operations. This fact permits the reuse of the algebraic specification of iterator operations simulating them as higher-order functions. Three axioms constitute the algebraic specification of the *select* operator for sets (as shown in Maude notation in Table 6). These are the arguments of select: BB is a variable that contains the boolean body, PL is a parameter list for the body operator, and Col is the original set. The first axiom considers the recursion case where there is more than one element in the set. If the body function validates to a true value, the element is added to the resulting set. Finally, the recursion over the rest of the elements continues. The second axiom considers the recursion case when only one element remains in the set so that the recursive trail ends. The third axiom considers the case where the set is empty.

  When the select operation is invoked, Maude returns the following result:

```
reduce in SOUP : Set{1,2,3,4,5,6}->select(isOdd ; empty-params ; empty-set) .
rewrites: 33 in -285521343301ms cpu (0ms real) (~ rewrites/second)
result Set{RAT}: Set{1,3,5}
```

```
eq Set{ N , M } -> select ( BB ; PL ; Col ) = if (N :: BB ( PL ; Col )) then Set{ N } -> including ( ( Set{ M } -> select ( BB ; PL ; Col )) ) -> flatten
    else Set{ M } -> select ( BB ; PL ; Col ) fi .
eq Set{ N } -> select ( BB ; PL ; Col ) = if (N :: BB ( PL ; Col )) then Set{ N } else empty-set fi .
eq empty-set -> select ( BB ; PL ; Col ) = empty-set .
```

Table 6. Axiomatic specification of the select operation for sets.

4 Algebraic specification of metamodels and models

The advantage of OCL is that user-defined types can be used in expressions to perform queries on software artefacts (namely models). User-defined types are the types that can be used in a model: classes, associations, enumerations, and so on. One of the keys to success in the use of the OCL algebraic specification is the integration with an industrial modelling environment. In this way, OCL expressions can be evaluated in a graphical model without having to prepare the information in a specific format manually.

In our case, we have chosen the Eclipse Modeling Framework (EMF). EMF is a modeling environment that is plugged into the Eclipse platform and provides a sort of implementation of the MOF. It brings code generation capabilities: for UML-like models, it generates the final structural code of the application; for metamodels, it generates a default tree-like editor that permits the definition of domain-specific models and the validation of the corresponding metamodel. EMF enables the automatic importation of software artefacts from heterogeneous data sources: UML models (by means of visual modeling environments), relational schemas of any relational database management system (through the Rational Rose tool), and XML schemas. Therefore, EMF has become an industrial framework for MDA.

To perform OCL queries over EMF software artefacts, two types of projection mechanisms have been specified. On the one hand, we have defined a projection mechanism that obtains the algebraic specification[3] that corresponds to a specific Ecore model automatically, by applying generative programming techniques. On the other hand, another projection mechanism permits us to project an Ecore model instance as a term of the algebra that corresponds to the Ecore model.

As shown in Fig. 2, an Ecore model may represent either a metamodel at the M2-layer (for instance, the UML metamodel) or a model at the M1-layer (for instance, a UML model). Similarly, an Ecore model instance may represent either a model that conforms to a metamodel at the M1-layer (for instance, a UML model) or a model instance at the M0-layer (for instance, the instances of a UML model). From now on, the ocl support is explained by using the example of the UML model, although it is exactly the same to define OCL queries over Ecore metamodels.

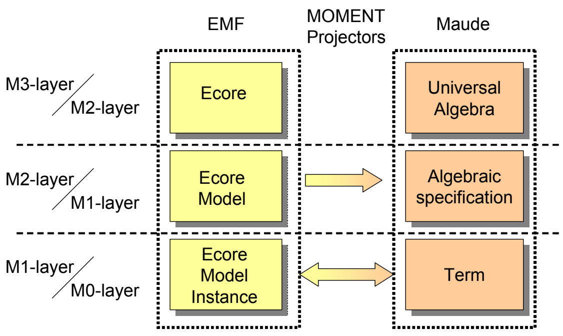

Fig. 2. EMF and Maude conceptual integration

4.1. A metamodel as an algebraic specification

To query a metamodel, it must be represented as an algebraic specification in Maude. This specification is split in two parts: the generic OCL specification that is defined in the parameterized module OCL-SUPPORT(X::TRIV) and the user-defined types in the model.

[3] The algebraic specification that is generated for a given metamodel (defined in EMF as an Ecore model) permits the representation of models as algebraic terms. Thus, models can be manipulated by our model management operators. Algebraic specifications of this kind do not specify operational semantics for the concepts of the metamodel, they only permit the representation of information for model management issues.

4.1.1 The algebraic signature of a metamodel

A metamodel in EMF is an Ecore model, which is mainly constituted by EClass instances (informally called classes) that are related to each other by means of inheritance relationships and EReference instances (informally called references in Ecore and associations in UML). They are used to generate an algebraic specification that is used as an actual parameter for the *OCL-SUPPORT(X::TRIV)* module. Informally, the following tasks are involved in the generation of Maude code from a model:

– Signature generation: An Ecore class is constituted by attributes and references. This information is used to generate a sort that represents the collection of instances of this class and a constructor, whose arguments are: an internal identifier, a group of arguments that represent the attributes (basic data types) and a group of identifier collections (representing references). For instance, the *Table* class in Fig. 1 is used to generate the following Maude code:
  ```
  sort Table .
  op `(Table_______`) : Qid String String OrderedSet {QID} OrderedSet {QID} OrderedSet {QID}
  OrderedSet {QID}  -> rdbms-Table [ctor] .
  ```
  where the first argument is the internal identifier of the instance, the second argument is the inherited *name* attribute, the third argument is the inherited *kind* attribute, the fourth argument is the *key* reference (UML role), the fifth argument is the *foreignKey* reference, the sixth argument is the *column* reference and the seventh argument is the *schema* reference. This template is only applied to classes that are not abstract. When a class is defined as abstract, the code only contains the declaration of the code and no constructor is generated, indicating that this class cannot be instantiated.
– Kind structure generation: class inheritance is represented by means of subsort relationships in Maude. In the example, the *Table* class is a specialization of the RModelElement class. In the Maude specification, the sort *Table* is defined as subsort of the RModelElement sort.
– Kind unification: all the sorts that are generated in an algebraic specification are encapsulated within the same kind, by adding a root supersort, called *rdbmsNode* (where *rdbms* is the name of the package that contains the model definition). This sort permits the definition of the view between the model algebraic specification and the formal parameter of the *OCL-SUPPORT(X::TRIV)* module. All the sorts that have no supersort must be related to root supersort by means of a subsort relationship. This allows all the terms that represent instances of model classes to belong to this sort, thereby becoming elements of a collection.
– Signature generation: the generated elements applying the previous templates are encapsulated within a module. The automatically generated module for the example model is called *sigrdbms*.
– View generation: to use the signature that is generated for a model as actual parameter for the module *OCL-SUPPORT(X::TRIV)*, a view is defined containing the mapping between the *Elt* sort of the *TRIV* theory and the generated root sort for the model. The generated view for the model in the example is as follows:
  ```
  view rdbms from TRIV to sigrdbms is sort Elt to rdbmsNode . endv
  ```

4.1.2 The algebraic specification of a metamodel

A model that conforms a metamodel is represented as a set of instances of the classes that constitute the metamodel. To define models as sets, the signature that is obtained from the definition of the metamodel is used as actual parameter for the parameterized module that contains the generic algebraic specification of OCL by means of the generated view.

The instantiation of the *OCL-SUPPORT(X::TRIV)* module is contained in another module, called *sprdbms* in the example. This module also contains the operations that are needed to navigate a model by means of either the attributes or the roles of a class.

The operation to navigate a reference (UML role) has the following skeleton:

```
op _::_`(_`) : X$Elt Fun{X} Collection{X} -> Collection{X} [ctor] .
```

where *X$Elt* represents an element of the model (a class instance), *Fun{X}* represents the symbol of the role, the Collection{X} argument represents the model to be navigated, and the Collection{X} return parameter represents the collection of class instances that is obtained by means of the navigation operation. For example, to navigate through the reference *column* of a *Table* class instance, called *t*, of the a relational schema, called *RdbmsSchema*, we use the following expression:

```
t :: column (RdbmsSchema)
```

The operation to obtain the value of an attribute has the following skeletons, depending on the attribute type:

```
op _::_ : X$Elt YFun{X}  -> Y [ctor] . (where Y = {Int, Float, Bool, String, Qid} )
```

For example, to obtain the value of the *name* attribute of a *Table* instance, called *t*, we use the following expression:

```
t :: name
```

Finally, the module *sprdbms* contains all the operations that are needed to define a relational schema (constructors to define collections, to define basic data type values and user-defined types) and to apply OCL queries to any relational model (collection operations, iterators, and user-defined navigation operations). Therefore, the membership equational theory that permits the query of the instances of a specific model is defined by the tuple (S_{OCL}, $OP_{OCL} \cup OP_{QUERY}$, E_{OCL}), where:

$S_{OCL} = S_{SimpleDataType} \cup S_{Collections} \cup S_{CollectionOperations} \cup S_{Iterators} \cup S_{UserDefinedClasses}$

$OP_{OCL} = OP_{SimpleDataTypes} \cup OP_{Collections} \cup OP_{UserDefinedTypes}$

$OP_{QUERY} = OP_{CollectionOperations} \cup OP_{Iterators} \cup OP_{UserDefinedNavigationOperations}$

$E_{OCL} = E_{CollectionOperators} \cup E_{Iterators} \cup E_{UserDefinedNavigationOperations}$

4.2. A relational model as an algebraic term

In this paper, OCL queries are performed over models. The definition of a model can be graphically performed by means of EMF. Then, it can be automatically serialized to a term that represents a set of instances of the classes that constitute the model. The serialization uses the algebraic specification that is generated for a model as mentioned above.

For example, a relational schema that conforms to the Rdbms metamodel (shown in Fig. 3) is serialized as a set (shown in Fig. 4), where all the elements are instances of the classes of the metamodel.

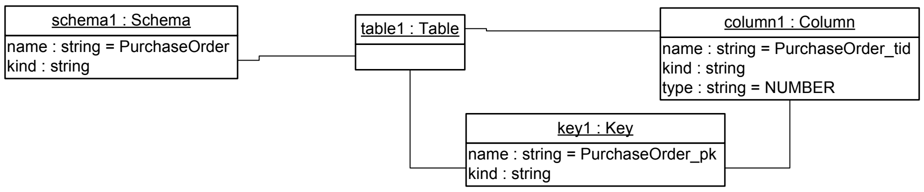

Fig. 3. Object diagram that represents a relational schema that conforms to the Rdbms metamodel defined as a UML model in Fig. 1

```
Set {
    (rdbms-Schema 'schema1 "purchaseOrder" "" OrderedSet { 'table1 } ),
    (rdbms-Table 'table1 "PurchaseOrder" "" OrderedSet { 'key1 } empty-orderedset OrderedSet {
    'column1 } OrderedSet {'schema1 } ),
    (rdbms-Key 'key1 "PurchaseOrder_pk" "" OrderedSet {'table1 } OrderedSet { 'column1 } ),
    (rdbms-Column 'column1 "PurchaseOrder_tid" "" "NUMBER" empty-orderedset OrderedSet { 'key1 }
    OrderedSet {'table1 } )
}
```

Fig. 4. Algebraic representation of the relational schema shown in Fig. 3.

The internal structure of a term is transparent to the user of the algebraic specification due to the navigation operations, which permit the user to navigate in an OCL-like way throughout the roles and attributes of the objects of the model instance.

Therefore, the invariant in Section 2 can be written by defining the body expression of the *forAll* iterator as a body operation. The body operation *inv::body0* defines the body expression of the *forAll* iterator, indicating that the types of the columns of a specific foreign key must be equal to the types of the corresponding columns of the referred primary key. The body operation *inv1* defines the body of the invariant, checking the size of the column collections that belong to the foreign key and to the referred primary key and invoking the forAll operation. The code generated for both body expressions is as follows:

```
var column1 : rdbms-Column . var self : rdbmsNode .

op inv::body0 : -> BoolBody{rdbms} [ctor].
ceq column1 :: inv::body0 ( ? self ; rdbmsModel ) =
    (((((self :: refersTo ( rdbmsModel ) :: column ( rdbmsModel )) -> asSequence) -> at(
        ((self :: column ( rdbmsModel )) -> indexOf(column1))
    )) :: type) == (column1 :: type))
if column1 :: rdbms-Column .
eq column1 :: inv::body0 ( PL ; rdbmsModel ) = false [owise].

op inv1 : -> BoolBody{rdbms} [ctor] .
ceq self :: inv1 ( PL ; rdbmsModel ) =
    (if (((self :: column ( rdbmsModel )) -> size) == ((self :: refersTo ( rdbmsModel ) :: column (
    rdbmsModel )) -> size)) then
        ((self :: column ( rdbmsModel )) -> forAll (inv::body0 ; ? self ; rdbmsModel ))
    else
        false
    fi)
if self :: rdbms-ForeignKey .
eq self :: inv1 ( PL ; rdbmsModel ) = false [owise] .
```

The invariant is checked by selecting the instances of the context class and by applying the invariant body by means of a forAll iterator as follows:
red *rdbms-instance ->* ***select*** *(oclIsTypeOf ; ? "ForeignKey" ; rdbms-instance) ->* ***forAll****(inv1 ; empty-params ; rdbms-instance) .*

5 Related works

Although OCL is not as well supported as UML in some CASE tools, there is a growing interest in providing support for OCL in order to achieve different goals. In [13], several tools that support OCL are studied. Taking them and others into account, some technological examples, which are classified by their main goal, are provided:
- Model transformation: MOMENT, ATL, YATL.
- Model verification: the KeY System.
- Requirements validation: ITP/OCL, the USE tool, the Dresden OCL Toolkit, the Kent OCL tool.
- Code generation (also for requirements validation): Octopus, OCLE.
- OCL Testing: HOL/OCL.

Nevertheless, only a few of them rely on formal methods to provide support for the operational semantics of OCL, and even fewer tools are integrated in (commercial) CASE tools. We focus on some tools that rely on formal methods in this section.

The USE tool [14] provides interactive validation of OCL constraints over a model. This tool reads the input model and the OCL constraints from textual resources, supporting class diagrams, object diagrams and sequence diagrams. Afterwards, objects and links can be graphically created to define a snapshot of a running system. This tool has been extended for the automatic generation of test cases and validation cases [15].

The KeY system [16] provides functionality for formal specification and deductive verification within a commercial CASE tool (Together Control Center). In this approach, the user defines a software artefact in UML that can be annotated with OCL constraints. The OCL constraints are translated into formulas of JavaDL (a dynamic logic for Java [17] that can be reduced by means of an interactive theorem prover.

The ITP/OCL tool [18] provides automatic validation of UML static class diagrams with respect to OCL constraints. It provides an algebraic OCL specification using Maude, where UML class diagrams and object diagrams are formalized by means of algebraic specifications in membership equational logic and where OCL constraints are defined as formulas in membership equational logic theories. A graphical front-end is being developed for the ITP/OCL tool, which permits the definition of class diagrams and the definition of correct object diagrams.

In these last three discussed approaches, only UML diagrams are considered for validating OCL expressions. In the MOMENT-OCL specification, OCL queries can be automatically applied either to metamodels or to models that may be defined in EMF by making use of the Maude parameterization mechanism, following a more automated model-driven oriented approach. In our approach, while Maude is used to execute OCL expressions and to guarantee formal features of the expressions, the

OCL expressions can be applied to graphical model-based software artefacts through the EMF.

6 Conclusions and Further Work

OCL is becoming a de-facto standard for defining constraints and queries in the Model-Driven Engineering field. The number of tools that provide support for this language is growing, and although the operational semantics of OCL is said to be formal, only a few tools rely on formal methods to define its operational semantics.

In this paper, we present an algebraic specification of part of the operational semantics of OCL 2.0 from an implementation point of view. This specification takes advantage of the parameterization mechanism and the attribute theory of Maude and provides a way to simulate collection operations as higher-order functions for the sake of reuse.

This specification is used to perform model queries in the EMF and to represent EMF software artefacts as algebraic specifications or as terms that can be manipulated by means of model management operators [19]. The OCL specification has been developed generically so that it can be used for any kind of metamodels, models or model instances. These operators use OCL queries to specify model transformations that can be run in Maude as detailed in [5]. Therefore, not only can OCL be studied in an algebraic setting, it can also be applied in the well-known modelling environment EMF.

As the syntax that was obtained for OCL in Maude is not complete standard-compliant, we are developing an OCL compiler to Maude code. In this way, the gap between OCL-practitioners and the algebraic specification will be transparent. Further work consists in exploiting the formal features of the OCL specification from a more-theoretical point of view and its application to real case studies.

References

1. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture--Practice and Promise. (2003)
2. Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. John Wiley & Sons (2004)
3. OMG: Meta Object Facility (MOF) 2.0 Core Specification, ptc/04-10-15. (2004)
4. OMG: MOF 2.0 QVT final adopted specification (ptc/05-11-01). (2005)
5. Boronat, A., Carsí, J.A., Ramos, I.: Algebraic Specification of a Model Transformation Engine. In: Proceedings of Fundamental Approaches to Software Engineering, FASE'06, LNCS, Vol. 3922. Springer Vienna, Austria (2006)
6. Bézivin, J., Dupe, G., Jouault, F., Pitette, G., Rougui, J.E.: First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In: Proceedings of OOPSLA 2003 Workshop, Anaheim, California (2003)
7. The Model Transformation Framework: http://www.alphaworks.ibm.com/tech/mtf
8. Warmer, J., Kleppe, A.: The Object Constraint Language, Second Edition, Getting Your Models Ready for MDA. Addison-Wesley (2004)

9. Chiorean, D., Bortes, M., Corutiu, D.: Proposals for a Widespread Use of OCL. In: Proceedings of Proceedings of the MoDELS'05 Conference Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends, Technical Report LGL-REPORT-2005-001, Vol. EPFL, Montego Bay, Jamaica (2005) 68--82
10. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: specification and programming in rewriting logic. Theor. Comput. Sci. **285** (2002) 187-243
11. The MOMENT Project: http://moment.dsic.upv.es
12. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude 2.2 manual and examples. (2005) http://maude.cs.uiuc.edu/maude2-manual/
13. Toval, A., Requena, V., Fernández, J.L.: Emerging OCL tools. Software and System Modeling **2** (2003) 248-261
14. Richters, M.: The USE tool: A UML-based Specification Environment. (2001) http://www.db.informatik.uni-bremen.de/projects/USE/
15. Gogolla, M., Bohling, J., Richters, M.: Validation of UML and OCL Models by Automatic Snapshot Generation. In: Proceedings of UML 2003 - The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference. Springer, San Francisco, CA, USA (2003) 265-279
16. Ahrendt, W., Baar, T., Beckert, B., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Schmitt, P.H.: The KeY System: Integrating Object-Oriented Design and Formal Methods. In: Proceedings of Fundamental Approaches to Software Engineering. 5th International Conference, FASE 20022306. Springer, Grenoble, France (2002) 327-330
17. Beckert, B.: A Dynamic Logic for Java Card. In: Proceedings of Proc. 2nd ECOOP Workshop on Formal Techniques for Java Programs, Cannes, France (2000)
18. Egea, M., Clavel, M.: The ITP/OCL tool. (2006) http://maude.sip.ucm.es/itp/ocl/
19. Boronat, A., Carsí, J.A., Ramos, I.: Automatic Support for Traceability in a Generic Model Management Framework. In: Proceedings of Model Driven Architecture - Foundations and Applications, First European Conference, ECMDA-FA 2005, LNCS, Vol. 3748. Springer Nuremberg, Germany (2005) 316-330