

# An Algebraic Specification of Generic OCL Queries within the Eclipse Modeling Framework♦

Artur Boronat, Joaquín Oriente, Abel Gómez, 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 | joriente | agomez | iramos | pcarsi}@dsic.upv.es

**Abstract.** In the Model-Driven Architecture initiative, software artefacts are represented by means of models that can be manipulated. Such manipulations can be performed by means of transformations and queries. The standard Query/Views/Transformations and the standard language OCL are becoming suitable languages for these purposes. This paper presents an algebraic specification of the operational semantics of part of the OCL 2.0 standard, focusing on queries. This algebraic specification of OCL can be used within the Eclipse Modeling Framework to represent models in an algebraic setting and to perform queries or transformations over software artefacts that can be represented as models: model instances, models, metamodels, etc. In addition, a prototype for executing such OCL queries and invariants over EMF models is presented. This prototype provides a compiler of the OCL standard language that targets an algebraic specification of OCL, which is run on the term rewriting system Maude.

**Keywords:** MDA, OCL queries and invariants, metamodeling, algebraic specification.

## 1 Introduction

Model-Driven Development is a field in Software Engineering that, for several years, has represented 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 constitutes what is called the metamodel of the model.

Interest in this field has grown in software development companies due to several factors. Previous experiences with Model Integrated Computing [1] (where embedded systems are designed and tested by means of models before generating them automatically) have shown that costs decrease in the development process. The consolidation of UML as a design language for software engineers has contributed to

---

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

software Model-Driven Development by means of several CASE tools that permit the definition of UML models and automated code generation. The emergence of important model-driven initiatives such as the Model-Driven Architecture [2], which is supported by OMG, and the Software Factories [3], which is supported by Microsoft, ensures a model-driven technology stock for the near future.

Model-Driven Development has evolved into the Model-Driven Engineering field, where not only design and code generation tasks are involved, but also traceability, model management, metamodeling issues, model interchange and persistence, etc. To fulfil these tasks, model transformations and model queries are relevant tasks that must be solved. In the MDA context, they are dealt with from an open-standard point of view. The standard Meta-Object Facility (MOF) [4] provides a way to define metamodels. The standard proposal Query/Views/Transformations (QVT) [5] will provide support for both transformations and queries. While model transformation technology is being developed [6-8], the Object Constraint Language (OCL) remains as the best choice for queries.

OCL [9] 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, support for model testing and simulation, ontology development and validation for the Semantic Web, among others. 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 proofs, and the execution of OCL expressions as well. This algebraic specification has been developed in the MOMENT framework (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; Section 5 provides the architecture of the prototype; Section 6 presents some related works; Section 7 provides some conclusions and ongoing work.

## **2 The Coach Company Example**

The Meta-Object Facility standard (MOF) [4] 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 modelled a simple coach company in UML. In this design, a coach has a specific number of seats and can be used for regular trips or for private trips. In regular trips, the tickets are bought individually. In private trips, the whole coach is rented for a trip. The model is shown in UML notation in Fig. 1. The example provides a specific UML model, and the queries are applied to its instances. 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.

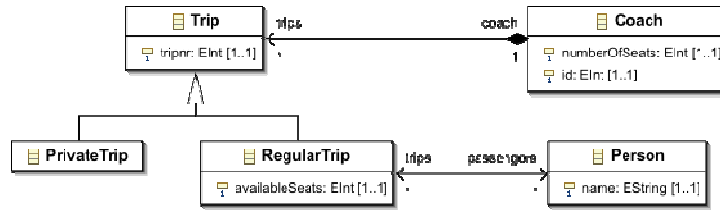


Fig. 1. Coach company model

OCL queries<sup>1</sup> permit a more precise definition of the above model by adding constraints. For instance, we can indicate that overbooking is not allowed in a regular trip by means of the following invariant:

```

context Coach:
inv: self.trips -> select( t: Trip | t.oclIsType(RegularTrip))
-> forAll( r: Trip | r.oclAsType(RegularTrip).passengers -> size()
<= r.coach.numberOfSeats -> sum()

```

### 3 Algebraic specification of generic<sup>2</sup> OCL queries

In this section, we describe the parameterized algebraic specification of OCL that permits the query of either metamodels or UML models. The Maude term rewriting system [10] has been used for this purpose. Maude provides an algebraic specification language that belongs to the OBJ family<sup>3</sup>. Its equational deduction mechanism animates the OCL algebraic specification over a specific model instance, providing the operational semantics for OCL expressions. We have developed a plug-in that embeds the Maude environment into the Eclipse framework so that we can use it for our purposes.

<sup>1</sup> We consider that an invariant is built on an OCL query that returns a Boolean value. Thus, although we talk about invariants, we are also using OCL queries.

<sup>2</sup> In this work, OCL genericity refers to the possibility of reusing the OCL specification for any software artefact that can be represented as a model, including metamodels.

<sup>3</sup> In this paper, we assume some basic knowledge about algebraic specifications and OBJ-like notation. We refer to [12] for more details.

### 3.1 Overview of the parameterized OCL algebraic specification

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:  $\Sigma$  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) that are 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.

OCL collection types and their operations have been defined in a parameterized algebraic specification, called  $OCL-SUPPORT\{X :: TRIV\}$ . Fig. 2 shows the elements involved in the parameter passing mechanism diagram.  $TRIV$  is the algebraic specification of the formal parameter, which is called theory in Maude.

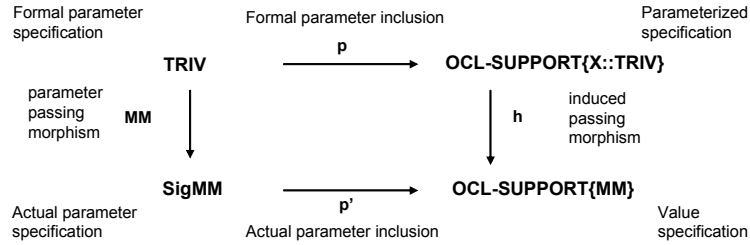


Fig. 2. The parameter passing diagram for the  $OCL-SUPPORT\{X :: TRIV\}$  parameterized module.

$SigMM$  is an algebraic specification that is automatically obtained from a specific metamodel automatically. The  $SigMM$  specification constitutes the actual parameter for the  $OCL-SUPPORT\{X :: TRIV\}$  module and provides a constructor for each type that is defined in the metamodel and an inheritance hierarchy among the types that appear in the metamodel. The  $MM$  view is the morphism that relates the elements of the  $TRIV$  formal parameter to the elements of the  $SigMM$  actual parameter.

OCL collection types and related operations have been generically specified 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  $El$  (referred to as  $XSEl$  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 operations as well.

In Fig. 2,  $p$  and  $p'$  are inclusion morphisms that indicate that the formal parameter specification is included in the parameterized specification, and that the actual parameter specification is included in the value specification, respectively. The  $h$  morphism is the induced passing morphism that relates the elements of the parameterized module to the elements of the  $OCL-SUPPORT\{MM\}$  value specification by using the  $MM$  parameter passing morphism.

### 3.2 Algebraic Specification of OCL Types

Types in OCL are divided into 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 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), substrng(substr), = (==), <> (=/=)

Table 1. OCL and Maude data-type correspondences

#### 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.

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 group of elements that are not ordered by means of the association and the commutativity attributes. The constructor for this sort has the symbol “,” and is associative and commutative. Thus, working with integers, “1, 2, 3” is the term that represents a valid *Magma*{*Int*}. In addition, we can state that “1,2,3” and “3,2,1” represent the same group of elements modulo the commutative and associative attributes.

Instead, the constructor of the sort *OrderedMagma*{*X*} does not have the commutativity property, producing terms that represent ordered concatenations of elements. The constructor for this sort has “::” as symbol and permits building ordered groups of elements by using the common syntax for lists in functional programming. Thus, the term “1 :: 2 :: 3” represents a valid ordered magma of integers, and “1 :: 2 :: 3” is different from “3 :: 2 :: 1” because the constructor “::” is not commutative.

<ol style="list-style-type: none"> <li>1. <i>sort</i> <math>Magma\{X\}</math> <math>OrderedMagma\{X\}</math> .</li> <li>2. <i>subsort</i> <math>XSElt &lt; Magma\{X\}</math> <math>OrderedMagma\{X\}</math> .</li> <li>3. <i>sorts</i> <math>Collection\{X\}</math> <math>Set\{X\}</math> <math>OrderedSet\{X\}</math> .</li> <li>4. <i>subsort</i> <math>Collection\{X\} &lt; XSElt</math> .</li> <li>5. <i>subsorts</i> <math>Set\{X\}</math> <math>OrderedSet\{X\} &lt; Collection\{X\}</math> .</li> <li>6. <i>op</i> <math>_</math> : <math>Magma\{X\}</math> <math>Magma\{X\} \rightarrow Magma\{X\}</math> [<i>assoc comm ctor</i>] .</li> <li>7. <i>op</i> <math>_</math> : <math>Magma\{X\}</math> <math>Magma\{X\} \rightarrow Magma\{X\}</math> [<i>assoc ctor</i>] .</li> <li>8. <i>op</i> <math>Set\{\_}</math> : <math>Magma\{X\} \rightarrow Set\{X\}</math> [<i>ctor</i>] .</li> <li>9. <i>op</i> <i>empty-set</i> : <math>\rightarrow Set\{X\}</math> [<i>ctor</i>] .</li> <li>10. <i>op</i> <math>OrderedSet\{\_}</math> : <math>OrderedMagma\{X\} \rightarrow OrderedSet\{X\}</math> [<i>ctor</i>] .</li> <li>11. <i>op</i> <i>empty-orderedset</i> : <math>\rightarrow OrderedSet\{X\}</math> [<i>ctor</i>] .</li> </ol>
--

**Table 2.** Specification of groups of elements

Terms of the sort  $Magma\{X\}$  are used to define sets (line 8), while terms of the sort  $OrderedMagma\{X\}$  are used in ordered sets (line 10). In Table 2, we show the Maude code that specifies the Set and OrderedSet types. In our specification, collections of collections are allowed by indicating that one collection can be an element of another collection (line 4). 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 9, 11). The types Bag and Sequence have also been specified, similarly to the Set and OrderedSet types, respectively. 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 Loop Operations or Iterators

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 while performing a specific action. In this paper, we focus on the second type of operations.

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 and the expression  $(i \mid i.mod(2) \langle \rangle 0)$  is the body. 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.

Using the example of the selection of odd numbers from an integer set, we study first how to specify the body of the select expression  $i \mid i.mod(2) <> 0$ . Expression bodies can be evaluated to several types depending on the kind of operator in which they are used. For instance, the body expression of a select evaluates to a boolean value. Depending on the return type of the body expression, a symbol is associated to it indicating the name of the body expression. For the example, we obtain:

$op \text{ isOdd} : \rightarrow BoolBody\{Int\} [ctor]$ .

The body expression is built by using the following operation:

$op \_::\_(\_;\_) : Magma\{X\} BoolBody\{X\} ParameterList Collection\{X\} \rightarrow Bool$ .

where the first argument is a term that represents a magma of elements, the second argument is the corresponding body symbol, the third argument is a variant list of parameters that can be empty, and the fourth argument is the whole initial collection to which the first argument belongs. To define a body function, the axioms must be provided by the user in Maude notation. For the example, we define the following equation:

$var \text{ intN} : Int . var \text{ intCol} : Collection\{Int\} . var \text{ PL} : ParameterList .$

$eq \text{ intN} :: \text{ isOdd} ( \text{ PL} ; \text{ intCol} ) = ((\text{intN} \text{ rem } 2) \neq 0) .$

Once the body expression has been defined, we provide an algebraic specification of the operational semantics of the select operation for sets. The different collection operations have been defined as function symbols (terms of the sorts that are shown in Table 3), depending on the return type of each operation. For instance, the select operation, which returns a collection of elements, is defined as follows:

$op \text{ select} : \rightarrow Fun\{X\} [ctor]$ .

	Return type	Collection operator symbols					Iterator symbols
		Collection	Set	OrderedSet	Bag	Sequence	Collection
Fun{X}	Collection	union, flatten, including, excluding, iterate	--, intersection	--, 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, isEmpty, notEmpty					one, forAll, forAll2 <sup>4</sup> , exists, isUnique
IntFun{X}	Integer value	count, size, sum, product		indexOf		indexOf	

**Table 3.** OCL collection operations that have been specified

The operational semantics of iterator operations is defined 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 4). These are the arguments of select: BB is a variable that contains the boolean body expression, 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

<sup>4</sup> The forAll2 operation has been included to provide support when two iterators are being used in the forAll operation.

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.

To invoke an iterator in an OCL-like way, the following operation is used:

```
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 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)* .

```
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 4.** Axiomatic specification of the select operation for sets.

## 4 Algebraic specification of metamodels and models

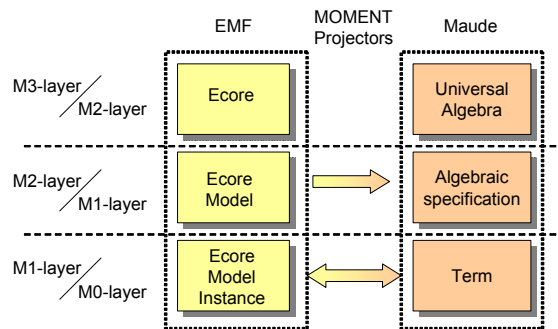
The advantage of OCL is that user-defined types can be used in expressions to perform queries on software artefacts (namely models). User-defined types are the types that can be used in a model: classes, associations, enumerations, and so on. One of the keys to success in the use of the OCL algebraic 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) [13]. 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. Moreover, third-party researchers and developers are bringing new tools to work on ontologies through EMF and graphical Domain Specific Languages. 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<sup>5</sup> 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. 3, 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 would be exactly the same as defining OCL queries over Ecore metamodels.



**Fig. 3.** EMF and Maude conceptual integration

#### 4.1. A model as an algebraic specification

To query a model, 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.

##### 4.1.1 The algebraic signature of a model (SigMM)

An Ecore model 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

<sup>5</sup> 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.

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 *RegularTrip* class in Fig. 1 is used to generate the following Maude code:
 

```
sorts RegularTrip .
op `(RegularTrip _____) : Qid Int Int OrderedSet {QID} OrderedSet {QID} -> RegularTrip [ctor] .
```

 where the first argument is the internal identifier of the instance, the second argument is the inherited *tripnr* attribute, the third argument is the *availableSeats* argument, the fourth argument is the inherited *coach* reference (UML role), and the fifth argument is the *passengers* reference (UML role). This template is only applied to specific classes. 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 *RegularTrip* class is a specialization of the *Trip* class. In the Maude specification, the sort *RegularTrip* is defined as a subsort of the *Trip* 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 *tripNode* (where *trip* 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 *sigtrip*.
- 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 vtrip from TRIV to sigtrip is sort Elt to tripNode . endv
```

#### 4.1.2 The algebraic specification of a model

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

The instantiated module  $OCL-SUPPORT\{MM\}$  is contained in another module, called *sptrip* in the example. This module also contains the operations that are needed to navigate the 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:

$op\_::\_(\_): X\$Elt Fun\{X\} Collection\{X\} \rightarrow Collection\{X\} [ctor]$ .

where  $X\$Elt$  represents an element of the model instance (a class instance),  $Fun\{X\}$  represents the symbol of the role, the  $Collection\{X\}$  argument represents the model instance (which is needed to navigate the model), 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 *coach* of a *RegularTrip* class instance, called *rt*, of the a model instance, called *ModelInstance*, we use the following expression:

$rt::coach(ModelInstance)$

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

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

For example, to obtain the value of the *availableSeats* of a *TripRegular* class instance, called *rt*, we use the following expression:

$rt::availableSeats$

Finally, the module *sptrip* contains all the operations that are needed to define an instance of a model (constructors to define collections, to define basic data type values and user-defined types) and to apply OCL queries to instances of any 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. An instance of a model as an algebraic term

OCL queries are performed over instances of 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, the instance of the model *Trip* (shown in Fig. 4) is serialized as a set, where all the elements are instances of the classes of the model, as follows:

$Set \{ (Coach \text{ 'coach1 1 10 } OrderedSet \{ \text{ 'person1 } \} ),$   
 $(RegularTrip \text{ 'trip1 1 9 } OrderedSet \{ \text{ 'coach1 } \} OrderedSet \{ \text{ 'person1 } \} ),$   
 $(Person \text{ 'person1 "Peter" } OrderedSet \{ \text{ 'trip1 } \} ) \}$

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.

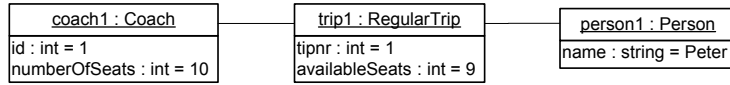


Fig. 4. Object diagram defined as an instance of the model defined in Fig. 1

Therefore, the query that is used in the invariant in Section 2 can be written by defining the body expression of the *forall* iterator as a body operation. This body operation checks that all the regular trips have a lower number of passengers than the number established by the *numberOfSeats* attribute of the corresponding Coach instance. The Maude code that is automatically obtained for the body expression of the *forall* operator in the example is as follows:

```

var self : trip-Trip . var tripModel : Set{trip} .
op notOverbooked : -> BoolBody{trip} [ctor].
ceq self :: notOverbooked ( PL ; tripModel ) =
  (((self :: oclAsType ( ? "RegularTrip" ; tripModel ) :: passengers ( tripModel ))
  -> size) <= ((self :: coach ( tripModel ) :: numberOfSeats) -> sum))
if self :: trip-Trip .
eq self :: notOverbooked ( PL ; tripModel ) = false [owise].
  
```

where *self* is a variable of type *Trip* and *tripModel* is a set that represents the model instance to be queried. The query that is used in the body of the invariant can be coded as follows:

```

red self :: trips ( tripModel ) -> select ( oclIsTypeOf ; "RegularTrip" ; tripModel ) -> forall
(notOverbooked ; empty-params ; tripModel) .
  
```

where *self* is a variable of type *Coach*, *tripModel* is a variable that contains the model instance to be checked, and the *select* and the *forall* operation provide the body expression of the invariant in Maude code by using the *oclIsTypeOf* operator and the above body expression *notOverbooked*.

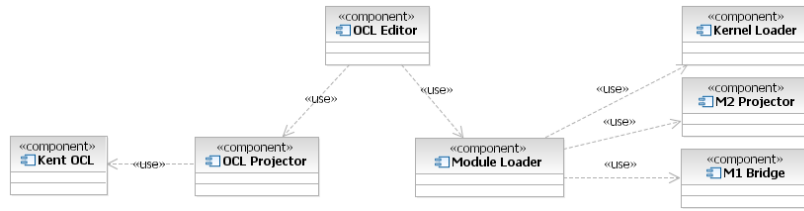
## 5 MOMENT-OCL: a prototype for executing algebraic OCL expressions within the Eclipse Modeling Framework

The OCL algebraic specification that has been presented in the paper permits both the representation of models as sets and the use of queries and invariants over them. This permits the use of OCL expressions in algebraic model transformations, such as those presented in [6]. In addition, we have developed a simple OCL editor that permits the evaluation of OCL queries and invariants over EMF models or model instances. In this section, we provide a brief description of the architecture of this prototype, which is called MOMENT-OCL.

Fig. 6 shows the components of the MOMENT-OCL prototype that permit the execution of algebraic OCL expressions over EMF models:

- The *OCL Projector* component is the module that projects the OCL expression to Maude code. It makes use of the *Kent OCL* library [14] to validate the syntax and the semantics of the expression.

The process of compilation from OCL to Maude follows the typical structure of a language processor. The process is divided in two phases: an initial analysis phase and a second synthesis phase.



**Fig. 5.** MOMENT-OCL Architecture

In the first phase, we have reused the OCL support of the Kent Modelling Framework (KMF) [15], which provides lexical, syntactical and semantical analysis of OCL expressions over an EMF model. KMF analyzes an OCL expression, taking into account the semantics of the model, and produces an Abstract Syntax Tree (AST) to represent the data that is needed in the synthesis phase.

In the second phase, once an OCL expression has been analyzed by KMF correctly, the AST is parsed and Maude code for body expressions, queries and invariants are produced in order to evaluate OCL expressions over EMF models in Maude.

- The *Module Loader* component obtains the algebraic specification from a metamodel, by instantiating the `OCL-SUPPORT{X::TRIV}` module with the signature obtained for a specific metamodel. This algebraic specification is extended with the Maude code obtained from the compilation of OCL expressions by means of the OCL Projector component. The *Module Loader* uses three other components: the M2 Projector, which projects a metamodel  $MM$  (the Coach model in the example) as the signature  $SigMM$ ; the M1 Bridge, which projects a model (model instance in the example) as a term of the corresponding algebraic specification `OCL-SUPPORT{MM}`; and the Kernel Loader, which instantiates the parameterized algebraic specification of OCL with the signature  $SigMM$ , providing the formal environment where OCL expressions for the model  $MM$  can be evaluated.
- The OCL Editor permits the definition of OCL queries and invariants over EMF models and provides syntactical and semantical analyses of the expressions by reusing this functionality from the KMF. It permits the evaluation of queries and invariants. Finally, there is an option to show the generated Maude code which specifies the operational semantics for the OCL expressions for debugging purposes.

The editor (shown in Fig. 7) consists of a tree viewer, a properties view, and a console to show results. The tree viewer is the main window of the application. The information is shown in a hierarchical structure. The first step is to consider a model and its instance specifying the paths of the respective XMI files, where they are defined. The second step consists in specifying different OCL contexts for the model selected in the first step. Finally, we can define groups of queries or invariants for each context. It is possible to evaluate all the invariants defined in a model or only those defined in a specific context. The queries and invariants have two attributes, a name to identify them, which is useful to visualize the result of a group of invariants, and the expression, which can be written in an OCL editor.

If we consider an invariant or query, we can analyze the expression syntactically and semantically, evaluate it by showing the result, or parse it to Maude code, as indicated in Fig. 7.

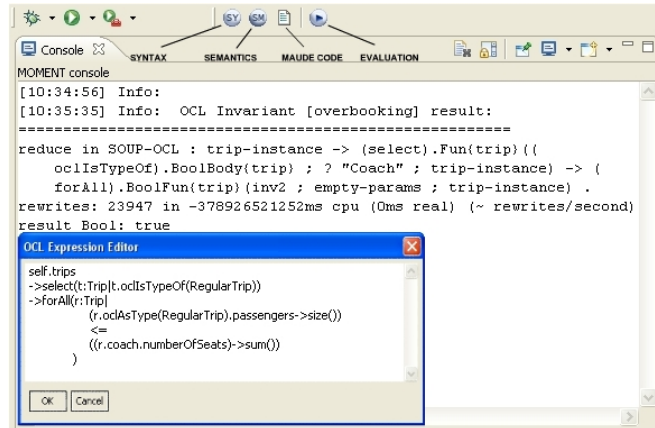


Fig. 6. MOMENT-OCL screenshot

## 6 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 [16], 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 KeY system [17] 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) that can be reduced by means of an interactive theorem prover.

The USE tool [18] 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.

The ITP/OCL tool [19] 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 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, the OCL expressions can be applied to graphical model-based software artefacts through the EMF. Whenever EMF, and related support, is used to develop a (domain specific or UML) modelling environment, we can use MOMENT-OCL to provide invariant checking and query evaluation. Thus, our philosophy does not consist in developing a new modelling environment to provide OCL support, we provide it for other existing modelling approaches. Other java-based approaches that integrate OCL within the EMF are [14, 20, 21], from which we took the Kent library to reuse the analysis phase for the ocl compilation.

Several instances of a model can be easily queried with our tool. Nevertheless, as the main goal of the algebraic specification of OCL in MOMENT-OCL was its use as query and data manipulation language for model transformation, the part of OCL that considers method validation has not been addressed yet.

## 7 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 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 [11]. The OCL specification has been developed generically so that it can be used for any kind of metamodel, model or model instance. These operators use OCL queries to specify model transformations that can be run in Maude as detailed in [6]. Thus, not only can OCL be studied in an algebraic setting, it can also be used in the well-known modelling environment EMF.

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. By now,

MOMENT-OCL permits the analysis, compilation and evaluation of ocl expressions over EMF models in Maude, but the output that is returned by Maude is not still processed. Further work consists in providing such processing, allowing the user to check the invariants that have failed in an evaluation.

## References

1. Sztipanovits, J., Karsai, G.: Model-Integrated Computing. IEEE Computer Society Press 30 (1997) 110-111
2. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture--Practice and Promise. (2003)
3. Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. John Wiley & Sons (2004)
4. OMG: Meta Object Facility (MOF) 2.0 Core Specification, ptc/04-10-15. (2004)
5. OMG: MOF 2.0 QVT final adopted specification (ptc/05-11-01). (2005)
6. 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)
7. 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)
8. The Model Transformation Framework: <http://www.alphaworks.ibm.com/tech/mtf>
9. Warmer, J., Kleppe, A.: The Object Constraint Language, Second Edition, Getting Your Models Ready for MDA. Addison-Wesley (2004)
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. 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
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. The Eclipse Modeling Framework: <http://www.eclipse.org/emf/>
14. Kent Object Constraint Language Library: <http://www.cs.kent.ac.uk/projects/ocl/index.html>
15. Kent Modeling Framework: <http://www.cs.kent.ac.uk/projects/kmf/>
16. Toval, A., Requena, V., Fernández, J.L.: Emerging OCL tools. Software and System Modeling 2 (2003) 248-261
17. 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
18. Richters, M.: The USE tool: A UML-based Specification Environment. (2001) <http://www.db.informatik.uni-bremen.de/projects/USE/>
19. Egea, M., Clavel, M.: The ITP/OCL tool. (2006) <http://maude.sip.ucm.es/itp/ocl/>
20. Vanwormhoudt, G.: EMF OCL Plugin. (2006) <http://www.enic.fr/people/Vanwormhoudt/siteEMFOCL/maven-reports.html>
21. Eclipse Modeling Framework Technologies. (2006) <http://www.eclipse.org/emft/projects/>