Equational Specification of UML+OCL Static Class Diagrams *

Manuel Clavel and Marina Egea

Universidad Complutense de Madrid, Spain

Abstract. In this paper we propose an equational specification of UML+OCL static class diagrams that provides a formal foundation for automatically validating UML object diagrams with respect to OCL constraints. Basically, class and object UML diagrams are specified as membership equational theories, and OCL expressions are represented as terms over extensions of those theories. Then, validating object diagrams with respect to invariants is reduced to checking whether the corresponding terms rewrite to true or false. Based on these ideas, we have developed a tool, named ITP/OCL, that provides automatic validation of object diagrams with respect to OCL constraints.

1 Summary

The Unified Modeling Language (UML) [11] is a general-purpose visual modeling language that is used to specify, visualize, construct, and document the artifacts of a software system. The UML notation is largely based on diagrams. However, for certain aspects of a model, diagrams often do not provide the level of conciseness and expressiveness that a textual language can offer. The Object Constraint Language (OCL) [10] is a textual constraint language with a notational style similar to common object oriented languages. OCL comes to provide help on precise information specification in UML models. Although designed to be a formal language, experience with OCL has shown that the language definition is not precise enough. In this regard, various authors have pointed out language issues related to ambiguities, inconsistencies or open interpretations [14, 15, 7, 9].

Validation and testing in software development have been recognised of key importance for long. There are many different approaches to validation: simulation, rapid prototyping, etc. We *validate* a model by checking whether its instances (also called "snapshots") fulfill the desired constraints. This can lead to several consequences with respect to the design. First, if there are reasonable snapshots that do not fulfill the constraints this may indicate that the constraints are too strong or the model is not adequate in general. On the other hand, constraints may bee too weak, allowing undesirable system states.

A number of CASE tools exists which facilitate drawing and documenting UML diagrams. However, there is little support for validating models during

* Research supported by Spanish MEC Projects TIC2003-01000 and TIN2005-09207-C03-03.

the design stage and generally no substantial support for constraints written in OCL. In this paper we propose an equational specification of UML+OCL static class diagrams that provides a formal foundation for validating UML object diagrams with respect to OCL constraints. This is our current contribution to an effort demanded by many actors in the software modeling community: "The number of modeling directions requesting the use of OCL increases significantly by the day. In these circumstances, the first steps are identifying the reasons of the unsatisfactory state of facts that persists in the OCL tool world and proposing reasonable solutions. A clear, unequivocal and complete language specification is among the preconditions for conceiving and implementing the OCL tools required by real-world projects" [4]. When designing our specification we did not look *only* for just another *clear*, *unequivocal* specification of the basic UML graphical modeling elements and of the usual OCL textual constraint constructors. Indeed, we set ourselves two additional goals:

1. To provide an *executable* specification that could be used for the automatic validation of object diagrams with respect to the invariants constraining their class diagrams.
2. To provide a *practical* specification that could serve as a firm, mathematical basis for industrial-strength software modeling tools, without requiring from the part of the users additional knowledge beyond the UML and OCL specification languages.

The results presented in this paper can be summarized as follows:

1. We provide three maps, $QueryTheory$, $QueryTerm$, and $AsTerm$, such that, if e is an OCL expression whose value over an object diagram $\mathcal{OD}$ is v, then

$$QueryTheory(\mathcal{OD}, e) \vdash QueryTerm(e) = AsTerm(v)\,,$$

   where $QueryTheory(\mathcal{OD}, e)$ is a membership equational theory, and $QueryTerm(e)$ and $AsTerm(v)$ are terms that can be proved equal in that theory using standard term rewriting techniques.[1] [2]
2. The three maps are implementable so that both the theory $QueryTheory(\mathcal{OD}, e)$ and the terms $QueryTerm(e)$ and $AsTerm(v)$ can be automatically generated from an appropriate textual description of the object diagram $\mathcal{OD}$ and from the OCL expressions e and v.

Although we do not cover yet all the standard UML modeling elements and OCL constraint constructors, based on the current results we have developed a

[1] Two comments about our claim. On the one hand, in order to prove it, we need first an official, formal, and complete language specification for UML and OCL, something that is still missing. On the other hand, we like to consider our equational specification of UML+OCL static class diagrams as a first step to fill this gap.

[2] By construction the module $QueryTheory(\mathcal{OD}, e)$ is Church-Rosser and terminating. The proof of this property is out of the scope of this paper. However, we hope to provide enough information about the construction of $QueryTheory(\mathcal{OD}, e)$ so as to justify our claim.

tool, named ITP/OCL [6], that provides automatic validation of object diagrams with respect to non-trivial OCL constraints. It is written entirely in Maude [5] making extensive use of its reflective capabilities to implement the user interface, thanks to which the tool's underlying equational semantics remains hidden to the user who only must be familiar with the standard notions of UML diagrams and OCL invariants. The latest version of the tool, with the available documentation and examples can be found at `http://maude.sip.ucm.es/itp/ocl/`. Finally, in the Appendix we show a screenshot of the Visual ITP/OCL, a Java graphical interface that is being developed as a front-end for the ITP/OCL tool.

Organization First, in Sections 2 and 3 we provide background material on UML+OCL static class and object diagrams and on membership equational logic. Then, in Sections 4, 5, and 6, we propose an equational specification of UML class and object diagrams as theories, and a representation of OCL queries (and invariants) as terms in extensions of those theories. The maps *QueryTheory* and *QueryTerm* are incrementally defined throughout Sections 4, 5, and 6. Finally, we report on related work and draw conclusions.

2 UML+OCL Static Class and Object Diagrams

The UML *static view* models concepts in the application domain as well as internal concepts invented as part of the implementation of an application. It does not describe the time-dependent behaviour of the system, which is described in other views. Key elements in the static view are classes and their relationships, which can be of different kinds, including association and generalization. The static view is displayed in class diagrams. A *class diagram* has the following components: a set of classes, a set of attributes for each class, a set of operations for each class, and a set of relationships between classes.

Example 1. Consider the class diagram TRAINWAGON shown in Figure 1. It models an example from a railway context. A train may own wagons, and wagons may be connected to other wagons (their predecessor and successor wagons). Wagons can be either smoking or non-smoking.

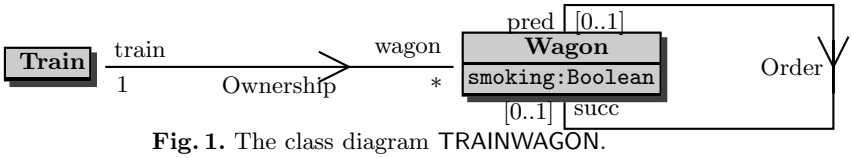

Fig. 1. The class diagram TRAINWAGON.

A system may be in different states as it changes over time. An *object diagram* models the objects and links that represent the state of a system at a particular moment. An *object* is an instance of a class. A *link* is an instance of an association. An object diagram is primarily a tool for research and testing. It can

be used to understand a problem by documenting examples from the problem domain. It can also be used during analysis and design to verify the accuracy of class diagrams.

Example 2. Consider now the object diagram TRAINWAGON-1 shown in Figure 2. It describes a snapshot of the railway system modeled by the class diagram TRAINWAGON, although possibly an "undesired" one since it describes a train with two wagons linked in a cyclic way!.

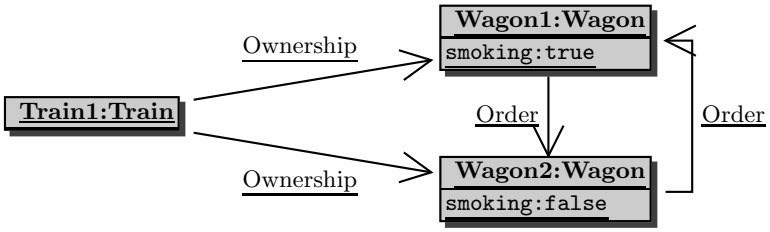

Fig. 2. The object diagram TRAINWAGON-1.

The language OCL is a pure specification language on top of UML. It is a textual language with a notational style similar to common object oriented languages. OCL defines a number of operations on the UML predefined types. For example, for the type Boolean, it includes the operations and and not. Each OCL expression is written in the context of a UML class diagram. Classes from the UML class diagram are also types in the OCL expressions that are attached to the model. The value of a property for an object is accessed by a dot (.) followed by the name of the property. Starting from a specific object, we can also navigate an association to refer to other objects and their properties, by using the opposite role. In OCL it is possible to use features defined on the classes themselves. A predefined feature on classes is allInstances which results in the collection of all instances of the class. Collections, like sets, ordered sets, bags, and sequences are predefined types in OCL. They have a large number of predefined operations on them. For example, the operation includes which checks whether an element belongs to a collection. The iterator exists is just another example. Many times one needs to know whether there is at least one element in a collection for which a constraint holds. The exists iterator in OCL allows you to specify a Boolean expression which must hold for at least one object in a collection.

$collection \rightarrow \mathsf{exists}(x : z \mid boolean\text{-}expression\text{-}with\text{-}x : z)$

The result is true if and only if *boolean-expression-with-*$x : z$ is true for at least one element of *collection*.

Example 3. Consider the following constraint over the class diagram TRAINWAGON: *There do not exist two different wagons linked to each other in a cyclic*

way. This constraint can be expressed using OCL as the following invariant notInCyclicWay over the class diagram TRAINWAGON:

```
not ((Wagon.allInstances)
    →exists(w1:Wagon | (Wagon.allInstances) →exists(w2:Wagon |
            w1:Wagon <> w2:Wagon
            and (w1:Wagon.succ) →includes(w2:Wagon)
            and (w2:Wagon.succ) →includes(w1:Wagon)))) .
```

The object diagram TRAINWAGON-1 does not satisfy this constraint, since there exists two wagons, namely Wagon1 and Wagon2, such that the successors of Wagon1 include Wagon2, and the successors of Wagon2 include Wagon1.

3 Membership Equational Logic

Membership equational logic (MEL) is an expressive version of equational logic; a full account of its syntax and semantics can be found in [3]. A *signature* in MEL is a triple $\Omega = (K, \Sigma, S)$, with K a set of *kinds*, Σ a many-kinded signature, and S a pairwise disjoint K-kinded family of sets of *sorts*. The basic intuition is that correct or well-behaved terms are those that can be proved to have a sort, whereas error or undefined terms are terms that have a kind but do not have a sort. For example, we can declare a kind `Class` for terms representing arbitrary objects, with two sorts `Train` and `Wagon`, for terms representing, respectively, trains and wagons. We can also declare a kind `ClassCol` for terms representing collections of arbitrary objects, with two sorts `TrainCol` and `WagonCol` for representing, respectively, collections of trains and collections of wagons. Then, assuming that `col` and `nil` are the operators for building collections, the term `col(Wagon1, col(Wagon2, nil))` will be a term of the kind `ClassCol` with sort `WagonCol`, while the term `col(Train1, col(Wagon2, nil))` will be a term of the kind `ClassCol` but with no sorts.[3]

The atomic formulas of MEL are either *equations* $t = t'$, where t and t' are terms of the same kind, or *membership assertions* of the form $t : s$, where the term t has kind k and $s \in S_k$. Sentences are Horn clauses on these atomic formulas, i.e., sentences of the form $\forall\{\boldsymbol{x}\}\, A_0$ *if* $A_1 \wedge \ldots \wedge A_n$, where each A_i is either an equation or a membership assertion. A theory (in other contexts called "theory presentation") is a pair (Ω, E), where E is a finite set of sentences in MEL over the signature Ω. For example, our theory for collections of trains and wagons will contain a conditional membership axiom that states that any term `col`$(X,\ XC)$ belongs to the sort `TrainCol` if X is of the sort `Train` and XC is of the sort `TrainCol`, with X a variable of the kind `Class` and XC a variable of the kind `ClassCol`. MEL inference system extends equational logic with rules for handling sort-memberships.

[3] The distinction between kinds and sorts is introduced in MEL to handle partiality. Although the example does not show the expressiveness of this formalism (better examples are data structures with partial constructors like priority queues, sorted lists, etc.), it serves a purpose in the context of this paper: to introduce the basic features of MEL and its use in our specification of UML+OCL static diagrams.

4 UML Class Diagrams

In this section we define a map *ClassTheory* from class diagrams to membership equational theories. For the sake of simplification, we do not consider here enumeration classes or association classes, neither we consider *query* class operations (that is, operations that do not change the state of the system) or multi-valued class attributes: they are specified in a similar fashion. Non-query operations are a different story: we discuss this issue in the concluding section. Finally, n-ary associations can be reduced to binary ones.

Let $\mathcal{CD}$ be a class diagram. Then, *ClassTheory*($\mathcal{CD}$) is defined as follows:

- We specify, using (conditional) membership and equational axioms, the UML predefined types Boolean, Integer, and String, with their operations, as the sorts `Boolean` (of the kind `Boolean?`), `Integer` (of the kind `Integer?`), and `String` (of the kind `String?`), with their operations.
- We introduce the kinds `Class` and `ClassCol` to contain the terms that represent, respectively, objects and collections of arbitrary objects;
- We introduce the operator `col`, and the constant `nil` to represent collections of arbitrary objects;
- For each class c, we introduce a sort c of the kind `Class` to contain the terms that represent the objects of the class c, and a sort c+`Col` (that is, the concatenation of c and `Col`) of the kind `ClassCol` to contain the terms that represent the collections of objects of the class c;
- For each class c, and each attribute at of c of type Boolean (respectively, Integer and String), we introduce an operator at and declare, using a conditional membership axiom, that $at(X)$ always returns a term of sort Boolean (respectively, Integer and String) when X is a term of sort c;
- For each generalization between classes c (subclass) and c' (superclass), we declare, using a conditional membership axiom, that X always has sort c' if it has sort c;
- For each pair of classes (not necessarily different) c and c', and each association, with roles rl and rl', between c and c', we introduce two operators rl and rl', and declare, using conditional membership axioms, that $rl(X)$ always returns a term of sort c+`Col` when X is a term of sort c', and that $rl'(X)$ always returns a term of sort c'+`Col` when X is a term of sort c.

Notice that in the specification of class diagrams, the sorts representing the classes are empty and the operators representing the attributes and roles are undefined. In the next section we propose a specification of object diagrams as extensions —sorts representing classes are filled with terms representing objects, and operators representing attributes and roles are defined for those terms— of the specifications of their class diagrams.

Example 4. We show in Figure 3 the specification `TRAINWAGON` corresponding to the theory *ClassTheory*(TRAINWAGON); we have omitted, however, the specification of the predefined types Boolean, Integer, and String. The presentation of `TRAINWAGON` follows the syntactical conventions of the Maude [5] language (an

efficient implementation of MEL), except for the fact that kinds are declared explicitely using the keyword `kind` along with their associated sorts (enclosed in square brackets). As in Maude, operator declarations start with the keyword `op` (or `ops` for operators with the same rank) and are followed by the operator's name, a list with the arguments' kinds, and the result's kind; membership assertions are introduced with the keyword `mb`, or `cmb` for conditional ones; and unconditional and conditional equations are similarly presented using the keywords `eq` and `ceq`.

Let X and XC be variables, respectively, of the kinds `Class` and `ClassCol`.

```
spec TRAINWAGON is
 kind Class = [Wagon, Train] .
 kind ClassCol = [WagonCol, TrainCol] .
 op nil : -> ClassCol .
 op col : Class ClassCol -> ClassCol .
 mb nil : WagonCol .
 cmb col(X, XC) : WagonCol if X : Wagon /\ XC : WagonCol .
 mb nil : TrainCol .
 cmb col(X, XC) : TrainCol if X : Train /\ XC : TrainCol .
 ops train wagon pred succ : Class -> ClassCol .
 cmb train(X) : TrainCol if X : Wagon .
 cmb wagon(X) : WagonCol if X : Train .
 cmb pred(X) : WagonCol if X : Wagon .
 cmb succ(X) : WagonCol if X : Wagon .
endspec
```

Fig. 3. The theory *ClassTheory*(TRAINWAGON).

5 UML Object Diagrams

In this section we define a map *ObjectTheory* from object diagrams to membership equational theories. The relation between class diagrams and object diagrams is reflected by the fact that object diagram theories extend their corresponding class diagram theories by instantiating the sorts representing their classes and the operators representing their attributes and roles.

Let $\mathcal{OD}$ be an object diagram of a class diagram $\mathcal{CD}$. Then, *ObjectTheory*($\mathcal{OD}$) is defined as the following extension of *ClassTheory*($\mathcal{CD}$):

- For each object o, if c is the class of the object o, we introduce a constant o of the kind `Class`, and declare, using a membership equational axiom, that the constant o has sort `c`;

– For each object o, if v is the value of the attribute at for the object o, then we declare, using an equational axiom, that the operator at returns the term that represents the value v when applied to o;
– For each object o, if $\{o'_1, \ldots, o'_n\}$ is the collection of objects linked to o under an association with roles rl and rl', we declare, using an equational axiom, that the operator rl' returns the term that represent the collection $\{o'_1, \ldots, o'_n\}$ when applied to o.

Example 5. We show in Figure 4 the specification `TRAINWAGON-1` corresponding to the theory *ObjectTheory*(TRAINWAGON-1). We use the keyword `extending` to declare that `TRAINWAGON-1` contains `TRAINWAGON`.

```
spec TRAINWAGON-1 is
 extending TRAINWAGON .
 op Train1 Wagon1 Wagon2 : -> Class .
 mb Train1 : Train .
 mb Wagon1 : Wagon .
 mb Wagon2 : Wagon .
 eq smoking(Wagon1) = true .
 eq smoking(Wagon2) = false .
 eq wagon(Train1) = col(Wagon1, col(Wagon2, nil)) .
 eq train(Wagon1) = col(Train1, nil) .
 eq pred(Wagon1) = nil .
 eq succ(Wagon1) = col(Wagon2, nil) .
 eq train(Wagon2) = col(Train1, nil) .
 eq pred(Wagon2) = col(Wagon1, nil) .
 eq succ(Wagon2) = col(Wagon1, nil) .
endspec
```

Fig. 4. The theory *ObjectTheory*(TRAINWAGON-1).

6 OCL Queries

In what follows, we call *iterating expressions* those of the form $(s \rightarrow \Upsilon(x : z \mid e))$, where Υ is an OCL iterator, like forAll, exists, collect, reject, or select; we call the variable $x : z$ the *iterator-variable* and the expression e the *body* of the iterating expression. We call *basic expressions* those that do not contain any iterating expressions. When the body of an iterating expression contains variables different from its iterator-variable, we say that the iterating expression is *parametarized* by those variables. For example, the innermost iterating expression in the invariant nonInCyclicWay in Example 3 is parameterized by the variable (w1:Wagon. Finally, we call *values* of an object diagram to the elements of a predefined type, the objects of a class, or the collection of elements of a predefined type or of the objects of a class.

6.1 An evaluator for basic OCL queries

In this section we define a map, *QueryBasicTheory*, from object diagrams to membership equational logic, which, informally, generates the equational specification of the OCL non-iterating operators over the models described by the given object diagrams. We also define a map *QueryBasicTerm* that, informally, translate OCL non-iterating expressions into terms in the signature of the theories generated by *QueryBasicTheory*. More formally, let e be a basic OCL expression over a class diagram $\mathcal{CD}$, and let $\mathcal{OD}$ be an object diagram of $\mathcal{CD}$. The two maps *QueryBasicTheory* and *QueryBasicTerm* are such that, if e evaluates to a value v in the object diagram $\mathcal{OD}$, then

$$QueryBasicTheory(\mathcal{OD}) \vdash QueryBasicTerm(e) = AsTerm(v)\,,$$

where $AsTerm(v)$ is the term that represents v in $QueryBasicTheory(\mathcal{OD})$, namely, objects are represented as constants, and collections are represented using col and nil.

We define the theory $QueryBasicTheory(\mathcal{OD})$ as the following extension of $ObjectTheory(\mathcal{CD})$:

- We introduce the kinds BooleanCol?, IntegerCol?, and StringCol?, with sorts BooleanCol, IntegerCol, and StringCol, to contain, respectively, the terms that represent collections of booleans, integers and strings;
- We introduce the (overloaded) operator col and the (overloaded) constant nil, to represent collections of booleans; and similarly, for the integers and the strings;
- We declare, using membership axioms, that nil is a term of sort BooleanCol, and that col(X, XC) is a term of sort BooleanCol, when X is a term of sort Boolean and XC is a term of sort BooleanCol; similarly, for the integers and the strings;
- We declare operators that specify the non-iterating operators over booleans, integers, strings, and collections of objects (like size, includes, asSet, and so on), and we define, using (conditional) membership and equational axioms, their semantics;
- We introduce the kind ClassId to contain the terms that represent class names;
- We introduce the operator allInstances;
- For each class c in the class diagram, we introduce a constant c of the kind ClassId; we declare, using a membership axiom, that allInstances(c) always return a term of sort c+Col, that is, a term representing a collection of objects of the class c; and we define, using an equational axiom, that allInstances(c) is equal to the collection of objects of the class c.

Example 6. We show in Figure 5 a fragment of the specification `QUERY-BASIC-TRAINWAGON-1` corresponding to the theory *QueryBasicTheory*(TRAINWAGON-1), in which we specify the operators includes and isEmpty for collections of arbitrary objects, and the operator allInstances for the classes Train and Wagon.

Let X, X' be variables of the kind `Class` and let XC be a variable of the kind `ClassCol`.

```
spec QUERY-BASIC-TRAINWAGON-1 is
 extending TRAINWAGON-1 .
 op isEmpty : ClassCol -> Boolean? .
 cmb isEmpty(XC) : Boolean if XC : TrainCol.
 cmb isEmpty(XC): Boolean if XC : WagonCol.
 eq isEmpty(nil) = true .
 eq isEmpty(cons(X,XC)) = false .
 op includes : ClassCol Class -> Boolean? .
 cmb includes(XC, X) : Boolean if XC : TrainCol ∧ X : Train .
 cmb includes(XC, X) : Boolean if XC : WagonCol ∧ X : Wagon .
 eq includes(nil, X) = true .
 ceq includes(cons(X, XC), X') = true
       if equal(X, X') = true.
 ceq includes(cons(X, XC), X') = includes(XC, X')
       if equal(X, X') = false.
 op Train : -> ClassId .
 op Wagon : -> ClassId .
 op allInstances : ClassId -> ClassId .
 mb allInstances(Wagon) : WagonCol .
 mb allInstances(Train) : TrainCol .
 eq allInstances(Wagon) = cons(Wagon1, cons(Wagon2, nil)) .
 eq allInstances(Train) = cons(Train1, nil) .
```

Fig. 5. A fragment of the theory *QueryBasicTheory*(TRAINWAGON-1).

We define the map *QueryBasicTerm* recursively over the structure of well-typed OCL expressions. We have grouped the clauses according to the type of the expression; in each group, the cases that are not covered are treated in a similar fashion. Let $Kind(x\!:\!z)$ denote the kind corresponding to the type z (that is, `Boolean?` for Boolean, `Class` for Wagon and Train, `ClassCol` for the types of the collections of objects of a class, and so on). Also, let $AsVar$ be a map that generates, for each expression $x : z$, a unique variable of the kind $Kind(x : z)$. Finally, let e, e_1, e_2, s be arbitrary expressions. Then,

Boolean expressions

$$\begin{aligned}
QueryBasicTerm(e) &\triangleq asTerm(e), \quad \text{if } e \in \{\mathsf{true}, \mathsf{false}\}.\\
QueryBasicTerm(\mathsf{not}\, e) &\triangleq \mathtt{not}(QueryBasicTerm(e))\,.\\
QueryBasicTerm(e_1 \,\mathsf{and}\, e_2) &\triangleq QueryBasicTerm(e_1)\,\mathsf{and}\,QueryBasicTerm(e_2)\,.\\
QueryBasicTerm(e_1 = e_2) &\triangleq \mathtt{equal}(QueryBasicTerm(e_1)\,, QueryBasicTerm(e_2))\,.\\
QueryBasicTerm(e_1 < e_2) &\triangleq QueryBasicTerm(e_1) < QueryBasicTerm(e_2)\,.\\
QueryBasicTerm(s \rightarrow \mathsf{includes}(e)) &\triangleq \mathtt{includes}(QueryBasicTerm(s)\,, QueryBasicTerm(e))\,.\\
QueryBasicTerm(s.\mathsf{isEmpty}()) &\triangleq \mathtt{isEmpty}(QueryBasicTerm(s))\,.\\
QueryBasicTerm(e.at) &\triangleq at(QueryBasicTerm(e)), \quad \text{if } at \text{ is a Boolean attribute.}
\end{aligned}$$

$$
\begin{aligned}
&\underline{\textsf{Integer} \text{ expressions}} \\
\mathit{QueryBasicTerm}(e) &\triangleq \mathit{asTerm}(e), \quad \text{if } e \text{ is an integer number.} \\
\mathit{QueryBasicTerm}(e_1 + e_2) &\triangleq \texttt{add}(\mathit{QueryBasicTerm}(e_1)\,, \mathit{QueryBasicTerm}(e_2))\,. \\
\mathit{QueryBasicTerm}(e_1.\textsf{max}(e_2)) &\triangleq \texttt{max}(\mathit{QueryBasicTerm}(e_1)\,, \mathit{QueryBasicTerm}(e_2))\,. \\
\mathit{QueryBasicTerm}(e.at) &\triangleq at(\mathit{QueryBasicTerm}(e)), \quad \text{if } at \text{ is an } \textsf{Integer} \text{ attribute.} \\
\mathit{QueryBasicTerm}(s \rightarrow \textsf{size}()) &\triangleq \texttt{size}(\mathit{QueryBasicTerm}(s))\,. \\
&\underline{\textsf{String} \text{ expressions}} \\
\mathit{QueryBasicTerm}(e) &\triangleq \mathit{asTerm}(e), \quad \text{if } e \text{ is a string.} \\
\mathit{QueryBasicTerm}(e_1.\textsf{concat}(e_2)) &\triangleq \texttt{concat}(\mathit{QueryBasicTerm}(e_1)\,, \mathit{QueryBasicTerm}(e_2))\,. \\
\mathit{QueryBasicTerm}(e.at) &\triangleq at(\mathit{QueryBasicTerm}(e)), \quad \text{if } at \text{ is a } \textsf{String} \text{ attribute.} \\
&\underline{\text{Collection expressions}} \\
\mathit{QueryBasicTerm}(id.\textsf{allInstances}) &\triangleq \texttt{allInstances}(\mathit{asCns}(id)), \quad \text{where } id \text{ is a class identifier.} \\
\mathit{QueryBasicTerm}(e.rl) &\triangleq rl(\mathit{QueryBasicTerm}(e)), \quad \text{if } rl \text{ is a role.} \\
&\underline{\text{Variables}} \\
\mathit{QueryBasicTerm}(x\!:\!z) &\triangleq \mathit{AsVar}(x\!:\!z)\,.
\end{aligned}
$$

Example 7. Consider the following expression over the object diagram TRAINWAGON-1:

(Wagon.allInstances) →includes(Wagon1)

Its value in the object diagram is true. *QueryBasicTerm* maps this expression to the term:

`includes(allInstances(Wagon), Wagon1)` ,

which is equal to `true` in $\mathit{QueryBasicTheory}$(TRAINWAGON1).

6.2 An evaluator for OCL queries

In this section we define two maps, *QueryTheory* and *QueryTerm*, that extend the maps *QueryBasicTheory* and *QueryBasicTerm* to cover also the specification of OCL iterating operators over the models described by the given object diagrams. Informally, *QueryTheory* is an extension of *QueryBasicTheory* in which, for each iterating expression $(s \rightarrow \Upsilon(x\!:\!z \mid e'))$ occurring in a given OCL expression e, we introduce an operator f and declare, using equational axioms, how this operator iterates over collections of values of type z. Then, $\mathit{QueryTerm}(e)$ is the result of mapping e with an extension of *QueryBasicTerm* in which each iterating expression $(s \rightarrow \Upsilon(x : z \mid e'))$ occurring in e, is mapped to the term $f(\mathit{QueryTerm}(s))$. More formally, let e be an OCL expression over a class diagram $\mathcal{CD}$, and let $\mathcal{OD}$ be an object diagram of $\mathcal{CD}$. The two maps *QueryTheory* and *QueryTerm* are such that if e evaluates to a value v in the object diagram $\mathcal{OD}$, then

$$\mathit{QueryTheory}(\mathcal{OD}, e) \vdash \mathit{QueryTerm}(e) = \mathit{AsTerm}(v)\,.$$

The maps *QueryTheory* and *QueryTerm* are formally defined in terms of an auxiliary map *QueryAux* from pairs $\langle e, \boldsymbol{x}:\boldsymbol{z}\rangle$, where $\boldsymbol{x}:\boldsymbol{z}$ denotes a list $(\boldsymbol{x_1}:\boldsymbol{z_1}, \ldots, \boldsymbol{x_n}:\boldsymbol{z_n})$ of iterating variables, to triples $\langle t, ops, eqs\rangle$, where *ops* are the operators for the iterating expression occurring in e and *eqs* are the equations that declare how these operators iterate over collections. Thus, $QueryTheory(\mathcal{OD}, e)$ is the theory that results from adding *ops* and *eqs* to the theory $QueryBasicTheory(\mathcal{OD})$, and $QueryTerm(e)$ is the term t.

The auxiliary map *QueryAux* is defined recursively over the structure of well-typed OCL expressions. To simplify the presentation, we use three auxiliary maps defined as follows: if $QueryAux(\langle e, \boldsymbol{x}:\boldsymbol{z}\rangle) = \langle t, ops, eqs\rangle$, then,

$$\begin{aligned} QueryAuxTerm(\langle e, \boldsymbol{x}:\boldsymbol{z}\rangle) &\triangleq t, \\ QueryAuxOps(\langle e, \boldsymbol{x}:\boldsymbol{z}\rangle) &\triangleq ops, \text{and} \\ QueryAuxEqs(\langle e, \boldsymbol{x}:\boldsymbol{z}\rangle) &\triangleq eqs. \end{aligned}$$

For non-iterating expressions, the definition of *QueryAux* follows the definition of *QueryBasicTerm*. Notice that non-iterating expressions may contain, in general, iterating subexpressions. We show below the clauses defining *QueryAux* for the Boolean expressions considered in our definition of the map *QueryBasicTerm*. Let e, e_1, e_2, and s be arbitrary expressions, and let $\boldsymbol{x}:\boldsymbol{z}$ be a list of parameters. Then,

$$\begin{aligned}
QueryAux(\langle e, \boldsymbol{x}:\boldsymbol{z}\rangle) &\triangleq \langle AsTerm(e), \emptyset, \emptyset\rangle, \quad \text{if } e \in \{\mathsf{true}, \mathsf{false}\}. \\
QueryAux(\langle x:\mathsf{Boolean}, \boldsymbol{x}:\boldsymbol{z}\rangle) &\triangleq \langle AsVar(x:\mathsf{Boolean}), \emptyset, \emptyset\rangle\,. \\
QueryAux(\langle \mathsf{not}\, e, \boldsymbol{x}:\boldsymbol{z}\rangle) &\triangleq \langle \texttt{not}(QueryAuxTerm(\langle e, \boldsymbol{x}:\boldsymbol{z}\rangle)), \\
&\quad QueryAuxOps(\langle e, \boldsymbol{x}:\boldsymbol{z}\rangle), QueryAuxEqs(\langle e, \boldsymbol{x}:\boldsymbol{z}\rangle)\rangle\,. \\
QueryAux(\langle e_1 \,\mathsf{and}\, e_2, \boldsymbol{x}:\boldsymbol{z}\rangle) &\triangleq \langle QueryAuxTerm(\langle e_1, \boldsymbol{x}:\boldsymbol{z}\rangle)\ \mathsf{and}\ QueryAuxTerm(\langle e_2, \boldsymbol{x}:\boldsymbol{z}\rangle), \\
&\quad \bigcup_{i=1,2} QueryAuxOps(\langle e_i, \boldsymbol{x}:\boldsymbol{z}\rangle), \bigcup_{i=1,2} QueryAuxEqs(\langle e_i, \boldsymbol{x}:\boldsymbol{z}\rangle)\rangle\,. \\
QueryAux(\langle e_1 = e_2, \boldsymbol{x}:\boldsymbol{z}\rangle) &\triangleq \langle \texttt{equal}(QueryAuxTerm(\langle e_1, \boldsymbol{x}:\boldsymbol{z}\rangle), QueryAuxTerm(\langle e_2, \boldsymbol{x}:\boldsymbol{z}\rangle)), \\
&\quad \bigcup_{i=1,2} QueryAuxOps(\langle e_i, \boldsymbol{x}:\boldsymbol{z}\rangle), \bigcup_{i=1,2} QueryAuxEqs(\langle e_i, \boldsymbol{x}:\boldsymbol{z}\rangle)\rangle\,. \\
QueryAux(\langle e_1 < e_2, \boldsymbol{x}:\boldsymbol{z}\rangle) &\triangleq \langle QueryAuxTerm(\langle e_1, \boldsymbol{x}:\boldsymbol{z}\rangle) < QueryAuxTerm(\langle e_2, \boldsymbol{x}:\boldsymbol{z}\rangle), \\
&\quad \bigcup_{i=1,2} QueryAuxOps(\langle e_i, \boldsymbol{x}:\boldsymbol{z}\rangle), \bigcup_{i=1,2} QueryAuxEqs(\langle e_i, \boldsymbol{x}:\boldsymbol{z}\rangle)\rangle\,. \\
QueryAux(\langle s \rightarrow \mathsf{includes}(e), \boldsymbol{x}:\boldsymbol{z}\rangle) &\triangleq \langle \texttt{includes}(QueryAuxTerm(\langle s, \boldsymbol{x}:\boldsymbol{z}\rangle), QueryAuxTerm(\langle e, \boldsymbol{x}:\boldsymbol{z}\rangle)), \\
&\quad QueryAuxOps(\langle s, \boldsymbol{x}:\boldsymbol{z}\rangle) \cup QueryAuxOps(\langle e, \boldsymbol{x}:\boldsymbol{z}\rangle), \\
&\quad QueryAuxEqs(\langle s, \boldsymbol{x}:\boldsymbol{z}\rangle) \cup QueryAuxEqs(\langle e, \boldsymbol{x}:\boldsymbol{z}\rangle)\rangle\,. \\
QueryAux(\langle s.\mathsf{isEmpty}(), \boldsymbol{x}:\boldsymbol{z}\rangle) &\triangleq \langle \texttt{isEmpty}(QueryAuxTerm(\langle s, \boldsymbol{x}:\boldsymbol{z}\rangle)), \\
&\quad QueryAuxOps(\langle s, \boldsymbol{x}:\boldsymbol{z}\rangle), QueryAuxEqs(\langle s, \boldsymbol{x}:\boldsymbol{z}\rangle)\rangle\,. \\
QueryAux(\langle e.at, \boldsymbol{x}:\boldsymbol{z}\rangle) &\triangleq \langle at(QueryAuxTerm(\langle s, \boldsymbol{x}:\boldsymbol{z}\rangle)), \\
&\quad QueryAuxOps(\langle e, \boldsymbol{x}:\boldsymbol{z}\rangle), QueryAuxEqs(\langle e, \boldsymbol{x}:\boldsymbol{z}\rangle)\rangle, \\
&\quad \text{if } at \text{ is a } \mathsf{Boolean} \text{ attribute.}
\end{aligned}$$

Remark 1. Let e be a well-typed basic OCL expression over an object diagram $\mathcal{OD}$. Then, $QueryTheory(\mathcal{OD}, e) = QueryBasicTheory(\mathcal{OD})$, and $QueryTerm(e) = QueryBasicTerm(e)$.

For iterating expressions, the definition of *QueryAux* depends on the iterator at the top of the expression. We show below the clauses defining exists-iterating expressions. The definition of *QueryAux* for the other iterating expression is entirely similar. By $x_0 : z_0 \oplus \boldsymbol{x} : \boldsymbol{z}$ we denote the result of appending $x_0 : z_0$ to the list $\boldsymbol{x} : \boldsymbol{z}$.

$$
\begin{aligned}
&QueryAux(\langle s \rightarrow \mathsf{exists}(x_0 : z_0 \mid e), \boldsymbol{x} : \boldsymbol{z} \rangle) \\
&\quad \triangleq \langle AsOp(\mathsf{exists}(x_0 : z_0 \mid e))(QueryAuxTerm(\langle s, \boldsymbol{x} : \boldsymbol{z} \rangle)), \\
&\qquad QueryAuxOps(\langle s, \boldsymbol{x} : \boldsymbol{z} \rangle) \cup QueryAuxOps(\langle e, (x_0 : z_0 \oplus \boldsymbol{x} : \boldsymbol{z}) \rangle) \\
&\qquad\quad \cup OpDecl(\mathsf{exists}(x_0 : z_0 \mid e), \boldsymbol{x} : \boldsymbol{z}), \\
&\qquad QueryAuxEqs(\langle s, \boldsymbol{x} : \boldsymbol{z} \rangle) \cup QueryAuxEqs(\langle e, (x_0 : z_0 \oplus \boldsymbol{x} : \boldsymbol{z}) \rangle) \\
&\qquad\quad \cup EqAxms(\mathsf{exists}(x_0 : z_0 \mid e), \boldsymbol{x} : \boldsymbol{z}, QueryAuxTerm(\langle s, \boldsymbol{x} : \boldsymbol{z} \rangle)) \rangle \,,
\end{aligned}
$$

where *AsOp* generates a unique operator for each iterating expression (by forming, for example, a single string of characters from the different strings of characters that build the expression, disregarding blank spaces), *OpDecl* generates the declaration corresponding to this operator, and *EqAxms* generates the axioms that define how this operator iterates over collections given its body. Of course, the definitions of *OpDecl* and *OpDecl* are specific for each iterator. Notice also that these definitions must take into account that iterating expressions can be parameterized. We show the definitions of *OpDecl* and *OpDecl* for expression of the form $\mathsf{exists}(x_0 : z_0 \mid e)$, with iterating variables $\boldsymbol{x} : \boldsymbol{z}$. In the (conditions of the) equations below, t is the term that results from $QueryAuxTerm(\langle e, (x_0 : z_0 \oplus \boldsymbol{x} : \boldsymbol{z}) \rangle)$. That is, t is the representation of the Boolean expression in the exists-iterating expression which must hold for at least on object in the collection. Notice, also, that, since *QueryBasicTerm* maps variables $x : z$ to variables $AsVar(x : z)$ of the kind $Kind(x : z)$, the variables in t must belong to the list $AsVar(x_0 : z_0), AsVar(x_1 : z_1), \ldots, AsVar(x_n : z_n)$.

$$
\begin{aligned}
&\texttt{op}\ AsOp(\mathsf{exists}(x : z \mid e)) \colon Kind(x_1 : z_1) \cdots Kind(x_n : z_n)\ z_0 \texttt{+Col -> Boolean?}\ . \\
&\\
&\texttt{eq}\ AsOp(\mathsf{exists}(x : z \mid e))(AsVar(x_1 : z_1), \ldots, AsVar(x_n : z_n), \texttt{nil}) = \mathsf{true}\ . \\
&\texttt{ceq}\ AsOp(\mathsf{exists}(x : z \mid e))(AsVar(x_1 : z_1), \ldots, AsVar(x_n : z_n), \texttt{col}(AsVar(x_0 : z_0), XC)) \\
&\quad = \mathsf{false} \quad \texttt{if}\ t = \mathsf{true}\ . \\
&\texttt{ceq}\ AsOp(\mathsf{exists}(x : z \mid e))(AsVar(x_1 : z_1), \ldots, AsVar(x_n : z_n), \texttt{col}(AsVar(x_0 : z_0), XC)) \\
&\quad = AsOp(\mathsf{exists}(x : z \mid e))(AsVar(x_1 : z_1), \ldots, AsVar(x_n : z_n), XC) \\
&\qquad\qquad \texttt{if}\ t = \mathsf{false}\ .
\end{aligned}
$$

Example 8. Consider the nonInCyclicWay invariant in Example 3. Its value in the object diagram TRAINWAGON-1 is false. We show in Figure 6 the specification `QUERY-TRAINWAGON-1-nonInCyclicWay` corresponding to the theory *QueryTheory*(TRAINWAGON-1, nonInCyclicWay). For this example, we have mapped the outermost exists-expression to the operator `exists1`, and the innermost one to the operator `exists2`. Notice that *QueryTerm* maps the nonInCyclicWay invariant to the term

`not(exists1(allInstances(Wagon)))`,

which is equal to `false` in the theory $QueryTheory$(TRAINWAGON-1,nonInCyclicWay).

```
QUERY-TRAINWAGON-1-nonInCyclicWay is
 extending QUERY-BASIC-TRAINWAGON-1 .
 op exists1 : ClassCol -> Boolean? .
 op exists2 : Class ClassCol -> Boolean? .

 eq exists1(nil) = false .
 ceq exists1(col(W1, XC)) = true
       if exists2(W1, allInstances(Wagon)) = true .
 ceq exists1(col(W1, XC)) = exists2(XC)
       if exists2(W1, allInstances(Wagon)) = false .

 eq exists2(W1, nil) = false .
 ceq exists2(W1, col(W2, XC)) = true
       if and(not(equal(W1, W2)),
          and(includes(succ(W1), W2),
          and(includes(succ(W2), W1)))) = true .
 ceq exists2(W1, col(W2, XC)) = exists2(W1, XC)
       if and(not(equal(W1, W2)),
          and(includes(succ(W1), W2),
          and(includes(succ(W2), W1)))) = false .
```

Fig. 6. The theory $QueryTheory$(TRAINWAGON-1, nonInCyclicWay).

7 Related Work

Here we will focus on related proposals for equational specifications for UML+OCL diagrams. A comparison between the ITP/OCL tool and the USE tool [13] (which also supports validation of OCL contraints, although based on a different semantics) can be found in [8, 6].

- RIVIERA [16] is a framework for the verification and simulation of UML class diagram models (without OCL constraints) and statecharts. It is based on the representation of class diagrams and statecharts as terms (not as theories, as in our proposal) in Maude modules that specify the UML metamodel [1].
- MOMENT [2] is a generic model management framework. It uses Maude modules to automatically serialize software artifacts. It supports OCL queries (but not OCL constraints over UML models). MOMENT is integrated in Eclipse, an open platform for tool integration. The specification of OCL queries is done manually, and it requires a deep understanding of both Maude and the MOMENT specific representation of UML diagrams.

– CASL-LTL [12] is the metalanguage adopted by the CoFI Group [17] to describe the semantics of UML models, including behavioural diagrams. It proposes a "flatten" representation of the different modeling elements (classes, attributes, operations, associations an so on) as constants (of the same type) whose modeling meaning and relationships must be defined with additional logical axioms. This representation of UML models must be done manually.

8 Conclusion and Future Work

In this paper we have proposed an equational specification of UML+OCL static diagrams that provides a formal foundation for automatically validating UML object diagrams with respect to OCL constraints. Basically, class and object UML diagrams are specified as membership equational theories, and OCL invariants are represented as Boolean terms over extensions of those theories. Then, validating object diagrams with respect to invariants is reduced to checking whether the corresponding Boolean terms rewrite to true or false. Based on this ideas, we have developed a tool, named ITP/OCL, that provides automatic validation of object diagrams with respect to OCL constraints. The ITP/OCL tool is written entirely in Maude [5], making extensive use of its reflective capabilities. The ITP/OCL interface effectively keeps the tool's underlying semantics hidden to the user.

We plan to extend this work to deal with the specification of constraints (pre- and post-conditions) on operations and methods. In our view, operations and methods are in a different semantic level with respect to the rest of the modeling elements in a class diagram. When evaluated, operations and methods may change the model as a whole. This, in our view, corresponds to a change in the membership equational theory specifying the model. To address this issue, we will take advantage of the reflective properties of membership equational logic, and of its implementation in the Maude language. Operations and methods will be specified as metalevel operators that take class theories as arguments and modify their declarations so as to reflect the changes in the models provoked by the evaluation of the operations and methods.

References

1. Jose Luís Fernández Alemán. *Una Propuesta de Formalizacin de la Arquitectura en Cuatro Capas de UML.* PhD thesis, Universidad de Murcia, 2001.
2. A. Boronat, J.A. Cars, and I. Ramos. Automatic support for traceability in a generic model management framework. In David Kreische, editor, *European Conference on Model-Driven Architecture - Foundations and Applications*, volume 3748 of *Lecture Notes in Computer Science*, pages 316–330, 2005.
3. Adel Bouhoula, Jean-Pierre Jouannaud, and Jose Meseguer. Specification and proof in membership equational logic. *Theor. Comput. Sci.*, 236(1-2):35–132, 2000.
4. Dan Chiorean, Maria Bortes, and Dyan Corutiu. Proposals for a widespread use of OCL. In Thomas Baar, editor, *Proceedings of the MoDELS'05 Conference Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends*, Technical Report LGL-REPORT-2005-001, pages 68–82. EPFL, 2005.

5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.
6. Manuel Clavel and Marina Egea. ITP/OCL: A rewriting-based validation tool for UML+OCL static class diagrams. Submitted for publication. `http://maude.sip.ucm.es/~clavel`.
7. Steve Cook, Anneke Kleppe, Richard Mitchell, Bernhard Rumpe, Jos Warmer, and Alan Wills. The Amsterdam Manifesto on OCL. In *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, pages 115–149, London, UK, 2002. Springer-Verlag.
8. Marina Egea. ITP/OCL: a theorem prover-based tool for UML+OCL class diagrams. Master's thesis, Facultad de Informática, Universidad Complutense de Madrid, September 2005. `http://maude.sip.ucm.es/~marina/`.
9. Ali Hamie, Franco Civello, John Howse, Stuart Kent, and Richard Mitchell. Reflections on the Object Constraint Language. In *1998: Selected papers from the First International Workshop on The Unified Modeling Language*, pages 162–172, London, UK, 1999. Springer-Verlag.
10. Object Management Group. Object Constraint Language specification, 2004. `http://www.omg.org`.
11. Object Management Group. Unified Modeling Language specification, 2004. `http://www.uml.org`.
12. G. Reggio, E. Astesiano, and C. Choppy. Casl-Ltl: A Casl Extension for Dynamic Reactive Systems - Summary. Technical report, DISI-Università di Genova, Italy, February 2000. DISI-TR-99-34.
13. Mark Richters. *The USE tool : A UML-based specification environment*, 2001. `http://www.db.informatik.uni-bremen.de/projects/USE/`.
14. Mark Richters. *OCL Constraints*. PhD thesis, Universitat Bremen, Berlin, 2002.
15. Mark Richters and Martin Gogolla. OCL: Syntax, semantics, and tools. In Tony Clark and Jos Warmer, editors, *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, pages 42–68. Springer, 2002.
16. J. Saez, A. Toval Alvarez, and J.L. Fernandez Aleman. Tool support for transforming UML models to a formal language. In J. Whittle et al., editor, *Workshop on Transformations in UML*, pages 111–115, 2001.
17. The CoFI Reactive System Group. The Common Framework Initiative for algebraic specification and development. `http://www.brics.dk/Projects/CoFI/`.

A The Visual ITP/OCL

The Visual ITP/OCL tool is simply a Java graphical front-end for the ITP/OCL tool.[4] Events on the Visual ITP/OCL's worksheets and toolbars are transformed into ITP/OCL commands and are interpreted and executed in a Maude process running the ITP/OCL tool. In fact, the Visual ITP/OCL tool does not contain any knowledge about the meaning of the UML modeling elements, neither about the semantics of OCL expressions.

In this screenshot we show a standard Visual ITP/OCL session: in one window (the main window) we see a class diagram (in this case, our TRAINWAGON

[4] The Visual ITP/OCL tool is being developed by F. Alcaraz, J. P. Gavela, and J. Arias as a Master's project.

example) under construction; in a second window (a pop-up window) we see a class property sheet that has been opened out (possibly, to edit a class already introduced); finally, in a third window (a shell terminal) we see the commands that have been sent to the ITP/OCL tool which is running on a Maude process in the background.

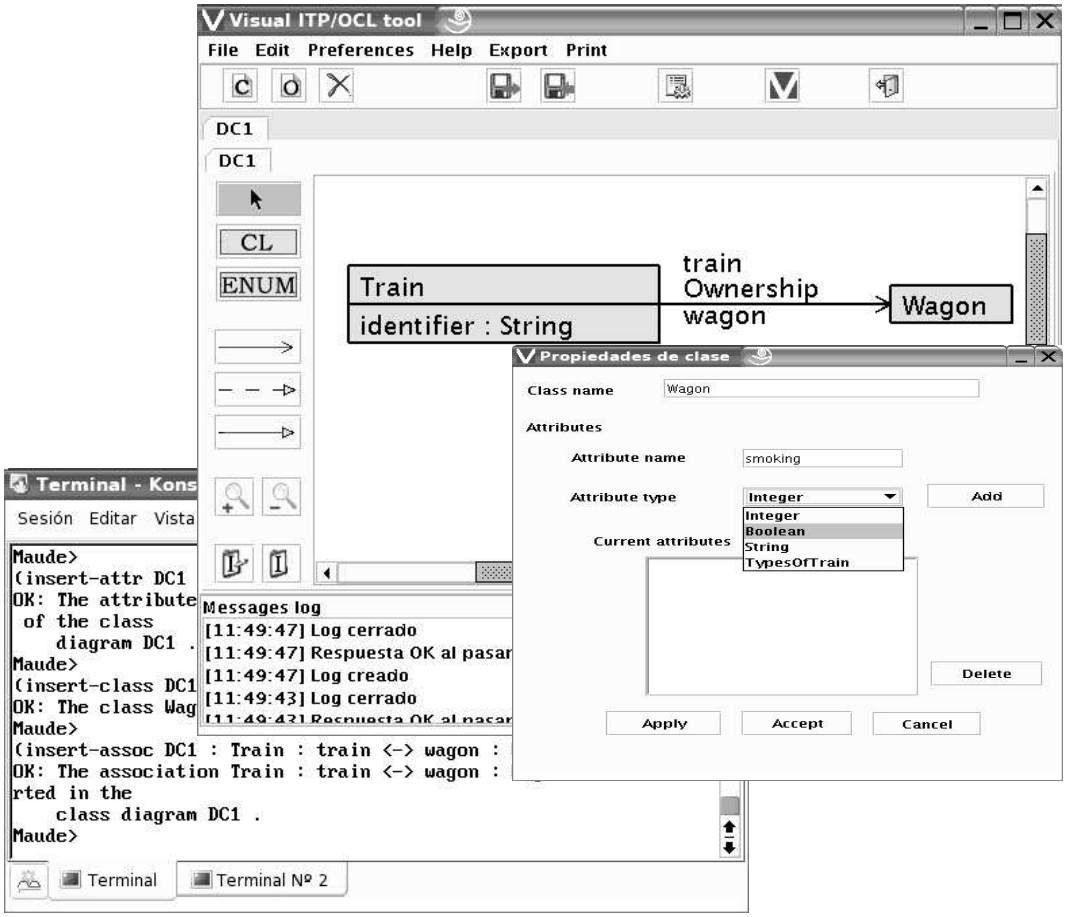

Fig. 7. A Visual ITP/OCL tool running example.