Using Reflection to Implement in Maude a Rewriting-Based Validation Tool for UML+OCL Static Class Diagrams *

Manuel Clavel and Marina Egea

Universidad Complutense de Madrid, Spain

Abstract. In this paper we present the ITP/OCL tool, a rewriting-based tool that supports automatic validation of UML static class diagrams with respect to OCL invariants. From a conceptual point of view, the ITP/OCL tool is directly based on the equational specification of UML+OCL class diagrams developed in [11], according to which: i) class and object diagrams are specified as membership equational theories; ii) invariants are represented as Boolean terms over extensions of those theories; and iii) checking invariants over object diagrams is reduced to inspecting whether the corresponding Boolean terms rewrite to true or false. From an implementation point of view, the ITP/OCL tool is written entirely in Maude [8], 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.

1 Introduction

The Unified Modeling Language (UML) [15, 20, 2] is a general-purpose visual modeling language that is used to specify, visualise, construct, and document the artifacts of a software system. The UML notation is largely based on diagrams; however, for certain aspects of a design, diagrams do not provide the level of conciseness and expressiveness that a textual language can offer. The Object Constraint Language (OCL) [14, 23] is a textual constraint language with a notational style similar to common object oriented languages. OCL came to help the modelers to specify and document with UML diagrams. 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 [18, 19, 10, 12].

Validation and testing in software development has 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

* Research supported by Spanish MEC Projects TIC2003-01000 and TIN2005-09207-C03-03, and by Comunidad de Madrid Program S-0505/TIC/0407.

instances (also called "snapshots") fulfill the desired invariants. This can lead to several consequences with respect to the design. First, if there are reasonable snapshots that do not fulfill the invariants this may indicate that the invariants are too strong or the model is not adequate in general. On the other hand, invariants 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 the design stage and generally no substantial support for invariants written in OCL. In this paper we present the ITP/OCL tool, a rewriting-based tool that supports automatic validation of UML static class diagrams with respect to OCL invariants. It is intended as a *lightweight* formal method: it should help software modelers to find flaws in UML class diagrams in the early phases of the software development process. It is intended also as a *practical* formal method: it should be directly usable for UML+OCL modelers. The ITP/OCL tool is directly based on the equational specification of UML+OCL class diagrams developed in [11]. This semantics has three important advantages: i) it uses a formal language, namely, membership equational logic [13], that already exists, is well-understood, and is implemented; ii) it formalizes the different UML+OCL modeling elements preserving their natural meaning and relationships; and iii) it provides validation methods for UML+OCL object diagrams that can be mechanized using a technique, namely, term-rewriting, that also exists, is well-understood, and is implemented. Although the equational semantics developed in [11] only covers UML+OCL static class diagrams, it provides a solid ground upon which to develop equational extensions to cope with the semantics of other UML diagrams.

The ITP/OCL tool is written entirely in Maude [8], a term-rewriting based programming language that implements membership equational logic (and rewriting logic). Maude is also a reflective programming language [9]. This means, in particular, that both its parser and its rewriting engine are available to the programmer as built-in operations: we have taken advantage of the latter to implement the tool's OCL parser and of the former to implement the tool's UML+OCL rewriting-based validation engine. Finally, Maude provides a generic input/output facility, that we have tailored to built an interface for the ITP/OCL that effectively keeps the tool's underlying semantics hidden to the user. The latest version of the ITP/OCL tool, with the available documentation and examples, can be found at `http://maude.sip.ucm.es/itp/ocl/`. We are also currently developing the Visual ITP/OCL, a Java visual front-end for the ITP/OCL tool; in Appendix B we show a screenshot of this graphical interface. Currently, the ITP/OCL tool does not share modules with the ITP tool [7]; code-wise, they are two independent tools. However, we envision a strong interaction with the ITP tool in the future: for example, when proving diagram transformation properties.

Organisation In Section 2 we provide background material: first, we introduce membership equational logic; then, we present UML+OCL diagrams and summarise the membership equational specification proposed for them in [11]. In Section 3 we present the ITP/OCL tool: we discuss its reflective design and ex-

plain its commands in relation with the tool's equational semantics. Finally, in Sections 4 and 5 we report on related work and draw conclusions. In general, we assume that the reader is familiar with the reflective capabilities of the Maude system and the style of functional metaprogramming that it supports.

2 An equational specification of UML+OCL class diagrams

In this section we provide background material: first, we introduce membership equational logic; then, we present UML+OCL class diagrams and summarise the membership equational specification proposed for them in [11].

2.1 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 [4]. 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.[1]

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.

[1] 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.

2.2 UML class 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 generalisation. 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. Consider the class diagram TRAIN-WAGON 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). Trains are of two types: monorail and high-speed; and can be identified by a string of characters. Finally, wagons can be either smoking or non-smoking and there is a special class of wagons, namely, the first-class wagons.

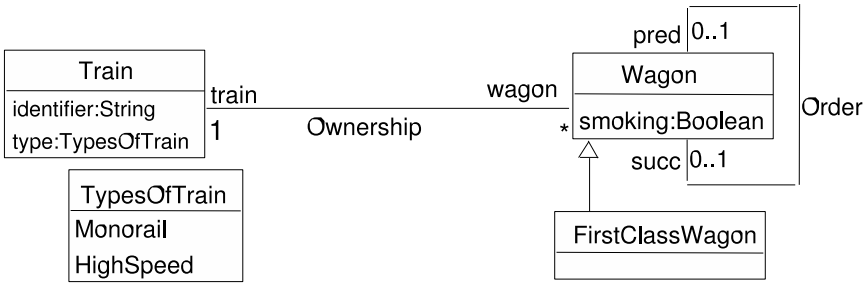

Fig. 1. The class diagram TRAIN-WAGON.

In [11] we propose an equational specification of class diagrams as membership equational theories, called "class-theories", where basically

- classes are represented by sorts of the kind `Class`;
- class collections are represented by sorts of the kind `Col`;
- attributes and roles are represented by operators of the appropriate ranks.

In Section 3.2, we will give a more detailed explanation of the construction of class-theories when presenting the ITP/OCL commands for describing class diagrams. Notice that, for the sake of simplification, we do not consider in [11] association classes, *query* class operations (that is, operations that do not change the state of the system), and multi-valued class attributes: they are, however, specified in a similar fashion. Non-query operations are a different story: we discuss this issue in the concluding section. Finally, notice that n-ary associations can be reduced to binary ones.

2.3 UML object diagrams

A system may be in different states as it changes over time. An *object diagram* models the objects (i.e., the instances of the classes) and links (i.e., the instances of the associations) that represent the state of a system at a particular moment. 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.

Consider the object diagram TRAIN-WAGON-1 shown in Figure 2. It describes an instance (also called a "snapshot") of the railway system modeled by the class diagram TRAIN-WAGON shown in Figure 1. In [11] we propose

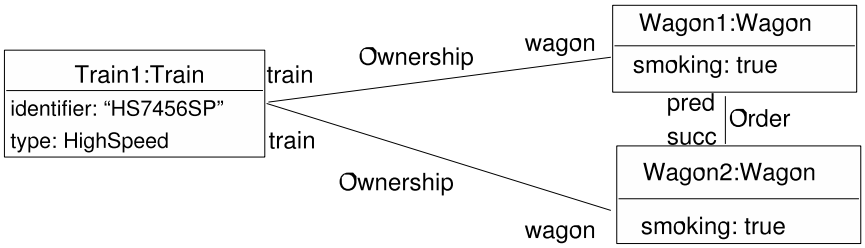

Fig. 2. The object diagram TRAIN-WAGON-1.

an equational specification of object diagrams as membership equational theories, called "object-theories". The relation between class diagrams and object diagrams is preserved by the fact that object-theories *instantiate* their corresponding class-theories by "filling" the sorts representing their classes and by defining the operators representing their attributes and roles. Basically,

- objects are represented by constants of the kind `Class`, which are declared to belong to the sorts representing their classes;
- values are declared for each object's attribute by defining the value of the operator representing the attribute when applied to the constant representing the object; and
- collections are associated to each object's role by defining the value of the operator representing the role when applied to the constant representing the object.

In Section 3.2 we will give a more detailed explanation of the construction of object-theories when presenting the ITP/OCL commands for describing object diagrams.

2.4 OCL invariants

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. It can be

used to state constraints concerning the static structure and the behaviour of a system. The most important uses of OCL expressions in UML diagrams are [23]: (1) the specification of invariants on classes and types in the class diagram; (2) the specification of constraints on operations and methods; (3) the description of pre- and post-conditions on operations; (4) the specification of initial values and derivation rules for attributes; (5) the specification of query operations; and (6) the introduction of new attributes and operations. The equational semantics for UML+OCL class diagrams proposed in [11] explicitely supports use (1) and, implicitly, uses (4), (5) and (6). Regarding uses (2) and (3), see our remarks in the concluding section.

In OCL, a number of basic types are predefined and available to the modeler. These types are Boolean (true, false), Integer (1, -5, 2, 34, ...), Real (1.5, 3.14, ...), and String ('To be or not to be...'). OCL defines a number of operations on the predefined types. For example, for the type Boolean, the operations and, not, and implies. 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 in OCL by a dot (.) followed by the name of the property. For example, if w:Wagon is the reference to an object, (w:Wagon).smoking is the value of the attribute smoking for w:Wagon. Starting from a specific object, we can also navigate an association to refer to other objects and their properties by using the opposite association-end. For example, if t:Train is the reference to an object, (t:Train).wagon is the collection of objects of type Wagon linked to t:Train under the association Ownership. 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. For example, Train.allInstances() is the collection of objects of class Train that exist in the system at the time the expression is evaluated.

Collections, like sets, ordered sets, bags, and sequences are predefined types in OCL. They have a large number of predefined operations on them. Here we only introduce those that we use later on in our examples. The value of a property for a collection is accessed by an arrow (→) followed by the name of the property. For example, if t:Train is the reference to an object, ((t:Train).wagon)→size() is the number of objects in the collection (t:Train).wagon; similarly, ((t:Train).wagon) →isEmpty() is true if and only if the collection (t:Train).wagon is empty. When we want to specify a collection which is derived from some other collection, but which contains different elements from the original collection (i.e., it is not a subcollection), we can use a collect operation. The syntax for the collect operation is:

collection →collect(v | *expression-with-*v)

The variable v is called the iterator-variable. The value of the collect operation is the collection of the results of the evaluation of *expression-with-*v for each element in *collection*. For example, the result of Train.allInstances→collect(t:Train | (t:Train).identifier) is the collection of all the train identifiers. Many times a constraint is needed on all elements of a collection. The forAll operation in OCL

allows specifying a Boolean expression, which must hold for all objects in a collection:

collection →forAll(v | *boolean-expression-with-*v)

This forAll expression results in a Boolean. The result is true if and only if the *boolean-expression-with-*v is true for all elements of *collection*. For example, the result of Wagon.allInstances→forAll(w:Wagon | (w:Wagon).smoking) is true if and only if smoking is allowed in all wagons.

Consider the following constraint over the class diagram TRAIN-WAGON: *A wagon and its successor wagon should belong to the same train.* This constraint can be expressed using OCL as the following invariant belongToTheSameTrain over the class diagram TRAIN-WAGON:

```
contex Wagon inv belongToTheSameTrain:
   (Wagon.allInstances)→forAll(self:Wagon |
    (self:Wagon).succ→notEmpty() implies
    (self:Wagon).train = (((self:Wagon).succ→collect(w:Wagon|(w:Wagon).train))→asSet()))
```

Of course, the same constraint can be expressed in different ways in OCL. In particular, one may use the self operator to contextualize the invariant and get rid of the initial Wagon.allInstances expression. For the sake of simplification, we consider here the general case.

In [11] we propose a formalisation of OCL invariants over UML class diagrams as Boolean terms over membership equational theories, called "invariant-theories", that extend the class-theories corresponding to the UML class diagrams. This extension basically consists on:

- the specification of the OCL operators over collections (like size, includes, asSet, and so on);
- the specification of the OCL operator allInstances; and
- the specification of the particular instances of the OCL iterator-operators (like forAll, exists, collect, and so on) that occur in the invariants.

In Section 3.2 we will give a more detailed explanation of the construction of the invariant-theories when presenting the ITP/OCL commands for stating and checking invariants.

3 The ITP/OCL: a validation tool for UML+OCL diagrams

In this section we present the ITP/OCL tool: we discuss its reflective design and explain its commands in relation with the tool's equational semantics. We assume that the reader is familiar with the reflective capabilities of the Maude system and the style of functional metaprogramming that it supports. See [9] for a complete description of the Maude system.

3.1 The ITP/OCL design

The implementation of an interactive tool in Maude comprises four different tasks [6]: defining a read-eval-print loop; defining the syntax for the commands; defining the interaction with the loop; and defining the processing of the commands.

The read-eval-print loop. Maude provides a generic input/output facility through its predefined module `LOOP-MODE`. This module declares an operator that builds terms of sort `System`—called "loop objects"— with an *input stream* (its first argument, of sort `QidList`), an *output stream* (its third argument, also of sort `QidList`), and a *state* (its second argument, of sort `State`). The way to distinguish the inputs passed to a loop object from the inputs passed to the Maude system is by enclosing the former in parentheses. In the module `LOOP-MODE` the sort `State` does not have any constructors: this gives complete flexibility for defining the terms that represent the state of the loop object in each interactive tool. In the case of the ITP/OCL tool, the sort `State` contains terms representing objects which basically have:

- An attribute `odb` that keeps a database in which the UML+OCL diagrams are stored.
- An attribute `input` that holds the next request to be processed by the tool.
- An attribute `output` that holds the next response to be output to the user.

The syntax for the commands. Maude provides great flexibility to define the syntax for an interactive tool thanks to its mixfix front-end and to the use of *bubbles* (any nonempty list of Maude identifiers). In the case of the ITP/OCL, the syntax for its commands is defined in the module `OCL-GRAMMAR` which introduces three sorts of bubbles: the sort `Token` for bubbles of length one; the sort `Bubble` for bubbles of any length; and the sort `NeTokenList` for bubbles of any length greater that one. In Appendix A we include the fragment of the module `OCL-GRAMMAR` that defines the ITP/OCL's command grammar. In this module commands are represented as terms of sort `Input`.

The interaction with the loop. The implementation in Maude of an interactive tool contains rewrite rules acting on loop objects to detect when a valid request has been entered by the user (and hence it must be processed), or when a valid result has been produced by the application (and hence it must be output). Detecting valid requests can be easily and efficiently achieved using the built-in operation `metaParse`. In the case of the ITP/OCL, a rule labelled `[in]` checks when a list of quoted identifiers placed in the input stream of its loop object corresponds to a valid command, that is, to a term of sort `Input` in the module `OCL-GRAMMAR`, and it places the metarepresentation of this term in the `input` attribute of the loop object's state. For the inverse direction of the interaction, a rule labelled `[out]` checks when the `output` attribute of the loop object's state holds a response to be output and it places it in the output stream of the loop object.

The processing of the commands. The processing of the requests made to an interactive tool can be defined in Maude by equations acting on the states of the loop objects, that is, acting on terms of sort `State`. The ITP/OCL's commands can be grouped in four classes:

- *Commands that create a diagram.* They are defined by equations that add an empty class (resp. object) diagram to the state's database. This database is implemented as a list of terms metarepresenting Maude modules: namely, the modules specifying the UML+OCL diagrams entered by the user. These commands add to this list the term that metarepresents the module that, according with the semantics introduced in [11], specifies an empty class (resp. object) diagram.
- *Commands that insert an element (class, attribute, association, and so on) in a diagram.* They are defined by equations that add to the module specifying the diagram in the state's database the declarations (sorts, operators, memberships, equations) that, according with the semantics introduced in [11], specify that the diagram has this element.
- *Commands that state an invariant over a class diagram.* They are defined by equations that associate to the module specifying the class diagram in the state's database the metarepresentation of the Boolean term that, according with the semantics introduced in [11], represents this invariant.
- *Commands that validate invariants over an object diagram.* They are defined by equations that check whether the Boolean terms representing the invariants reduce to `true` or `false` in the module that, according with the semantics introduced in [11], specifies the union of the invariant-theory corresponding to these invariants and the object-theory corresponding to the object diagram.

In the next section we introduce in more detail the different ITP/OCL commands, their syntax and their effects on the state of the tool's loop object. Regarding their implementation, the equations defining the ITP/OCL commands make extensive use of the functions predefined in Maude to create, modify, and execute modules at the metalevel.

3.2 The ITP/OCL commands

In this section we introduce the commands available in the ITP/OCL tool to describe UML class and object diagrams, and to state and validate OCL constraints over these diagrams.

UML class diagrams To create a class diagram we use the command `(create-class-diagram` *CD* `.)`, where *CD* is the class diagram's name. Internally, this command adds a module *CD*, which specifies an empty class diagram, to the state's database.

To insert a class we use the command `(insert-class` *CD* `:` *C* `.)`, where *CD* is the class diagram's name and *C* is the class' name. Internally, this command adds the sorts *C* and *C*`Col`, of the kinds, respectively, `Class` and `Col`, to the

module CD in the state's database. It also adds the membership axioms that define the set of terms that represent C-collections.

To insert a generalisation relation we use the command `(insert-subclass` CD `:` C `->` C' `.)`, where CD is the class diagram's name, C is the sub-class' name, and C' is the super-class' name. Internally, this command adds a subsort relation between the sorts C and C' to the module CD in the state's database. The subsort relation makes C to inherit the attributes and properties declared for its superclass C'.

To insert an enumeration class we use the command `(insert-enum-class` CD `:` C `-` W_1 ... W_n `.)`, where CD is the class diagram's name, C is the class' name, and W_i are the class' values, for $i = 1, \ldots, n$. Internally, this command adds the sort C of the kind `Class` to the module CD in the state's database. It also adds the operators W_i as constants of the kind `Class`. Finally, it adds the membership axioms that declare that the constants W_i represent values in the enumeration class C.

To insert a class' attribute we use the command `(insert-attr` CD `:` C `(`A`,`V`) .)`, where CD is the class diagram's name, C is the class' name, A is the attribute, and V is its values' type. Internally, this command adds an operator A, with rank `Class ->` $Kind(V)$, where $Kind(V)$ is the kind corresponding to the type V, to the module CD in the state's database.

To insert an association relation we use the command `(insert-assoc` CD `:` C `:` R `<->` R'`:` C'`.)`, where CD is the class diagram's name, C and C' are the names of the classes at each end of the association, and R and R' are their roles in the association. Internally, this command adds two operators R and R', with ranks `Class -> Col`, to the module CD in the state's database.

To insert multiplicities at each end of an association we use the command `(insert-multiplicity` CD `:(`C `:` R`,` M `)<->(` M'`,` R' `:` C'`).)`, where CD is the class diagram's name, C and C' are the names of the classes at each end of the association, R and R' are their roles in the association, and M and M' are the multiplicities at each end of the association. A multiplicity is either (`<` k `>`), with k a natural number or the symbol `*`, or (`<` n, k `>`), with n a natural number and k a natural number or the symbol `*`. Internally, multiplicities are automatically added to the list of invariants associated to the module CD in the state's database. We will go back to this point in Section 3.2.

UML object diagrams To create an object diagram we use the command `(create-object-diagram` CD `:` OD `.)`, where OD is the object diagram's name and CD is its class diagram's name. Internally, this command adds a module OD, which specifies an empty instance of the object diagram OD, to the state's database.

To insert an object we use the command `(insert-object` OD `:` C `:` O `.)`, where OD is the object diagram's name, O is the object's name, and C is the name of the class of which O is an instance. Internally, this command adds the operator O as a constant of the kind `Class`. It also adds the membership axiom that declares that the constant O represents an object in the class C. Finally, for

each operator-role R in OD, it adds the equation that declares that the object O is initially linked in its role R to the empty collection of objects.

To insert the value of an object's attribute we use the command `(insert-attr-value` OD `:` C `:` O `:` A `:` V `->` E `.)`, where OD is the object diagram's name, O is the object's name, C is the name of the class of which O is an instance, A is the attribute, V is its type, and E is its value for the object O. Internally, this command adds to the module OD in the state's database the equation that declares that the term $A(O)$ is equal to E.

To insert a link we use the command `(insert-link` OD `|` C `:` R `<->` R'`:` C' `|` O `<->` O' `.)`, where OD is the object diagram's name, O and O' are the names of the objects at each end of the link, C and C' are the names of their classes, and R and R' are their roles in the association of which the link is an instance. Internally, this command modifies the right-hand side of the equation that define the value of the operator R (resp. R') over the constant O' (resp. O) by union-ing the collection with the object O (resp. O') to the collection of objects already linked to O' (resp. O) in its role R.

OCL invariants To state an invariant we use the command `(insert-invariant` CD `::` INV `.)` where CD is the class diagram's name and INV is the invariant. A few warnings, however, are in order regarding invariant expressions:[2]

- To denote the collection of objects associated to an object's role, we must prefix the symbol `#` to the role (and insert a blank space between the expression denoting the object and the role).
- We must also prefix the allInstances operator with the symbol `#` (and insert a blank space between the class' name and the operator).
- We can not apply an iterator with several iterator-variables; instead, we must apply the iterator in nested form.
- Finally, the iterator-variable must be used along with its class (separated by a semicolon).

Internally, this command associates the Boolean term representing INV to the module CD in the state's database.

Multiplicities are considered as invariants. Internally, a command to insert multiplicities at each end of an association in a class diagram is transformed into a command that state the corresponding invariant over the class diagram.

Validating invariants To check whether an object diagram satisfies the invariants stated over the class diagram of which it is an instance, we use the command `(check-invariants` CD `:` OD `.)` where CD is the class diagram's name and OD is the object diagram's name. Internally, for each invariant INV associated to the module CD in the state's database, this command:

[2] These restrictions will be removed in the near future, with a more adequate definition of ITP/OCL's bubbles and tokens in the module `OCL-GRAMMAR`.

– First, it creates a new module as a result of extending the module *OD* with i) the specification of the OCL operators over collections; ii) the specification of the OCL operator allInstances for the module *OD*; and iii) the specification of the particular instances of the OCL iterator-operators that occurs in *INV*. Each of these extensions is explained below.
– Then, it reduces in this new module the Boolean term representing the invariant *INV*. Since, by construction, this module is Church-Rosser, terminating, and sufficiently complete for ground Boolean terms, the result of this reduction must be either `true` or `false`.[3]
– Finally, it changes the state's attribute `output` to hold a message informing the user of the reduction result.

The specification of the OCL operators over collections (like size, includes, asSet, and so on) as equationally defined operators can be found in the module `OCL-BASIC` which is part of the ITP/OCL distribution.

The specification of the OCL operator allInstances as an equationally defined operator with rank `ClassId->ClassCol` is accomplished as follows. First, for each class C_i in the object diagram, we declare an operator C_i as a constant of the kind `ClassId`. Then, for each class C_i in the object diagram, we define an equation that declare that the operator `allInstances`, when applied to the constant C_i, is equal to the collection of constants representing the instances of C_i in the object diagram.

Finally, the specification of the particular instances of the OCL iterator-operators (like forAll, exists, collect, and so on) also as equationally defined operators is accomplished as follows. First, for each iterator-expression $iter \rightarrow (id : c \mid exp)$ that occurs in *INV*, we declare an operator $iter@n$, for n a unique natural number. Then, we define the equations that declare the value of $iter@n$ over collections. The generation of these equations is a generic process (formally specified in [11]), parameterized by the kind of the iterator *iter* and the expression *exp*.

4 Related Work

Here we will focus on related proposals for an algebraic semantics for UML(+OCL) models. A comparison between the ITP/OCL tool and the USE tool [17] can be found in [11].

– RIVIERA [21] 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].

[3] The proof of these properties would be published elsewhere. Detail information about the construction of the invariant-theory modules can be found in [11].

– MOMENT [3] is a generic model management framework. It uses Maude modules to automatically serialize software artifacts. It supports OCL constraints over UML models. As in the case of RIVIERA, it is based on the representation of class diagrams as terms (not as theories, as in our proposal) in Maude module that specifies the UML metamodel. MOMENT is integrated in Eclipse, an open platform for tool integration.
– CASL-LTL [16] is the metalanguage adopted by the CoFI Group [22] to describe the semantics of UML models, including behavioural diagrams. It proposes a "flatten" representation of the different modeling elements (classes, attributes, operations, associations, and 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. In our approach, we formalize the various modeling elements using different formal elements (sorts, subsorts, operators, constants and terms, equations and memberships) in an attempt to preserve their natural meaning and relationships. This formalization is automatized in the ITP/OCL tool.

5 Conclusion and Future Work

The equational specification of UML+OCL static class diagrams proposed in [11] 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" [5].

In this paper we have presented the ITP/OCL tool, an OCL tool directly based on our equational semantics for UML+OCL static class diagrams. The seminal work [11] only contained a basic description of the tool, still implemented as an extension of the ITP tool, without many of its actual modeling features, and lacking its current user interface. The ITP/OCL tool is written entirely in Maude [8], making extensive use of its reflective capabilities. The ITP/OCL interface effectively keeps the tool's underlying semantics hidden to the user. The tool is intended as a lightweight formal method: it will help to find defects in UML class diagrams in the early phases of the software developing process.

Remarks on the equational specification of non-query class operations. However, there are uses of OCL expressions in UML class diagrams that are not yet covered by our equational specification. In particular, we plan to extend this work to deal with the specification of constraints (pre- and post-conditions) on class non-query operations. In our view, non-query operations are in a different semantic level with respect to the rest of the modeling elements in a class diagram. When evaluated, non-query operation 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. Non-query operations will be specified as metalevel operators that take class-theories modules as arguments and modify their declarations so as to reflect the changes in the models provoked by the evaluation of the non-query operations.

References

1. J. L. F. Alemán. *Una Propuesta de Formalizacin de la Arquitectura en Cuatro Capas de UML*. PhD thesis, Universidad de Murcia, 2001.
2. M. Blaha and J. Rumbaugh. *Object-oriented modeling and design with UML, 2nd edition*. Prentice Hall, Upper Saddle River, N.J., 2005.
3. A. Boronat, J. Cars, and I. Ramos. Automatic support for traceability in a generic model management framework. In D. Kreische, editor, *European Conference on Model-Driven Architecture - Foundations and Applications*, volume 3748 of *Lecture Notes in Computer Science*, pages 316–330, 2005.
4. A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theor. Comput. Sci.*, 236(1-2):35–132, 2000.
5. D. Chiorean, M. Bortes, and D. Corutiu. Proposals for a widespread use of OCL. In T. 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.
6. M. Clavel. Strategies and user interfaces in Maude at work. *Electronic Notes in Theoretical Computer Science*, 86(4), 2003.
7. M. Clavel. The ITP tool's home page. `http://maude.sip.ucm.es/itp`, 2005.
8. 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.
9. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude manual (version 2.2). SRI International, December 2005, `http://maude.cs.uiuc.edu`.
10. S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. 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.
11. M. 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/`.
12. A. Hamie, F. Civello, J. Howse, S. Kent, and R. 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.
13. J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer-Verlag, 1998.
14. Object Management Group. Object Constraint Language specification, 2004. `http://www.omg.org`.

15. Object Management Group. Unified Modeling Language specification, 2004. http://www.uml.org.
16. 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.
17. M. Richters. *The USE tool : A UML-based specification environment*, 2001. http://www.db.informatik.uni-bremen.de/projects/USE/.
18. M. Richters. *OCL Constraints*. PhD thesis, Universitat Bremen, Berlin, 2002.
19. M. Richters and M. Gogolla. OCL: Syntax, semantics, and tools. In T. Clark and J. Warmer, editors, *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, pages 42–68. Springer, 2002.
20. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual, 2nd Edition*. Addison-Wesley, 2004.
21. J. Saez, A. T. Alvarez, and J. F. Aleman. Tool support for transforming UML models to a formal language. In J. W. et al., editor, *Workshop on Transformations in UML*, pages 111–115, 2001.
22. The CoFI Reactive System Group. The Common Framework Initiative for algebraic specification and development. http://www.brics.dk/Projects/CoFI/.
23. J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA, Second Edition*. Object Technology Series. Addison Wesley, 2003.

A The command grammar

```
---------------------------------
--- Commands to create diagrams
---------------------------------
op create-class-diagram_. : Token -> Input .
op create-object-diagram_:_. : Token Token -> Input .
---------------------------------
--- Commands to insert elements in diagrams
---------------------------------
op insert-class_:_. : Token Token -> Input .
op insert-enum-class_:_-_. : Token Token Bubble -> Input .
op insert-attr_:_(_,_). : Token Token Token Token -> Input .
op insert-assoc_:_:_<->_:_. : Token Token Token Token Token -> Input .
op insert-subclass_:_->_. : Token Token Token -> Input .
op insert-multiplicity_:(_:_,_)<->(_,_:_). :
     Token Token Token Multi Multi Token Token -> Input .
op insert-object_:_:_. : Token Token Token -> Input .
op insert-link_|_:_<->_:_|_<->_. : Token Token Token Token Token
     Token Token -> Input .
op insert-attr-value_:_:_:_->_. : Token Token Token Token Token
     Token -> Input .
---------------------------------
--- Commands to state invariants over diagrams
---------------------------------
op insert-invariant_::_. : Token Bubble -> Input .
---------------------------------
--- Commands to validate diagrams
```

```
---------------------------------
op check-invariants_:_. : Token Token -> Input .
```

B 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 Figure 3 we show the screenshot of a standard Visual ITP/OCL session: in one window (the main window) we see a class diagram (in this case, our TRAINWAGON 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.

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

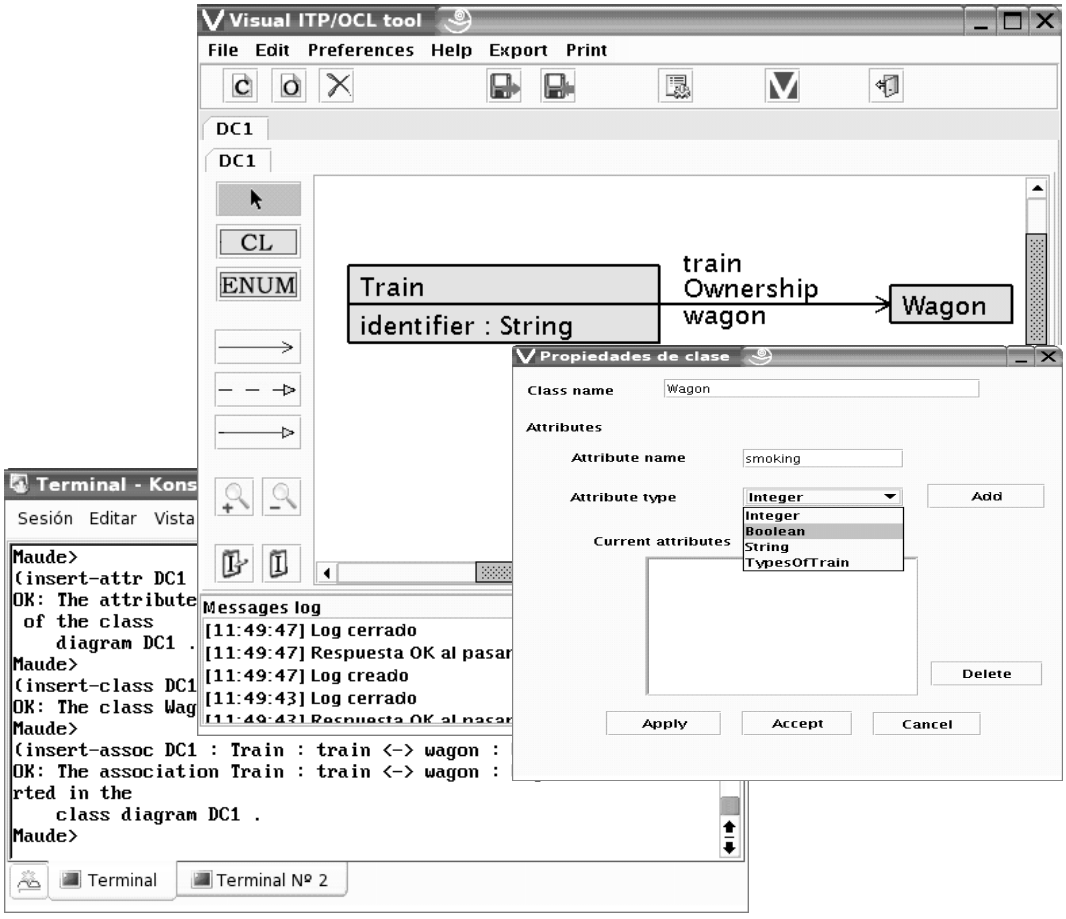

Fig. 3. A Visual ITP/OCL tool running example.