Electronic Communications of the EASST
Volume 36 (2010)

Proceedings of the Workshop on OCL and Textual Modelling (OCL 2010)

Dynamic Validation of OCL Constraints with mOdCL

Manuel Roldán and Francisco Durán

18 pages

Guest Editors: Jordi Cabot, Tony Clark, Manuel Clavel, Martin Gogolla
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Dynamic Validation of OCL Constraints with mOdCL

Manuel Roldán[1]* **and Francisco Durán**[2]†

[1] mrc@lcc.uma.es
[2] duran@lcc.uma.es
Dpto. de Lenguajes y Ciencias de la Computación
Universidad de Málaga, España

Abstract: This paper presents mOdCL, a Maude based evaluator of OCL expressions and validator of OCL constraints. Given its OCL expression evaluator, the use of execution strategies allows mOdCL, not only validating invariant constraints on concrete system states, but also dynamically validating invariants and pre- and post-conditions on the successive states obtained during system execution.

Keywords: OCL, Dynamic validation, Maude

1 Introduction

The Object Constraint Language (OCL) [OCLa] is a textual formal language which provides the necessary formal notation to complete the UML diagrams with precise and unambiguous specifications. An important use of OCL is to complete class diagrams with the constraints to be satisfied. Specifically, class invariants, and pre- and post-conditions on methods can be specified on them.

The growing interest in OCL, mainly thanks to its use in Model-Driven Software Development (MDSD), has motivated the development of a number of tools (see, e.g., the OCL Portal [OCLb]), which allow different kinds of analysis as, for example, consistency checks to guarantee that the OCL constraints are not contradictory, or validation of OCL constraints.

We can find OCL tools using very different approaches. For example, the validation environment USE [GBR07], the Dresden toolkit [Thea] or OCLE [CPC+04] are based on Java, the proof environment HOL-OCL [BW08] is implemented as a shallow embedding of OCL into the Higher-Order Logic (HOL), the UMLtoCSP tool [CCR07] uses Constraints Logic Programming, etc.

Some tools validate OCL constraints directly on UML models by representing class diagrams in the supporting formalism and checking whether such constraints hold on given system states (snapshots), which can be obtained, for example, from object diagrams or from states of a program being executed [RG03]. Other tools are designed to dynamically validate OCL constraints on the implementation of UML models, typically by codifying the constraints validation somehow and injecting it into the implementation of the system.

We present the mOdCL validator, a tool which is able to both evaluate OCL expressions on given object configurations and execute Maude prototypes of UML models, checking the

* This author was supported by Spanish Research Projects TIN2008-03107 and P07-TIC-03184.
† This author was supported by Spanish Research Projects TIN2008-03107 and P07-TIC-03184.

satisfaction of their OCL constraints. Maude [CDE+02, CDE+07] is an executable formal specification language based on rewriting logic, with a rich set of validation and verification tools [CDE+07, CDH+07], increasingly used as support to the development of UML, MDA, and OCL tools (see, e.g., [BM08, RRDV07, Theb]). Furthermore, Maude has demonstrated to be a good environment for rapid prototyping, and also for application development (see, e.g., [CDE+07, Chapter 20]). Moreover, Maude has shown to be very good as a logical and semantic framework in which different logics and formalisms can be expressed and executed [MM02]. For all this, we believe that Maude is a good candidate to express prototypes of UML models to be used to dynamically validating the OCL constraints on them. Nevertheless, we do not intend to describe here systematic transformations of UML diagrams into Maude (for that, see, e.g., [Kna00, FT01, MB09]). We will describe however the Maude representations of class diagrams and object diagrams as used by mOdCL. Some of the tools that automatically generate Maude modules representing UML, MOF or Ecore class and object diagrams are MOVA [Theb], MOMENT [BCR06], and e-Motions [RDV09].

Some OCL tools, as the Dresden toolkit, can perform dynamic validation, but they require a complete implementation of the system to be validated. On the other hand, the tools that support the validation of UML models without their complete implementation can only statically validate given snapshots representing concrete system states. Our mOdCL validator improves the existing validation tools because it allows the user to statically and dynamically validate UML models on very high level prototypes. This way, some errors in the model or in the constraints definition could be detected early, during the modeling phase.

We focus here on UML class diagrams. In UML class diagrams, we can provide invariant constraints on specific classes, as well as pre- and post-conditions on methods. The invariants of a class must be satisfied by all objects of such a class, and pre- and post-conditions in all the executions of the methods they are associated to. However, these constraints are not required to be satisfied in all states of the system, but only in those relevant states [WK03]. Invariants are required to be satisfied when an operation terminates, but not during its execution. Similarly, pre- and post-conditions must be satisfied, respectively, before and after the execution of the operations they are associated to.

Thus, the dynamic validation of the constraints of a model has two key parts: the identification of those states on which constraints must be satisfied, and the evaluation of the satisfaction of the appropriate constraints. To deal with these issues, our proposed system mOdCL includes two differentiated components: an OCL expression evaluator and a strategy that controls the execution of the system and checks the appropriate OCL constraints on the appropriate states.

The mOdCL evaluator can evaluate OCL expressions on object configurations. It can be used to statically validate constraints on object diagrams or to evaluate any other OCL expression on them. It can be used as an independent component, and has in fact already been integrated into the e-Motions tool [RDV09, RDV10], a tool for defining the behavior of visual languages.[1]

Our execution strategy, named `metaOCLRewrite`, works on a prototype of the user's UML

[1] The e-Motions tool is a Domain Specific Modeling Language (DSML) and graphical framework developed for Eclipse that supports the specification, simulation, and formal analysis of real-time systems. It extends standard in-place rules so that time and action statements can be included in the behavioral specifications of a DSL. The behavioral models generated with e-Motions can be fully integrated in MDE processes. See http://atenea.lcc.uma.es/e-Motions for additional details.

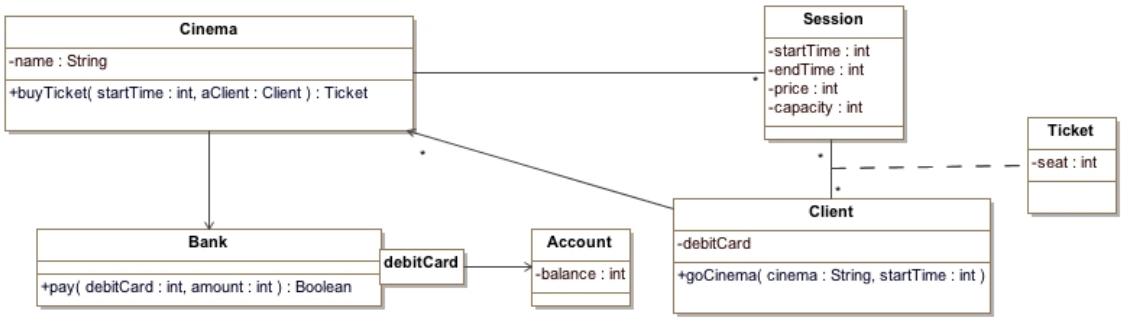

Figure 1: Class diagram for the cinema example.

model, specified as a rewriting theory in Maude. Exploiting the metalevel capabilities of Maude, the strategy controls the execution of the rewrite rules specifying the behavior of the system and checks the corresponding OCL constraints on the appropriate states.

We are aware that end users are interested in validating UML diagrams, not in the validation process itself. In practice, mOdCL users do not need any knowledge on the internals of our validation tool, the evaluator and the strategy remains hidden to them. Furthermore, the integration of the facilities proposed here into generic development environments is a goal for us. This is, e.g., the case in the use of mOdCL in e-Motions, where an ATL transformation is used to get mOdCL Maude configurations from Ecore object models, using mOdCL as a backend tool to evaluate OCL expressions.

The paper is structured as follows. Section 2 introduces a running example, which will be used in the rest of the paper to illustrate our approach. Section 3 serves as a brief introduction to rewriting logic and Maude, with emphasis on its support for object-oriented systems, and covering those features of Maude relevant to understanding the current paper. Section 4 discusses how UML concepts must be represented in Maude to be used by the mOdCL validator. Section 5 then shows how to validate OCL constraints with mOdCL. Section 6 gives some details on the implementation of the validator. Finally, Section 7 draws some conclusions and future work.

2 Our Running Example

Figure 1 shows a class diagram which models part of a very simple ticket sale system for cinemas, to be used along the paper.

A `Cinema` has a `name` and offers a number of sessions, for which it sells tickets, managing the payment through a single `bank`. A `Session` has a `capacity`, a ticket `price`, starting/ending times (`startTime`/`endTime`), and the tickets sold to clients. A `Client` knows some cinemas, pays with a `debitCard` and has access to the tickets he has bought. The association class `Ticket` represents a ticket bought by a client for a given session; each ticket has a `seat` number. A `Bank` knows the accounts that support the debit cards, modeled as a qualified association with the `debitCard` number as key. Each `Account` has a `balance`.

Three public methods have been considered: `goCinema`, in the `Client` class, allows a client purchase a ticket for a given cinema, for a given time; `buyTicket`, in the `Cinema` class, allows to buy one ticket for the session starting at a given time; and `pay`, in the `Bank` class, allows to charge a given `amount` to the account associated to a given `debitCard`.

We impose the `avoid-overlapping` OCL invariant to the `Client` class, which states that a client cannot buy two tickets for overlapping sessions, and the `seats-in-session` invariant to the `Session` class, which states that the number of tickets sold for a session does not exceed its capacity.

```
context Client inv avoid-overlapping :
    ticket -> forAll(T1 | ticket -> forAll(T2 |
        (T1 = T2)
        or (T1.session.endTime < T2.session.startTime)
        or (T2.session.endTime < T1.session.startTime)))

context Session inv seats-in-session :
    capacity >= ticket -> size()
```

The `buyTicket` method assumes, as a pre-condition, that the cinema must offer a session at the requested time and, as a post-condition, that the returned value is either `null` or a new ticket, which is the only ticket added to the tickets of the requested session.

```
context Cinema::buyTicket(st:Integer, cl:Client): Ticket
pre:  session->select(S | S.startTime = st)->size() = 1
post: (result = null) or
      (session->select(S | S.startTime = st).ticket
          -> includes(result)
       and
       ((session->select(S | S.startTime = st).ticket->asSet()) -
        (session->select(S | S.startTime = st).ticket@pre->asSet()))
          -> size() = 1)
```

3 Rewriting Logic and Maude

Membership equational logic (MEL) [Mes98] is a Horn logic whose atomic sentences are equalities $t = t'$ and *membership assertions* of the form $t : S$, stating that a term t has sort S. Such a logic extends order-sorted equational logic, and supports sorts, subsort relations, subsort polymorphic overloading of operators, and definition of partial functions with equationally defined domains.

Rewriting logic (RL) [Mes92] is a logic of change that can naturally deal with state and with highly nondeterministic concurrent computations. In RL, the state space of a distributed system is specified as an algebraic data type in terms of an equational specification (Σ, E), where Σ is a signature of sorts (types) and operations, and E is a set of equational axioms. The dynamics of a system in RL is then specified by rewrite *rules* of the form $t \longrightarrow t'$, where t and t' are Σ-terms. This rewriting happens modulo the equations E, describing in fact local transitions $[t]_E \longrightarrow [t']_E$ These rules describe the local, concurrent transitions possible in the system, i.e. when a part of

the system state fits the pattern t (modulo the equations E) then it can change to a new local state fitting pattern t'. Notice the potential of this type of rewriting, and the very high level of abstraction at which systems may be specified, to perform, e.g., rewriting modulo associativity or associativity-commutativity.

RL is parameterized by its underlying equational logic. In what follows, we will use the syntax of Maude [CDE+02, CDE+07], a wide spectrum programming language directly based on RL, with MEL as its underlying equational theory, to present rewrite theories. Thus, Maude integrates an equational style of functional programming with RL computation.

Maude supports the modeling of object-based systems by providing sorts representing the essential concepts of object (`Object`), message (`Msg`), and configuration (`Configuration`). A configuration is a multiset of objects and messages (with the empty syntax, associative commutative, union operator __) that represents a possible system state.

Although the user is free to define any syntax for objects and messages, several additional sorts and operators are introduced as a common notation. Maude provides sorts `Oid` for object identifiers, `Cid` for class identifiers, `Attribute` for attributes of objects, and `AttributeSet` for multisets of attributes (with _,_ as union operator). Given a class C with attributes a_i of types S_i, the objects of this class are then record-like structures of the form

$$< O : C \mid a_1 : v_1 , \ldots , a_n : v_n >$$

where O is the identifier of the object, and v_i are the current values of its attributes (with appropriate types). See [CDE+07] for additional details on how object-oriented systems are represented in Maude, including explanations on how to represent inheritance, syntax for object-oriented modules, different forms of object communication, etc.

The following Maude definitions specify a class `Account` of bank accounts, with messages `withdraw` and `transfer` to operate with such bank accounts. The `Account` class is defined with a single attribute `balance`, of sort `Int`, representing the balance of an account. The `withdraw` message has two parameters, namely the addressee of the message and the amount of money to withdraw from the account. The `transfer` message will make the amount of money specified as its third argument to be transferred from the account given as first argument to the one given as second argument.

```
sort Account .
subsort Account < Cid .
op Account : -> Account .
op balance :_ : Int -> Attribute .
op withdraw : Oid Int -> Msg .
op transfer : Oid Oid Int -> Msg .
```

Rules `debit` and `transfer` below represent local transitions of the system that specify the behavior of bank accounts upon the reception of such messages. E.g., if an `Account` object receives a `withdraw` message and the amount of money to withdraw is smaller or equal than the balance of the account receiving the message, then the message is 'consumed' and the balance of the account is decremented by such an amount. Notice the synchronization of `Account` objects in the `transfer` rule.

```
vars A B : Oid .
vars BalA BalB M : Int .

crl [debit] :
  < A : Account | balance : BalA >
  withdraw(A, M)
  => < A : Account | balance : BalA - M >
  if BalA >= M .
crl [tranfer] :
  < A : Account | balance : BalA >
  < B : Account | balance : BalB >
  transfer(A, B, M)
  => < A : Account | balance : BalA - M >
     < B : Account | balance : BalB + M >
  if BalA >= M .
```

Notice that, since the __ operator is declared associative, commutative, and with identity element, we do not need to worry about the order in which objects and messages appear in the rules. And since rules describe local transitions, we do not need to worry about the rest of the objects and messages in the configuration either.

Well-formedness of objects may be automatically checked by Maude's typing system. For example, we can add declarations constraining `Account` objects:

```
sort AccountObject .
subsort AccountObject < Object .

var O : Oid .
var Bal : Int .

mb < O : Account | balance : Bal > : AccountObject .
```

Notice that with these declarations, an object `< O : Account | >` is a valid term of sort `Object`, but since the membership cannot be applied on it, it is not of type `AccountObject`.

4 The mOdCL Representation of UML Class and Object Diagrams

Different parts of UML and OCL have already been represented in MEL and Maude by different authors for different purposes, and following different approaches. For instance, in the Maude-based modeling framework MOVA [Theb], both UML class and object diagrams are formalized as MEL theories. We follow an approach similar to those used in metamodeling frameworks such as MOMENT [BM08], Maudeling [RRDV07], and e-Motions [RDV09], where class diagrams are represented as Maude object-oriented modules, and system states as configurations of objects. There are however differences on the representation of attributes, links and methods. And since we also provide support for dynamic validation, our configurations will be of objects and messages. We briefly describe in the following sections the representation of class diagrams used illustrating it with our running example.

4.1 OCL Basic Types and Expressions

OCL basic predefined types are directly mapped on predefined Maude sorts. Thus, the basic OCL types *Boolean*, *Integer*, *Real* and *String* are, respectively, mapped into the Maude predefined sorts `Bool`, `Int`, `Float`, and `String`. User-defined types are mapped on the Maude predefined sort `Oid`. OCL predefined collection types are mapped into sorts `Set`, `Bag`, `OrderedSet`, and `Sequence`. All these sorts are subsorts of `OclType`, which defines any predefined OCL type.

OCL expressions are represented as terms in the `OclExp` sort, providing syntax for the OCL operators. Thus, we can represent, e.g., the `seats-in-session` OCL invariant presented in Section 2 as:

```
op seats-in-session : -> OclExp .
eq seats-in-session
  = context Session inv : capacity >= ticket -> size() .
```

Notice that the syntax for OCL expressions in mOdCL is close to the syntax of OCL. However, they are defined in equations, being declared as constants for later reference. And because of Maude limitations to define the lexical level, spaces must be left around operators to break tokens.

Invariants, pre- and post-conditions are then specified by operators `inv`, `pre` and `post`, to build the `OclExp` to be checked when the validation process requires to check invariants, pre- or post-conditions of a given method, respectively.

```
op inv : -> OclExp .
ops pre post : OpName -> OclExp .
```

The `inv` operator defines (the conjunction of) the class invariants in the system. For our example, given the above `seats-in-session` expression and the `avoid-overlapping` expression,

```
op avoid-overlapping : -> OclExp .
eq avoid-overlapping
  = context Client inv :
       ticket -> forAll(T1 | ticket -> forAll(T2 |
           (T1 = T2)
           or (T1 . session . endTime < T2 . session . startTime)
           or (T2 . session . endTime < T1 . session . startTime))))
```

the `inv` constant can be defined as:

```
eq inv = seats-in-session and avoid-overlapping .
```

The `pre` and `post` operators must be defined for each method. For example, for the method `buyTicket` in our running example we have:

```
eq pre(buyTicket)
  --- the number of sessions starting at the given time must be 1
  = session -> select(S | S . startTime = startTime) -> size() = 1 .
```

```
eq post(buyTicket)
  = (result = null)
    or
    (--- tickets of the session must include the result ticket
     session -> select(S | S . startTime = startTime) . ticket
       -> includes(result) .
     and
     --- the number of tickets increases in 1 unit
     ((session -> select(S | S . startTime = startTime) . ticket)
         -> asSet() -
      (session -> select(S | S . startTime = startTime)
         . ticket @pre) -> asSet()) -> size() = 1) .
```

4.2 The mOdCL Representation of the System Structure

In this section, we show how to represent in mOdCL the elements of given UML class models, providing the mapping of each element in the class diagram.

Classes are represented following the standard representation of classes in Maude (see Section 3 and [CDE+07]). The `Attribute` sort provides the syntax for attributes and associations in the objects, given by its name, of sort `AttributeName`, and its type, of sort `OclType`.

```
op _:_ : AttributeName OclType -> Attribute [ctor] .
```

The only difference with the standard way to represent classes in Maude is that attributes and associations are represented as constants of the mOdCL sort `AttributeName`. Thus, for the `Cinema` class in our running example we have:

```
sort Cinema .
subsort Cinema < Cid .
op Cinema : -> Cinema [ctor] .
ops name bank session : -> AttributeName [ctor] .
```

Associations with multiplicity 1 are represented as attributes of sort `Oid`, and associations with multiplicity $\star$ as attributes of sort `Set` (for `Oid` sets). Constraints as ordered, unique, multiplicities, etc. on the associations will adjust this rule appropriately; e.g., if the association is ordered, a sequence will be used instead of a set.

An operation $op(arg_1 : type_1, \ldots, arg_n : type_n) : type$ is represented as a constant op, of sort `OpName` (of operation names), and constants $arg_1, \ldots, arg_n$, of sort `Arg` (of arguments).

Thus, e.g., for the `buyTicket` operation we have declarations

```
op buyTicket : -> OpName [ctor] .
ops startTime aClient : -> Arg [ctor] .
```

4.3 The System Behavior

The behavior of a system is given by the behavior of the methods specified in its class diagrams, which invoke one to another and perform actions, and may produce changes in the state of the

system. We assume a single thread of execution and synchronous method invocation: when a given method m_1 invokes a method m_2, the caller gets blocked until the completion of the invoked method. To manage the chaining of method invocations—an invoked method can invoke another one (or recursively to itself)—the validator uses an execution stack in which the necessary information is stored. A multithreaded execution would only require to handle multiple stacks and to identify each method invocation with the corresponding thread.

The behavior of each method will be represented as a sequence of Maude rules, and to allow the management of the stack and transparently gather the data implied in the validation of invariants, and pre- and post-conditions, these rules are assumed to follow a basic scheme which requires the use of specific messages. The use of these messages allows us to detect and handle the invocation and completion of a method (`call` and `return` messages) and, once an invoked method has been completed, to reactivate the caller (with a `resume` message), which was blocked after invoking the method. This is the only assumption, the modeler can implement the rest of the prototype according to his preferences.

The Maude rules modeling a method may need to access some information related to its invocation, as the object invoking the method or the actual parameters used in the invocation. We assume that such information is stored in an object of class `Context`, representing the execution context with the appropriate information for the running method, with the form

```
< ctx : Context | op : m, self : id, args : vars >
```

where `ctx` is the identifier of the object, `m` is the name of the active method, `id` is the identifier of the current object, and `vars` is a set of variable name - value pairs corresponding to the arguments of the method invocation and local variables. The validator infrastructure is responsible for transparently creating and handling such objects, so that the modeler can make use of it in the rules representing the behavior of methods when the value of an argument variable or a local variable is required.

The scheme for method definitions is as follows. First, although Maude allows any syntax for message declarations, we assume that messages are sent using the `call` operator. Thus, when the user wants to send an `m(args-list)` message to an object `O`, a `call(m, O, args-list)` term is placed in the configuration of objects and messages representing the system state. This `call` message is transparently processed by the validator infrastructure, which will create an object of class `Context`, with the name of the invoked method, the self object, and the actual parameters. This object is the initial execution context to be used by the specification of the method `m`.

When a `call` message is sent in a rule to invoke a given method `m`, the flow of control has to be blocked until the execution of the requested method is completed. To this end, the blocked rule waits for a `resume(m, Result)` message with the result of the invoked method.

Finally, the last rule of the implementation of a method `m` is assumed to send a message `return(result)` with the result of such a method. This `return` message will be transparently handled by the validator infrastructure, which will send a `resume` message to unblock the caller.

Basically, mOdCL uses the `call` operator to detect the invocation of a method, in which case a context for such a method is created and the necessary information is added to the execution

stack. Upon the detection of a `return` operator, mOdCL replaces the current context with the one at the top of the stack, and places a `resume` message to proceed with the execution. We sketch the `CALL` and `RETURN` infrastructure rules that manage these messages. Since a `CALL` rule is executed after a method invocation, and a `RETURN` rule is executed after a method completion, it can be used from the execution strategy to determine when to validate OCL constraints.

```
rl [CALL] :
 call(op-nm, self, args-list)
 stack(... contents of the stack ...)
 => < context : Context | ... >          --- new execution context
    stack(... new contents of the stack ...) .

rl [RETURN] :
 return(R:OclType)
 < context : Context | ... >             --- old execution context
 stack(< new-context : Context | ...>
       ... rest of the contents of the stack ...)
 => resume(op-nm, R:OclType)
    < new-context : Context | ... >      --- new execution context
    stack(... new contents of the stack ...) .
```

As an example, we show below some of the Maude rules to represent the system behavior in our running example. The `goCinema` method begins by sending a `buyTicket` message to the cinema specified as argument for the given start time. Depending on whether there are tickets available or not for that session, a `Ticket` object or a `null` will be returned. And then, depending on the result of this message, the ticket would be paid. We illustrate the approach by giving a few rules.

The `GO-CINEMA-1` rule is the first rule to be executed for the `goCinema` method (see Section 2). Upon the occurrence of a

```
call(goCinema, Self, (arg(cinema, Cn), arg(startTime, St))
```

the `CALL` rule above handles the stack as appropriate and puts in the configuration an object

```
< ctx : Context | op : goCinema, self : Self,
                  args : arg(cinema, Cn), arg(startTime, St))
```

This rule receives the execution context in the `Context` object and invokes the `buyTicket` method on a given cinema, to get a ticket for the session.

```
rl [GO-CINEMA-1] :
  < ctx : Context | op : goCinema, self : Self,
                        args : arg(cinema, Cn), arg(startTime, St))
  < Self : Client | cinema : Set{C, LC}, Atts1 >
  < C : Cinema | name : Cn, session : Set{S, LS}, Atts2 >
  < S : Session | startTime : St, Atts3 >
  => < Self : Client | cinema : Set{C, LC}, Atts1 >
     < C : Cinema | name : Cn, session : Set{S, LS}, Atts2 >
     < S : Session | startTime : St, Atts3 >
```

```
    < ctx : Context | op : goCinema, self : Self,
                      args : arg(cinema, Cn), arg(startTime, St))
    call(buyTicket, C, (arg(startTime, St), arg(client, Self))) .
```

The BUY-TICKET-1-NO-FREE-SEATS rule below models the first execution step in the implementation of the buyTicket method. This rule is applied if there are no free seats in the chosen session, in which case the method execution ends, returning a null ticket in a return message.

```
crl [BUY-TICKET-1-NO-FREE-SEATS] :
  < ctx : Context | op : buyTicket, self : Self,
                    args : arg(startTime, St), arg(client, Cl))
  < Self : Cinema | session : Set{S, LS}, Atts1 >
  < S : Session | startTime : St, ticket : TS, capacity : C, Atts2 >
  => < Self : Cinema | session : Set{S, LS}, Atts1 >
     < S : Session | startTime : St, ticket : TS,
                     capacity : C, Atts2 >
     < ctx : Context | op : buyTicket, self : Self,
                       args : arg(startTime, St), arg(client, Cl))
     return(null)
  if size(TS) >= C .
```

The GO-CINEMA-2-FAIL rule is the second rule in the implementation of the goCinema method. The flow of control is blocked after calling the buyTicket method, and it is released when the resume message, is received. Furthermore, as this rule ends the execution of the goCinema method, it sends a return message with the result. In this case we do not use any information of the execution context, and therefore, we do not use the Context object.

```
rl [GO-CINEMA-2-FAIL] :
  resume(buyTicket, null)
  => return(false) .
```

5 Validation of OCL Constraints

The mOdCL evaluator is the component responsible for evaluating OCL expressions on configurations representing given system states. It provides the eval function, which specifies how to interpret each valid OCL expression on a given Configuration representing the system state.

```
 op eval : OclExp Configuration -> OclType .
```

For example, given the following state configuration, of sort Configuration, corresponding to a given object diagram for the Cinema model,

```
op state : -> Configuration .
eq state
 = < c : Cinema | name : "Cnml", bank : b, session : Set{s1 ; s2} >
```

```
< s1 : Session  | startTime : 1100, endTime : 1230,
                  ticket : Set{t2}, capacity : 40, price : 5 >
< s2 : Session  | startTime : 1300, endTime : 1430,
                  ticket : Set{t1}, capacity : 20, price : 6 >
< j : Client | cinema : Set{c}, ticket : Set{t1}, debitCard : 4 >
< m : Client | cinema : Set{c}, ticket : Set{t2}, debitCard : 7 >
< b : Bank | debitCards : qas(4, a1) $$ qas(7, a2) >
< a1 : Account | balance : 100 >
< a2 : Account | balance : 1000 >
< t1 : Ticket | seat : 1, session : s2, client : j >
< t2 : Ticket | seat : 1, session : s1, client : m > .
```

we can use the `eval` function as follows to statically validate, e.g., the above OCL expression `seats-in-session`:[2]

```
Maude> red in mOdCL : eval(seats-in-session, state) .
result: true
```

To dynamically validate an UML model, we must provide both the Maude representation of the system behavior and the mOdCL representation of the OCL constraints. Given the module `CINEMA`, which implements our model example, according to the scheme proposed in Section 4.3, and defines `state` as an initial configuration, and given the `CINEMA-CONSTRAINTS` module, which defines functions `inv`, `pre` and `post`, as in Section 4.1, we can use the mOdCL `validate` command to validate the execution of the prototype of our system as follows:

```
Maude> (validate CINEMA with CINEMA-CONSTRAINTS from state .)
```

This way, the modeler can specify a Maude prototype as usual object-oriented modules, but taking into account that the system structure for the model must be specified according to the description in Section 4.2, and the rules which specify the system behavior must use the basic scheme proposed in Section 4.3, concerning the invocation of methods.

If, as in this case, the system can be executed without violating any restriction, we get its final configuration as result. Otherwise, the execution is aborted and an error message is given, reporting on the the state and the constraint involved in such an error.

6 The mOdCL Validator Architecture

The mOdCL validator lays on two components, the OCL evaluator and the `metaOCLRewrite` strategy. In the following sections we sketch the rationale of these components.

6.1 The mOdCL Evaluator

The evaluator component has been designed as a shallow embedding of OCL in Maude. For this purpose, it provides OCL syntax to Maude and defines an `eval` operator to evaluate valid OCL

[2] The Maude `red` (or `reduce`) command simplifies the given term using the equations in the specified module.

expressions. This is completed with the library COLLECTION, which supports the implementation of the OCL Standard Library.

The evaluator pivots around the definition of sorts OclType, to represent any predefined OCL type, and OclExp, to represent valid OCL expressions, and works on system states represented as object configurations (of sort Configuration), where the elements in the UML class diagram have been represented as explained in Section 4.2.

As we mentioned in Section 4.1, predefined OCL types are directly mapped on predefined Maude types, and furthermore, collection types are built by using the facilities provided by the COLLECTION library. All these types are subsorts of OclType.

```
subsort Int Float String Bool Oid < BasicType .
subsort Set Bag OrderedSet Sequence  < Collection .
subsort BasicType Collection < OclType .
```

The SYNTAX module provides definitions for all valid OCL expressions as terms of sort OclExp, and the EVAL module provides equations to evaluate them. The evaluation of expressions in the context of pre- and post-conditions of operations requires two system states, one representing the state before executing the operation and another representing the state after its completion, whereas others expressions only require the current state. Thus, we provide two eval functions:

```
op eval : OclExp Configuration Configuration -> OclType .
op eval : OclExp Configuration -> OclType .
```

Only to provide some intuition on the general scheme used, we show the syntactic definitions in Maude for the includes operator on OCL collections:

```
op _-> includes(_) : OclExp OclExp -> OclExp .

vars E1 E1 : OclExp .
var  C : Configuration .

eq eval(E1 -> includes(E2), C) = eval(E1, C) in eval(E2, C) .
```

where the infix operator _in_ is provided by the COLLECTION library to determine if a given element is contained in a given collection.

6.2 Implementation of the `metaOCLRewrite` Strategy

Maude provides key metalevel functionality for metaprogramming and for writing strategies to control the execution of systems. In general, strategies are defined in extensions of the predefined META-LEVEL module by using predefined functions in it as building blocks. The META-LEVEL module also provides sorts Term and Module, so that the representations of a term T and of a module M are, respectively, a term $\overline{T}$ of sort Term and a term $\overline{M}$ of sort Module. Constants (resp. variables) are metarepresented as quoted identifiers that contain the name of the constant (resp. variable) and its type separated by a dot (resp. colon), e.g.,

`'true.Bool` (resp. `'B:Bool`). Then, a term is constructed in the usual way, by applying an operator symbol to a comma-separated list of terms. For example, the term `eval(S, true)` of sort `OclExp` is metarepresented as the term `'eval['S:State, 'true.Bool]` of sort `Term`. In what follows, we will use the over-line notation to denote the metarepresentation of modules and terms.

Of particular interest for our purposes are the `META-LEVEL` partial functions `metaReduce` and `metaXapply`:[3]

```
op metaReduce : Module Term ~> Term .
op metaXapply : Module Term Qid ~> Term .
```

`metaReduce` takes a module $\overline{M}$ and a term $\overline{T}$, and returns the metarepresentation of the normal form of T in M, that is, the result of reducing T as much as possible using the equations in M. `metaXapply` takes as arguments a module $\overline{M}$, a term $\overline{T}$, and a rule label L, and returns the metarepresentation of the term resulting from applying the rule with label L in M on the term T once.

Rewrite systems can be executed in Maude with commands *rewrite* and *frewrite*, which use different built-in strategies to control the way to apply the rules on the current state. However, since we want to validate OCL constraints, we need a different way to control the rule application that allow to detect the system states to validate, and to check the required OCL constraints. To this end, we define a `metaOCLRewrite` strategy (see Figure 2), which is responsible for monitoring rule execution.

The `metaOCLRewrite` strategy executes the rules modeling the system until a reached term, representing a system state, cannot be further rewritten, or violates given constraints. In each execution step, the strategy tries to apply all rules in the module. Once a rule can be applied, and the corresponding constraints are satisfied, the resulting term becomes the current term and it starts again. Otherwise, an error message is generated.

To try all the possible rules in some given order, the `REW-OCL` module in Figure 2 has an `ITERATOR` theory as parameter. This theory defines a sort `Iterator`, which comes with a function `iterator` that returns an iterator on the labels of a module, and functions that allow us to iterate on them: `hasNext`, which determines whether there are remaining labels in the structure; `next`, which returns the iterator with the next label set to be the current one; and `getLabel`, which return the current label in the sequence.

The `metaOCLRewrite` operator uses the `inv` function to get the class invariants. After using the `eval` function to check them on the initial state, it calls the `metaOCLRewriteAux` auxiliary function to execute and validate the system according to the constraints defined.

The `metaOCLRewriteAux` uses the iterator to get a candidate rule. If it is not possible (`not hasNext(C)`), the execution stops. Otherwise, it gets a rule label and it uses the predefined `metaXapply` function to execute the rule. If it can be applied (notice the `T' :: Term` to check that the result is of a valid sort), the system state is validated; otherwise, it gets an error term in `T'`, and it resumes the execution to try the next rule.

As we explained in Section 4.3, the mOdCL evaluator uses an execution stack to manage method invocations. Basically, it executes a `CALL` rule after a method invocation and a `RETURN`

[3] We have simplified the form of these functions for presentation purposes, since we do not need here their complete functionality. See [CDE+07] for their actual descriptions.

```
mod REW-OCL{X :: ITERATOR} is
  inc META-LEVEL .
  op error : String -> Msg .
  op metaOCLRewrite : Module Term ~> Term .
  op metaOCLRewriteAux : Module Term X$Iterator ~> Term .
  op checkCall   : Module Term ~> Term .
  op checkReturn : Module Term Term ~> Term .

  var M : Module .       vars T T' I P Q opN : Term .
  var C : X$Iterator .   var  L : Qid .

  ceq metaOCLRewrite(M, T) = metaOCLRewriteAux(M, T, iterator(M))
    if I := metaReduce(M, 'inv.OclExp)
    /\ metaReduce(M, 'eval[I, T]) = 'true.Bool .
  ceq metaOCLRewriteAux(M, T, C) = T if not hasNext(C) .
  ceq metaOCLRewriteAux(M, T, C)
    = if T' :: Term
      then (if L == 'CALL
            then checkCall(M, T')
            else (if L == 'RETURN
                  then checkReturn(M, T, T')
                  else metaOCLRewriteAux(M, T', iterator(M))
                  fi)
            fi)
      else metaOCLRewriteAux(M, T, next(C))
      fi
    if L := getLabel(C) /\ T' := metaXapply(M, T, L)
    [owise] .

  ceq checkCall(M, T)
    = if metaReduce(M, 'eval[P, T]) == 'true.Bool
      then metaOCLRewriteAux(M, T, iterator(M))
      else 'error['"Precondition failed"]
      fi
    if opN := metaReduce(M, 'getOpName[T])
    /\ P := metaReduce(M, 'pre[opN]) .
  ceq checkReturn(M, T, T')
    = if metaReduce(M, 'eval[Q, T]) == 'true.Bool
      then (if metaReduce(M, 'eval[I, T]) == 'true.Bool
            then metaOCLRewriteAux(M, T', iterator(M))
            else 'error['"Invariant failed"]
            fi)
      else 'error['"Postcondition failed"]
      fi
    if opN := metaReduce(M, 'getOpName[T])
    /\ Q := metaReduce(M, 'post[opN])
    /\ I := metaReduce(M, 'inv.OclExp) .
endm
```

Figure 2: The REW-OCL module.

rule on its completion. They are infrastructure rules, hidden to the user, but we make use of them in the `metaOCLRewriteAux` operation to detect a method invocation (when the label of the executed rule is `'CALL`) or a method completion (when it is `'RETURN`). In these cases, the suitable validation is performed by using the `checkCall` or `checkReturn` operators, respectively.

Now, we can validate the dynamics of our example. Assuming a `REW-OCL` module which defines the `metaOCLRewrite` execution strategy, and given the module `CINEMA`, which implements our model example and defines `state` as an initial configuration, and given the module `CINEMA-CONSTRAINTS`, which defines functions `inv`, `pre` and `post` as above, we can create the `CINEMA-TEST` module as follows:

```
mod CINEMA-TEST is
  inc REW-OCL{IteratorSeq} + CINEMA + CINEMA-CONSTRAINTS .
endm
```

And we can execute the system from its initial configuration as follows:

```
Maude> red metaOCLRewrite(CINEMA-TEST, state) .
```

This is in fact the computation that takes place when using the `validate` command, which hides all references to meta-representations, stategies, etc., which are kept hidden to the user.

7 Conclusions and future work

We have presented the mOdCL system, which can both evaluate OCL expressions on given system configurations and validate OCL constraints on the execution of Maude system prototypes.

Our validator lays on both an OCL expression evaluator and an execution strategy, which is responsible for controlling when and how to validate each OCL constraint. The user only has to specify the system behavior and the constraints to fulfill. The strategy is independent of the system to validate, and this level of modularity allows us to extend the strategy if we are interested on experimenting with new validation approaches.

We provide guidelines to represent UML class and object diagrams in mOdCL. Furthermore, we provide a basic scheme to represent the method related concepts. The system behavior representation must comply with the scheme proposed, but we do not impose any further restriction: the user can complete the system according to his preferences.

We are currently working on the automatic generation of mOdCL representations from UML class and object diagrams, and on the use of UML behavior diagrams to (semi-)automate the Maude representation of the system behavior. Different authors have previously explored such a approach for different purposes. First steps on the use of sequence diagrams to systematically obtain skeletons for the system behavior representation, which can be later completed by the modeler, have already been taken.

Given the specification of a system with its OCL constraints, many possibilities are opened using Maude formal environment: theorem proving, model checking, reachability analysis, etc. We are also exploring other uses of the approach, taking advantage of the Maude system and tools around it. E.g., the possibility of using our validator in conjunction with the reachability tools of Maude to automatically generate test case violating the OCL constraints.

Bibliography

[BCR06] A. Boronat, J. A. Carsí, I. Ramos. Algebraic Specification of a Model Transformation Engine. In Baresi and Heckel (eds.), *FASE*. Lecture Notes in Computer Science 3922, pp. 262–277. Springer, 2006.

[BM08] A. Boronat, J. Meseguer. An Algebraic Semantics for MOF. Pp. 377–391 in [FI08].

[BW08] A. D. Brucker, B. Wolff. HOL-OCL: A Formal Proof Environment for UML/OCL. Pp. 97–100 in [FI08].

[CCR07] J. Cabot, R. Clarisó, D. Riera. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In Stirewalt et al. (eds.), *ASE*. Pp. 547–548. ACM, 2007.

[CDE+02] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, J. F. Quesada. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.* 285(2):187–243, 2002.

[CDE+07] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. L. Talcott (eds.). *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Lecture Notes in Computer Science 4350. Springer, 2007.

[CDH+07] M. Clavel, F. Durán, J. Hendrix, S. Lucas, J. Meseguer, P. C. Ölveczky. The Maude Formal Tool Environment. In Mossakowski et al. (eds.), *CALCO*. Lecture Notes in Computer Science 4624, pp. 173–178. Springer, 2007.

[CPC+04] D. Chiorean, M. Pasca, A. Cârcu, C. Botiza, S. Moldovan. Ensuring UML Models Consistency Using the OCL Environment. *Electr. Notes Theor. Comput. Sci.* 102:99–110, 2004.

[FI08] J. L. Fiadeiro, P. Inverardi (eds.). *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Lecture Notes in Computer Science 4961. Springer, 2008.

[FT01] J. L. Fernández-Alemán, A. Toval-Álvarez. Seamless Formalizing the UML Semantics through Metamodels. In *Unified Modeling Language: Systems Analysis, Design and Development Issues*. Pp. 224–248. 2001.

[GBR07] M. Gogolla, F. Büttner, M. Richters. USE: A UML-based specification environment for validating UML and OCL. *Sci. Comput. Program.* 69(1-3):27–34, 2007.

[Kna00] A. Knapp. Generating Rewrite Theories from UML Collaborations. In Futatsugi et al. (eds.), *CAFE: An Industrial-Strength Algebraic Formal Method*. Pp. 97–120. Elsevier, 2000.

[MB09] F. Mokhati, M. Badri. Generating Maude Specifications From UML Use Case Diagrams. *Journal of Object Technology* 8(2):319–136, 2009.

[Mes92] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science* 96:73–155, 1992.

[Mes98] J. Meseguer. Membership Algebra as a Logical Framework for Equational Specification. In Parisi-Presicce (ed.), *Recent Trends in Algebraic Development Techniques*. Lecture Notes in Computer Science 1376, pp. 18–61. Springer-Verlag, 1998.

[MM02] N. Martí-Oliet, J. Meseguer. Rewriting Logic as a Logical and Semantic Framework. In Gabbay and Guenthner (eds.). Volume 9, pp. 1–87. Kluwer Academic Publishers, second edition, 2002.

[OCLa] Object Management Group. Object Constraint Language (OCL) Specification (formal/2010-02-01).

[OCLb] OCL Portal. http://st.inf.tu-dresden.de/oclportal/.

[RDV09] J. E. Rivera, F. Duran, A. Vallecillo. A graphical approach for modeling time-dependent behavior of DSLs. In *Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. VLHCC '09, pp. 51–55. 2009.

[RDV10] J. E. Rivera, F. Durán, A. Vallecillo. On the Behavioral Semantics of Real-Time Domain Specific Visual Languages. In Ölveczky (ed.), *WRLA*. Lecture Notes in Computer Science 6381, pp. 174–190. Springer, 2010.

[RG03] M. Richters, M. Gogolla. Aspect-Oriented Monitoring of UML and OCL Constraints. In *In AOSD Modeling With UML Workshop, 6th International Conference on the Unified Modeling Language (UML*. 2003.

[RRDV07] J. R. Romero, J. E. Rivera, F. Durán, A. Vallecillo. Formal and Tool Support for Model Driven Engineering with Maude. *Journal of Object Technology* 6(9):187–207, 2007.

[Thea] The Dresden OCL Team. The Dresden OCL toolkit. Available at http://dresden-ocl.sourceforge.net/.

[Theb] The MOVA Group. The MOVA tool: a validation tool for UML. Available at http://maude.sip.ucm.es/mova/.

[WK03] J. B. Warmer, A. Kleepe. *The Object Constraint Language*. Addison Wesley, 2003.