Validating OCL Constraints on Maude Prototypes of UML Models

Francisco Durán Manuel Roldán

September 8, 2012

Abstract

Current tools to dynamically validate OCL constraints require a complete system implementation, whereas those tools which allow to validate constraints directly on the UML model only can do it statically, on concrete system states. We propose the use of high level Maude prototypes of UML models on which constraints can be dynamically validated during their simulation, prior the final implementation.

We provide the necessary infrastructure to implement our approach and basic guidelines to obtain Maude prototypes from UML class diagrams. Our tool supports either single thread or multi thread execution and allows synchronous and asynchronous messages.

This document shows our proposal and presents some use cases, including the simulation and validation of single-threaded and multi-threaded prototypes.

1 Introduction

The Unified Modeling Language [35] (UML) is a standard of the Object Management Group (OMG) widely accepted and used to model software systems using an object oriented approach, which relies on the use of a set of diagrams to model the structure and behavior of the system from different perspectives.

The Object Constraint Language [42, 34] (OCL) is a formal language, defined as part of the standard UML in order to complete the definition of the different elements of a UML model with precise textual descriptions. An interesting use of OCL expressions is to describe constraints on the UML model (invariants and pre- and post-conditions) to be checked at runtime. This has clear benefits, among which we may highlight the possibility of detecting situations in which implementations deviate from their designs (either during its development or maintenance), or simply to facilitate testing by using constraints as test oracles.

The violation of a given constraint is a sign of an incorrect implementation of the model, but it can also come from an incorrect model specification. For this reason, most OCL tools support the validation of OCL constraints directly on UML models (see, e.g., [20, 41, 3, 12]). These tools represent class diagrams in the supporting formalism and statically check whether such constraints hold on given system states (snapshots), which can be obtained, for example, from object diagrams. They are, however, unable to perform the validation of dynamic design models. But, if we are able to specify the behavior of systems, why not supporting its dynamic validation?. It would be very useful to have tools to analyze dynamic design models, e.g., as described by UML sequence or activity diagrams, so that we can detect design mistakes in the structure of models as well as on their dynamics. We would say more, in our opinion, mistakes in the dynamics of systems are typically more subtle and more difficult to detect, and such kind of tools would be even more relevant. And not only for complete models, but also for partial models so that these checks can be accomplished all along the system design, from its very early stages. This way, some errors in the model, or in the constraints on them, could be detected earlier, during the modeling phase, long before an implementation is available.

This document describes our proposal to dynamically validate UML models by checking that the OCL constraints imposed to the model are fulfilled at runtime. Instead of monitor the execution of the final implementation of the model, we propose to play with Maude specifications which can be used as high level prototypes of UML models.

Maude [9, 10] is an executable formal specification language based on rewriting logic, with a rich set of validation and verification tools [10, 11], increasingly used as support to the development of UML, MDA, and OCL tools (see, e.g., [4, 40, 12]). Furthermore, Maude has demonstrated to be a good environment for rapid prototyping, and also for application development (see surveys [10, 31]). 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 [27]. 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.

Several authors have already explored how to represent different UML diagrams in Maude (see, e.g., [23, 18, 32]), and we do not intend to describe here systematic transformations of UML diagrams into Maude. We focus on class and object diagrams, and we provide a basic scheme for specifying UML dynamic models as rewrite systems. The use of the proposed scheme allows us, not only to simulate UML models by executing their corresponding Maude specifications, but also to dynamically validate the OCL constraints included in their class diagrams. Furthermore, our Maude prototypes are not restricted to these uses, we could use the tools of the Maude environment to perform other kinds of analyses. For example, we have already explored the use of our Maude prototypes in conjunction with the Maude reachability-analisis tool to locate scenarios which fulfill or violate given constraints [13], but further uses of Maude's reachability analysis tool and model checker, as the ones already proposed for domain specific languages in [37], could be considered, as uses of Maude's theorem prover to prove properties on OCL constraints, as their independence or unsatisfiability.

Basically, the dynamic validation of the constraints of a model requires two tasks: the identification of those states on which constraints must be satisfied, and the evaluation of the satisfaction of the appropriate constraints. Our approach provides support for the location of relevant states by using a given *scheme* to specify system prototypes, and the evaluation of the OCL constraints on such states is made by mOdCL, our OCL interpreter for Maude, which can evaluate, not only OCL constraints, but OCL expressions in general (see [15][i] for a detailed description of mOdCL).

[i] referencias

We are aware that end users are interested in validating UML models, not in the validation process itself. In practice, users do not need any knowledge on the internals of our validation tool. In fact, the only task a user has to do is to provide a Maude prototype for the system, and a module with the constraints imposed to such a prototype.

Our long term goal is to use Maude as back end of a graphical tool which hides Maude details from basic users. It would be desirable to provide the user the possibility of obtaining the Maude prototype directly from the diagrams representing the UML model. Although this goal is not near yet, we are working on the the automation of the prototype specification from behavior diagrams. Our first step in this direction is the use of sequence diagrams to systematically obtain skeletons for the system behavior representation, which can be later completed by the modeler.

The document 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 rest of the paper. Section 4 presents our proposal to represent UML models as Maude prototypes, considering their static and dynamic aspects and dealling with single threaded and multi-threaded execution. Later, Section 5 shows how to validate OCL constraints on those prototypes, introducing the mOdCL evaluator ant its use for static and dynamic validation. Section 6 revises related work. Appendices A and B show a complete example for simulating a single-threaded example, Appendix C shows an example where we simulate multi-threaded prototypes, Appendix E shows our extension of mOdCL for dealing with the *@pre* operator in the context of the dynamic validation. Finally, Appendix D shows examples where we validate OCL constraints during the execution of single-threaded and multi-threaded examples.

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 report. In it, 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 tick-

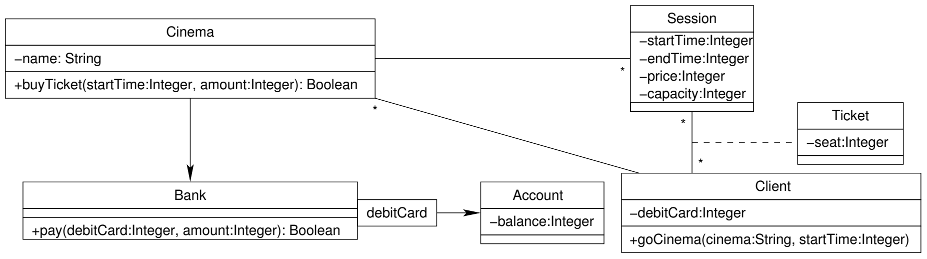

Figure 1: Class diagram for the cinema example.

ets 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`.

We consider three public operations: `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` operation 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)
```

In coming sections, we will specify this UML model in Maude, and we will use the Maude interpreter to simulate such a model and our validation tool to validate the imposed constraints.

3 Rewriting Logic and Maude

Maude [9, 10] is a high-performance reflective language and system that integrates an equational style of functional programming with rewriting logic computation, supporting specification and programming for a wide range of applications [10, 31].

Rewriting logic is parameterized by its underlying equational logic. In the case of Maude, the underlying equational logic is *membership equational logic* [29], 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 [28] is a logic of change that can naturally deal with state and with highly nondeterministic concurrent computations. In particular, it supports very well concurrent object-oriented computation [30]. In rewriting logic, 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 rewriting logic is then specified by rewrite *rules* of the form $t \rightarrow t'$, where t and t' are Σ-terms. This rewriting happens modulo the equations E, describing in fact local transitions $[t]_E \rightarrow [t']_E$. 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 state in which t has been replaced by 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.

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

$$\texttt{<}\ O\ \texttt{:}\ C\ \texttt{|}\ a_1\texttt{:}v_1\texttt{,}\ \ldots\texttt{,}\ a_n\texttt{:}v_n\ \texttt{>}$$

where O is the identifier of the object, and v_i are the current values of its attributes (with appropriate types). See [10] 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.

4 UML Prototyping with Maude

Rewriting logic defines a suitable framework to formally specify different languages and systems [27, 30, 10], and Maude provides an efficient interpreter with which to execute the specifications, which can be used as a rapid prototype of systems under study. Different authors have previously proposed rewriting logic and Maude as a vehicle to formalize UML [18, 12, 4, 32, 7] and to build early prototypes during the analysis and design [47] phases. In this section we present our proposal to develop Maude specifications representing UML models which can be used with different purposes. We will show how to represent the structure and the behavior of a UML model and how to execute the system. Later, in Section 5 we will show how given constraints can be added and how they can be validated.

4.1 Representation of system structure

In UML the system structure is given by its class diagram, therefore, we must provide the mapping of the elements in class diagrams to Maude. We follow an approach similar to those used in metamodeling frameworks such as MOMENT [4], Maudeling [40], and e-Motions [38], where class diagrams are represented as Maude object-oriented modules, and system states as configurations of objects (of sort `Configuration`). There are however differences on the representation of attributes, associations and methods. And since we will also provide support for dynamic validation, our configurations will be of objects and messages. Furthermore, as we will use the mOdCL evaluator (see Section 5.1) to validate given OCL constraints, we represent the elements of class diagrams by means of mOdCL sorts.

Classes are represented following the standard representation of classes in Maude for object oriented systems (see Section 3 and [10]). The `Attribute` sort provides the syntax for attributes and associations in the objects. An attribute is given by its name, of the mOdCL sort `AttributeName`, and its type, of sort `OclType`(ii).

(ii) Cambiar OclType por OclAny

```
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` and its value is represented by OCL types (of sort `OclType` in mOdCL). 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 `*` 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.

Similarly, we use mOdCL sorts to map the elements of operations. An operation *op* (arg_1 : $type_1$, . . . , arg_n : $type_n$) : *type* is represented as a constant *op* , of sort `OpName` (of operation names), and constants arg_1, . . . , 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] .
```

Appendix A contains the specification of the module `CLASSES-CINEMA`, with the complete representation of the system structure for the cinema model.

4.2 Representation of system behavior. Single-threaded execution

The behavior of a system is given by the behavior of the operations specified in its class diagram. Each of these operations may invoke others of these operations, perform actions or produce changes in the state of the system. We will introduce our approach with the simplest case, given by a single thread of execution and synchronous messages—when a given operation invokes another operation, the caller gets blocked until the completion of the invoked one—, Section 4.5 presents its generalization to multi-threaded execution.

The behavior of each operation will be specified as a sequence of Maude rules, which may require access to information related to its invocation or its computation, as the object invoking the operation or the actual parameters used in its invocation. These rules may assume that such information is stored in an object of class `Context`, representing the execution context with the appropriate information for the running operation, 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 operation, *id* is the identifier of the current object, and *vars* is a set of pairs with the name of a parameter or a local variable and its actual value.

To manage the chaining of operation invocations—an invoked operation can invoke another one (or recursively to itself)—we provide an execution stack in which the execution contexts of pending operations are stored. The stack management is specified in a `STACK` module (see Section 4.3), which is responsible

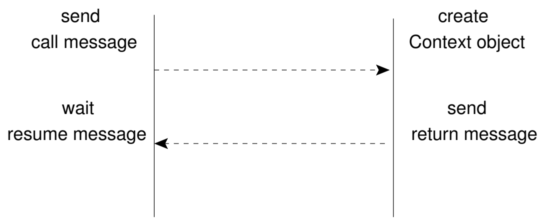

Figure 2: Basic scheme for the specification of operation behavior.

for creating and handling execution contexts. The execution stack is completely transparent for the user modeling the system, which simply uses the execution context of the active operation when necessary during the specification of the behavior of an operation. For example, if he needs to know the value of some parameter, it can be retrieved from the `args` attribute of the active `Context` object.

To make the management of the stack transparent to the user, the invocation and the completion of operations must be identified. For this reason we assume that the rules specifying the behavior of a given operation follow a basic scheme, which requires the use of specific messages, namely, `call`, `return`, and `resume` messages, as depicted in Figure 2.

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 a message `m(args-list)` to an object `O`, a message `call(m, O, args-list)` is placed in the configuration of objects and messages representing the system state. This `call` message is processed by the stack infrastructure, which will create an object of class `Context`, with the name of the invoked operation, the *self* object, and the actual parameters. This object is the initial execution context to be used by the specification of the operation `m`.

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

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

Basically, we use the `call` operator to detect the invocation of an operation, in which case a context for such an operation is created and the necessary information is added to the execution stack. Upon the detection of a `return` message, the current context is replaced by the one at the top of the stack, and a new `resume` message is sent to proceed with the execution.

To illustrate this rule scheme, we show below some of the Maude rules to represent the system behavior in our running example. Figure 3 shows the sequence diagram for the `goCinema` operation, to buy a ticket for a given session

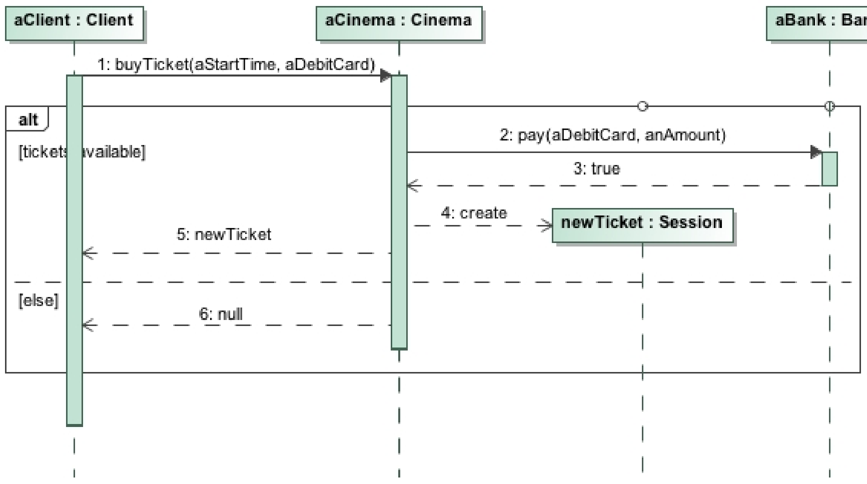

Figure 3: Sequence diagram for the `goCinema` operation.

in a cinema. It begins by sending a `buyTicket` message to the cinema for a given start time. Depending on whether there are tickets available or not for the specified session, a `Ticket` object or a `null` value will be returned. And then, depending on the result of this message, the ticket would be paid.

The `GO-CINEMA-1` rule below is the first rule to be executed for the `goCinema` operation. Upon the arrival of a message

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

for variables of appropriate types `Self`, `Cn`, and `St`, the stack infrastructure handles the stack as expected and puts in the configuration a `Context` object

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

The `GO-CINEMA-1` rule receives the execution context in the `Context` object and sends a `buyTicket` message to a cinema of its choice, 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} >
  < C : Cinema | name : Cn, session : Set{S, LS} >
  < S : Session | startTime : St >
  => < Self : Client | cinema : Set{C, LC} >
     < C : Cinema | name : Cn, session : Set{S, LS} >
     < S : Session | startTime : St >
     < 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 operation. This rule is applied if there are no free seats in the chosen session, in which case the operation 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} >
  < S : Session | startTime : St, ticket : TS, capacity : C >
  => < Self : Cinema | session : Set{S, LS} >
     < S : Session | startTime : St, ticket : TS, capacity : C >
     < 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 operation. The flow of control is blocked after calling the buyTicket operation, and it is released when the resume message is received. Furthermore, as this rule ends the execution of the goCinema operation, it sends a return message with the result. In this case we do not use any information of the execution context.

```
rl [GO-CINEMA-2-FAIL] :
  < ctx : Context | op : goCinema >
  resume(buyTicket, null)
  => < ctx : Context | op : goCinema >
     return(false) .
```

The rest of the rules specifying the goCinema operation and the complete Maude specification of this example can be found in Appendix B.

4.3 The STACK module

The proposed scheme assumes an execution stack to deal with execution contexts, provided as part of the infrastructure of the proposal. As we will show in Section 5, the infrastructure responsible for validating OCL constraints pivots around the equations that manage the stack. The definition of the stack is provided by the STACK module in Figure 4.

The STACK module defines classes Stack and Context. A stack is represented by an object of class Stack, with only one attribute state, which maintains a list of objects representing the sequence of execution contexts of those operations waiting for the completion of any previous call. The state attribute is declared of sort List{Object}, which has nil as empty list and _::_ as concatenation operator. As explained above, a Context object contains the current operation name (opN), its arguments (args) and the invoking object (obj), where OpName and List{Arg} are sorts defined by the mOdCL evaluator.

```
mod STACK is
  pr mOdCL .
  pr LIST{Object} * {op __ to _::_} .

  op Stack : -> Cid [ctor] .
  op state :_ : ListM{Object} -> Attribute [gather (&)] .

  op Context : -> Cid [ctor] .
  op opN :_ : OpName -> Attribute [ctor gather (&)] .
  op args :_ : ArgsList -> Attribute [ctor gather (&)] .
  op obj :_ : Oid -> Attribute [ctor gather (&)] .

  op call : OpName Oid ArgsList -> Msg [ctor] .
  op return : OclType -> Msg [ctor] .
  op resume : OpName OclType -> Msg [ctor] .

  op [_] : Configuration -> Configuration .

  vars Op Op' : OpName .
  vars AL AL' : ArgsList .
  vars S S' O : Oid .
  var R : OclType .
  var ST : List{Object} .

  ops stack ctx : -> Oid .

  eq [INIT] : [ Cf ] = < stack : Stack | state : nil > Cf  .
  eq [CALL] :
    call(Op', S', AL')
    < ctx : Context | opN : Op, obj : S, args : AL >
    < stack : Stack | state : ST >
    = < ctx : Context | opN : Op', obj : S', args : AL' >
      < stack : Stack |
        state : < ctx : Context |
                    opN : Op, obj : S, args : AL > :: ST > .
  eq [RETURN] :
    return(R)
    < ctx : Context | opN : Op, obj : S, args : AL >
    < stack : Stack |
      state : < ctx : Context |
                  opN : Op', obj : S', args : AL' > :: ST >
    = resume(Op, R)
      < ctx : Context | opN : Op', obj : S', args : AL' >
      < stack : Stack | state :  ST > .
endm
```

Figure 4: The STACK module.

```
mod CINEMA-TEST is
  pr CINEMA .

  op init-state : -> Configuration .
  eq init-state
   = < c : Cinema | name : "Cnml", bank : b, session : Set{s1, s2} >
     < s1 : Session  | startTime : 1100, endTime : 1230,
                       ticket : Set{t1}, capacity : 40, price : 5 >
     ... the rest of sessions
     < j : Client | cinema : Set{c}, ticket : Set{}, debitCard : 4 >
     ... the rest of clients
     < b : Bank | debitCards : qas(4, a1) >
     < a1 : Account | balance : 100 >
     ... the rest of accounts
     < t1 : Ticket | seat : 1, session : s2, client : h >
     ... the rest of sold tickets
endm
```

Figure 5: Part of the CINEMA-TEST module.

Finally, the STACK module defines call, return and resume messages, and also CALL and RETURN equations to handle them. The stack management is hidden from the user, which simply uses call, return and resume messages as above. To this end, the initial system state is expected to be given as a term of sort Configuration enclosed by square brackets. An equation INIT includes an empty stack in such a system configuration.

The CALL equation is responsible for processing call messages, by storing the actual execution context in the stack and creating a new execution context for the invoked operation. The RETURN equation is responsible for processing return messages, by replacing the actual execution context by the top of the stack and including a resume message in the system configuration. This allows to unlock the rule waiting for the completion of the invoked operation.

4.4 The system execution

Once we have a Maude prototype for the UML model we can define an initial state and use the Maude rewrite command to execute the system. For our cinema model we can, for example, create a CINEMA-TEST module which imports the CINEMA module with the system prototype and defines an init-state constant with a configuration representing the initial state as show in Figure 5[1].

We can then execute the system as is usual in Maude, obtaining a final state with some tickets sold:

```
  Maude> rewrite in CINEMA-TEST : [ init-state ] .
  result Configuration :
```

[1]See in Appendix B the complete configuration and the real execution results

```
    < c : Cinema | name : "Cnml", bank : b, session : Set{s1} >
    < s1 : Session | startTime : 1100, endTime : 1230,
                     ticket : Set{t1, t2}, capacity : 40, price : 5 >
    ... the rest of sessions
    < j : Client | cinema : Set{c}, ticket : Set{t2}, debitCard : 4 >
    ... the rest of clients
    < b : Bank | debitCards : qas(4, a1) >
    < a1 : Account | balance : 95 >
    ... the rest of accounts
    < t1 : Ticket | seat : 1, session : s2, client : h >
    < t2 : Ticket | seat : 1, session : s1, client : j >
    ... the rest of sold tickets
```

4.5 Multi-Threaded Execution

Multi-threaded execution requires some way to activate new execution threads when necessary. Furthermore, a different stack management is necessary to deal with the chaining of invocation in each active thread. We use a `newThread` message which should be thrown to create a new execution thread.

```
  op newThread : Oid -> Msg [ctor] .
```

Each new thread is identified by a given `Oid` which should be used in `call`, `return` and `resume` messages to select the active thread where they are managed. For example, in a multi-threaded version of our cinema example (see Appendix C) we could execute our system by activating three thread where three clients buy a ticket simultaneously, each one using a different thread of execution. In this case, we could use the same initial state `init-state` and we would include `newThread` messages. In our example we would create three threads, labeled with `t1`, `t2` and `t3`, to buy tickets for three clients[2].

```
mod CINEMA-TEST-MT is
  pr CINEMA-TEST .

  ops t1 t2 t3 : -> Oid .
  op init-state-1-mt : -> Configuration .
  eq init-state-1-mt
    = init-state
      newThread(t1)
      call(goCinema, j, (arg(cinema,"Cnml"), arg(startTime,1100)), t1) .
      newThread(t2)
      call(goCinema, cyd, (arg(cinema,"Cnml"), arg(startTime,1200)), t2) .
      newThread(t3)
      call(goCinema, bob, (arg(cinema,"Cnml"), arg(startTime,1100)), t3) .
endm
```

[2]We simplify here the way to invoke the `goCinema` operation, you can find the real code in Appendix C

As we previously mentioned, multi-threaded execution requires the management of multiple execution stacks. Thus, we provide a multi-threaded version of the `STACK` module (see Figure 6) with some differences from the single-thread version. Now a `NEW-THREAD` equation is used (instead of the `INIT` equation) to create an empty stack when the `newThread` message is thrown. Furthermore, `call`, `return` and `resume` messages are parameterized with the thread identifier and the `CALL` and `RETURN` rules make use of the thread identifier to locate the stack concerned.

When we presented the single-thread version we assumed synchronous messages, however the multi-threaded version allows the use of asynchronous messages too. The management of asynchronous messages does not require any modification of the scheme proposed to represent operations as messages `call` and `return`, but obviously as synchronization is not required, `resume` messages would not be used in rules modeling asynchronous behavior.

5 Validation of OCL Constraints

Once we have a Maude prototype of a given UML model, we can perform different types of analyses on it. For example, we could be interested in statically validating OCL constraints on snapshots representing relevant system states, given by object diagrams, or we could be interested in dynamically validating such constraints during the execution of the prototype. Furthermore, we could use the formal support provided by the Maude environment to perform other kinds of analysis by using, for example, its reachability tool to locate scenarios violating given constraints, as in [13], or its model checker [10].

We first present the mOdCL evaluator, an interpreter of OCL for Maude which allows us to evaluate, not only OCL constraints, but any OCL expression on a given configuration, and we later show how to use it to validate OCL constraints either statically, on given snapshots, or dynamically during the execution of a Maude prototype of a UML model.

5.1 The mOdCL evaluator

Maude has good qualities as a logical and semantical framework in which to represent logics and languages, and were to give semantics to them [27, 10, 31]. The facilities provided by Maude allow us to design mOdCL as a shallow embedding of OCL in Maude. This way, it is not necessary to use complex representations of the elements involved in the OCL expressions and in the UML models they refer to. It is enough to define the mapping from each element in OCL to the Maude element that represents it and to provide a way to evaluate each valid OCL expression on such representation. The mOdCL evaluator provides OCL syntax to Maude and defines an `eval` operator to evaluate OCL 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`.

```
mod STACK-MT is
  pr mOdCL .
  pr LIST{Object} * {op __ to _::_} .

  op Stack : -> Cid [ctor] .
  op state :_ : List{Object} -> Attribute [gather (&)] .

  op Context : -> Cid [ctor] .
  op id :_ : Oid -> Attribute [ctor gather (&)] .
  op opN :_ : OpName -> Attribute [ctor gather (&)] .
  op args :_ : ArgsList -> Attribute [ctor gather (&)] .
  op obj :_ : Oid -> Attribute [ctor gather (&)] .

  op call : OpName Oid ArgsList Oid -> Msg [ctor] .
  op return : OclType Oid -> Msg [ctor] .
  op resume : OpName OclType Oid -> Msg [ctor] .
  op newThread : Oid -> Msg [ctor] .

  op [_] : Configuration -> Configuration .

  vars Op Op' : OpName .
  vars AL AL' : ArgsList .
  vars S S' O T : Oid .
  var R : OclType .
  var ST : List{Object} .

  ops stack ctx : -> Oid .

  eq [NEW-THREAD] :
    newThread(O) = < stack : Stack | id : O, state : nil > .

  eq [CALL] :
    call(Op', S', AL', T)
    < ctx : Context | id : T, opN : Op, obj : S, args : AL >
    < stack : Stack | id : T, state : ST >
    = < ctx : Context | id : T, opN : Op', obj : S', args : AL' >
      < stack : Stack |
        id : T,
        state : < ctx : Context |
                    id : T, opN : Op, obj : S, args : AL > :: ST > .
  eq [RETURN] :
    return(R, T)
    < ctx : Context | id : T, opN : Op, obj : S, args : AL >
    < stack : Stack |
      id : T,
      state : < ctx : Context |
                  id : T, opN : Op', obj : S', args : AL' > :: ST >
    = resume(Op, R, T)
      < ctx : Context | id : T, opN : Op', obj : S', args : AL' >
      < stack : Stack | id : T, state :  ST > .
endm
```

Figure 6: The STACK-MT module.

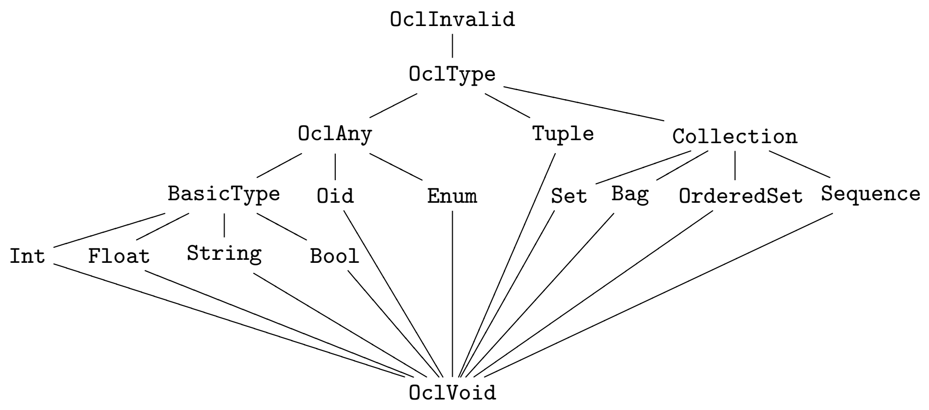

Figure 7: Sort structure of mOdCL.

User-defined types are represented as Maude classes, and their references as elements of the Maude predefined sort `Oid`. This is completed with the *template* types, namely ***Tuple(T1,T2)*** and the collection types ***Collection(T)***, ***Set(T)***, ***Bag(T)***, ***OrderedSet(T)***, and ***Sequence(T)***, which are implemented by respective homonymous types in mOdCL. As shown in Figure 7, `Collection(T)` is an abstract class, supertype of all other collection types. Sort relations in Figure 7[iii] correspond to subsort relations in the mOdCL specification of OCL in Maude.

[iii] Actualizar figura

Key sorts in the definition of the mOdCL evaluator are `OclType`[iv], which represents any predefined OCL type, and `OclExp`, which represents valid OCL expressions. mOdCL assumes system states represented as object configurations (of sort `Configuration`), where the elements in the UML class diagram have been represented as explained in Section 4.1.

[iv] OclAny?

In mOdCL, simple predefined sorts as well as collection sorts are subsorts of `OclType`, and sorts `OclVoid` and `OclInvalid` are introduced to represent, respectively, the ***OclVoid*** and ***OclInvalid*** OCL types (used to deal with undefinedness and errors).

Constants `invalid` and `null` are the only literals in sorts `OclInvalid` and `OclVoid`, respectively.

```
op invalid : -> OclInvalid .
op null : -> OclVoid .
```

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. Because of Maude limitations to define the lexical level, spaces must be left around operators to break tokens.

OCL is a language with a great number of operators and high-level features, and mOdCL supports almost completely all of them, with minor lexical limitations as mentioned above. We do not intend to show here a complete description of the functionality and implementation of mOdCL.[3] Instead, we will provide some intuition on the general principles used in its design. Specifically, we will show how mOdCL supports the syntax and semantics of OCL operators and how it supports the use of query operations in OCL expressions.

The mOdCL evaluator is designed in three main modules: the `SYNTAX` module provides definitions for all valid OCL expressions as terms of sort `OclExp`; the `EVAL` module provides equations to evaluate them by using the `eval` function; and the `COLLECTION` module provides functions to implement basic operations on collections.

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

In general, any OCL operator has a syntactic definition in the `SYNTAX` module and equations to provide the interpretation of the `eval` function for such an operator, possibly using the functionality of operators defined in the `COLLECTION` module. For example, for the `includes` operator on OCL collections we have the following syntax definition in the `SYNTAX` module

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

and the following equation in the `EVAL` module to provide its evaluation

```
vars E1 E1 : OclExp .
var  C : Configuration .
eq eval(E1 -> includes(E2), C) = eval(E1, C) in eval(E2, C) .
```

which is supported by the infix operator `_in_`, provided by the `COLLECTION` module to determine whether a given element is contained in a given collection. Similar equations define the semantics and the behavior of all the different operators available in mOdCL, and thus, the evaluation of any OCL expression is resolved by the recursive invocation of the `eval` operator on the successive subexpressions, and the auxiliary operations defined inside mOdCL.

Given the specification of mOdCL, it is possible to use mOdCL to evaluate any OCL expression. For example, we can calculate the average price of the sessions in the state `init-state` introduced in Section 4.4 as follows:[4]

[3]See [15] for a complete description of the design and implementation of mOdCL, and [14] to get the result of checking the OCL benchmark proposed by Kuhlmann et al. in [25] on the mOdCL evaluator

[4]The Maude `reduce` command simplifies the given term using the equations in the specified module.

```
Maude> reduce in TEST-CINEMA :
         eval(let sum = Session . allInstances() . price -> sum(),
                  num = Session . allInstances() -> size()
              in sum / num,
              init-state) .
result NzNat: 6
```

We end our brief presentation of mOdCL by showing how it supports query operations. That is, those operations defined in the UML class diagram that have no side effects. They can be defined by using OCL expressions and other expressions can invoke them when necessary.

A query operation will be defined in mOdCL by a constant with the name of the operation and one Maude equation whose right part is the OCL expression that defines the operation body. This simple implementation is possible because we have defined the OCL syntax in Maude, thus allowing the definition in the righthand side of the equation, and because we provide a syntax for the definition for this type of equations as

```
op _(_) : OpName List{OclExp} -> OclExp .
```

and syntax to allow the invocation to query operations from OCL expressions as

```
op ._(_) : OpName List{OclExp} -> NavStep .
```

where `NavStep` is a sort defined in mOdCL to represent a valid navigation step in an OCL expression. For example, if we have the following operation to calculate the number of free seats in any session of a given cinema:

```
context Cinema::globalCapacity():Integer
body: self.sessions.capacity->sum()
```

we would define a constant `globalCapacity` of sort `OpName`, as introduced in Section 4.1, and an equation with the body expression in the right side

```
op globalCapacity : -> OpName [ctor] .
eq globalCapacity() = self . sessions . capacity -> sum() .
```

The evaluation of the invocation to a given query operation *`E.Op(args)`* is based on the reduction of the term representing such an invocation, using the corresponding Maude equation defining the body of the operation. The result is a new OCL expression defining the body, which is finally evaluated with the `eval` function. We show below a simplified version of the `eval` equation responsible for processing the invocation to a query operation, where we first use the `eval-EL` function to evaluate the expressions defining the list of actual parameters `L`, later the `Op` equation is reduced as mentioned, and, finally, the new OCL expression defining the body of the query operation is evaluated, using the `eval` function again.

```
  eq eval(E . Op(L), Cf)
    = eval(Op(eval-EL(L, env(VL) Cf, Cf')), Cf) .
```

This way, we can use the `eval` function to evaluate expressions containing invocations to query operations and, for example, we can use the above OCL `globalCapacity` operation to calculate the name of the cinema which offers more seats in its sessions as follows:

```
  Maude> reduce in TEST-CINEMA :
           eval(if cnm1 . globalCapacity() > cnm2 . globalCapacity()
                then cnm1 . name
                else cnm2 . name
                endif,
                state) .
```

5.2 Constraints for dynamic validation

As seen above, the mOdCL evaluator can be used to statically validate given constraints on snapshots representing system states. For its dynamic validation we will use invariants, and pre- and post-conditions, and since they will be referred to from several places in different situations, we assume certain conventions. Given a UML model *M*, we would provide a module, say *M*-`CONSTRAINTS`, which imports the mOdCL evaluator and the system structure, represented as in Section 4.1, and defining the constraints imposed to the model *M*.

```
mod M-CONSTRAINTS is
  pr mOdCL .
  pr M-CLASSES .
```

The mOdCL evaluator declares operators `inv`, `pre` and `post`, to build the OCL expression to be checked. It is assumed that the `inv` operator defines (the conjunction of) the class invariants, and operators `pre` and `post` define, respectively, the pre- and post-condition to be checked for a given operation. Thus, we must define such operators for the constraints assumed in the model *M*. For example, given the above `seats-in-session` expression and the following `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))))
```

we can define the `inv` constant for the `Cinema` model as:

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

The pre and post operators must be defined for each operation which imposes some pre- or post-condition to the model. For example, for the operation 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) .
endm
```

Now, assuming constants state-pre and state-post for configurations representing states which must fulfill given invariants, pre- and post-condition, we can use the eval function as follows:

```
  Maude> red eval(inv and pre(buyTicket), state-pre) .
  result Bool : true .

  Maude> red eval(inv and post(buyTicket), state-post) .
  result Bool : false .
```

5.3 Dynamic validation of OCL constraints with mOdCL

In Section 4, we proposed a scheme to obtain an executable Maude prototype of a UML model. Now, we show how to dynamically validate the OCL constraints in its class diagram, during the execution of such a prototype. Basically, we need to be able to control system execution to locate the system states which must fulfil given constraints and a way to check that they are validated. In the following, we show how to use the stack infrastructure presented in Section 4.3 for the first task and the mOdCL evaluator presented in Section 5.1 for the second one.

An invariant is a Boolean expression that states a condition that must always be met by all instances of the class for which it is defined. To be more precise, an invariant must be true in all consistent states of the system. While the system is, for example, executing an operation, it is not in a consistent state, and the invariant need not be true. Of course, when the execution of the operation has finished, the invariant must again be true [43]. Pre- and post-conditions are

Boolean expressions that must be satisfied, respectively, before and after the execution of the operations they are associated to.

Assuming that the Maude prototype of the UML model follows the scheme proposed in Section 4, we know that CALL and RETURN equations are executed to handle, respectively, call and return messages. This allows us to locate states before and after the execution of operations. We will make use of that to check pre-conditions and invariants before the execution of operations and post-conditions and invariants after their completion. Thus, we will modify the basic stack management presented in Section 4 by including validation checks in CALL and RETURN equations.

The stack management presented in Section 4 only considered the execution of prototypes, not their validation. For this reason, only execution contexts were stored in the stack. Now, we want to perform validation checks and the situation is different, because OCL expressions in post-conditions can use the *@pre* operator to refer to the system state before the execution of the operation. As post-conditions are evaluated *after* the operation completion, and their evaluation can use the system state *before* the operation execution, our validation stack stores information about the system state before calling the operation, *not only its execution context*.

We use now a stack of configurations, representing system states. The module VALIDATION-STACK below defines a class Stack with a single attribute state defined of sort List{Configuration}. Furthermore, the VALIDATION-STACK module redefines the CALL and RETURN equations to perform validation checks when necessary. As we did in Section 4.2, a configuration enclosed by square brackets is used to represent initial states, and furthermore, now we use a Configuration enclosed by curly brackets to represent any other state, thus allowing to gather the complete system state when necessary.

```
mod VALIDATION-STACK is
  pr mOdCL .
  pr LIST{Configuration} * (op __ to _::_).

  class Stack | state : List{Configuration} .

  op [_] : Configuration -> Configuration .
  op {_} : Configuration -> Configuration .

  eq [INIT] : [ Cf ] = { < stack : Stack | state : nil > Cf } .
```

The CALL equation is always executed upon the processing of call messages, which are sent to perform operation invocations. Thus, we know that the actual configuration Cf corresponds to the system state before the execution of the operation Op'. Class invariants and pre-conditions of operation Op' must be satisfied at such a system state Cf. We check that it in fact happens by using the eval function on the invariants defined for the inv constant and on the pre-condition defined for the operator pre(Op').

Invariants expressions do not refer to data concerning the operation invocation. Therefore, when we use eval to check invariants we do not provide any information about the invoked operation. However, pre-condition expressions may refer to the *self* object or to actual parameters of the operation, and the `eval` function needs them in the configuration representing the system state. We use a `state-pre` function to include the `self` object in an `env` message and the parameters in an `OpEnv` message[5] as it is required by the mOdCL evaluator.

If some constraint is violated, we stop the system execution and we provide an `error` message. Otherwise, we create a new `Context` object for the invoked operation (`Op'`) and we store in the execution stack the `Context` object corresponding to the invoking operation (`Op`) and the information corresponding to the actual state which could be required to evaluate post-condition expressions on the operation completion. We use a `filter-pre` function which inspects the actual state and the post-condition expression for the operation, returning a configuration with all the objects corresponding to the actual state which could be required to evaluate occurrences of the `@pre` operator [6]. Obviously, if the `@pre` operator is not used in post-conditions, an empty configuration would be returned.

```
op state-pre : Oid List{Arg} Configuration -> Configuration .
eq state-pre(S, AL, Cf) = env(self <- S) OpEnv(AL) Cf .

msg error : String Configuration -> Msg .

eq [CALL] :
  { call(Op', S', AL')
    < ctx : Context | opN : Op, obj : S, args : AL >
    < stack : Stack | state : ST >
    Cf }
  = if (eval(inv, Cf) and eval(pre(Op'), state-pre(S', AL', Cf))
    then {< stack : Stack |
            state : < ctx : Context | opN : Op,obj : S,args : AL >
            filter-pre(post(Op'), state-pre(S', AL', Cf)) :: ST >
          < ctx : Context | opN : Op', obj : S', args : AL' >
          Cf}
    else {error("Invariant or pre-condition violation", Cf)}
    fi .
```

Similarly, the `RETURN` equation is always executed upon the reception of a `return` message. That is, after executing the last rule in a operation `Op`. We make use of the `eval` function to check invariants (as we did before) and the post-conditions of the `Op` operation. As we also mentioned before, the evaluation of post-conditions requires two system states, those before and after the operation execution. We use `state-pre` to provide the system state before the execution, by retrieving it from the top of the stack the part of such a state required to evaluate the `@pre` operator, and we use `state-post` to provide the actual state, now including the value resulting of the operation execution (contained in the

[5] `OpEnv` and `env` are predefined messages in the mOdCL evaluator.

[6] See Appendix E to get the complete specification of the `filter-pre` function

`return` message) as a new argument in the parameter list of the operation. mOdCL represents the result of an operation as a parameter of name `result`.

```
  op state-post :
               Oid OclType List{Arg} Configuration -> Configuration .
  eq state-post(S, R, AL, Cf) = env(self <- S)
                                OpEnv(arg(result, R), AL) Cf .
  eq [RETURN] :
    { return(R)
      < ctx : Context | opN : Op, obj : S, args : AL >
      < stack : Stack |
        state : < ctx : Context |
                        opN : Op', obj : S', args : AL' > Cf' :: ST >
      Cf }
    = if (eval(inv, Cf) and
          eval(post(Op),
               state-post(S, R, AL, Cf), state-pre(S, AL, Cf')))
      then { resume(Op, R)
             < ctx : Context | opN : Op', obj : S', args : AL' >
             < stack : Stack | state :  ST >
             Cf }
      else { error("Invariant or post-condition violation", Cf) }
      fi .
endm
```

Now, using the new validation stack and mOdCL we can execute our prototype as we did in Section 4.4 [7]. However, now the constraints imposed to the model would be checked when necessary and the system would be stopped in case of any violation.

We could, for example, validate the constraints imposed to the cinema model by providing a `CINEMA-CONSTRAINTS` module and using the `CINEMA-TEST` module in Section 4.

```
  mod CINEMA-VALIDATION-TEST is
    pr CINEMA-TEST .
    pr CINEMA-CONSTRAINTS .
  endm
```

We may now execute the system with the `rewrite` command, obtaining an error message if some constraint is violated, or the final configuration, otherwise.

```
  Maude> rewrite in CINEMA-VALIDATION-TEST : [ init-state ] .
  result Configuration:
       { error("Precondition violation",
            --- Here the complete state violating the pre-condition,
            --- removed to improve readability) }
```

[7]Appendix D includes the complete code for validating some examples, either of single-threaded execution or multi-threaded execution

Note that the user specifies his prototype as he did for simulating his model. The only task required to dynamically validating the model is to write the OCL constrains in a module, by defining `inv`, `pre` and `post` operators. The infrastructure for the validation stack and mOdCL are responsible for the validation, hiding the user all the details.

6 Related Work

The use of OCL is supported by a number of tools, which use a diverse range of technological solutions. In the following we comment on those proposals most related to our contributions in this paper: the mOdCL OCL evaluator and the Maude-based tool for dynamically validating UML prototypes.

The mOdCL evaluator has been designed to provide OCL support inside Maude, thus contributing to the development of tools with which to handle OCL specifications from the Maude formal environment, at the time we benefit from the tools in such environment to analyze and reason about OCL constraints. In a certain extent this approach is similar to the one used in the Eclipse OCL (MDT/OCL) [33] and the Dresden OCL projects [41], which intend to provide OCL support to the Eclipse platform by providing different APIs with which to parse and evaluate OCL expressions.[8] One can use these APIs from Java programs to have OCL support for applications and environments developed in Eclipse, as you can use mOdCL from Maude specifications to have OCL support in Maude-based systems. One can, of course, combine them as convenient, as was done, e.g., in the implementation of e-Motions [2], which uses Eclipse as front end and, as has Maude as back-end tool on which models are executed, it uses mOdCL as evaluation tool for OCL expressions inside Maude.

Other users are interested in adding OCL constraints to their UML (or MOF) models, with support for the evaluation of OCL expressions, and not in building tools with OCL support. In this case they do not need an API, but a tool with a suitable GUI. Tools like Dresden OCL and MDT/OCL offer several plug-ins to handle models and queries on them, which allows them to be used from Eclipse, or case tools like TopCased or MagicDraw. A similar approach is used in the USE tool [20], although in this case its own graphical environment is provided, and which furthermore includes its own textual language to define UML class diagrams and object diagrams, and queries on them. As the mOdCL evaluator has been designed as a back-end tool to be used from other tools it does not provide a GUI itself.

The level of compliance with OCL by the mentioned tools is similar. They are however not easily comparable. Kuhlmann et al. present in [25] a benchmark to analize OCL compliance level by providing a set of tests, which may be a good starting point for it. Although the authours claim in [25] that they have used the benchmark to test USE, MDT/OCL and DresdenOCL, the concrete results are not available, and only global results on accomplishment rates are provided. We have used the benchmark to test mOdCL with excellent results.

[8]The Dresden OCL tool can also be used as a stand-alone library for Java.

The details of this evaluation can be found in [14], where one can see that the level of compliance of mOdCL is similar the ones for these other tools. The only significative difference is that MDT/OCL does not support the evaluation of *@pre* expressions, which implies that post-conditions and body expressions cannot be evaluated. Moreover, mOdCL complies with the latest version of the OCL Standard currently available [34], whilst some of the other tools present several divergences, for example, related to the handling of null values and undefinedness.

There exist two other OCL evaluators for Maude, namely, ITP/OCL [12] and MOMENT-OCL [3]. Both tools only provide support for part of OCL, lacking, for example, support for user-defined operations, definition of pre- and post-conditions, and providing a limited support for collections. Whereas mOdCL provides OCL syntax to Maude, and is able to directly evaluate expressions by means of its `eval` function, both ITP/OCL and MOMENT-OCL change their representation, thus restricting the use on them. Specifically, ITP/OCL translates UML object diagrams and OCL expressions on them to corresponding representations in equational logic. MOMENT-OCL, as mOdCL, represents OCL expressions as terms and UML object diagrams as configurations of objects. However, MOMENT-OCL translates the OCL expressions before their evaluation. The need of the translations introduces additional difficulties when considering the use of these alternative implementations as external components, and even when considering them for dynamic validation, where both the configurations of objects and the OCL expressions are continuously changing.

We have used the mOdCL evaluator to develop a tool with which to dynamically validate OCL constraints during the execution of Maude prototypes of UML models. The idea of using Maude specifications as rapid prototypes of systems is not new [10]. In the context of UML, different authors have proposed the use of Maude to formalize UML models [18] or some of its diagrams [23, 26, 32], and as support to rapid prototyping. E.g., Wirsing and Knapp propose in [47] a formal approach to design object-oriented systems based on the use of rewriting logic and UML diagrams. In their proposal, diagrams are semi-automatically translated into formal specifications, i.e., a diagram is automatically translated into an incomplete formal specification which then has to be completed by hand. The resulting specification would be used as a prototype of the system.

We do not know of any tool that provide the possibility of dynamically validating OCL constraints on specifications of UML models as we propose. All tools that currently support the run-time checking of OCL constraints work on final implementations of models. Dynamic validation of prototypes could be situated somewhere between the validation of concrete states (snapshots), which can be accomplished with OCL evaluators as USE, Dresden OCL, MDT or mOdCL, and the run-time checking of real implementations. Dynamic validation shares some characteristics with run-time checking, as it is necessary to decide how to capture those states to be validated, and also how to deal with the *@pre* operator. We revise below how current approaches have dealt with these issues in the scope of run-time checking tools.

Run-time checking tools generate the code to check invariants and pre- and post-conditions, which is then somehow injected in the implementation code. Different approaches have been used for that (see [19] for an overview on the different proposals), from the generation of skeletons for Java classes, as in Octopus [22] or OCLE [36], to the use of wrappers, as in the first version of the Dresden OCL Toolkit [44]. The general tendency of these tools is to isolate the code generated for models from the code generated to check the constraints. For this reason, recent tools use Aspect-Oriented Programming (AOP) [21], as in the last version of Dresden OCL [45], in the ocl2j [16] tool, or in the proposal by Cheon et al. [8]. Our implementation shares this goal, and we completely separate the Maude specification of the model from the monitoring of the constraints by integrating all required tasks in the management of the stack, which remains hidden to the user.

Due to the cost of its complete support, run-time checking tools have limitations on the use of the *@pre* operator. At present, there are three approaches: to restrict the use of the *@pre* operator to situations where concrete previous values can be easily stored in auxiliary variables, as it occurs in ocl2j [16]; to rely on the user, as in Dresden OCL [46], which assumes that the user will provide a method `createCopy` with which to save the part of the state required for the evaluation of the *@pre* operator in post-condition expressions; and to enrich the generated code to capture the part of the state to be saved, as it is proposed in [24]. In our proposal, we opted for analyzing the models to determine the part of the state which could be used in *@pre* expressions, saving the required objects in the execution stack when the *@pre* is used in post-condition expressions. Although our proposal is less efficient, both in time and space, than these other three alternatives, since we deal with model prototypes, the requirements are less pressing, both space- and time-wise.

In addition to evaluation and run-time checking tools, there is another group of tools whose aim is to verify OCL properties on models. These tools use theorem proving techniques to prove whether given OCL constraints are violated in the model, or to locate constraints which are inconsistent or contradictory. Tools like KeyTool [1], which uses dynamic logic, the HOL-OCL tool [5], based on high-order logic, or UML2CSP [6], which is based on constraint programming, translate OCL constraints to their corresponding formalisms and reason on them on the available theorem provers or alternative formal tools. Although the tools presented in this paper have not been used to verify OCL properties, the support provided by our mOdCL evaluator opens the possibility of using the Maude formal environment to verify OCL constraints in the future. In this line, the mOdCL evaluator has already been used in combination with Maude's reachability-analysis tool to locate scenarios which fulfill or violate given OCL constraints [13]. Maude's invariant analyzer [39], model checker [17] and theorem prover [10] suggest a number of interesting applications.

References

[1] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY Tool. *Software and System Modeling*, 4:32–54, 2005.

[2] Atenea: System Modeling Group. The e-Motions Tool. Available at `http://maude.sip.ucm.es/e-Motions`.

[3] A. Boronat. MOMENT-OCL: A Tool for OCL Validation and OCL Query Evaluation, 2007. Available at `http://moment.dsic.upv.es/content/view/32/75/`.

[4] A. Boronat and J. Meseguer. An Algebraic Semantics for MOF. In J. L. Fiadeiro and P. Inverardi, editors, *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 200*, volume 4961 of *LNCS*, pages 377–391. Springer, 2008.

[5] A. D. Brucker and B. Wolff. HOL-OCL: A Formal Proof Environment for UML/OCL. In J. L. Fiadeiro and P. Inverardi, editors, *Fundamental Approaches to Software Engineering (FASE08)*, volume 4961 of *LNCS*, pages 97–100. Springer, 2008.

[6] J. Cabot, R. Clarisó, and D. Riera. UMLtoCSP: A Tool for the Formal Verification of UML/OCL Models Using Constraint Programming. In R. E. K. Stirewalt, A. Egyed, and B. F. 0002, editors, *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, ASE 2007*, pages 547–548. ACM, 2007.

[7] A. Chaoui, O. Tibermacine, and A. R. Zerek. Formal verification of a subset of UML diagrams: An approach using Maude. In *Handbook of Research on E-Services in the Public Sector: E-Government Strategies and Advancements*. IGI Global, 2011.

[8] Y. Cheon, C. Avila, S. Roach, and C. Munoz. Checking Design Constraints at Run-Time Using OCL and AspectJ. *International Journal of Software Engineering*, 2(3):5–28, 2009.

[9] 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(2):187–243, 2002.

[10] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007.

[11] M. Clavel, F. Durán, J. Hendrix, S. Lucas, J. Meseguer, and P. C. Ölveczky. The Maude Formal Tool Environment. In T. Mossakowski, U. Montanari,

and M. Haveraaen, editors, *Algebra and Coalgebra in Computer Science, Second International Conference, CALCO 2007, Proceedings*, volume 4624 of *LNCS*, pages 173–178. Springer, 2007.

[12] M. Clavel and M. Egea. ITP/OCL: A rewriting-based validation tool for UML+OCL static class diagrams. In M. Johnson and V. Vene, editors, *Algebraic Methodology and Software Technology, 11th International Conference, AMAST 2006, Proceedings*, volume 4019 of *Lecture Notes in Computer Science*, pages 368–373. Springer, 2006.

[13] F. Durán, M. Gogolla, and M. Roldán. Tracing Properties of UML and OCL Models with Maude. In F. Durán and V. Rusu, editors, *Proceedings of International Workshop on Algebraic Methods in Model-Based Software Engineering (AMMSE 2011)*, Electronic Proceedings in Theoretical Computer Science, 2011.

[14] F. Durán and M. Roldán. El que sea. Technical Report EL QUE TENGA, University of Málaga, 2012. Available at `LAQUESEA`.

[15] F. Durán and M. Roldán. Evaluation of OCL expressions in Maude. The Evaluator mOdCL. Technical report, University of Málaga, 2012. Available at .

[16] W. J. Dzidek, L. C. Briand, and Y. Labiche. Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java. In J.-M. Bruel, editor, *Satellite Events at the MoDELS 2005 Conference, MoDELS 2005*, volume 3844 of *Lecture Notes in Computer Science*, pages 10–19. Springer, 2006.

[17] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL Model Checker and its Implementation. In T. Ball and S. K. Rajamani, editors, *Model Checking Software, 10th International SPIN Workshop, 2003, Proceedings*, volume 2648 of *Lecture Notes in Computer Science*, pages 230–234. Springer, 2003.

[18] J. L. Fernández-Alemán and A. Toval-Álvarez. Seamless Formalizing the UML Semantics through Metamodels. In K. Siau and T. Halpin, editors, *Unified Modeling Language: Systems Analysis, Design and Development Issues*, pages 224–248. Idea Group Publishing, 2001.

[19] L. Froihofer, G. Glos, J. Osrael, and K. M. Goeschka. Overview and Evaluation of Constraint Validation Approaches in Java. *International Conference on Software Engineering*, pages 313–322, 2007.

[20] M. Gogolla, F. Büttner, and M. Richters. USE: A UML-based Specification Environment for Validating UML and OCL. *Science of Computer Programming*, 69(1-3):27–34, 2007.

[21] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353. Springer, 2001.

[22] Klasse Objecten. The Klasse Objecten OCL checker Octopus. Available at `http://sourceforge.net/projects/octopus/`.

[23] A. Knapp. Generating Rewrite Theories from UML Collaborations. In K. Futatsugi, A. Nakagawa, and T. Tamai, editors, *CAFE: An Industrial-Strength Algebraic Formal Method*, pages 97–120. Elsevier, 2000.

[24] P. Kosiuczenko. On the Implementation of @pre. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FASE '09, pages 246–261, Berlin, Heidelberg, 2009. Springer-Verlag.

[25] M. Kuhlmann, L. Hamann, M. Gogolla, and F. Büttner. A Benchmark for OCL Engine Accuracy, Determinateness, and Efficiency. *Software and System Modeling*, 11(2):165–182, 2012.

[26] M. S. Lund and K. Stølen. Deriving tests from UML 2.0 sequence diagrams with neg and assert. In *Proceedings of the 2006 international workshop on Automation of software test*, AST 2006, pages 22–28. ACM, 2006.

[27] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume 9, pages 1–87. Kluwer Academic Publishers, second edition, 2002. First published as SRI Technical Report SRI-CSL-93-05, 1993.

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

[29] J. Meseguer. Membership Algebra as a Logical Framework for Equational Specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques*, volume 1376 of *LNCS*, pages 18–61. Springer, 1998.

[30] J. Meseguer. Rewriting logic and Maude: A wide-spectrum semantic framework for object-based distributed systems. In S. Smith and C. Talcott, editors, *Formal Methods for Open Object-Based Distributed Systems IV, IFIF TC6/WG6.1 Fourth International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2000)*, volume 177 of *IFIP Conference Proceedings*, pages 89–117. Kluwer Academic Press, 2000.

[31] J. Meseguer. 20 years of rewriting logic. Technical report, University of Illinois at Urbana-Champaign, 2012. Available at `http://hdl.handle.net/2142/32096`.

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

[33] MDT/OCL. `http://www.eclipse.org/modeling/mdt/?project=ocl`.

[34] Object Management Group. Object Constraint Language (OCL) (Version 2.3.1 Specification (formal/2012-01-01), 2012.

[35] Object Management Group. Unified Modeling Language v2.4.1. Specification (formal/2012-05-06), 2012.

[36] Object Constraint Language Environment. Available at http://lci.cs.ubbcluj.ro/ocle/.

[37] J. E. Rivera, F. Durán, and A. Vallecillo. Formal Specification and Analysis of Domain Specific Models Using Maude. *Simulation*, 85(11-12):778–792, 2009.

[38] J. E. Rivera, F. Durán, and 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 2009, pages 51–55. IEEE Computer Society, 2009.

[39] C. Rocha and J. Meseguer. Proving Safety Properties of Rewrite Theories. In A. Corradini, B. Klin, and C. Cîrstea, editors, *Algebra and Coalgebra in Computer Science - 4th International Conference, CALCO 2011, Proceedings*, volume 6859 of *Lecture Notes in Computer Science*, pages 314–328. Springer, 2011.

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

[41] Software Technology Group. DresdenOCL. Available at `\url{http://www.dresden-ocl.org/}`.

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

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

[44] R. Wiebicke. Utility Support for Checking OCL Business Rules in Java Programs. Master's thesis, Dresden University of Technology, 2000.

[45] C. Wilke. Java Code Generation for Dresden OCL2 for Eclipse. Available at `http://www.claaswilke.de/publications/study/beleg.pdf`.

[46] C. Wilke. Java Code Generation for Dresden OCL2 for Eclipse (Minor Thesis). Technical report, Technische Universitat Dresden, February 2009.

[47] M. Wirsing and A. Knapp. A formal approach to object-oriented software engineering. *Theoretical Computer Science*, 285:519–560, 2001.

A The cinema model. System structure

This appendix shows the definition of the CLASSES-CINEMA module, which contains the declaration of the elements in the class diagram for the cinema example. These declarations will be used from the Maude specification of the system behavior and from mOdCL, when it is necessary to evaluate some constraint on a given system state (object configuration).

```
mod CLASSES-CINEMA is
  inc CLASSES-SGNT .

  --- CLASES
  --- class Cinema | name : String, bank : Oid, sessions : Set{Oid} .
  sort Cinema .
  subsort Cinema < Cid .
  op Cinema : -> Cinema [ctor] .
  op name : -> AttributeName [ctor] .
  op bank : -> AttributeName [ctor] .
  op sessions : -> AttributeName [ctor] .

  --- class Session | startTime : Int, endTime : Int, capacity : Int,
  ---                 ticket : Set{Oid}, price : Int .
  sort Session .
  subsort Session < Cid .
  op Session : -> Session [ctor] .
  op startTime : -> AttributeName [ctor] .
  op endTime : -> AttributeName [ctor] .
  op capacity : -> AttributeName [ctor] .
  op price : -> AttributeName [ctor] .
  op ticket : -> AttributeName [ctor] .

  --- class Client | cinemas : Set{Oid}, debitCard : Nat,
  ---                ticket : Set{Oid} .
  sort Client .
  subsort Client < Cid .
  op Client : -> Client [ctor] .
  op cinemas   : -> AttributeName [ctor] .
  op debitCard : -> AttributeName [ctor] .

  --- class Ticket | seat : int, cinema : Oid, client : Oid .
  --- Association class
  sort Ticket .
  subsort Ticket < Cid .
  op Ticket : -> Ticket [ctor] .
  op seat : -> AttributeName [ctor] .
  op session : -> AttributeName [ctor] .
  op client : -> AttributeName [ctor] .

  --- class Bank | cards : Q-Assoc .
  --- Qualified association
  sort Bank .
  subsort Bank < Cid .
  op Bank : -> Bank [ctor] .
  op cards : -> AttributeName [ctor] .

  --- class Account | bal : Float .
  sort Account .
  subsort Account < Cid .
  op Account : -> Account [ctor] .
  op bal : -> AttributeName [ctor] .

  --- Operations
  op goCinema : -> OpName .
  op cinema : -> Arg [ctor] .
  op startTime : -> Arg [ctor] .

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

  op pay : -> OpName .
  op debitCard : -> Arg [ctor] .
  op amount : -> Arg [ctor] .
endm
```

B The cinema model. Single-Threaded execution

This appendix describes the complete executable specification of the system behavior for the cinema model. It uses a single thread of execution and synchronous messages, and it follows the scheme introduced in Section 4.2 for specifying the operations.

We define a `CINEMA` module which imports from the `CLASSES-CINEMA` module (see Appendix A) the definition of the class diagram and from the `STACK` module the stack management infrastructure. We use natural numbers as identifiers of objects of class `Ticket` and constants explicitly defined of sort `Oid` as identifiers of the rest of objects.

```
mod CINEMA is
  pr CLASSES-CINEMA .
  pr STACK .

  subsort Nat < Oid .

  var Self C S AC  : Oid .
  vars AS-1 AS-2 AS-3 : AttributeSet .
  vars LC LC' LS LS' LT : List .
  var RESULT : OclType .
  var A : ArgsList .
```

During the execution users buy tickets for given sessions on cinemas. There is no restriction to specify such behavior, the user can do it as he prefer. We have used here a sequence of `goCinema` messages, which represent a given sequence of buy ticket requests.

```
  op go-cinema : Oid String Nat Nat -> Msg [msg] .
```

A `goCinema` message has as arguments the identifier of the object executing the request, the name of the cinema where he want to buy a ticket, the starting time for the requested session and a number of sequence, used to determine the order in which the requests should be processed. The sequence number will be used to guarantee that all requests (and therefore the invocations to the `goCinema` operation) are executed in the order given by the `goCinema` messages. The rule `SEQUENCE-GO` is responsible for that by using `next-goCinema-call` messages.

```
  op next-goCinema-call : Nat -> Msg [msg] .
```

In this example we want that those rules modeling each operation be executed in a given order. We use messages `seq-goCinema` and `seq-buyTicket` for that. They are tokens which select the rules able to be executed in any time (we use messages for that, but other approaches would valid as well). Finally, we use a message `next-ticket` for generating a new ticket identifier for each sold ticket.

```
op seq-goCinema : Nat -> Msg [msg] .
op seq-buyTicket : Nat -> Msg [msg] .
op next-ticket : Nat -> Msg [msg] .
```

The rules `SEQUENCE-GO` and `SEQUENCE-GO-RT` are responsible, respectively, for invoking the `goCinema` operation to buy a ticket (by means of a message `call(goCinema,...)`) and for locking future invocations until the completion of the actual one. The use of the `next-goCinema-call` guarantees that any invocation is executed in the order stated in `goCinema` messages. The use of the message `resume(goCinema, RESULT)` allows to lock coming invocations until the completion of the actual invocation. Note as the arguments used in the invocation to the `goCinema` operation (the name of the cinema and the time for the session) are included as argument of the `call` message, in a pair list `arg(`*`nm`*`, `*`val`*`)`, where *`nm`* (of sort `Arg`) is the name of the constant defined for such an argument (in the `CLASSES-CINEMA` module) and *`val`* is the value of the actual parameter in the invocation.

```
rl [SEQUENCE-GO] :
  go-cinema(Cl:Oid, Cn:String, St:Nat, I:Nat)
  next-goCinema-call(I:Nat)
  => next-goCinema-call(I:Nat)
     seq-goCinema(1)
     call(goCinema, Cl:Oid, (arg(cinema, Cn:String),
                             arg(startTime, St:Nat))) .

rl [SEQUENCE-GO-RT] :
  resume(goCinema, RESULT)
  next-goCinema-call(I:Nat)
  => next-goCinema-call(s I:Nat) .
```

Finally, we include the specification of the three operation described in the class diagram. The `goCinema` operation is modeled by using three rules. The rule `OP-GO-CINEMA-1` checks if the cinema and session requested are valid and uses a `call` message for invoking the `buyTicket` operation. The others rules use the `resume` message to lock until the completion of the invocation. If the invocation returns a valid ticket the execution continues by executing the rule `OP-GO-CINEMA-2-OK`, using a `return` message to return `true` as the result of the `goCinema` operation; otherwise, the rule `OP-GO-CINEMA-2-FAIL` is executed, which returns `false`. Note how all these rules receive and propagate the `ctx` object with the execution context.

```
rl [OP-GO-CINEMA-1] :
  < ctx : Context | opN : goCinema, obj : Self ,
          args : (arg(cinema, Cn:String), arg(startTime, St:Nat)) >
  seq-goCinema(1)
  < Self : Client | cinemas : Set{LC’, C, LC}, AS-1 >
```

```
    < C : Cinema | name : Cn:String,
                   sessions : Set{LS', S, LS}, AS-2 >
    < S : Session   | startTime : St:Nat, AS-3 >
    => < Self : Client | cinemas : Set{LC', C , LC}, AS-1 >
       < C : Cinema | name : Cn:String,
                      sessions : Set{LS', S ,LS}, AS-2 >
       < S : Session   | startTime : St:Nat, AS-3 >
       seq-goCinema(2)
       seq-buyTicket(1)
       < ctx : Context | opN : goCinema, obj : Self,
               args : (arg(cinema, Cn:String),
                       arg(startTime, St:Nat)) >
       call(buyTicket, C, (arg(startTime, St:Nat),
                           arg(aClient, Self))) .

  crl [OP-GO-CINEMA-2-OK] :
    < ctx : Context | opN : goCinema, obj : Self, args : A >
    resume(buyTicket, RESULT)
    seq-goCinema(2)
    < Self : Client | ticket : Set{LT}, AS1 >
    =>
      < Self : Client | ticket : Set{RESULT, LT}, AS1 >
      < ctx : Context | opN : goCinema, obj : Self, args : A >
      return(true)
  if RESULT =/= null .

  rl [GO-CINEMA-2-FAIL] :
    < ctx : Context | opN : goCinema, obj : Self,
                      args : (arg(cinema, Cn:String),
                              arg(startTime, St:Nat)) >
    resume(buyTicket, null)
    seq-goCinema(2)
    => < ctx : Context | opN : goCinema, obj : Self,
                         args : (arg(cinema, Cn:String),
                                 arg(startTime, St:Nat)) >
       return(false) .
```

The `buyTicket` operation is specified by four rules. The `BUY-TICKET-1-FREE-SEAT` and `BUY-TICKET-1-NO-FREE-SEAT` rules model the first execution step in the operation, where we check if there are available seats in the requested session. If we have, the `pay` operation is invoked to perform the payment; otherwise a `null` ticket is returned. Finally, the rules `BUY-TICKET-2-CHARGE-OK` and `BUY-TICKET-2-CHARGE-FAIL` model the second execution step. If the payment is right a new ticket is created and returned; otherwise, a `null` ticket is returned again.

```
  crl [BUY-TICKET-1-FREE-SEAT] :
    < ctx : Context | opN : buyTicket, obj : Self,
            args : (arg(startTime, St:Nat), arg(aClient, Cl:Oid)) >
    seq-buyTicket(1)
    < Self : Cinema | bank : B:Oid,
                      sessions : Set{LS', S, LS}, AS-1 >
    < S : Session | startTime : St:Nat, ticket : TS:Set,
                    capacity : Cp:Nat, price : P:Nat, AS-2 >
    < Cl:Oid : Client | debitCard : Cn:Nat, AS-3 >
    => < Self : Cinema | bank : B:Oid,
                         sessions : Set{LS', S, LS}, AS-1 >
       < S : Session | startTime : St:Nat, ticket : TS:Set,
                       capacity : Cp:Nat, price : P:Nat, AS-2 >
       < Cl:Oid : Client | debitCard : Cn:Nat, AS-3 >
       < ctx : Context | opN : buyTicket, obj : Self,
                         args : (arg(startTime, St:Nat),
                                 arg(aClient, Cl:Oid)) >
       call(pay, B:Oid, (arg(debitCard, Cn:Nat), arg(amount, P:Nat)))
       seq-buyTicket(2)
  if | TS:Set | < Cp:Nat .

  crl [BUY-TICKET-1-NO-FREE-SEAT] :
    < ctx : Context | opN : buyTicket, obj : Self,
                      args : (arg(startTime, St:Nat),
                              arg(aClient, Cl:Oid)) >
    < Self : Cinema | sessions : Set{LS', S, LS}, AS-1 >
    < S : Session | startTime : St:Nat, ticket : TS:Set,
                    capacity : Cp:Nat, AS-2 >
    seq-buyTicket(1)
    => < Self : Cinema | sessions : Set{LS', S, LS}, AS-1 >
       < S : Session | startTime : St:Nat, ticket : TS:Set,
                       capacity : Cp:Nat, AS-2 >
       < ctx : Context | opN : buyTicket, obj : Self,
                         args : (arg(startTime, St:Nat),
                                 arg(aClient, Cl:Oid)) >
      return(null)
  if | TS:Set | >= Cp:Nat .

  rl [BUY-TICKET-2-CHARGE-OK] :
    resume(pay, true)
    < ctx : Context | opN : buyTicket, obj : Self,
            args : (arg(startTime, St:Nat), arg(aClient, Cl:Oid)) >
    next-ticket(TK:Nat)
    seq-buyTicket(2)
    < Self : Cinema | sessions : Set{LS', S, LS}, AS-1  >
    < S : Session | startTime : St:Nat, ticket : Set{LT}, AS-2 >
    => < TK:Nat : Ticket | seat : 0,  session : S, client : Cl:Oid >
       < Self : Cinema | sessions : Set{LS', S, LS}, AS-1  >
       < S : Session | startTime : St:Nat,
                       ticket : Set{TK:Nat , LT}, AS-2 >
       next-ticket(s TK:Nat)
       < ctx : Context | opN : buyTicket, obj : Self,
                         args : (arg(startTime, St:Nat),
                                 arg(aClient, Cl:Oid)) >
      return(TK:Nat) .

   rl [BUY-TICKET-2-CHARGE-FAIL] :
     resume(pay, false)
     < ctx : Context | opN : buyTicket, obj : Self,
             args : (arg(startTime, St:Nat), arg(aClient, Cl:Oid)) >
     seq-buyTicket(2)
     => < ctx : Context | opN : buyTicket, obj : Self,
                          args : (arg(startTime, St:Nat),
                                  arg(aClient, Cl:Oid)) >
        return(null) .
```

Finally, we use rules PAY-OK and PAY-FAIL to specify the pay operation, which returns true or false depending if the the account linked to the debit card has enough balance to perform the payment. Note that the cards attribute in the Bank class is a qualified association which relates card numbers with identifiers of accounts.

```
  crl [PAY-OK] :
    < ctx : Context | opN : pay, obj : Self,
                      args : (arg(debitCard, Cn:Nat),
                              arg(amount, Amt:Nat)) >
    < Self : Bank | cards : qas(Cn:Nat,AC) $$ AA:Q-Assoc >
    < AC : Account | bal : B:Nat >
    =>
       < Self : Bank | cards : qas(Cn:Nat,AC) $$ AA:Q-Assoc >
       < AC : Account | bal : sd(B:Nat,Amt:Nat) >
       < ctx : Context | opN : pay, obj : Self,
                         args : (arg(debitCard, Cn:Nat),
                                 arg(amount, Amt:Nat)) >
       return(true)
    if B:Nat >=  Amt:Nat .

  crl [PAY-FAIL] :
    < ctx : Context | opN : pay, obj : Self,
                      args : (arg(debitCard, Cn:Nat),
                              arg(amount, Amt:Nat)) >
    < Self : Bank | cards : qas(Cn:Nat,AC) $$ AA:Q-Assoc >
    < AC : Account | bal : B:Nat >
    =>
       < Self : Bank | cards : qas(Cn:Nat,AC) $$ AA:Q-Assoc >
       < AC : Account | bal : B:Nat >
       < ctx : Context | opN : pay, obj : Self,
                         args : (arg(debitCard, Cn:Nat),
                                 arg(amount, Amt:Nat)) >
       return(false)
   if B:Nat < Amt:Nat .
```

Now we can define some initial configurations and simulate system execution from these initial states. For example, we can define a module TEST-CINEMA which imports the CINEMA module and defines the init-state configuration with data corresponding to cinemas, banks and clients. We use configurations init-state1 and init-state2 which extend init-state with some go-cinema requests to buy some tickets to given sessions.

```
mod TEST-CINEMA is
   pr CINEMA .

   op cnm1  : -> Oid .
   op bbva  : -> Oid .
   ops s1 s2 s3 : -> Oid .
   ops bob ada tom : -> Oid .
   ops acc1 acc2 acc3 : -> Oid .

   op init-state : -> Configuration .
   eq init-state =
      next-ticket(1)
      < cnm1 : Cinema | name : "Coronet",
                        bank : bbva, sessions : Set{s1 , s2 , s3} >

      < s1 : Session  | startTime : 1100, endTime : 1150,
                        capacity : 10, price : 5, ticket : Set{} >
      < s2 : Session  | startTime : 1200, endTime : 1250,
                        capacity : 10, price : 8, ticket : Set{} >
      < s3 : Session  | startTime : 1300, endTime : 1350,
                        capacity : 10, price : 5, ticket : Set{} >

      < bob : Client | cinemas : Set{cnm1},
                       ticket : Set{}, debitCard : 111 >
      < ada  : Client | cinemas : Set{cnm1},
                        ticket : Set{}, debitCard : 222 >
      < tom : Client | cinemas : Set{cnm1},
                       ticket : Set{}, debitCard : 333 >

      < bbva : Bank | cards : qas(111,acc1)
                              $$ qas(222,acc2) $$ qas(333,acc3) >

      < acc1 : Account | bal :   100 >
      < acc2 : Account | bal :  1000 >
      < acc3 : Account | bal : 10000 > .

  op init-state1 : -> Configuration .
  eq init-state1 = init-state
                   next-goCinema-call(1)
                   go-cinema(bob, "Coronet", 1100, 1)
                   go-cinema(ada,  "Coronet", 1100, 2)
                   go-cinema(bob, "Coronet", 1200, 3)
                   go-cinema(ada,  "Coronet", 1200, 4)
                   go-cinema(tom, "Coronet", 1300, 5) .
  op init-state2 : -> Configuration .
  eq init-state2 = init-state
                   next-goCinema-call(1)
                   go-cinema(ada, "Coronet", 1200, 1)
                   go-cinema(ada, "Coronet", 1200, 2) .
endm
```

We execute the system to simulate these situations and we get the following results:

```
Maude> rewrite in TEST-CINEMA : [init-state1] .
rewrites: 125 in 0ms cpu (23ms real) (~ rewrites/second)
result Configuration: < stack : Stack | state : nil >
    < cnm1 : Cinema | name : "Coronet",
                      bank : bbva,sessions : Set{s1, s2, s3} >
    < bbva : Bank | cards : (qas(111, acc1)
                             $$ qas(222, acc2) $$ qas(333, acc3)) >
    < s1 : Session | startTime : 1100,endTime : 1150,capacity : 10,
                     price : 5,ticket : Set{2, 1} >
    < s2 : Session | startTime : 1200,endTime : 1250,capacity : 10,
                     price : 8,ticket : Set{4, 3} >
    < s3 : Session | startTime : 1300,endTime : 1350,capacity : 10,
                     price : 5,ticket : Set{5} >
    < bob : Client | ticket : Set{3, 1},
                     cinemas : Set{cnm1},debitCard : 111 >
    < ada : Client | ticket : Set{4, 2},
                     cinemas : Set{cnm1},debitCard : 222 >
    < tom : Client | ticket : Set{5},
                     cinemas : Set{cnm1},debitCard : 333 >
    < acc1 : Account | bal : 87 >
    < acc2 : Account | bal : 987 >
    < acc3 : Account | bal : 9995 >
    < 1 : Ticket | seat : 0,session : s1,client : bob >
    < 2 : Ticket | seat : 0,session : s1,client : ada >
    < 3 : Ticket | seat : 0,session : s2,client : bob >
    < 4 : Ticket | seat : 0,session : s2,client : ada >
    < 5 : Ticket | seat : 0,session : s3,client : tom >

Maude> rewrite in CINEMA-VALIDATION-TEST : [init-state2] .
rewrites: 53 in 0ms cpu (1ms real) (~ rewrites/second)
result Configuration: < stack : Stack | state : nil >
    < cnm1 : Cinema | name : "Coronet",
                      bank : bbva,sessions : Set{s1, s2, s3} >
    < bbva : Bank | cards : (qas(111, acc1)
                             $$ qas(222, acc2) $$ qas(333, acc3)) >
    < s1 : Session | startTime : 1100,endTime : 1150,capacity : 10,
                     price : 5,ticket : Set{mt-ord} >
    < s2 : Session | startTime : 1200,endTime : 1250,capacity : 10,
                     price : 8,ticket : Set{2, 1} >
    < s3 : Session | startTime : 1300,endTime : 1350,capacity : 10,
                     price : 5,ticket : Set{mt-ord} >
    < bob : Client | ticket : Set{mt-ord},
                     cinemas : Set{cnm1},debitCard : 111 >
    < ada : Client | ticket : Set{2, 1},
                     cinemas : Set{cnm1},debitCard : 222 >
    < tom : Client | ticket : Set{mt-ord},
                     cinemas : Set{cnm1},debitCard : 333 >
    < acc1 : Account | bal : 100 >
    < acc2 : Account | bal : 984 >
    < acc3 : Account | bal : 10000 >
    < 1 : Ticket | seat : 0,session : s2,client : ada >
    < 2 : Ticket | seat : 0,session : s2,client : ada >
```

C The cinema model. Multi-Threaded execution

This appendix contains the complete executable specification of the system behavior for the cinema example modeled for multi-threaded execution. In this case the Maude specification is similar to the one presented in the Appendix B, however it includes two differences:

- Messages `call`, `return` and `resume` include the thread identifier.
- The specification of the `buyTicket` operation is modified to avoid that the number of sold ticked in a session exceeds its capacity. As the cinema version presented in Appendix B only considers one execution thread, it is not necessary to lock the tickets requested in a session until their payment is completed. However, now we have several execution threads, and several requests of tickets are possible for a given session. For this reason we include a new attribute `reserved` in the class `Session`, which helps us to control that the number of sold and pending of payment tickets does not exceed the capacity of a given session. The mentioned modification only requires to change the `BUY-TICKET-1-FREE-SEAT` and `BUY-TICKET-1-NO-FREE-SEAT` rules.

```
mod CINEMA-MT is
   pr CLASSES-CINEMA-MT .
   pr STACK-MT .

   subsort Nat < Oid .

   vars Self C S AC  : Oid .
   var T : Oid .
   vars AS-1 AS-2 AS-3 : AttributeSet .
   vars LC LC' LS LS' LT : List .
   var RESULT : OclType .
   var A : ArgsList .

   op go-cinema : Oid Oid String Nat Nat -> Msg [msg] .
   op next-goCinema-call : Oid Nat -> Msg [msg] .
   op seq-goCinema :  Oid Nat -> Msg [msg] .
   op seq-buyTicket :  Oid Nat -> Msg [msg] .
   op seq-pay :  Oid Nat -> Msg [msg] .
   op next-ticket : Nat -> Msg [msg] .

   rl [SEQUENCE-GO] :
     go-cinema(T, Cl:Oid, Cn:String, St:Nat, I:Nat)
     next-goCinema-call(T, I:Nat)
     => next-goCinema-call(T, I:Nat)
        seq-goCinema(T, 1)
        call(goCinema, Cl:Oid, (arg(cinema, Cn:String),
                                arg(startTime, St:Nat)),Cl:Oid) .
```

```
rl [SEQUENCE-GO-RT] :
    resume(goCinema, RESULT, T)
    next-goCinema-call(T, I:Nat)
    => next-goCinema-call(T, s I:Nat) .

--- -----------------------------------------------------------
--- CLASS: Client
--- -----------------------------------------------------------
rl [OP-GO-CINEMA-1] :
  < ctx : Context | id : T, opN : goCinema, obj : Self ,
          args : (arg(cinema, Cn:String),arg(startTime,St:Nat)) >
  < Self : Client | cinemas : Set{LC', C, LC}, AS-1 >
  < C : Cinema | name : Cn:String,
                 sessions : Set{LS', S, LS}, AS-2 >
  < S : Session | startTime : St:Nat, AS-3 >
  seq-goCinema(T, 1)
    =>
      < Self : Client | cinemas : Set{LC', C, LC}, AS-1 >
      < C : Cinema | name : Cn:String,
                     sessions : Set{LS', S, LS}, AS-2 >
      < S : Session   | startTime : St:Nat, AS-3 >
      seq-goCinema(T, 2)
      seq-buyTicket(T, 1)
      < ctx : Context | id : T, opN : goCinema, obj : Self,
              args : (arg(cinema, Cn:String),
                      arg(startTime, St:Nat)) >
      call(buyTicket, C, (arg(startTime, St:Nat),
                          arg(aClient, Self)), T) .

crl [OP-GO-CINEMA-2-OK] :
  < ctx : Context | id : T, opN : goCinema, obj : Self, args : A >
  resume(buyTicket, RESULT, T)
  seq-goCinema(T, 2)
  < Self : Client | ticket : Set{LT, AS:AttributeSet >
  =>
    < Self : Client | ticket : Set{RESULT, LT}, AS:AttributeSet >
    < ctx : Context | id : T, opN : goCinema, obj : Self, args : A >
    return(true, T)
if RESULT =/= null .

rl [GO-CINEMA-2-FAIL] :
  < ctx : Context | id : T, opN : goCinema, obj : Self,
                    args : (arg(cinema, Cn:String),
                            arg(startTime, St:Nat)) >
  resume(buyTicket, null, T)
  seq-goCinema(T, 2)
  => < ctx : Context | id : T, opN : goCinema, obj : Self,
                       args : (arg(cinema, Cn:String),
                               arg(startTime, St:Nat)) >
```

```
     return(false, T) .

--- ------------------------------------------------------------
--- CLASS: Cinema
--- ------------------------------------------------------------
crl [BUY-TICKET-1-FREE-SEAT] :
  < ctx : Context | id : T, opN : buyTicket, obj : Self,
                    args : (arg(startTime, St:Nat),
                            arg(aClient, Cl:Oid)) >
  < Self : Cinema | bank : B:Oid,
                    sessions : Set{LS', S, LS}, AS-1 >
  < S : Session | startTime : St:Nat, ticket : TS:Set,
                  reserved : Rsv:Int, capacity : Cp:Nat,
                  price : P:Nat, AS-2 >
  < Cl:Oid : Client | debitCard : Cn:Nat, AS-3 >
  seq-buyTicket(T, 1)
  => < Self : Cinema | bank : B:Oid,
                       sessions : Set{LS', S, LS}, AS-1 >
     < S : Session | startTime : St:Nat, ticket : TS:Set,
                     reserved : Rsv:Int + 1, capacity : Cp:Nat,
                     price : P:Nat, AS-2 >
     < Cl:Oid : Client | debitCard : Cn:Nat, AS-3 >
     < ctx : Context | id : T, opN : buyTicket, obj : Self,
                       args : (arg(startTime, St:Nat),
                               arg(aClient, Cl:Oid)) >
     seq-pay(T, 1)
     call(pay, B:Oid, (arg(debitCard, Cn:Nat),
                       arg(amount, P:Nat)), T)
     seq-buyTicket(T, 2)
if | TS:Set | + Rsv:Int < Cp:Nat .

crl [BUY-TICKET-1-NO-FREE-SEAT] :
  < ctx : Context | id : T, opN : buyTicket, obj : Self,
                    args : (arg(startTime, St:Nat),
                            arg(aClient, Cl:Oid)) >
  < Self : Cinema | sessions : Set{LS', S, LS}, AS-1 >
  < S : Session | startTime : St:Nat, ticket : TS:Set,
                  reserved : Rsv:Int, capacity : Cp:Nat, AS-2 >
  seq-buyTicket(T, 1)
  => < Self : Cinema | sessions : Set{LS', S, LS}, AS-1 >
     < S : Session | startTime : St:Nat, ticket : TS:Set,
                     reserved : Rsv:Int, capacity : Cp:Nat, AS-2 >
     < ctx : Context | id : T, opN : buyTicket, obj : Self,
                       args : (arg(startTime, St:Nat),
                               arg(aClient, Cl:Oid)) >
    return(null, T)
if | TS:Set | + Rsv:Int >= Cp:Nat .

rl [BUY-TICKET-2-CHARGE-OK] :
  resume(pay, true, T)
  < ctx : Context | id : T, opN : buyTicket, obj : Self,
          args : (arg(startTime, St:Nat), arg(aClient, Cl:Oid)) >
  next-ticket(TK:Nat)
  < Self : Cinema | sessions : Set{LS', S, LS}, AS-1  >
  < S : Session | startTime : St:Nat,
                  ticket : Set{LT}, reserved : Rsv:Int, AS-2 >
  seq-buyTicket(T, 2)
  => < TK:Nat : Ticket | seat : 0,  session : S, client : Cl:Oid >
     < Self : Cinema | sessions : Set{LS', S, LS}, AS-1  >
     < S : Session | startTime : St:Nat, ticket : Set{TK:Nat, LT},
                     reserved : Rsv:Int - 1, AS-2 >
     next-ticket(s TK:Nat)
    < ctx : Context | id : T, opN : buyTicket, obj : Self,
                      args : (arg(startTime, St:Nat),
                              arg(aClient, Cl:Oid)) >
    return(TK:Nat, T) .

rl [BUY-TICKET-2-CHARGE-FAIL] :
  resume(pay, false, T)
  < ctx : Context | id : T, opN : buyTicket, obj : Self,
          args : (arg(startTime, St:Nat), arg(aClient, Cl:Oid)) >
  < S : Session | startTime : St:Nat, reserved : Rsv:Int, AS-1 >
  seq-buyTicket(T, 2)
  => < S : Session | startTime : St:Nat, reserved : Rsv:Int - 1, AS-1 >
     < ctx : Context | id : T, opN : buyTicket, obj : Self,
                       args : (arg(startTime, St:Nat),
                               arg(aClient, Cl:Oid)) >
     return(null, T) .

--- -------------------------------------------------------------
--- CLASS: Bank
 --- -------------------------------------------------------------
crl [PAY] :
  < ctx : Context | id : T, opN : pay, obj : Self,
                    args : (arg(debitCard, Cn:Nat),
                            arg(amount, Amt:Nat)) >
  < Self : Bank | cards : qas(Cn:Nat,AC) $$ AA:Q-Assoc >
  < AC : Account | bal : B:Nat >
  seq-pay(T, 1)
  =>
     < Self : Bank | cards : qas(Cn:Nat,AC) $$ AA:Q-Assoc >
     < AC : Account | bal : sd(B:Nat,Amt:Nat) >
     < ctx : Context | id : T, opN : pay, obj : Self,
                       args : (arg(debitCard, Cn:Nat),
                               arg(amount, Amt:Nat)) >
     return(true, T)
  if B:Nat >= Amt:Nat .

crl [PAY] :
  < ctx : Context | id : T, opN : pay, obj : Self,
                    args : (arg(debitCard, Cn:Nat),
                            arg(amount, Amt:Nat)) >
     < Self : Bank | cards : qas(Cn:Nat,AC) $$ AA:Q-Assoc >
     < AC : Account | bal : B:Nat >
     seq-pay(T, 1)
     =>
        < Self : Bank | cards : qas(Cn:Nat,AC) $$ AA:Q-Assoc >
        < AC : Account | bal : B:Nat >
        < ctx : Context | id : T, opN : pay, obj : Self,
                          args : (arg(debitCard, Cn:Nat),
                                  arg(amount, Amt:Nat)) >
        return(false, T)
     if B:Nat < Amt:Nat .
endm
```

Now we can perform some simulations by defining some initial configurations and using the rewrite command as usual in Maude. For example, we define the following module `TEST-CINEMA-MT`, which uses the `init-state` configuration defined in the `TEST-CINEMA` module and creates some execution threads to buy a couple of tickets.

```
mod TEST-CINEMA-MT is
  pr TEST-CINEMA .

  op init-state-1-mt : -> Configuration .
  eq init-state-1-mt
    = init-state

      newThread(bob)
      next-goCinema-call(bob, 1)
      go-cinema(bob, bob, "Coronet", 1100, 1)
      go-cinema(bob, bob, "Coronet", 1200, 2)

      newThread(ada)
      next-goCinema-call(ada, 1)
      go-cinema(ada, ada, "Coronet", 1100, 1)

      newThread(tom)
      next-goCinema-call(tom, 1)
      go-cinema(tom, tom, "Coronet", 1300, 1) .

  op init-state-2-mt : -> Configuration .
  eq init-state-2-mt
    = init-state
      newThread(bob)
      next-goCinema-call(bob, 1)
      go-cinema(bob, bob, "Coronet", 1100, 1)
      go-cinema(bob, bob, "Coronet", 1100, 2)
      go-cinema(bob, bob, "Coronet", 1200, 3)
      go-cinema(bob, bob, "Coronet", 1300, 4)
      newThread(ada)
      next-goCinema-call(ada, 1)
      go-cinema(ada, ada, "Coronet", 1100, 1)

      newThread(tom)
      next-goCinema-call(tom, 1)
      go-cinema(tom, tom, "Coronet", 1100, 1)
      go-cinema(tom, tom, "Coronet", 1200, 2)
      go-cinema(tom, tom, "Coronet", 1300, 3) .
endm
```

We use the rewrite command and we obtain the following results:

```
Maude> rewrite in TEST-CINEMA : init-state-1-mt . rewrites: 134 in
0ms cpu (7ms real) (~ rewrites/second)
    result Configuration: next-ticket(5)
    < stack : Stack | state : nil,id : bob >
    < stack : Stack | state : nil,id : ada >
    < stack : Stack | state : nil,id : tom >
    < cnm1 : Cinema | name : "Coronet",
                      bank : bbva,sessions : Set{s1, s2, s3} >
    < bbva : Bank | cards : (qas(111, acc1)
                             $$ qas(222, acc2) $$ qas(333, acc3)) >
    < s1 : Session | startTime : 1100,endTime : 1150,capacity : 10,
                     price : 5,ticket : Set{2, 1},reserved : 0 >
    < s2 : Session | startTime : 1200,endTime : 1250,capacity : 10,
                     price : 8,ticket : Set{4},reserved : 0 >
    < s3 : Session | startTime : 1300,endTime : 1350,capacity : 10,
                     price : 5,ticket : Set{3},reserved : 0 >
    < bob : Client | ticket : Set{4, 1},
                     cinemas : Set{cnm1},debitCard : 111 >
    < ada : Client | ticket : Set{2},
                     cinemas : Set{cnm1},debitCard : 222 >
    < tom : Client | ticket : Set{3},
                     cinemas : Set{cnm1},debitCard : 333 >
    < acc1 : Account | bal : 87 >
    < acc2 : Account | bal : 995 >
    < acc3 : Account | bal : 9995 >
    < 1 : Ticket | seat : 0,session : s1,client : bob >
    < 2 : Ticket | seat : 0,session : s1,client : ada >
    < 3 : Ticket | seat : 0,session : s3,client : tom >
    < 4 : Ticket | seat : 0,session : s2,client : bob >

Maude> rewrite in TEST-CINEMA : init-state-2-mt .
rewrites: 280 in 0ms cpu (7ms real) (~ rewrites/second)
result Configuration:
    < stack : Stack | state : nil,id : bob >
    < stack : Stack | state : nil,id : ada >
    < stack : Stack | state : nil,id : tom >
    < cnm1 : Cinema | name : "Coronet",
                      bank : bbva,sessions : Set{s1, s2, s3} >
    < bbva : Bank | cards : (qas(111, acc1)
                             $$ qas(222, acc2) $$ qas(333, acc3)) >
    < s1 : Session | startTime : 1100,endTime : 1150,capacity : 10,
                     price : 5,ticket : Set{4,3,2,1},reserved : 0 >
    < s2 : Session | startTime : 1200,endTime : 1250,capacity : 10,
                     price : 8,ticket : Set{6, 5},reserved : 0 >
    < s3 : Session | startTime : 1300,endTime : 1350,capacity : 10,
                     price : 5,ticket : Set{8, 7},reserved : 0 >
    < bob : Client | ticket : Set{8, 6, 4, 1},
                     cinemas : Set{cnm1},debitCard : 111 >
    < ada : Client | ticket : Set{2},
                     cinemas : Set{cnm1},debitCard : 222 >
    < tom : Client | ticket : Set{7, 5, 3},
                     cinemas : Set{cnm1},debitCard : 333 >
    < acc1 : Account | bal : 77 >
    < acc2 : Account | bal : 995 >
    < acc3 : Account | bal : 9982 >
    < 1 : Ticket | seat : 0,session : s1,client : bob >
    < 2 : Ticket | seat : 0,session : s1,client : ada >
    < 3 : Ticket | seat : 0,session : s1,client : tom >
    < 4 : Ticket | seat : 0,session : s1,client : bob >
    < 5 : Ticket | seat : 0,session : s2,client : tom >
    < 6 : Ticket | seat : 0,session : s2,client : bob >
    < 7 : Ticket | seat : 0,session : s3,client : tom >
    < 8 : Ticket | seat : 0,session : s3,client : bob >
```

The first case is a correct situation, however the second one violates some constraints as we will see in Appendix D.

D Validation of OCL constraints. Examples

This appendix describes how to validate constraints on UML models specified as as Maude system modules. We will consider validation of prototypes with only one execution thread and with multiple execution threads which have already been used with simulation purposed in Appendices B and C.

We start considering the single-thread case and we will use the constraints described in Section 5.2. We need a module with the Maude specification of the cinema model and we use the `CINEMA` module as we did in Appendix B but now importing the `VALIDATION-STACK` module instead of the `STACK` module

```
mod CINEMA is
  pr CLASSES-CINEMA .
  pr VALIDATION-STACK .

  ...
endm
```

Furthermore, we need a module which defines constants `inv`, `pre` and `post` for invariants and pre- and post-conditions. We use the following module `CINEMA-CONSTRAINTS`:

```
mod CINEMA-CONSTRAINTS  is
  pr CLASSES-CINEMA .

  ops $C $T $S $T1 $T2 : -> Vid .

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

  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))))


  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))))) .

  --- ---------------- Invariants  ----------------------
  eq inv =  (seats-available and avoid-overlapping) .

  --- ---------------- Preconditions --------------------
  eq pre(buyTicket)
    = (sessions
          -> select($S | $S . startTime = startTime) -> size() = 1) .

  --- ---------------- Postconditions -------------------
  eq post(buyTicket)
    = ((result = null)
       or
      (sessions -> select($S |
             $S . startTime = startTime) . ticket -> includes(result)
      and
      (sessions -> select($S |
             $S . startTime = startTime) . ticket -> asSet()
       - sessions -> select($S |
             $S . startTime = startTime) . ticket @pre -> asSet())
       -> size() = 1)) .
endm
```

Finally, we need some configuration representing given initial states. Now we use the `init-state1` and `init-state2` configurations, already used in Appendix B with simulation purposes. The `init-state1` defines a correct execution and we get a correct final state:

```
Maude> rewrite in CINEMA-VALIDATION-TEST : [ init-state1 ] .
rewrite in CINEMA-VALIDATION-TEST : [init-state1] .
rewrites: 28421 in 4ms cpu (16ms real) (7105250 rewrites/second)
result Configuration+: {next-goCinema-call(6) next-ticket(6)
    < stack : Stack | state : nil >
    < cnm1 : Cinema | name : "Coronet",
                      bank : bbva,sessions : Set{s1, s2, s3} >
    < bbva : Bank | cards : (qas(111, acc1)
                             $$ qas(222, acc2) $$ qas(333, acc3)) >
    < s1 : Session | startTime : 1100,endTime : 1150,
                     capacity : 10,price : 5,ticket : Set{2, 1} >
    < s2 : Session | startTime : 1200,endTime : 1250,
                     capacity : 10,price : 8,ticket : Set{4, 3} >
    < s3 : Session | startTime : 1300,endTime : 1350,
                     capacity : 10,price : 5,ticket : Set{5} >
    < bob : Client | ticket : Set{3, 1},cinemas : Set{cnm1},
                     debitCard : 111 >
    < ada : Client | ticket : Set{4, 2},cinemas : Set{cnm1},
                     debitCard : 222 >
    < tom : Client | ticket : Set{5},cinemas : Set{cnm1},
                     debitCard : 333 >
    < acc1 : Account | bal : 87 >
    < acc2 : Account | bal : 987 >
    < acc3 : Account | bal : 9995 >
    < 1 : Ticket | seat : 0,session : s1,client : bob >
    < 2 : Ticket | seat : 0,session : s1,client : ada >
    < 3 : Ticket | seat : 0,session : s2,client : bob >
    < 4 : Ticket | seat : 0,session : s2,client : ada >
    < 5 : Ticket | seat : 0,session : s3,client : tom >}
```

However, the `init-state2` configuration defines an erroneous computation because `ada` tries to buy 2 ticket for the same sessions, thus violating the `avoid-overlapping` invariant.

```
Maude> rewrite in CINEMA-VALIDATION-TEST : [ init-state2 ] .
rewrite in CINEMA-VALIDATION-TEST : [init-state2] .
rewrites: 7175 in 4ms cpu (4ms real) (1793750 rewrites/second)
result Configuration+: {error("Invariant or postcondition violation",
    goCinema,next-goCinema-call(2) next-ticket(3)
    < cnm1 : Cinema | name : "Coronet",
                     bank : bbva,sessions : Set{s1, s2, s3} >
    < bbva : Bank | cards : (qas(111,acc1)
                            $$ qas(222, acc2) $$ qas(333, acc3)) >
    < s1 : Session | startTime : 1100,endTime : 1150,
                    capacity : 10,price : 5,ticket : Set{mt-ord} >
    < s2 : Session | startTime : 1200,endTime : 1250,
                    capacity : 10,price : 8,ticket : Set{2, 1} >
    < s3 : Session | startTime : 1300,endTime : 1350,
                    capacity : 10,price : 5,ticket : Set{mt-ord} >
    < bob : Client | ticket : Set{mt-ord},cinemas : Set{cnm1},
                    debitCard : 111 >
    < ada : Client | ticket : Set{2, 1},cinemas : Set{cnm1},
                    debitCard : 222 >
    < tom : Client | ticket : Set{mt-ord},cinemas : Set{cnm1},
                    debitCard : 333 >
    < acc1 : Account | bal : 100 >
    < acc2 : Account | bal : 984 >
    < acc3 : Account | bal : 10000 >
    < 1 : Ticket | seat : 0,session : s2,client : ada >
    < 2 : Ticket | seat : 0,session : s2,client : ada >)}
```

Now we will consider the multi-thread case using the `CINEMA-MT` module described in Appendix C but now importing the `VALIDATION-STACK-MT` to manage stacks considering multi-threaded execution.

```
mod CINEMA-MT is
  pr CLASSES-CINEMA-MT .
  pr VALIDATION-STACK-MT .

  ...
endm
```

In multi-threaded execution we can not use the postcondition previously used in the single-threaded version. It stated that after executing the `buyTicket` operation the ticket obtained is the only ticket added to the tickets of the requested session. However, now new ticked could be added by executing the `buyTicket` operations in a different thread. Thus, we define now a different postcondition for the `buyTicket` operation which guarantees that the balance of the account used to pay the ticket has been correctly updated.

We use now the constraints defined in the `CINEMA-CONSTRAINTS` module, but replacing the equation defining the postcondition for the `buyTicket` operation with the following:

```
eq post(buyTicket)
  = ((result = null)
     or
    (sessions -> select($S |
          $S . startTime = startTime) . ticket -> includes(result)
    and
    (Bank . allInstances() -> asSequence()
         -> first() . cards [aClient . debitCard] . bal
       + sessions -> select($S |
          $S . startTime = startTime) . price -> sum()
     = Bank . allInstances() -> asSequence()
         -> first() . cards [aClient . debitCard] . bal @pre))) .
```

We need again some configuration representing given computations. We use now configurations `init-state2-mt` (which defined an erroneous situation because *bob* tries to buy 2 tickets for the same session) and `init-state3-mt`, which defines a correct execution.

```
op init-state2-mt : -> Configuration .
eq init-state2-mt
  = init-state
    newThread(bob)
    next-goCinema-call(bob, 1)
    go-cinema(bob, bob, "Coronet", 1100, 1)
    go-cinema(bob, bob, "Coronet", 1100, 2)
    go-cinema(bob, bob, "Coronet", 1200, 3)
    go-cinema(bob, bob, "Coronet", 1300, 4)

    newThread(ada)
    next-goCinema-call(ada, 1)
    go-cinema(ada, ada, "Coronet", 1100, 1)

    newThread(tom)
    next-goCinema-call(tom, 1)
    go-cinema(tom, tom, "Coronet", 1100, 1)
    go-cinema(tom, tom, "Coronet", 1200, 2)
    go-cinema(tom, tom, "Coronet", 1300, 3) .

op init-state3-mt : -> Configuration .
eq init-state3-mt
  = init-state
```

```
    newThread(bob)
    next-goCinema-call(bob, 2)
    go-cinema(bob, bob, "Coronet", 1100, 2)
    go-cinema(bob, bob, "Coronet", 1200, 3)
    go-cinema(bob, bob, "Coronet", 1300, 4)

    newThread(ada)
    next-goCinema-call(ada, 1)
    go-cinema(ada, ada, "Coronet", 1100, 1)

    newThread(tom)
    next-goCinema-call(tom, 1)
    go-cinema(tom, tom, "Coronet", 1100, 1)
    go-cinema(tom, tom, "Coronet", 1200, 2)
    go-cinema(tom, tom, "Coronet", 1300, 3) .
```

After executing our cinema model from the `init-state2-mt` state we obtain an error:

```
Maude> rewrite in CINEMA-VALIDATION-TEST : [ init-state2 ] .
rewrite in CINEMA-VALIDATION-TEST : {init-state2-mt} .
rewrites: 19462 in 12ms cpu (14ms real) (1621833 rewrites/second)
result Configuration+: {error("Invariant or postcondition violation",
    ...
    Here the state violating the constrains
    ...
```

However, if we execute our system from the the `init-state3-mt` state we can see that all the constraints are validated and we reach a correct final state.

```
==========================================
Maude> rewrite in CINEMA-VALIDATION-TEST : [ init-state3-mt ] .
rewrite in CINEMA-VALIDATION-TEST : {init-state3-mt} .
rewrites: 48009 in 8ms cpu (54ms real) (6000374 rewrites/second)
result Configuration+: {next-ticket(8) next-goCinema-call(bob, 5)
    next-goCinema-call(ada, 2) next-goCinema-call(tom, 4)
    < stack : Stack | state : nil,id : bob >
    < stack : Stack | state : nil,id : ada >
    < stack : Stack | state : nil,id : tom >
    < cnm1 : Cinema | name : "Coronet",
                      bank : bbva,sessions : Set{s1, s2, s3} >
    < bbva : Bank | cards : (qas(111, acc1)
                             $$ qas(222, acc2) $$ qas(333, acc3)) >
    < s1 : Session | startTime : 1100,endTime : 1150,capacity : 10,
                     price : 5,ticket : Set{3, 2, 1},reserved : 0 >
    < s2 : Session | startTime : 1200,endTime : 1250,capacity : 10,
                     price : 8, ticket : Set{5, 4},reserved : 0 >
    < s3 : Session | startTime : 1300,endTime : 1350,capacity : 10,
                     price : 5,ticket : Set{7, 6},reserved : 0 >
    < bob : Client | ticket : Set{6, 4, 1},cinemas : Set{cnm1},
                     debitCard : 111 >
    < ada : Client | ticket : Set{2},cinemas : Set{cnm1},
                     debitCard : 222 >
    < tom : Client | ticket : Set{7, 5, 3},cinemas : Set{cnm1},
                     debitCard : 333 >
    < acc1 : Account | bal : 82 >
    < acc2 : Account | bal : 995 >
    < acc3 : Account | bal : 9982 >
    < 1 : Ticket | seat : 0,session : s1,client : bob >
    < 2 : Ticket | seat : 0,session : s1,client : ada >
    < 3 : Ticket | seat : 0,session : s1,client : tom >
    < 4 : Ticket | seat : 0,session : s2,client : bob >
    < 5 : Ticket | seat : 0,session : s2,client : tom >
    < 6 : Ticket | seat : 0,session : s3,client : bob >
    < 7 : Ticket | seat : 0,session : s3,
    client : tom >}
```

E Dealing with the *@pre* operator

Postconditions constraints can use the *@pre* operator to refer the value of given properties before the execution of the operation constrained by such postconditions. Thus, the implementation of the *@pre* operator requires to store some information concerning the state before the execution of the operation to which they refer.

When the goal is to validate dynamically OCL constraints during the execution of system *implementations*, it is necessary to provide an efficient implementation of the evaluation of @pre operator that avoid too much waste of time and memory. If only primitive types would involved we could think about storing such values in auxiliary variables. However, this is not the general situation because *@pre* can be used to refer to instances of classes which are not known in the state previous to the execution of the operation. For example, we could have an expression as *person.age@pre* that refers to the age of given person before the execution of the operation, whereas the person can change during the execution of the operation.

For this reason, runtime checking tools have limitations on the use of the *@pre* operator and we can find three different approaches:

- To assume restrictions on the syntax of postcondition expressions, using an approach similar to the one used in Eiffel. In that, only are valid expressions those with format $t_0[t_1...t_n]$, where t_0 is a term which does not uses *@pre* and any term t_i uses `@pre`. This is the approach followed, for example in oclj [16].
- To rely on the user, as in Dresden OCL [46], which assumes that the user will provide a method `createCopy` with which to save the part of the state required for the evaluation of the *@pre* operator.
- To enrich the generated code by using Aspect Oriented Programming to capture the part of the state to be saved, as it is proposed in [24].

In our case we do not work with final implementations and therefore we have less pressing requirements. For this reason we decided to analyze the models to determine the part of the state which could be used in *@pre* expressions, saving the required objects in the execution stack when the *@pre* is used in post-condition expressions.

We evaluate OCL expressions with mOdCL which only requires that we provide one configuration with those objects necessary to evaluate the *@pre* operator in OCL expressions. The rest of objects in configurations are not necessary. Thus, we can filter the configuration representing the system state before executing the operation, excluding those objects that are not implied in the evaluation. For example, for our previous example we know that only objects of class `Person` would be stored.

We use a `filter-pre` operator which, given an OCL expression, returns a new configuration containing only those objects required for the evaluation of the *@pre* operator. Obviously, if the expression does not contains the *@pre* operator an empty configuration (`none`) would be returned.

```
op filter-pre : OclExp Configuration -> Configuration .
```

Once we have the new configuration, it can be stored in the stack and used to provide to mOdCL the previous state implied in the evaluation of the post-condition expression when necessary. The `filter-pre` function is not part of mOdCL, so that we define a module `FILTER-PRE` where it is defined and we provide a new module `mOdCL-pre` which extends mOdCL by importing the `FILTER-PRE` module. Thus, if we want to dynamically validate OCL constraints on Maude prototypes, the `mOdCL-pre` module must be imported instead of the `mOdCL` module.

```
mod mOdCL-pre is
  pr mOdCL .
  pr FILTER-PRE .
endm
```

The implementation of the `filter-pre` function is simple, the resulting configuration is formed by all the instances of classes involved in the navigation expression until to reach a property accessed by using the *@pre* operator. Thus, we start on the object starting the expression and we determine its class and we follow by navigating an determining the classes of those associations implied in navigation expression.

```
op filter-pre : OclExp Configuration -> Configuration .

eq filter-pre(V:AttributeName @pre, Cf)
  = get-obj($eval(self, Cf, none), Cf) .

eq filter-pre((V:AttributeName QL:QF-List) @pre, Cf)
  = get-obj($eval(self, Cf, none), Cf)
    and-furthermore filter-pre-QF-List(QL:QF-List, Cf) .

eq filter-pre(W:OpName @pre (), Cf)
  = get-obj($eval(self, Cf, none), Cf) .

eq filter-pre(W:OpName @pre (LO), Cf)
  = get-obj($eval(self, Cf, none), Cf) and-furthermore filter-pre-args(LO, Cf) .

eq filter-pre(self . V:AttributeName @pre, Cf)
  = get-obj($eval(self, Cf, none), Cf) .

eq filter-pre(self . (V:AttributeName QL:QF-List) @pre, Cf)
  = get-obj($eval(self, Cf, none), Cf)
    and-furthermore filter-pre-QF-List(QL:QF-List, Cf) .

eq filter-pre(self . W:OpName @pre (), Cf)
```

```
  = get-obj($eval(self, Cf, none), Cf) .

eq filter-pre(self . W:OpName @pre (L), Cf)
  = get-obj($eval(self, Cf, none), Cf)
    and-furthermore filter-pre-args(L, Cf) .

eq filter-pre(E1 W:OCL-Attr @pre, Cf)
  = filter-pre(E1, Cf) and-furthermore
    if source-self(E1)
    then filter-objs(collect-class(
                    get-class($eval(self, Cf, none), Cf), E1), Cf)
    else filter-objs(collect-class(locate-class(E1), E1), Cf)
    fi .

eq filter-pre(E1 (W:OCL-Attr QL:QF-List) @pre, Cf)
  = (filter-pre(E1, Cf)
    and-furthermore filter-pre-QF-List(QL:QF-List, Cf))
    and-furthermore
    if source-self(E1)
    then filter-objs(collect-class(
                    get-class($eval(self, Cf, none), Cf), E1), Cf)
    else filter-objs(collect-class(locate-class(E1), E1), Cf)
    fi .

eq filter-pre(E1 . Op @pre(), Cf)
  = filter-pre(E1, Cf) and-furthermore
    if source-self(E1)
    then filter-objs(collect-class(
                    get-class($eval(self, Cf, none), Cf), E1), Cf)
    else filter-objs(collect-class(locate-class(E1), E1), Cf)
    fi .

eq filter-pre(E1 . Op @pre(LO), Cf)
  = (filter-pre(E1, Cf) and-furthermore filter-pre-args(LO, Cf))
    and-furthermore
    if source-self(E1)
    then filter-objs(collect-class(
                    get-class($eval(self, Cf, none), Cf), E1), Cf)
    else filter-objs(collect-class(locate-class(E1), E1), Cf)
    fi .

--- ----------------------------------------
--- Equations to propagate the @pre detection in subexpressions
---    The and-furthermore operator guarantees that we have not
---    duplicated objects
eq filter-pre(E1 = E2, Cf)
  = filter-pre(E1, Cf) and-furthermore filter-pre(E2, Cf) .
eq filter-pre(E1 <> E2, Cf)
  = filter-pre(E1, Cf) and-furthermore filter-pre(E2, Cf) .
eq filter-pre(E1 > E2, Cf)
```

```
  = filter-pre(E1, Cf) and-furthermore filter-pre(E2, Cf) .
eq filter-pre(E1 < E2, Cf)
  = filter-pre(E1, Cf) and-furthermore filter-pre(E2, Cf) .
eq filter-pre(E1 >= E2, Cf)
  = filter-pre(E1, Cf) and-furthermore filter-pre(E2, Cf) .
eq filter-pre(E1 <= E2, Cf)
  = filter-pre(E1, Cf) and-furthermore filter-pre(E2, Cf) .

eq filter-pre(- E1, Cf) = filter-pre(E1, Cf) .
eq filter-pre(E1 + E2, Cf)
  = filter-pre(E1, Cf) and-furthermore filter-pre(E2, Cf) .
eq filter-pre(E1 * E2, Cf)
  = filter-pre(E1, Cf) and-furthermore filter-pre(E2, Cf) .
eq filter-pre(E1 / E2, Cf)
  = filter-pre(E1, Cf) and-furthermore filter-pre(E2, Cf) .
eq filter-pre(E1 - E2, Cf)
  = filter-pre(E1, Cf) and-furthermore filter-pre(E2, Cf) .

eq filter-pre(not E1, Cf) = filter-pre(E1, Cf) .
eq filter-pre(E1 or E2, Cf)
  = filter-pre(E1, Cf) and-furthermore filter-pre(E2, Cf) .
eq filter-pre(E1 and E2, Cf)
  = filter-pre(E1, Cf) and-furthermore filter-pre(E2, Cf) .
eq filter-pre(E1 xor E2, Cf)
  = filter-pre(E1, Cf) and-furthermore filter-pre(E2, Cf) .
eq filter-pre(E1 implies E2, Cf)
  = filter-pre(E1, Cf) and-furthermore filter-pre(E2, Cf) .
eq filter-pre(if E1 then E2 else E3 endif, Cf)
  = filter-pre(E1, Cf) and-furthermore (filter-pre(E2, Cf)
    and-furthermore filter-pre(E3, Cf)) .

eq filter-pre((E1 . O:Name0), Cf) = filter-pre(E1, Cf) .
eq filter-pre((E1 . O:Name1(E2)), Cf)
   = filter-pre(E1, Cf) and-furthermore filter-pre(E2, Cf) .
eq filter-pre((E1 . O:Name2(E2, E3)), Cf)
  = filter-pre(E1, Cf) and-furthermore (filter-pre(E2, Cf)
    and-furthermore filter-pre(E3, Cf)) .
eq filter-pre(E1 -> O:Name0(), Cf) = filter-pre(E1, Cf) .
eq filter-pre(E1 -> O:Name1(E2), Cf)
  = filter-pre(E1, Cf) and-furthermore filter-pre(E2, Cf) .
eq filter-pre(E1 -> O:Name2(E2, E3), Cf)
  = filter-pre(E1, Cf) and-furthermore (filter-pre(E2, Cf)
    and-furthermore filter-pre(E3, Cf)) .
eq filter-pre(E1 -> I:IteratorName  (V | E2), Cf)
  = filter-pre(E1, Cf) and-furthermore filter-pre(E2, Cf) .
eq filter-pre(E1 -> I:IteratorName  (E2), Cf)
  = filter-pre(E1, Cf) and-furthermore filter-pre(E2, Cf) .

eq filter-pre(Set{mt-ord}, Cf) = none .
eq filter-pre(Set{E1 , L}, Cf)
```

```
  = filter-pre(E1, Cf) and-furthermore filter-pre(Set{L}, Cf) .
eq filter-pre(Bag{mt-ord}, Cf) = none .
eq filter-pre(Bag{E1 , L}, Cf)
  = filter-pre(E1, Cf) and-furthermore filter-pre(Bag{L}, Cf) .
eq filter-pre(Sequence{mt-ord}, Cf) = none .
eq filter-pre(Sequence{E1 , LO}, Cf)
  = filter-pre(E1, Cf) and-furthermore filter-pre(Sequence{LO}, Cf) .
eq filter-pre(OrderedSet{mt-ord}, Cf) = none .
eq filter-pre(OrderedSet{E1 , LO}, Cf)
  = filter-pre(E1,Cf) and-furthermore filter-pre(OrderedSet{LO},Cf) .

eq filter-pre(E1 . oclIsUndefined(), Cf) = filter-pre(E1, Cf) .
eq filter-pre(E1 . oclIsKindOf(C:Cid), Cf) = filter-pre(E1, Cf) .
eq filter-pre(E1 . oclIsTypeOf(C:Cid), Cf) = filter-pre(E1, Cf) .

eq filter-pre(C:Cid . allInstances(), Cf) = none .
eq filter-pre(C:Cid . allInstances, Cf) = none .

eq filter-pre(V:Vid, Cf) = none .

eq filter-pre(E1 W:OCL-Attr, Cf) = filter-pre(E1, Cf) .
eq filter-pre(E1 (W:OCL-Attr QL:QF-List), Cf)
  = filter-pre(E1, Cf)
    and-furthermore filter-pre-QF-List(QL:QF-List, Cf) .
eq filter-pre(E1 . Op(), Cf) = filter-pre(E1, Cf) .
eq filter-pre(E1 . Op(LO), Cf)
   = filter-pre(E1, Cf) and-furthermore filter-pre-args(LO, Cf) .

eq filter-pre(V:AttributeName, Cf) = none .
eq filter-pre(V:AttributeName QL:QF-List, Cf)
  = filter-pre-QF-List(QL:QF-List, Cf) .
eq filter-pre(W:OpName(), Cf) = none .
eq filter-pre(W:OpName(LO), Cf) = filter-pre-args(LO, Cf) .
eq filter-pre(V:OclAny, Cf) = none .

--- Auxiliary functions
---
--- -------------------------------------------------------
--- We assume that one class-of equation has been included in the
--- CLASSES-m module
---   - for each attribute of user defined class
---   - for each function returning an object ob user defined class
op class-of : Cid AttributeName -> Cid .
op class-of : Cid OpName -> Cid .

--- -------------------------------------------------------
op _and-furthermore_ : Configuration Configuration -> Configuration .
eq (O:Object Cf) and-furthermore (O' Cf')
  = if (O == O') then Cf and-furthermore (O' Cf')
    else Cf and-furthermore (O O' Cf') fi .
```

```
eq (M:Msg Cf) and-furthermore (M':Msg Cf')
  = if (M:Msg == M':Msg) then Cf and-furthermore (M':Msg Cf')
    else Cf and-furthermore (M:Msg M':Msg Cf') fi .
eq Cf and-furthermore Cf' =  Cf Cf' [owise] .

--- --------------------------------------------------------
op get-obj : Oid Configuration -> Object .
eq get-obj(O:Oid, < O:Oid : C:Cid | AS:AttributeSet > Cf)
  = < O:Oid : C:Cid | AS:AttributeSet > .

--- --------------------------------------------------------
op filter-objs : Cid Configuration -> Configuration .
eq filter-objs(C1:Cid, < O:Oid : C2:Cid | AS:AttributeSet > Cf)
  = if (C1:Cid == C2:Cid) or isSubClass(C2:Cid, C1:Cid)
    then < O:Oid : C2:Cid | AS:AttributeSet > filter-objs(C1:Cid,Cf)
    else filter-objs(C1:Cid, Cf)
    fi .
eq filter-objs(C1:Cid, Cf) = none
[owise] .

--- --------------------------------------------------------
op filter-pre-QF-List : QF-List Configuration -> Configuration .
eq filter-pre-QF-List([mt-ord], Cf) = none .
eq filter-pre-QF-List([E1 , LO], Cf)
  = filter-pre(E1,Cf) and-furthermore filter-pre-QF-List([LO],Cf) .

--- --------------------------------------------------------
op filter-pre-args :  List{OclExp} Configuration -> Configuration .
eq filter-pre-args(mt-ord, Cf) = none .
eq filter-pre-args((E1 , LO), Cf)
  = filter-pre(E1, Cf) and-furthermore filter-pre-args(LO, Cf) .

--- --------------------------------------------------------
--- Given an expression which does not start with self (explicit
--- or implicit), it starts with C:Cid . allInstances().
--- locate-class returns the class
op locate-class : OclExp -> Cid .
eq locate-class(C:Cid . allInstances()) = C:Cid .
eq locate-class(C:Cid . allInstances) = C:Cid .
eq locate-class(E1 . E2) = locate-class(E1) .
eq locate-class(E1 . E2 @pre) = locate-class(E1) .
eq locate-class(E1 . E2 QL:QF-List) = locate-class(E1) .
eq locate-class(E1 . E2 QL:QF-List @pre) = locate-class(E1) .
eq locate-class(E1 . W:OpName()) = locate-class(E1) .
eq locate-class(E1 . W:OpName (LO)) = locate-class(E1) .
eq locate-class(E1 . W:OpName @pre ()) = locate-class(E1) .
eq locate-class(E1 . W:OpName @pre (LO)) = locate-class(E1) .
eq locate-class(E1 -> O:Name0() ) = locate-class(E1) .
eq locate-class(E1 -> O:Name1(E2) ) = locate-class(E1) .
eq locate-class(E1 -> O:Name2(E2, E3) ) = locate-class(E1) .
```

```
--- --------------------------------------------------------
--- source-self determines if a given expression starts with
--- explicit self or with AttributeName or invocation a function
--- (self implicit)
op source-self : OclExp -> Bool .
eq source-self(self) = true .
eq source-self(V:AttributeName) = true .
eq source-self(V:AttributeName QL:QF-List) = true .
eq source-self((V:AttributeName QL:QF-List) @pre) = true .
eq source-self(W:OpName()) = true .
eq source-self(W:OpName (LO)) = true .
eq source-self(V:AttributeName @pre) = true .
eq source-self(W:OpName @pre ()) = true .
eq source-self(W:OpName @pre (LO)) = true .

eq source-self(E1 . E2) = source-self(E1) .
eq source-self(E1 . E2 @pre) = source-self(E1) .
eq source-self(E1 . E2 QL:QF-List) = source-self(E1) .
eq source-self(E1 . E2 QL:QF-List @pre) = source-self(E1) .
eq source-self(E1 . W:OpName()) = source-self(E1) .
eq source-self(E1 . W:OpName (LO)) = source-self(E1) .
eq source-self(E1 . W:OpName @pre ()) = source-self(E1) .
eq source-self(E1 . W:OpName @pre (LO)) = source-self(E1) .
eq source-self(E1 -> O:Name0() ) = source-self(E1) .
eq source-self(E1 -> O:Name1(E2) ) = source-self(E1) .
eq source-self(E1 -> O:Name2(E2, E3) ) = source-self(E1) .
eq source-self(E1 -> O:IteratorName(V | E2)) = source-self(E1) .
eq source-self(E1 -> O:IteratorName(E2)) = source-self(E1) .

eq source-self(E1) = false
[owise] .

--- --------------------------------------------------------
--- Given an expression which starts in an object of a given class,
--- returns the class of the object containing the attribute
--- referred by @pre. In case of complex expression with more that
--- one @pre clauses it returns the class concerning the last.
--- For example, in    a . b @pre . c . d @pre it returns the
--- class which contains the attribute 'd'
op collect-class : Cid OclExp -> Cid .

eq collect-class(C:Cid, self) = C:Cid .
eq collect-class(C:Cid, C':Cid . allInstances()) = C':Cid .
eq collect-class(C:Cid, C':Cid . allInstances) = C':Cid .

eq collect-class(C:Cid, AT) = class-of(C:Cid, AT) .
eq collect-class(C:Cid, AT @pre) = class-of(C:Cid, AT) .
eq collect-class(C:Cid, AT QL:QF-List) = class-of(C:Cid, AT) .
eq collect-class(C:Cid, AT QL:QF-List @pre) = class-of(C:Cid, AT) .
eq collect-class(C:Cid, E1 . AT)
  = class-of(collect-class(C:Cid, E1), AT) .
eq collect-class(C:Cid, E1 . AT @pre)
  = class-of(collect-class(C:Cid, E1), AT) .
eq collect-class(C:Cid, E1 . AT QL:QF-List)
  = class-of(collect-class(C:Cid, E1), AT) .
eq collect-class(C:Cid, E1 . AT QL:QF-List @pre)
  = class-of(collect-class(C:Cid, E1), AT) .

eq collect-class(C:Cid, Op()) = class-of(C:Cid, Op) .
eq collect-class(C:Cid, E1 . Op())
  = class-of(collect-class(C:Cid, E1), Op) .
eq collect-class(C:Cid, Op @pre()) = class-of(C:Cid, Op) .
eq collect-class(C:Cid, E1 . Op @pre())
  = class-of(collect-class(C:Cid, E1), Op) .
eq collect-class(C:Cid, Op(LO)) = class-of(C:Cid, Op) .
eq collect-class(C:Cid, E1 . Op(LO))
  = class-of(collect-class(C:Cid, E1), Op) .
eq collect-class(C:Cid, Op @pre(LO)) = class-of(C:Cid, Op) .
eq collect-class(C:Cid, E1 . Op @pre(LO))
  = class-of(collect-class(C:Cid, E1), Op) .

eq collect-class(C:Cid,E1 -> O:Name0()) = collect-class(C:Cid,E1) .
eq collect-class(C:Cid,E1 -> O:Name1(E2)) = collect-class(C:Cid,E1) .
eq collect-class(C:Cid,E1 -> O:Name2(E2,E3)) = collect-class(C:Cid,E1) .

eq collect-class(C:Cid,E1 -> select(V | E2)) = collect-class(C:Cid,E1) .
eq collect-class(C:Cid,E1 -> reject(V | E2)) = collect-class(C:Cid,E1) .
eq collect-class(C:Cid,E1 -> any(V | E2)) = collect-class(C:Cid, E1) .
eq collect-class(C:Cid,E1 -> sortedBy(V | E2)) = collect-class(C:Cid,E1) .

eq collect-class(C:Cid, E1 -> collect(V | E2))
  = collect-class(collect-class(C:Cid, E1), E2) .
eq collect-class(C:Cid, E1 -> collectNested(V | E2))
  = collect-class(collect-class(C:Cid, E1), E2) .
```