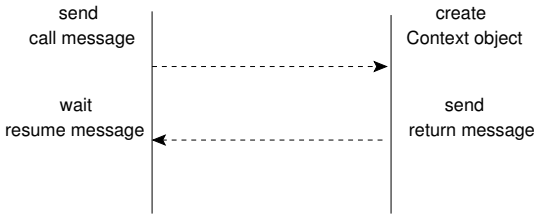

Figure 1: Basic scheme for methods behavior specification.

The cinema model. Maude specification of the system behavior

Rewriting logic defines a suitable framework to formally specify different languages and systems, and Maude provides an efficient interpreter with which to execute the specifications, which can be used as a rapid prototype of systems under study. In the following we will show our proposal to specify UML models in Maude. If we follow the proposed guidelines to specify the model in Maude we can take the specification obtained and use mOdCL to dynamically validate the constraints defined in the class diagram during the execution of the Maude prototype of the UML model.

Although we consider either single threaded or multi-threaded execution, we assume here 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. The behavior of each method will be represented as a sequence of Maude rules, which may need to access some information related to its invocation, as the object invoking the method or the actual parameters used in the invocation. These rules can assume that such information is stored in an object of class `Context`, representing the execution context with the appropriate information for the running method, with the form

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

where *ctx* is the identifier of the object, *m* is the name of the active method, *id* is the identifier of the current object, and *vars* is a set of pairs with the name of a parameter or a local variable and its actual value.

To manage the chaining of method invocations—an invoked method can invoke another one (or recursively to itself)— we handle an execution stack in which we store the execution contexts of pending methods. The stack management is provided by a `STACK` module (see below) which is responsible for creating and handling execution contexts, and it is transparent for the user modeling the system, which simply uses the execution context of the active method when necessary during the specification of a method behavior. For example, if he need to know the value of some parameter, it can be found in the `args` attribute of the active `Context` object.

We assume that the rules implementing a given method follow a basic scheme, which requires the use of specific messages, (see Figure 1), as follows:

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

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

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

As an example, we show below some of the Maude rules to represent the system behavior of the cinema system. Figure 2 shows the sequence diagram for the `goCinema` method, to buy a ticket for a given session in a cinema. It begins by sending a `buyTicket` message to the cinema for the given start time. Depending on whether there are tickets available or not for that session, a `Ticket` object or a `null` will be returned. And then, depending on the result of this message, the ticket would be paid. We illustrate the approach by giving a few rules.

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

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

the stack infrastructure handles the stack as appropriate and puts in the configuration an object

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

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

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

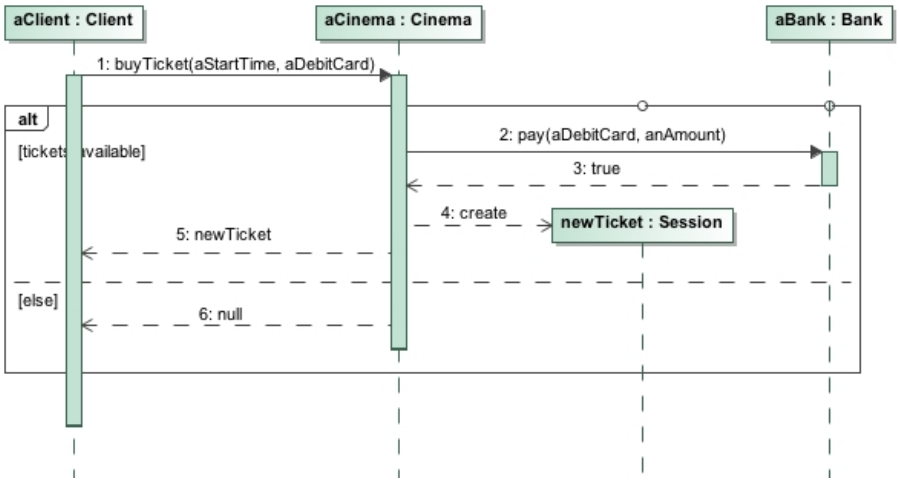

Figure 2: Sequence diagram for the goCinema operation.

```
< S : Session | startTime : St, Atts3 >
< ctx : Context | op : goCinema, self : Self,
                  args : arg(cinema, Cn), arg(startTime, St))
call(buyTicket, C, (arg(startTime, St), arg(client, Self))) .
```

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

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

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

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

The proposed scheme assumes an execution stack to deal with execution contexts. Although it is transparent to the user, which can specify the system behavior using the scheme described previously and without having any information about the stack specification, if you are interested yo can get the stack specification in Section 2.1 (normal users can skip this section).

1 Complete cinema specification

Now we include the complete executable specification of the system behavior for the cinema model, according to the scheme introduced before for specifying the operations. We define a `CINEMA` module which imports from the `CLASSES-CINEMA` module 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 .
```

2 The System Execution

Once we have a Maude prototype for the UML model we can define an initial state and use the `rewrite` command to execute the system. For our cinema model we can, for example, create 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
```

and to execute the system as is usual in Maude obtaining final states with some tickets sold.

```
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 >
```

2.1 The execution stack

The stack is an object of class `Stack` with only one attribute `state` with a list of objects representing the sequence of execution contexts of those methods waiting for the completion of any previous call. We define the `state` attribute of sort `ListM{Object}`, using a view `Object` to instantiate the parameterized module `LISTM`, which defines lists of items separated by `::`.

```
view Object from TRIV to CONFIGURATION is
  sort Elt to Object .
endv
```

A `Context` object contains the current operation name (`opN`), its arguments (`args`) and the invoking object (`obj`), where `OpName` and `ArgsList` are sorts defined by the mOdCL evaluator.

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 we introduced above. To this end, the initial system state is represented as an special situation by using a `Configuration` enclosed by square brackets, and an `INIT` equation transparently 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 method. 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 method.

```
mod STACK is
  pr mOdCL .
  pr LISTM{Object} .

  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 : ListM{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
```