How to use mOdCL

The mOdCL evaluate offers an `eval` function which allows to evaluate OCL expressions. This function can be used with three different interfaces, depending on the context required for the evaluation of the expression.

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

If the OCL expression is not referred to a given UML state the `eval` function only needs the expression to be evaluated. Otherwise, it refers to a given state and the `eval` function needs the object configuration representing such a state. Finally, those postcondition expressions using the *@pre* operator refer to two states (the actual state and the previous state referred be means of *@pre*) and therefore the `eval` function would use two objects configurations.

Evaluation of expressions in general

The mOdCL evaluator allows to evaluate OCL expressions in general, not only OCL constraints. For example, we could use it to calculate the number of sessions in a given state. Let's suppose that we have a module `CLASSES-CINEMA` which contains the representation of the UML class diagram, and a module `TEST-CINEMA` which contains the definition of a constant state with the configuration representing the system state referred from the expression. We only need two simple steps:

- To load the module `TEST-CINEMA` in the Maude tool (remember that this module imports the `mOdCL` module and the `CLASSES-CINEMA` module).

  ```
  mrc:~/cinema$ maude test-cinema.maude
  ```

- To use the `reduce` command from the Maude command line to execute the `eval` function on the `state` configuration.

  ```
  Maude> reduce in TEST-CINEMA :
              eval(Session . allInstances() -> size(), state) .
  result result NzNat: 3
  ```

Validation of constraints

Now we will show how to use the `eval` function to evaluate OCL expressions representing invariants or pre- and post-conditions.

An invariant is an OCL expressions which return a logical result. There is no difference between the evaluation of general expressions and the validation of invariants. We only have to provide the expression representing the invariants and the state which they refer. In our *Cinema* model we could validate if the object diagram represented by the `state` constant fulfill the invariants imposed in the class diagram as follows:

```
Maude> reduce in TEST-CINEMA :
           eval(avoid-overlapping and seats-in-session, state) .
result Bool: true
```

Pre- and post-condition are expressions which refer, respectively, to the states before and after executing a given operation. We use the following module `TEST-CINEMA-2` to define constants `state-pre` and `state-post` representing such states. The system state before the invocation is represented by the term `state` (as it was used in the previous example) and the system state after the execution of the operation is represented by a new term.

```
mod TEST-CINEMA-1 is
  protecting TEST-CINEMA .

  op state-pre : -> Configuration .
  eq state-pre = state .

  op state-post : -> Configuration .
  eq state-post =
     < cn : Cinema | bank : bbva, sessions : Set{s1 ; s2 ; s3} >
     < s1 : Session | startTime : 1100, endTime : 1150,
              capacity : 10, price : 5, ticket : Set{1 ; 3 ; 4} >
     < s2 : Session | startTime : 1200, endTime : 1250,
              capacity : 10, price : 8, ticket : Set{2} >
     < s3 : Session | startTime : 1300, endTime : 1350,
              capacity : 10, price : 5, ticket : Set{} >
     < paul : Client | cinemas : Set{cn}, ticket : Set{1 ; 2},
                       debitCard : 111 >
     < eve  : Client | cinemas : Set{cn}, ticket : Set{3},
                       debitCard : 222 >
     < bob : Client | cinemas : Set{cn}, ticket : Set{4} ,
                       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 : 99995 >
     < 1 : Ticket | seat : 1, session : s1, client : paul >
     < 2 : Ticket | seat : 1, session : s2, client : paul >
     < 3 : Ticket | seat : 2, session : s1, client : eve >
     < 4 : Ticket | seat : 3, session : s1, client : bob > .
```

Although the `state-pre` and `state-post` configurations represent the information described in the object diagrams, they do not contain any information concerning the invocation. The mOdCL evaluator requires that the object on which an operation is executed (`object`) be in the active variables environment (a message `env(self <- object)`). Furthermore, the actual parameters of the invocation must be included in an `OpEnv` message, as pairs with the name of the argument and its actual value. Finally, the result of

the operation must be a special argument whose name is `result`. In this example we define new constants `state-pre'` and `state-post'` which include such messages, thus preparing the terms as they are required by mOdCL. The message `OpEnv(arg(aClient, bob), arg(startTime, 1100))` represents the execution environment before the invocation of the `buyTicket` operation on an object `cinema` to buy a ticket for the session starting at `1100`. The message `OpEnv(arg(aClient,bob),arg(startTime,1100), arg(result,4))` represents the execution environment in the final state, after the execution of the invoked operation, where it returns a ticket labeled as 4.

```
  op state-pre' : -> Configuration .
  eq state-pre'
    = state-pre env(self <- cn)
      OpEnv((arg(aClient, bob), arg(startTime, 1100))) .

  op state-post' : -> Configuration .
  eq state-post'
    = state-post env(self <- cn)
      OpEnv((arg(aClient, bob),
             arg(startTime, 1100), arg(result, 4))) .
endm
```

Now we are able to validate the preconditions of the operation on the object diagram corresponding to the state before the invocation, by evaluating the expression `pre-buyTicket`

```
Maude> reduce in TEST-CINEMA-1 : eval(pre-buyTicket, state-pre') .
result Bool: true
```

and to validate the postconditions on the object diagram corresponding to the states before and after the execution of the operation, by evaluating the expressions `post-buyTicket`

```
Maude> reduce in TEST-CINEMA-1 :
          eval(post-buyTicket, state-post', state-pre') .
result Bool: true
```