Dynamic validation of OCL constraints with mOdCL

Manuel Roldán   Francisco Durán

Universidad de Málaga, Spain

OCL 2011

Our aims

- In model-driven developments, particular attention should be paid to checking crucial properties on models to guarantee software quality.
- Tools support for validating OCL constraints on UML models:
  - A number of tools allows static validation of models.
  - Some tools allow dynamic validation on the implementation of the system.
- The Maude language allows to obtain an executable model of an UML model.
  - We can dynamically validate OCL constraints on the model.

Our approach

- We translate the UML/OCL models into the algebraic specification language and system Maude.
- Specifically, using mOdCL
  - invariants are represented by state predicates,
  - operations by Maude rules, and
  - pre- and postconditions by predicates as well.
- An execution strategy controls the rules execution and checks the constraints.

The Maude system

- Formal notation and system
  - high-level language and a high-performance interpreter and compiler in the OBJ algebraic specification family
  - supports MEL and RL specification and programming
- Supported by a formal toolkit
  - execution of specifications
  - reachability analysis
  - model-checking
  - theorem proving
  - etc.
- Used in many different areas
  - Models of computation
  - Semantics of programming languages and software analysis
  - Modeling and analysis of networks and distributed systems
    - Distributed architectures and components
    - Specification and analysis of communication protocols
    - Modeling and analysis of security protocols
  - ...

Classes, objects, messages, and configurations

- Classes

```
sort Account .
subsort Account < Cid .
op Account : -> Account .
op balance :_ : Int -> Attribute .
```

- `Object` of objects

```
op <_:_|_> : Oid Cid AttributeSet -> Object .
```

```
< a : Account | balance : 5 >
```

- `Msg` of messages

```
op withdraw : Oid Int -> Msg .
```

```
withdraw(a, 3)
```

- `Configuration` of multisets of objects and messages

```
sort Configuration .
subsorts Object Message < Configuration .
op none : -> Configuration .
op __ : Configuration Configuration -> Configuration
    [assoc comm id: none] .
```

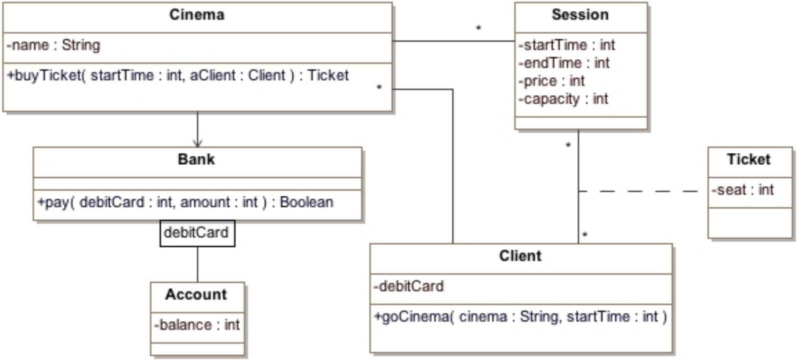
```
< a : Account | balance : 5 >
withdraw(a, 3)
```

Concurrent rewriting

- Concurrent states are represented as configurations of objects and messages
- that evolve by concurrent rewriting
- using rules that describe the effects of the communication events of objects and messages.

$$
\begin{array}{l}
\texttt{crl [}r\texttt{] :} \\
\quad \texttt{<}\ O_1\ \texttt{:}\ C_1\ \texttt{|}\ atts_1\ \texttt{>}\ \ldots\ \texttt{<}\ O_n\ \texttt{:}\ C_n\ \texttt{|}\ atts_n\ \texttt{>} \\
\quad M_1\ \ldots\ M_m \\
\quad \texttt{=>}\ \texttt{<}\ O_{i_1}\ \texttt{:}\ C'_{i_1}\ \texttt{|}\ atts'_{i_1}\ \texttt{>}\ \ldots\ \texttt{<}\ O_{i_k}\ \texttt{:}\ C'_{i_k}\ \texttt{|}\ atts'_{i_k}\ \texttt{>} \\
\qquad \texttt{<}\ Q_1\ \texttt{:}\ C''_1\ \texttt{|}\ atts''_1\ \texttt{>}\ \ldots\ \texttt{<}\ Q_p\ \texttt{:}\ C''_p\ \texttt{|}\ atts''_p\ \texttt{>} \\
\qquad M'_1\ \ldots\ M'_q \\
\quad \texttt{if}\ Cond\ \texttt{.}
\end{array}
$$

Running example: ticket sale system



Invariants

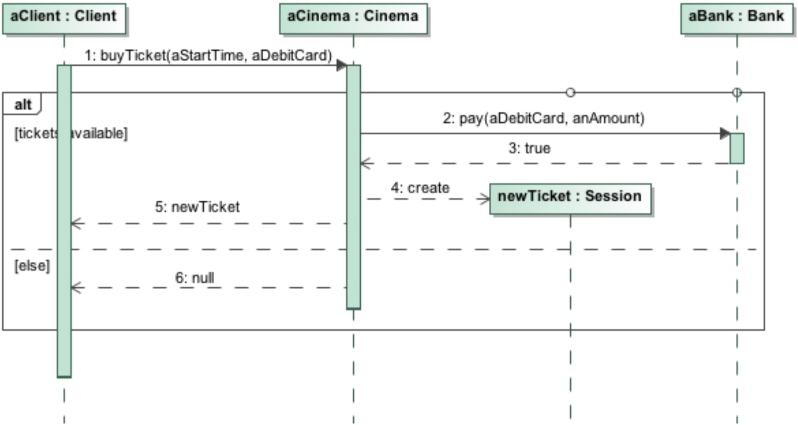
```
context Client inv avoid-overlapping :
    tickets->forAll(T1 |
      tickets->forAll(T2 | (T1 = T2)
                or (T1.session.endTime < T2.session.startTime)
                or (T2.session.endTime < T1.session.startTime)))

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

Pre- and post-conditions of the `buyTicket` operation

```
context Cinema::buyTicket(st:Integer, cl:Client):Ticket
pre : sessions -> select(S | S.startTime = st) -> size() = 1 .
post: (result = null)
       or
      -- tickets of the session must include the result ticket
      (sessions -> select(S | S.startTime = st).tickets -> includes(result)
       and
      -- the number of tickets increases in 1 unit
       ((sessions -> select(S | S.startTime = st).tickets->asSet())
        - (sessions -> select(S | S.startTime = st).tickets
             @pre->asSet()))
       -> size() = 1)
```

Running example: sequence diagram



The mOdCL representation of the system structure

- User-defined classes are represented as Maude classes. Attributes and associations are represented as constants of the mOdCL sort `AttributeName`.

```
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).
- An operation *op*(arg_1 : $type_1$, . . . , arg_n : $type_n$) : *type* is represented as an `OpName` constant *op* and `Arg` constants arg_1, . . ., arg_n.

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

OCL expressions in mOdCL: invariants

- OCL expressions are represented as terms of sort `OclExp`.

```
ops seats-in-session avoid-overlapping : -> OclExp .

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

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))))
```

- A constant `inv` is defined for invariants.

```
op inv : -> OclExp .

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

OCL expressions in mOdCL: pre- and post-conditions

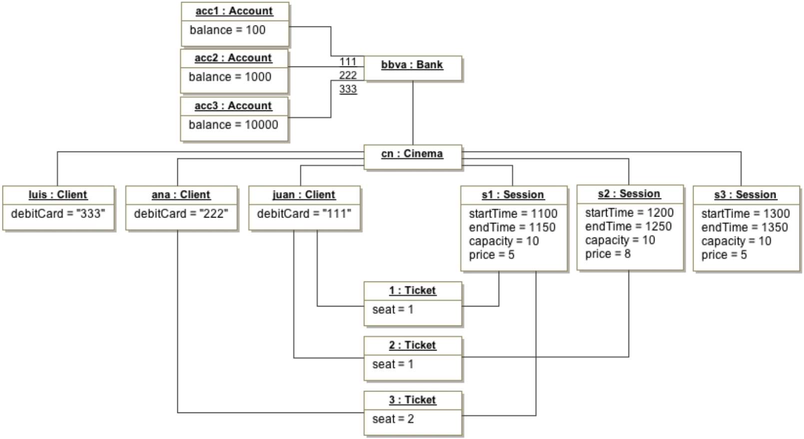
- pre and post operators must be defined for each method.

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

eq pre(buyTicket)
  = session -> select(S | S . startTime = startTime) -> size() = 1 .

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

Validating with mOdCL: an object diagram



The mOdCL representation of the object diagram

```
mod CINEMA-TEST is
  pr CINEMA .               --- Cinema model definition
  pr CINEMA-CONSTRAINTS .   --- Constraints for the Cinema model

  op state : -> Configuration .
  eq state
    = < cn : Cinema | bank : bbva, sessions : Set{s1, s2, s3} >
      < s1 : Session | startTime : 1100, endTime : 1150, capacity : 10,
                       price : 5, ticket : Set{1, 3} >
      < 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{} >
      < juan : Client | cinemas : Set{cn}, ticket : Set{1, 2}, debitCard : 111 >
      < ana : Client | cinemas : Set{cn}, ticket : Set{2}, debitCard : 222 >
      < luis : Client | cinemas : Set{cn}, 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 >
      < 1 : Ticket | seat : 1, session : s1, client : juan >
      < 2 : Ticket | seat : 1, session : s2, client : juan >
      < 3 : Ticket | seat : 2, session : s1, client : ana > .
 endm
```

Static validation

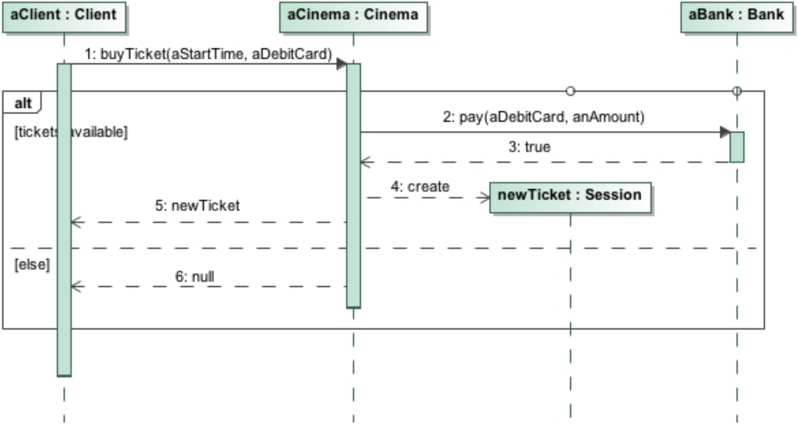
- mOdCL offers an eval function to evaluate an OCL expression.

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

E.g., we can validate the state object diagram.

```
Maude> red in CINEMA-TEST : eval(seats-in-season, state).
result: true
```

Running example: sequence diagram



The mOdCL representation of the system behavior

- mOdCL expects that methods are invoked with a message

  ```
  call(<method-name>, <addressee>, <argument-list>)
  ```

  and upon their completion they send a return message of the form

  ```
  return(<return-value>)
  ```

- When a method `m` calls a given operation `m'` the rules representing the `m` method block until the completion of `m'`. A `resume` message must be used to block such rules and to get the result.

  ```
  resume(m', <return-value>)
  ```

Infrastructure for operation calls

- The infrastructure of mOdCL will intercept and process these messages.
- The processing of a `call` operator results in the execution of a method, for which a context object, representing the execution context, is generated.

```
< Ctx : Context | op : M, self : Id, args : Args >
```

- A `return(<return-value>)` message will be replaced by a resume message of the form

```
resume(<return-value>)
```

- To manage the chaining of method invocations the validator uses an execution stack in which the necessary information is stored.

The execution stack

- The stack infrastructure is built around CALL and RETURN rules. We make use of this in the metaOCLRewrite strategy to locate the states where some constraints must be validated.

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

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

An example: the goCinema method

- A message call activates its execution.

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

- First rule. A call to buyTicket.

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

- It blocks and waits for a resume message from buyTicket.

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

An example: the buyTicket method

- Last rule of the buyTicket method (no free seats).

```
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 .
```

Dynamic validation

- We provide the `validate` command to validate the OCL constraints of an UML during its execution.

```
Maude> (validate in TEST-CINEMA with CINEMA-CONSTRAINTS from state .)
result Configuration:
  < cn : 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{1, 2} >
  < s2 : Session | startTime : 1200, endTime : 1250, capacity : 10, price : 8,
                   ticket : Set{3, 4} >
  < s3 : Session | startTime : 1300, endTime : 1350, capacity : 10, price : 5,
                   ticket : Set{5} >
  < juan : Client | ticket : Set{1, 3}, cinemas : Set{cn}, debitCard : 111 >
  < ana : Client | ticket : Set{2, 4}, cinemas : Set{cn}, debitCard : 222 >
  < luis : Client | ticket : Set{5},cinemas : Set{cn}, debitCard : 333 >
  < acc1 : Account | bal : 87 >
  < acc2 : Account | bal : 987 >
  < acc3 : Account | bal : 9995 >
  < 1 : Ticket | seat : 0, session : s1, client : juan >
  < 2 : Ticket | seat : 0, session : s1, client : ana >
  < 3 : Ticket | seat : 0, session : s2, client : juan >
  < 4 : Ticket | seat : 0, session : s2, client : ana >
  < 5 : Ticket | seat : 0, session : s3, client : luis >
  next-goCinema-call(6)
  next-ticket(6)
```

Dynamic validation

- In case of error we inform about the kind of error and the erroneous state.

```
Maude> (validate in TEST-CINEMA with CINEMA-CONSTRAINTS from state-1 .)
result Error:
  error("Precondition violation",
        ... name of operation ...
        session -> select(S | S . startTime = startTime) -> size() = 1,
        ... here the erroneous state ...)
```

Architecture: the mOdCL evaluator

- Sorts definition

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

- Syntax for OCL

```
vars E1 E1 : OclExp .
var  C : Configuration .

op _-> includes(_) : OclExp OclExp -> OclExp .
eq eval(E1 -> includes(E2), C) = eval(E1, C) in eval(E2, C) .
```

- The eval function

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

Architecture: the metaOCLRewrite strategy

- metaOCLRewrite controls the execution of the Maude rules specifying the UML model.
- It is implemented at the metalevel, and using the eval function to evaluate a given OclExp at a given state.

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

The checkCall and checkReturn functions

They check whether a given state satisfies given constraints represented as inv, pre and post terms.

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

Conclusions and future work

- UML dynamics is represented as Maude rules.
- Our mOdCL tool allows both static and dynamic validation on the UML model.
- The infrastructure used is hidden to the user.
- We plan to automate the generation of skeletons for the Maude rules representing UML models from sequence diagrams.
- The Maude formal environment open many possibilities
  - Reachability analysis
  - Verification
  - ...