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 .  
  < 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.

crl \ [r] : 
< O_1 : C_1 | atts_1 > ... < O_n : C_n | atts_n > 
M_1 ... M_m 
=> < O_{i_1} : C'_{i_1} | atts'_{i_1} > ... < O_{i_k} : C'_{i_k} | atts'_{i_k} > 
< Q_1 : C''_1 | atts''_1 > ... < Q_p : C''_p | atts''_p > 
M'_1 ... M'_q 
if Cond .
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 \texttt{buyTicket} operation

```plaintext
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

```
alt
[tickets available]

1: buyTicket(aStartTime, aDebitCard)

2: pay(aDebitCard, anAmount)

3: true

4: create

newTicket : Session

5: newTicket

6: null
```

```
[aClient : Client]

[aCinema : Cinema]

[aBank : Bank]
```
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.

```plaintext
sort Cinema .
subsort Cinema < Cid .
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 \(op(arg_1: type_1, \ldots, arg_n: type_n): type\) is represented as an OpName constant \(op\) and Arg constants \(arg_1, \ldots, arg_n\).

```plaintext
op buyticket : -> OpName [ctor] .
ops startTime aClient : -> Arg [ctor] .
```
OCL expressions in mOdCL: invariants

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

```plaintext
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.

```plaintext
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.

```plaintext
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
Validating with mOdCL

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
Validating with mOdCL

Static validation

- mOdCL offers an `eval` function to evaluate an OCL expression.
  
  \[
  \text{op eval : OclExp Configuration \to OclType .}
  \]

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

  \[
  \text{Maude}\rangle \quad \text{red in CINEMA-TEST : eval(seats-in-season, state).}
  \]

  \[
  \text{result : true}
  \]
Running example: sequence diagram
The mOdCL representation of the system behavior

- mOdCL expects that methods are invoked with a message
  \[
  \text{call}(<\text{method-name}>, <\text{addressee}>, <\text{argument-list}>)
  \]
  and upon their completion they send a return message of the form
  \[
  \text{return}(<\text{return-value}>)
  \]

- 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.
  \[
  \text{resume}(m', <\text{return-value}>)
  \]
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.

\[
\langle \text{Ctx} : \text{Context} \mid \text{op} : \text{M}, \text{self} : \text{Id}, \text{args} : \text{Args} \rangle
\]

- A return\(<\text{return-value}>\) message will be replaced by a resume message of the form

\[
\text{resume}(\text{<return-value>})
\]

- 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.

\[
\text{call}(\text{goCinema}, \text{Self}, (\text{arg}(\text{cinema}, \text{Cn}), \text{arg}(\text{startTime}, \text{St})))
\]

- First rule. A call to buyTicket.

\[
\text{rl [GO-CINEMA-1] :}
\]

\[
\begin{align*}
< & \text{ctx} : \text{Context} \mid \text{op} : \text{goCinema}, \text{self} : \text{Self}, \\
& \quad \text{args} : (\text{arg}(\text{cinema}, \text{Cn}), \text{arg}(\text{startTime}, \text{St})) \\
< & \text{Self} : \text{Client} \mid \text{cinema} : \text{Set}\{C, \text{LC}\}, \text{Atts1} > \\
< & \text{C} : \text{Cinema} \mid \text{name} : \text{Cn}, \text{session} : \text{Set}\{S, \text{LS}\}, \text{Atts2} > \\
< & \text{S} : \text{Session} \mid \text{startTime} : \text{St}, \text{Atts3} > \\
\Rightarrow & < \text{Self} : \text{Client} \mid \text{cinema} : \text{Set}\{C, \text{LC}\}, \text{Atts1} > \\
& < \text{C} : \text{Cinema} \mid \text{name} : \text{Cn}, \text{session} : \text{Set}\{S, \text{LS}\}, \text{Atts2} > \\
& < \text{S} : \text{Session} \mid \text{startTime} : \text{St}, \text{Atts3} > \\
& < \text{ctx} : \text{Context} \mid \text{op} : \text{goCinema}, \text{self} : \text{Self}, \\
& \quad \text{args} : (\text{arg}(\text{cinema}, \text{Cn}), \text{arg}(\text{startTime}, \text{St})) \\
& \quad \text{call}(\text{buyTicket}, \text{C}, (\text{arg}(\text{startTime}, \text{St}), \text{arg}(\text{client}, \text{Self}))) .
\end{align*}
\]

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

\[
\text{rl [GO-CINEMA-2-FAIL] :}
\]

\[
\begin{align*}
\text{resume}(\text{buyTicket}, \text{null}) \\
\Rightarrow \quad \text{return(false)} .
\end{align*}
\]
An example: the buyTicket method

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

```plaintext
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 ...)
```
The mOdCL validator architecture

Architecture: the mOdCL evaluator

- **Sorts definition**
  
<table>
<thead>
<tr>
<th>Subsort Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>subsort Int Float String Bool Oid &lt; BasicType .</code></td>
</tr>
<tr>
<td><code>subsort Set Bag OrderedSet Sequence &lt; Collection .</code></td>
</tr>
<tr>
<td><code>subsort BasicType Collection &lt; OclType .</code></td>
</tr>
</tbody>
</table>

- **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.

\[
\begin{align*}
\text{ceq} \; \text{metaOCLRewrite}(M, T) &= \text{metaOCLRewriteAux}(M, T, \text{iterator}(M)) \\
&\quad \text{if } I := \text{metaReduce}(M, \text{'inv.OclExp}) \\
&\quad \land \; \text{metaReduce}(M, \text{'eval}[I, T]) = \text{'true.Bool} . \\
\text{ceq} \; \text{metaOCLRewriteAux}(M, T, C) &= T \text{ if not hasNext}(C) . \\
\text{ceq} \; \text{metaOCLRewriteAux}(M, T, C) \\
&= \text{if } T' :: \text{Term} \\
&\quad \text{then (if } L == \text{'CALL} \\
&\quad \quad \text{then checkCall}(M, T') \\
&\quad \text{else (if } L == \text{'RETURN} \\
&\quad \quad \text{then checkReturn}(M, T, T') \\
&\quad \quad \text{else metaOCLRewriteAux}(M, T', \text{iterator}(M)) \\
&\quad \text{fi) \\
&\quad \text{else metaOCLRewriteAux}(M, T, \text{next}(C)) \\
&\quad \text{fi} \\
&\quad \text{if } L := \text{getLabel}(C) \land T' := \text{metaXapply}(M, T, L) \\
&\text{[owise].}
\end{align*}
\]
The checkCall and checkReturn functions

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

```plaintext
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
  - ...