Expressions: Standard library (2)

- **Finite quantification**
  - $c \rightarrow \text{forall}(i : T \mid e) = c \rightarrow \text{iterate}(i : T; \ a : \text{Boolean} = \text{true} \mid a \ \text{and} \ e)$
  - $c \rightarrow \text{exists}(i : T \mid e) = c \rightarrow \text{iterate}(i : T; \ a : \text{Boolean} = \text{false} \mid a \ \text{or} \ e)$

- **Selecting values**
  - $c \rightarrow \text{any}(i : T \mid e)$  
    some element of $c$ satisfying $e$
  - $c \rightarrow \text{select}(i : T \mid e)$  
    all elements of $c$ satisfying $e$

- **Collecting values**
  - $c \rightarrow \text{collect}(i : T \mid e)$
    collection of elements with $e$ applied to each element of $c$
  - $c.p$
    collection of elements $v.p$ for each $v$ in $c$
    (short-hand for collect)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C \text{.allInstances()}$</td>
<td>all current instances of classifier $C$</td>
</tr>
<tr>
<td>$o \text{.oclIsInState}(s)$</td>
<td>is $o$ currently in state machine state $s$?</td>
</tr>
<tr>
<td>$v \text{.oclIsUndefined()}$</td>
<td>is value $v$ null or invalid?</td>
</tr>
<tr>
<td>$v \text{.oclIsInvalid()}$</td>
<td>is value $v$ invalid?</td>
</tr>
</tbody>
</table>
Evaluation

• Strict evaluation with some exceptions
  • (if (1/0 = 0) then 0.0 else 0.0 endif).oclIsInvalid() = true
  • (1/0).oclIsInvalid() = true

• Short-cut evaluation for and, or, implies
  • (1/0 = 0.0) and false = false
  • true or (1/0 = 0.0) = true
  • false implies (1/0 = 0.0) = true
  • (1/0 = 0.0) implies true = true
  • if (0 = 0) then 0.0 else 1/0 endif = 0.0

• In general, OCL expressions are evaluated over a system state.

E.g., represented by an object diagram
Connection to UML

- Import of classifiers and enumerations as types
- Properties accessible in OCL
  - Attributes
    - $p\cdot\text{milesCard}$ (with $p : \text{Passenger}$)
  - Association ends
    - $p\cdot\text{flight}, p\cdot\text{booking}, p\cdot\text{booking}[\text{flight}]$
  - \{ query \} operations
  - Access to stereotypes via \(v\cdot\text{stereotype}\)

- Representation of multiplicities

\[
\begin{array}{c|c}
 a[1] : T & a : T \\
 a[0..1] : T & a : \text{Set}(T) \text{ or } T \\
 a[m..n] : T & a : \text{Set}(T) \\
 a[*] : T \{ \text{unordered} \} & a : \text{Set}(T) \\
 a[*] : T \{ \text{ordered} \} & a : \text{OrderedSet}(T) \\
 a[*] : T \{ \text{bag} \} & a : \text{Bag}(T)
\end{array}
\]
Invariants

context classifier

context Passenger
inv: ma.statusMiles > 10000 implies
status = Status::Albatros

Notational variants

context classifier

context Passenger
inv statusLimit: self.ma.statusMiles > 10000 implies
self.status = Status::Albatros

context p : Passenger
inv statusLimit: p.ma.statusMiles > 10000 implies
p.status = Status::Albatros
Semantics of invariants

- Restriction of valid states of classifier instances
  - when observed from outside

- Invariants (as all constraints) are inherited via generalizations
  - but how they are combined is not predefined

- One possibility: Combination of several invariants by **conjunction**

\[
\text{context } C \\
\text{inv: } I_1 \\
\text{context } C \\
\text{inv: } I_2
\]

\[
\rightarrow
\]

\[
\text{context } C \\
\text{inv: } I_1 \text{ and } I_2
\]
Pre-/post-conditions

- In UML models, pre- and post-conditions are defined separately
  - not necessarily as pairs
  - «precondition» and «postcondition» as constraint stereotypes

```plaintext
context Passenger::consumeMiles(b : Booking) : Boolean
pre: ma->notEmpty() and
    ma.flightMiles >= b.flight.miles

context Passenger::consumeMiles(b : Booking) : Boolean
post: ma.flightMiles = ma.flightMiles@pre-b.flight.miles and
    result = true
```

- Some constructs only available in post-conditions
  - values at pre-condition time
  - result of operation call
  - whether an object has been newly created
  - messages sent
Semantics of pre-/post-conditions

- **Standard interpretation**
  - A pre-/post-condition pair \((P, Q)\) defines a relation \(R\) on system states such that \((\sigma, \sigma') \in R\), if \(\sigma \models P\) and \((\sigma, \sigma') \models Q\).
    - system state \(\sigma\) on operation invocation
    - system state \(\sigma'\) on operation termination (\(Q\) may refer to \(\sigma\) by \(\@pre\)).
  - Thus \((P, Q)\) equivalent to \((\text{true}, P\@pre \text{ and } Q)\).

- **Meyer’s contract view**
  - A pre-/post-condition pair \((P, Q)\) induces benefits and obligations.
  - benefits and obligations differ for implementer and user

<table>
<thead>
<tr>
<th></th>
<th>obligation</th>
<th>benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>user</strong></td>
<td>satisfy (P)</td>
<td>(Q) established</td>
</tr>
<tr>
<td><strong>implementer</strong></td>
<td>if (P) satisfied, establish (Q)</td>
<td>(P) established</td>
</tr>
</tbody>
</table>
Combining pre-/post-conditions

- Standard interpretation
  - joining pre- and post-conditions conjunctively

  \[
  \text{context } C::\text{op}() \\
  \text{pre: } P_1 \quad \text{post: } Q_1 \\
  \text{context } C::\text{op}() \\
  \text{pre: } P_2 \quad \text{post: } Q_2 \\
  \]

- Alternative interpretation
  - case distinction (like in protocol state machines)
  - only useful for pre-/post-condition pairs

  \[
  \text{context } C::\text{op}() \\
  \text{pre: } P_1 \quad \text{post: } Q_1 \\
  \text{context } C::\text{op}() \\
  \text{pre: } P_2 \quad \text{post: } Q_2 \\
  \]

  \[
  \text{context } C::\text{op}() \\
  \text{pre: } P_1 \text{ or } P_2 \quad \text{post: } (P_1@\text{pre} \implies Q_1) \text{ and } (P_2@\text{pre} \implies Q_2) \\
  \]
context Subject::hasChanged()  
post: observer^update(self)  
in calls on hasChanged,  
some update message with argument  
self will have been sent to observer

context Subject::hasChanged()  
post: observer^update(? : Subject)  
the actual argument  
does not matter

context Subject::hasChanged()  
post: let messages : Set(OclMessage) =  
observer^^update(? : Subject)  
in messages->notEmpty() and  
messages->forall(m |  
result of message call m.result().oclIsUndefined() and  
whether it has finished m.hasReturned() and  
its actual parameter value m.subject = self)  
all those messages
Initial values and derived properties

- **Initial values**
  - fix the initial value of a property of a classifier

```uml
package Booking
  context Passenger::status
    init: Status::Swallow
endpackage
```

- **{ derived } properties**
  - define how the value of a property is derived from other information

```uml
context Passenger::currentFlights : Sequence(Flight)
derive: self->collect(booking)
  ->select(date = today()) . flight->asSequence()
```
Query bodies and model features

- Bodies of `{ query }` operations
  - define the value returned by a query operation
  - can be combined with a precondition

  ```
  context TravelHandling :: delay() : Minutes
  body: tsh.delay->sum()
  ```

- Definition of additional model features
  - defined for the context classifier

  ```
  context TravelStageHandling
  def: isEarly() : Boolean = self.delay < 0

  context TravelHandling
  def: someEarly() : Boolean = tsh->exists(isEarly())
  ```
Wrap up

• Formal language for specifying
  • invariants
    \[ \text{context } C \text{ inv: } I \]
  • pre-/post-conditions
    \[ \text{context } C::\text{op () : T} \]
    \[ \text{pre: } P \text{ post: } Q \]
  • query operation bodies
    \[ \text{context } C::\text{op () : T body: } e \]
  • initial values
    \[ \text{context } C::p : T \text{ init: } e \]
  • derived attributes
    \[ \text{context } C::p : T \text{ derive: } e \]
  • modelling attributes and operations
    \[ \text{context } C \text{ def: } p : T = e \]

• Side-effect free
• Typed language

• OCL specifications provide
  • verification conditions
  • assertions for implementations
Meta-Object Facility 2
OMG’s standards UML and MOF

Modelling with UML, with semantics
Relations between UML 2 and MOF 2

- MOF meta-meta-model of UML 2
- MOF is (based on) the core of UML 2
- UML 2 is a drawing tool of the MOF 2
- Definition synchronization
Meta-Object Facility (MOF)

- A *meta-data management framework*
- A language to be used for defining languages
  - i.e., it is an OMG-standard meta-modelling language.
  - The UML metamodel is defined in MOF.
- **MOF 2.0 shares a common core with UML 2.0**
  - Simpler rules for modelling metadata
  - Easier to map from/to MOF
  - Broader tool support for metamodeling (i.e., any UML 2.0 tool can be used)

- MOF has **evolved** through several versions
  - MOF 1.x is the one most widely supported by tools
  - MOF 2.0 is the current standard, and it has been substantially influenced by UML 2.0
  - MOF 2.0 is also critical in supporting transformations, e.g., QVT and Model-to-text

[http://www.omg.org/spec/MOF/2.0](http://www.omg.org/spec/MOF/2.0)
MOF 2.0 Structure

- MOF is separated into **Essential MOF (EMOF)** and **Complete MOF (CMOF)**
- EMOF corresponds to facilities found in OOP and XML.
  - Easy to map EMOF models to JMI, XMI, etc.
- CMOF is what is used to specify metamodels for languages such as UML 2.
  - It is built from EMOF and the core constructs of UML 2.
  - Both EMOF and CMOF are based on variants of UML 2.
MOF 2.0 Relationships (1)
MOF 2.0 Relationships (2)
EMOF Types — merged from UML Infrastructure
EMOF Classes — merged from UML Infrastructure (2)
EMOF Data Types — merged from UML Infrastructure
EMOF Packages — merged from UML Core:Basic