Object Constraint Language (OCL)

- Annke Kleppe’s company and OCL centre: http://www.klasse.nl/ocl.
OCL Overview

- A formal notation for users of UML to add more precision to their specifications.
- All of the power of logic and discrete mathematics is available.
- However, only ASCII characters (rather than conventional mathematical notation) should be used.

- Like an OOPL, an OCL expression involves operators operating on objects.
- However, the result of a complete expression must always be a Boolean.
- The objects can be instances of the OCL Collection class, of which Set and Sequence are two subclasses.
Setting Context:

```text
model my_model -- OCLE specific
    package my_package
        context Class1
            inv ... (invariant or whatever)
            context Class1::operation1(v1: Integer)
                inv ...
        endpackage
    endmodel
```

The context can be set to any model element (package, class, interface, component) or some sub-elements such as operation, attribute and in some cases (e.g. interaction diagram) an instance.
Invariants:

class Class1
  inv attr1 > 100

class Class2
  inv secondInvariant: attr2 < 10

Pre and Post Conditions:

class Class1::method1(v1: Integer): Integer
  pre valueIsLargeEnough: v1 >= 100
  post: attr1 >= attr1@pre + 100 and result > v/10

Query Definitions:

class Class1::query1(v: Integer): Integer
  body: v + 100 + attr1
OCL Quick Reference

Definitions:

*Introduce a Query*
context Class1
def: getTotal() : Integer = items.value->sum()

*Define an Initial Value*
context Class1::attr1
init: 100

*Define a Derived Attribute*
context Class1::attr2
derive: attr1/100

*Introduce a New Derived Value*
context Class1
def: attr2 : Integer = attr1/100
Basic Types:

- **Integer, Real**: =, <>, <=, >=, +, -, *, /, x.mod(y), x.div(y) (div is integer division)
- **String**: s.concat(t), s.size(), s.toLower(), s.toUpper(), s.subString(start, end) (indexing is “1-based”).
  - OCLE also offers
    - ocontains(subStr : String):Boolean,
    - opos(subStr : String):Integer and
    - osplit(separators : String):Sequence(String).
- **Boolean**: and, or, not, xor, =, <>, implies, “if b1 then ... else ..., endif”.

Collection Types

- **Set** – no duplicates, no order.
- **Bag** – duplicates, no order.
- **OrderedSet** – no duplicates, ordered.
- **Sequence** – duplicates, ordered.

In navigation expressions (e.g. item1.subItems.value),

- navigation through a single 1:m relationship (e.g. item.subItem) returns a set,
- navigation through more than one such relationship (e.g. item1.subItems.value) returns a bag,
- navigation through an {ordered} relationship results in a sequence.

These types are subtypes of an abstract base called **Collection**.

Collection type objects can be converted between types using built-in cast-like operators such as seq1->asSet() (see below).
Collection Manipulation Operations:

- `=` / `<>` Are the collections identical (not identical).
- `->` Return the value of the set difference of the arguments (Set and OrderedSet only).
- `append(obj), prepend(obj)` Append/prepend obj to an ordered collection.
- `asBag(), asSet(), asOrderedSet(), asSequence()` Type conversion operations (available to/from all collection types).
- `at(idx)` Return object at index of ordered collection.
- `count(obj)` Number of times that obj appears in a collection.
- `excludes(obj), includes(obj)` Does count(obj) = 0 / > 0 hold for all items in collection?
- `excludesAll(coll), includesAll(coll)` Does count(obj) = 0 / > 0 hold for all items in collection coll?
- `first(), last()` The first/last item in the ordered collection.
- `isEmpty(), notEmpty` Is collection's size() = 0 / > 0?
- `indexOf(obj)` The index value of the (first) occurrence of an object in an ordered collection.
- `insertAt(idx, obj)` Value of the collection with the specified object inserted at the index of the ordered collection.
- `intersection(coll)` Value of the intersection of the unordered collection and the unordered collection coll.

Operations are applied to collections using the "-" operator (e.g. items->isEmpty(), where "items" is a collection).

Collection Expressions

Type{initialiser} Set{1, 2, 3} Bag{'one', 'two', 'three', 'two'} OrderedSet{true, false} Sequence{1..30} Special case for integers Set{Set{1,2}, Set{2,3}} = Set{1,2,3} - flattening
Loop Operations:

**collect(expr)** Returns a *bag* containing the value of the expression for each of the items in the collection (e.g. `items->collect(value)`). A simpler synonym for this operation is the period (".") operator (e.g. `items.value`).

```plaintext
collection->collect( v : Type | expression-with-v )
collection->collect( v | expression-with-v )
collection->collect( expression )
```

**forAll(expr)** Does expression expr hold for all items in the collection?

**exists(expr)** Does expression expr hold for any items in the collection?

**reject(expr)** Returns the sub-collection of items in a collection for which expression expr does not hold.

```plaintext
collection->reject( v : Type | boolean-expression-with-v ) =
collection->select( v : Type | not (boolean-expression-with-v) )
```

**select(expr)** Returns the sub-collection of items in a collection for which expression expr holds.

```plaintext
set1->select(attr1 > 10)
set1->select(i | i.attr1 > 10)
```

These two examples are equivalent.

"i" is an "iterator" variable and can be thought of as being set to each of the elements of set1 in turn.

**one(expr)** Returns the value of the expression `coll->select(expr)->size()==1`

Let Expressions

Used to define temporary attributes or operations to allow them to be used in constraints to simplify the constraint or avoid repetition.

```plaintext
context Class1
inv abc:
    let val1:Boolean = att1 > 100 and att2 < 25
    let largeEnough(v :Integer):Boolean = v > 100
    in (val1 and attr3.mod(5)=0) or
       (val1 and attr4/5 > 10) or
       (largeEnough(val3))
```
Messages in OCL

Calling operations and sending signals:

context Subject::hasChanged()
post: observer^update(12, 14)

To specify that communication has taken place, the hasSent (^) operator is used. Update() is either an Operation that is defined in the class of observer, or it is a Signal specified in the UML model.

Accessing result values:

context Person::giveSalary(amount : Integer)
post: let message : OclMessage = company^getMoney(amount) in
message.hasReturned() -- getMoney was sent and returned
and
message.result() = true -- the getMoney call returned true

The standard operation result() of OclMessage contains the return value of the called operation.
# Examples

## Banking

<table>
<thead>
<tr>
<th>Account</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>balance</td>
<td>Real = 0</td>
</tr>
<tr>
<td>deposit()</td>
<td></td>
</tr>
<tr>
<td>withdraw()</td>
<td></td>
</tr>
<tr>
<td>getBalance()</td>
<td>Real</td>
</tr>
</tbody>
</table>

context Account::withdraw(amount : Real)
pre: amount <= balance
post: balance = balance@pre - amount

context Account::getBalance() : Real
post: result = balance
public void testWithdraw() {
    Account account = new Account();
    account.deposit(500);
    float balanceAtPre = account.getBalance();
    float amount = 250;
    account.withdraw(amount);
    assertTrue(account.getBalance() == balanceAtPre + amount);
}
Banking

Account account = new Account();
Customer customer = new Customer();
customer.accounts = new Account[] {account};
account.holder = customer;

customer.accounts.balance = 0  not ok
customer.accounts->select(id = 2324).balance = 0  ok
Examples

Person

<<enumeration>>

<table>
<thead>
<tr>
<th>Sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>MALE = 1</td>
</tr>
<tr>
<td>FEMALE = 2</td>
</tr>
</tbody>
</table>

Person

- sex : Sex
- marry(p : Person)

context Person::marry(p : Person)
pre cannot_marry_self: not (p = self)
pre not_same_sex: not (p.sex = self.sex)
-- neither person can be married already
pre not_already_married: self.spouse->size() = 0 and p.spouse->size() = 0
post: self.spouse = p and p.spouse = self
### Class sex

```java

```static final int MALE = 1;
static final int FEMALE = 2;
```

### Class Person

```java
class Person {
    public int sex;
    public Person spouse;
    public void marry(Person p) {
        assert p != this;
        assert p.sex != this.sex;
        assert this.spouse = null && p.spouse = null;
        this.spouse = p;
        p.spouse = this;
    }
}
```

### Defensive programming style

#### Example

```java
class Person {
    public int sex;
    public Person spouse;

    public void marry(Person p) throws ArgumentException {
        if(p == this) {
            throw new ArgumentException("cannot marry self");
        }
        if(p.sex == this.sex) {
            throw new ArgumentException("spouse is same sex");
        }
        if((p.spouse != null || this.spouse != null) {
            throw new ArgumentException("already married");
        }
        this.spouse = p;
        p.spouse = this;
    }
}
```
Examples

**Person**

- **context** Person **inv:**
  - self.age > 0

- **context** Person **inv:**
  - self.employer->size() < 3

- **context** Person::income(d : Date) : Integer
  - **post:** result = 5000

- **context** Person::income(d: Date) : Integer
  - **post:** result = age * 1000

- **context** Person **inv:**
  - self.wife->notEmpty() implies (self.wife.gender = Gender::female and self.wife.age >= 18)
Examples

**Person**

```plaintext
context Person::getCurrentSpouse() : Person
pre: self.isMarried = true
body: self.mariages->select( m | m.ended = false ).spouse
```

```plaintext
context Person::income : Integer
init: parents.income->sum() * 1% -- pocket allowance
derive: if underAge
    then parents.income->sum() * 1% -- pocket allowance
    else job.salary -- income from regular job
endif
```

```plaintext
context Person inv:
let income : Integer = self.job.salary->sum()
in if isUnemployed then
    income < 100
else
    income >= 100
endif
```

```plaintext
context Company::hireEmployee(p : Person)
post: employees = employees@pre->including(p) and
    stockprice() = stockprice@pre() + 10
```

```plaintext
context Company inv:
self.employee->select(age > 50)->notEmpty()
context Company inv:
self.employee->select(p | p.age > 50)->notEmpty()
context Company inv:
self.employee.select(p : Person | p.age > 50)->notEmpty()
```
## Examples

### Person

**context Company inv:**
```
self.employee->reject( isMarried )->isEmpty()
self.employee->collect( birthDate )->asSet()
```

**context Company inv:**
```
system.employee->collect(birthdate) =
    self.employee.birthdate
```

**context Company inv:**
```
self.employee->collect(birthdate) =
    self.employee.birthdate
```

**context Company inv:**
```
self.employee->forAll( age <= 65 )
self.employee->forAll( p | p.age <= 65 )
self.employee->forAll( p : Person | p.age <= 65 )
```

**context Company inv:**
```
self.employee->forAll( e1, e2 : Person | e1 <> e2 implies e1.forename <> e2.forename)
```

**context Company inv:**
```
self.employee->exists( forename = 'Jack' )
```

**context Company inv:**
```
collection->collect(x : T | x.property) -- is identical to:
collection->iterate(x : T; acc : T2 = Bag{} |
    acc->including(x.property))
```

**context Company::hireEmployee(p : Person)**
```
**post:**
employees = employees@pre->including(p) and
stockprice() = stockprice@pre() + 10
```

**context Company inv:**
```
self.employee->select(age > 50)->notEmpty()
```

**context Company inv:**
```
self.employee->select(p | p.age > 50)->notEmpty()
```

**context Company inv:**
```
self.employee.select(p : Person | p.age > 50)->notEmpty()
```
Examples

States


object.oclInState(On)
object.oclInState(Off)
object.oclInstate(Off::Standby)
object.oclInState(Off::NoPower)
Examples

context Train inv atLeastOneWagon:
  self.wagon->size() >= 1

context Wagon inv belongToTheSameTrain:
  self.succ->notEmpty() implies self.succ->forAll(w | w.train = self.train)

context Train inv sameNumberOfWagons:
  Train.allInstances->forAll(t1 | (self.wagon->size() = t1.wagon->size()))

context Wagon inv notInCyclicWay:
  (Wagon.allInstances)->forAll(w2 |
   self <> w2 implies not ((self.succ)->includes(w2) and (w2.succ)->includes(self)))
Train

Equivalent to
context C inv: b->includesAll(d)
OclModelElementType represents the types of elements that are ModelElements in the UML metamodel. It is used to be able to refer to states and classifiers in e.g. oclInState(...) and oclIsKindOf(...)
Type Conformance

conformsTo(c : Classifier) : Boolean  
true, if the self Classifier conforms to the argument c.

class context Classifier
    inv Reflexivity: self.conformsTo(self)

context Classifier
inv Transitivity: Classifier.allInstances()->forAll(x, y|  
    (self.conformsTo(x) and x.conformsTo(y)) implies self.conformsTo(y))

context Classifier
Inv Anti-symmetry: Classifier.allInstances()->forAll(t1, t2 |  
    (t1.conformsTo(t2) and t2.conformsTo(t1)) implies t1 = t2)
**Definition A.1 (Classes)**
The set of classes is a finite set of names \( \text{Class} \subseteq \mathcal{N} \).

**Definition A.2 (Attributes)**
Let \( t \in T \) be a type. The attributes of a class \( c \in \text{Class} \) are defined as a set \( \text{Attr}_c \) of signatures \( a : t_c \to t \) where the attribute name \( a \) is an element of \( \mathcal{N} \), and \( t_c \in T \) is the type of class \( c \). \( \Box \)

**Definition A.3 (Operations)**
Let \( t \) and \( t_1, \ldots, t_n \) be types in \( T \). Operations of a class \( c \in \text{Class} \) with type \( t_c \in T \) are defined by a set \( \text{Op}_c \) of signatures \( \omega : t_c \times t_1 \times \cdots \times t_n \to t \) with operation symbols \( \omega \) being elements of \( \mathcal{N} \). \( \Box \)

**Definition A.4 (Associations)**
The set of associations is given by

1. a finite set of names \( \text{Assoc} \subseteq \mathcal{N} \),

2. a function associates:
   \[
   \begin{cases} 
   \text{Assoc} \to \text{Class}^+ \\
   as \mapsto (c_1, \ldots, c_n) \text{ with } (n \geq 2)
   \end{cases}
   \]
**Definition A.5 (Role Names)**

Let $as \in \text{Assoc}$ be an association with $\text{associates}(as) = \langle c_1, \ldots, c_n \rangle$. Role names for an association are defined by a function

\[
\text{roles} : \left\{ \begin{array}{c} \text{Assoc} \rightarrow \mathcal{N}^+ \\ as \mapsto \langle r_1, \ldots, r_n \rangle \text{ with } (n \geq 2) \end{array} \right. 
\]

where all role names must be distinct, i.e.,

\[
\forall i, j \in \{1, \ldots, n\} : i \neq j \implies r_i \neq r_j .
\]

\[
\text{participating} : \left\{ \begin{array}{c} \text{CLASS} \rightarrow \mathcal{P}(\text{ASSOC}) \\ c \mapsto \{as \mid as \in \text{ASSOC} \land \text{associates}(as) = \langle c_1, \ldots, c_n \rangle \land \exists i \in \{1, \ldots, n\} : c_i = c \} 
\end{array} \right. 
\]

\[
\text{navends} : \left\{ \begin{array}{c} \text{CLASS} \times \text{ASSOC} \rightarrow \mathcal{P}(\mathcal{N}) \\ (c, as) \mapsto \{r \mid \text{associates}(as) = \langle c_1, \ldots, c_n \rangle \land \text{roles}(as) = \langle r_1, \ldots, r_n \rangle \land \exists i, j \in \{1, \ldots, n\} : (i \neq j \land c_i = c \lland r_j = r) \} 
\end{array} \right. 
\]

\[
\text{navends}(c) : \left\{ \begin{array}{c} \text{CLASS} \rightarrow \mathcal{P}(\mathcal{N}) \\ c \mapsto \bigcup_{as \in \text{participating}(c)} \text{navends}(c, as) 
\end{array} \right. 
\]

**Definition A.6 (Multiplicities)**

Let $as \in \text{ASSOC}$ be an association with $\text{associates}(as) = \langle c_1, \ldots, c_n \rangle$. The function $\text{multiplicities}(as) = \langle M_1, \ldots, M_n \rangle$ assigns each class $c_i$ participating in the association a non-empty set $M_i \subseteq \mathbb{N}_0$ with $M_i \neq \{0\}$ for all $1 \leq i \leq n$. \hfill \Box
Formal Semantics

GENERALIZATION

Definition A.7 (Generalization hierarchy)
A generalization hierarchy $\prec$ is a partial order on the set of classes $\text{CLASS}$.

$$\forall c_1, c_2 \in \text{CLASS} : c_1 \prec c_2 \implies I(c_1) \subseteq I(c_2).$$

Definition A.8 (Full descriptor of a class)
The full descriptor of a class $c \in \text{CLASS}$ is a structure $\text{FD}_c = (\text{ATT}_c^*, \text{OP}_c^*, \text{navends}^*(c))$ containing all attributes, user-defined operations, and navigable role names defined for the class and all of its parents.

$$\text{parents} : \begin{cases} \text{CLASS} \to \mathcal{P}(\text{CLASS}) \\ c \mapsto \{c' \mid c' \in \text{CLASS} \land c \prec c' \} \end{cases}$$

$$\text{ATT}_c^* = \text{ATT}_c \cup \bigcup_{c' \in \text{parents}(c)} \text{ATT}_{c'}$$

$$\text{OP}_c^* = \text{OP}_c \cup \bigcup_{c' \in \text{parents}(c)} \text{OP}_{c'}$$

$$\text{navends}^*(c) = \text{navends}(c) \cup \bigcup_{c' \in \text{parents}(c)} \text{navends}(c')$$
**Formal Semantics**

**Definition A.9 (Syntax of object models)**

The syntax of an object model is a structure

\[ M = (\text{CLASS}, \text{ATT}_c, \text{OP}_c, \text{Assoc}, \text{associates}, \text{roles}, \text{multiplicities}, \prec) \]

where

1. \text{CLASS} is a set of classes (Definition A.1).
2. \text{ATT}_c \text{ is a set of operation signatures for functions mapping an object of class } c \text{ to an associated attribute value (Definition A.2).}
3. \text{OP}_c \text{ is a set of signatures for user-defined operations of a class } c \text{ (Definition A.3).}
4. \text{Assoc} \text{ is a set of association names (Definition A.4).}
   
   (a) \text{associates} \text{ is a function mapping each association name to a list of participating classes (Definition A.4).}
   
   (b) \text{roles} \text{ is a function assigning each end of an association a role name (Definition A.5).}
   
   (c) \text{multiplicities} \text{ is a function assigning each end of an association a multiplicity specification (Definition A.6).}

5. \prec \text{ is a partial order on CLASS reflecting the generalization hierarchy of classes (Definitions A.7 and A.8).}
Interpretation of Object Models

DEFINITION A.10 (OBJECT IDENTIFIERS)

i. The set of object identifiers of a class $c \in \text{CLASS}$ is defined by an infinite set $\text{oid}(c) = \{c_1, c_2, \ldots\}$. 

ii. The domain of a class $c \in \text{CLASS}$ is defined as $I_{\text{CLASS}}(c) = \bigcup \{\text{oid}(c') \mid c' \in \text{CLASS} \land c' \preceq c\}$.

DEFINITION A.11 (LINKS)

Each association $as \in \text{ASSOC}$ with associates$(as) = (c_1, \ldots, c_n)$ is interpreted as the Cartesian product of the sets of object identifiers of the participating classes: $I_{\text{ASSOC}}(as) = I_{\text{CLASS}}(c_1) \times \cdots \times I_{\text{CLASS}}(c_n)$. A link denoting a connection between objects is an element $l_{as} \in I_{\text{ASSOC}}(as)$.

DEFINITION A.12 (SYSTEM STATE)

A system state for a model $M$ is a structure $\sigma(M) = (\sigma_{\text{CLASS}}, \sigma_{\text{ATT}}, \sigma_{\text{ASSOC}})$.

i. The finite sets $\sigma_{\text{CLASS}}(c)$ contain all objects of a class $c \in \text{CLASS}$ existing in the system state: $\sigma_{\text{CLASS}}(c) \subseteq \text{oid}(c)$.

ii. Functions $\sigma_{\text{ATT}}$ assign attribute values to each object: $\sigma_{\text{ATT}}(a) : \sigma_{\text{CLASS}}(c) \rightarrow I(t)$ for each $a : t_c \rightarrow t \in \text{ATT}_c^*$.

iii. The finite sets $\sigma_{\text{ASSOC}}$ contain links connecting objects. For each $as \in \text{ASSOC}$: $\sigma_{\text{ASSOC}}(as) \subseteq I_{\text{ASSOC}}(as)$. A link set must satisfy all multiplicity specifications defined for an association (the function $\pi_i(l)$ projects the $i$th component of a tuple or list $l$, whereas the function $\bar{\pi}_i(l)$ projects all but the $i$th component):

$$\forall i \in \{1, \ldots, n\}, \forall l \in \sigma_{\text{ASSOC}}(as) :$$

$$|\{l' \mid l' \in \sigma_{\text{ASSOC}}(as) \land (\bar{\pi}_i(l') = \bar{\pi}_i(l))\}| \in \pi_i(\text{multiplicities}(as))$$