

Object Constraint Language (OCL)

- Annke Kleppe's company and OCL centre: <http://www.klasse.nl/ocl>.
- *The Object Constraint Language*, 2nd ed., Jos Warmer and Anneke Kleppe, Addison Wesley, 2003.
- OCLE, a freely available OCL tool: <http://ici.cs.ubbcluj.ro/ocle>.
- OCL 1.5 Specification: <http://www.omg.org/cgi-bin/apps/doc?formal/03-03-13>.
- OCL 2.0 Specification: <http://www.omg.org/cgi-bin/apps/doc?ptc/2003-10-14>.

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 OOP, 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.

OCL Quick Reference

UML 1.5 Standard

[http://www.eoinwoods.info/doc/ocl_quick_reference.pdf]

Setting Context:

```
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.

OCL Quick Reference

UML 1.5 Standard

Invariants:

```
context Class1
    inv attr1 > 100
context Class2
    inv secondInvariant: attr2 < 10
```

Can have as many “inv” statements as required, optionally named, and the resulting invariant is their conjunction (“i1 and i2 and ...”).

Pre and Post Conditions:

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

The “@pre” notation refers to the “before” state (VDM’s “hook” notation) and “result” is a reserved word for the result of the operation (if it has one).

Query Definitions:

```
context Class1::query1(v: Integer) : Integer
body: v + 100 + attr1
```

Queries don’t change state and so pre/post-condition form isn’t used to define them. Instead, they use a single expression in a “body” statement.

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
```

OCL Quick Reference

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
 - contains(subStr : String): Boolean,
 - pos(subStr : String): Integer and
 - split(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.
- 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).

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**.

OCL Quick Reference

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

Collection Manipulation Operations:

= / <>

- 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 ? / Is count(obj) > 0 ?

excludesAll(coll), includesAll(coll) Does count(obj) = 0 / count(obj) > 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 / size() > 0?

size() Number of items in the collection.

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).

OCL Quick Reference

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).

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

```
collection.propertyname =  
collection->collect(propertyname)
```

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.

```
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.

```
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

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))
```

OCL Quick Reference

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:

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

```
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
```

Examples

Banking

Account
balance : Real = 0
deposit(amount : Real) withdraw(amount : Real) getBalance() : Real

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

context Account::getBalance() : Real
post : result = balance

Examples

Account
balance : Real = 0
deposit(amount : Real) withdraw(amount : Real) getBalance() : Real

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

context Account::getBalance() : Real
post : result = balance
```

```
class Account
{
    private float balance = 0;

    public void withdraw(float amount) {
        assert amount <= balance;

        balance = balance - amount;
    }

    public void deposit(float amount) {
        balance = balance + amount;
    }

    public float getBalance() {
        return balance;
    }
}
```

```
public void testWithdraw()
{
    Account account = new Account();

    account.deposit(500);

    float balanceAtPre = account.getBalance();

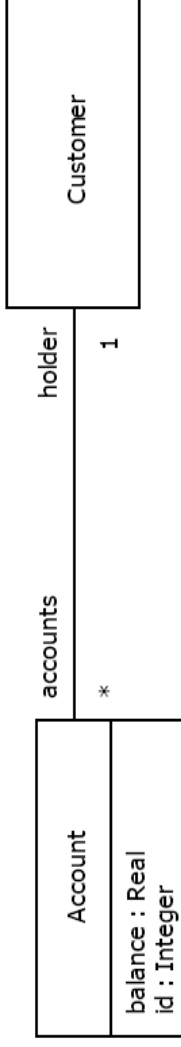
    float amount = 250;

    account.withdraw(amount);

    assertTrue(account.getBalance() == balanceAtPre . amount);
}
```

Examples

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
```

```
class Account
{
    public double balance;
    public int id;
}

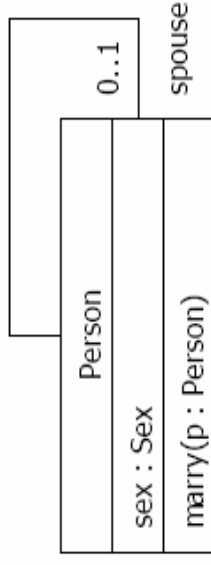
class Customer
{
    Account[] accounts;

    public Account SelectAccount(int id)
    {
        Account selected = null;
        for(int i = 0; i < accounts.length; i++)
        {
            Account account = accounts[i];
            if(account.id == id)
            {
                selected = account;
                break;
            }
        }
        return selected;
    }
}
```

Examples

Person

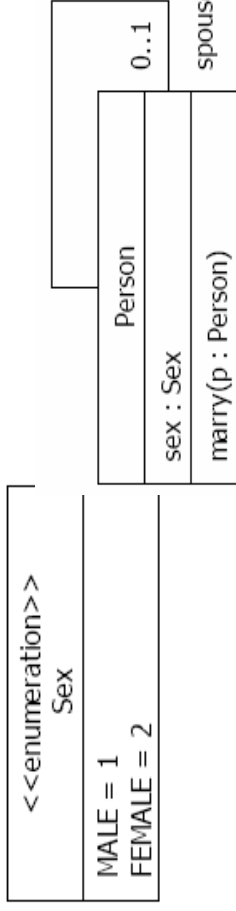
<<enumeration>> Sex
MALE = 1
FEMALE = 2



```
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
```

Person

Examples



```
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

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

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

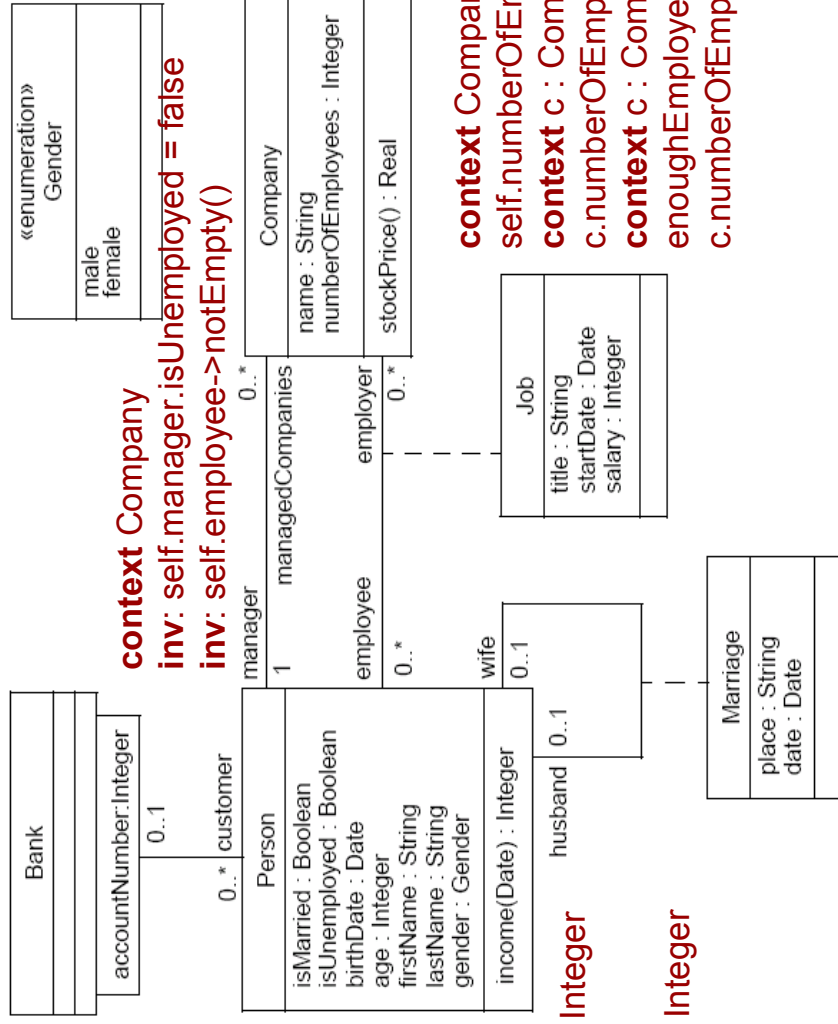
```
class Person
{
  public int sex;
  public Person spouse;

  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;
  }
}
```

Person

<http://www.omg.org/docs/ptc/03-10-14.pdf>

Examples



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 Company
inv: self.manager.isUnemployed = false
inv: self.employee->notEmpty()

context Company **inv**:
self.numberOfEmployees > 50
context c : Company **inv**:
c.numberOfEmployees > 50
context c : Company **inv**
enoughEmployees:
c.numberOfEmployees > 50

context Person **inv**:

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

Examples

Person

<http://www.omg.org/docs/ptc/03-10-14.pdf>

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

```
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
```

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

```
context Person
def: income : Integer = self.job.salary->sum()
def: nickname : String = 'Little Red Rooster'
def: hasTitle(t : String) : Boolean = self.job->exists(title = t)
```

```
context Person
inv: self.oclIsTypeOf( Person ) -- is true
inv: self.oclIsTypeOf( Company ) -- is false
```

```
context Person inv:
Person.allInstances()->forAll(p1, p2 |
    p1 <> p2 implies p1.name <> p2.name)
```

```
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()
16
```

Examples

Person

<http://www.omg.org/docs/ptc/03-10-14.pdf>

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

```
self.employee->collect(birthdate) =  
self.employee.birthdate
```

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

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

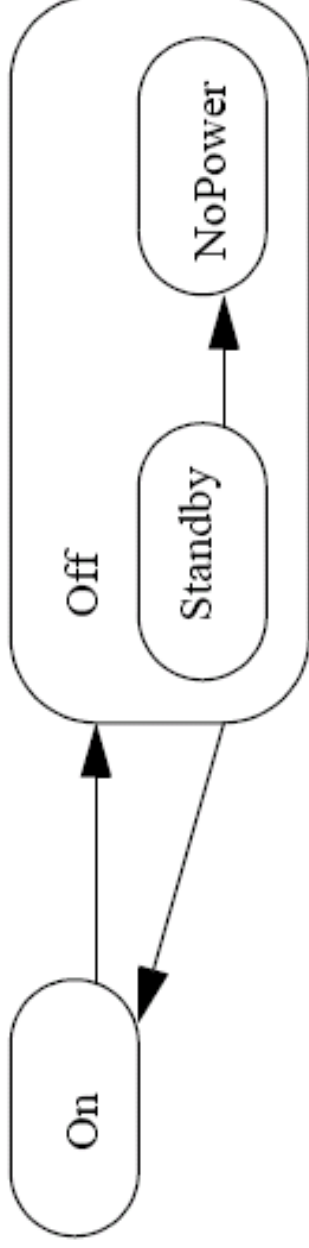
```
context Company::hireEmployee(p : Person)  
post: employees = employees@pre->including(p) and  
stockprice() = stockprice@pre() + 10
```

```
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()  
17
```

Examples

States

<http://www.omg.org/docs/ptc/03-10-14.pdf>

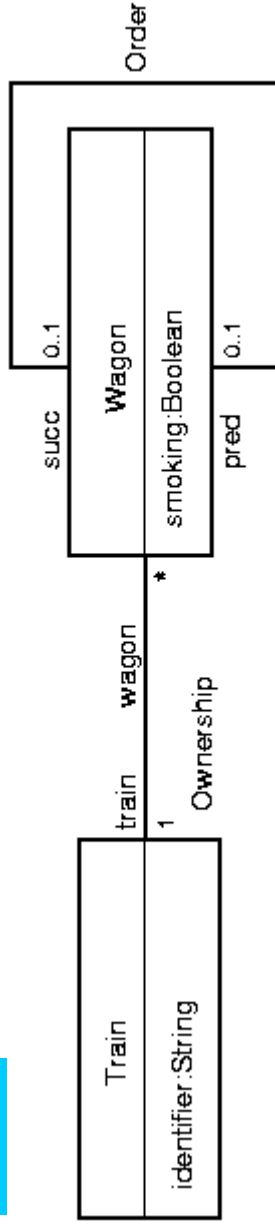


```
object.ocInState(On)
object.ocInState(Off)
object.ocInState(Off::Standby)
object.ocInState(Off::NoPower)
```

Examples

<http://maude.sip.ucm.es/ftp/ocl/tutorial.html>

Train



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

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

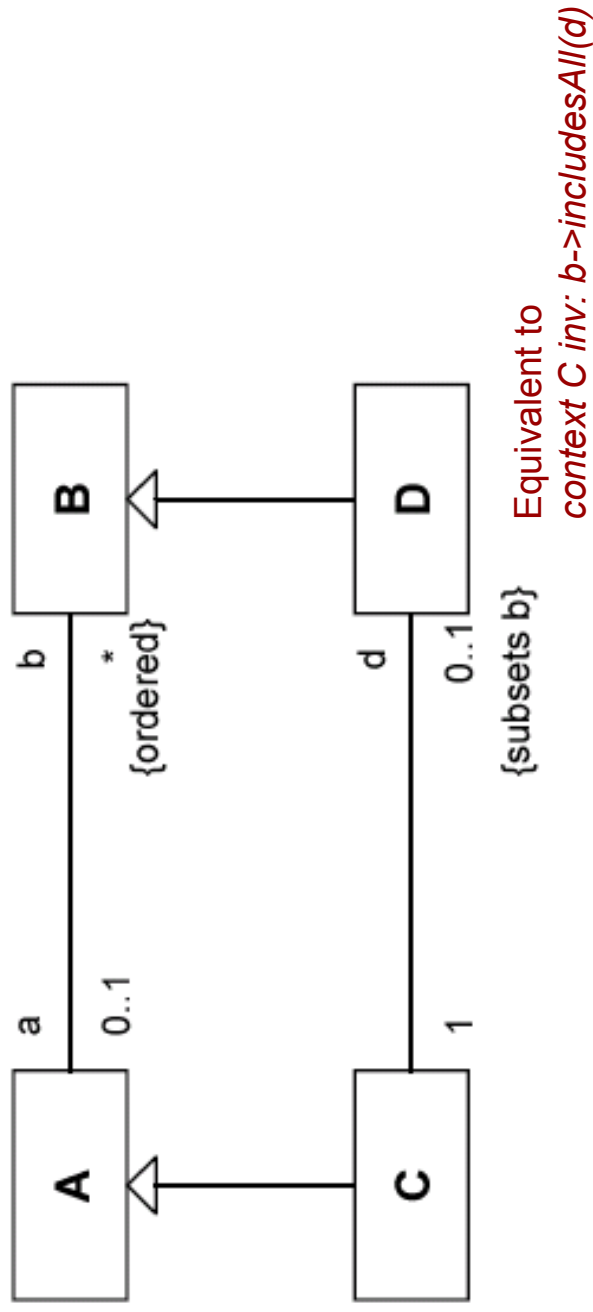
context Train inv sameNumberOfWagons:
Train.allInstances->forAll(t1 | (self.wagon->size() = t1.wagon->size()))

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

Examples

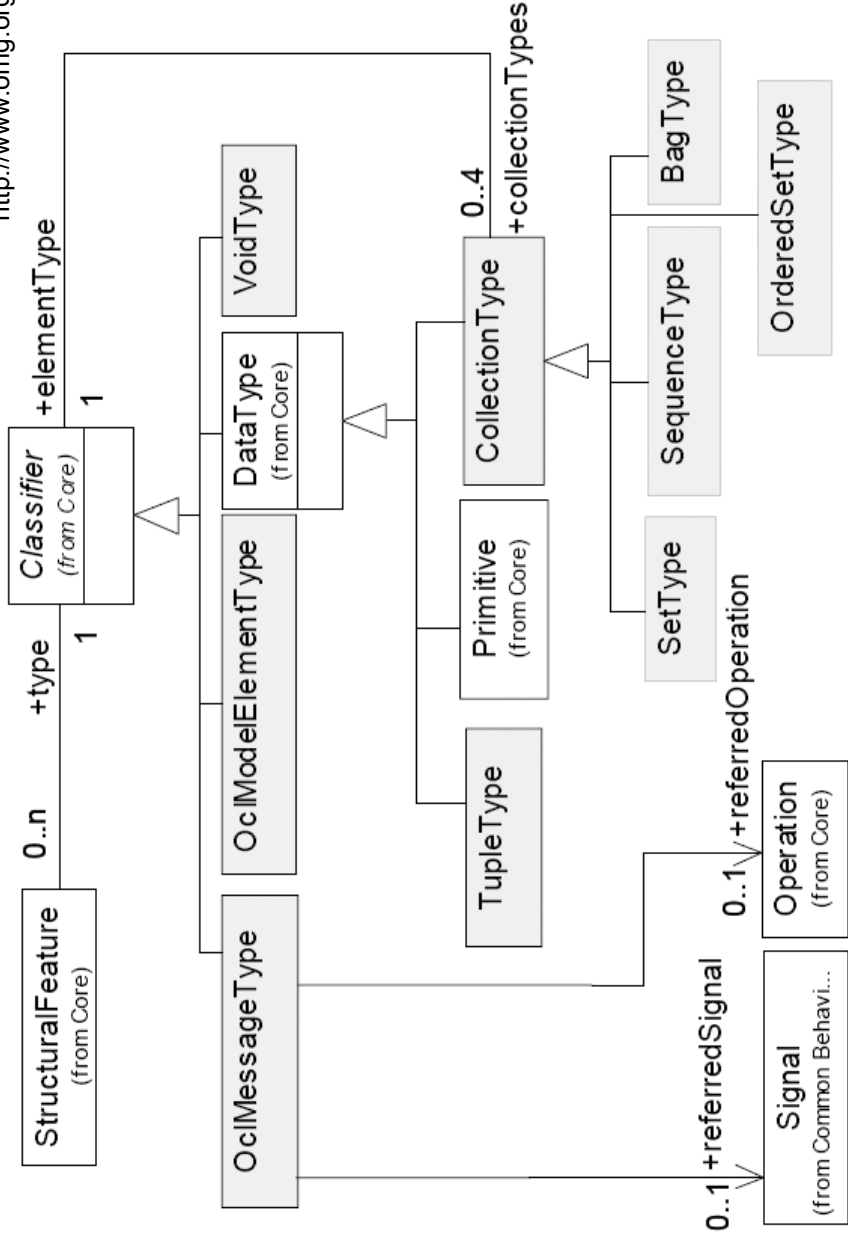
[OML-UML 2.0 Superstructure 04-10-02.pdf]

Train



Abstract Syntax Kernel Metamodel for OCL Types

UML 2.0 OCL Specification
<http://www.omg.org/docs/ptc/03-10-14.pdf>



OcIModelElementType 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*.

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)

Formal Semantics

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 ATT_c of signatures $a : t_c \rightarrow t$ where the attribute name a is an element of \mathcal{N} , and $t_c \in T$ is the type of class c . \square

DEFINITION A.3 (OPERATIONS)

Let t and t_1, \dots, 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 OP_c of signatures $\omega : t_c \times t_1 \times \dots \times t_n \rightarrow t$ with operation symbols ω being elements of \mathcal{N} . \square

DEFINITION A.4 (ASSOCIATIONS)

The set of associations is given by

- i. a finite set of names $\text{ASSOC} \subseteq \mathcal{N}$,
- ii. a function associates :
$$\begin{cases} \text{ASSOC} \rightarrow \text{CLASS}^+ \\ as \mapsto \langle c_1, \dots, c_n \rangle \text{ with } (n \geq 2) \end{cases}$$

Formal Semantics

DEFINITION A.5 (ROLE NAMES)

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

$$\text{roles} : \begin{cases} \text{ASSOC} \rightarrow \mathcal{N}^+ \\ as \mapsto \langle r_1, \dots, r_n \rangle \text{ with } (n \geq 2) \end{cases}$$

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

$$\forall i, j \in \{1, \dots, n\} : i \neq j \implies r_i \neq r_j .$$

$$\begin{aligned} & \text{participating} : \begin{cases} \text{CLASS} \rightarrow \mathcal{P}(\text{ASSOC}) \\ c \mapsto \{as \mid as \in \text{ASSOC} \wedge \text{associates}(as) = \langle c_1, \dots, c_n \rangle \\ \quad \wedge \exists i \in \{1, \dots, n\} : c_i = c\} \end{cases} \\ & \text{navends} : \begin{cases} \text{CLASS} \times \text{ASSOC} \rightarrow \mathcal{P}(\mathcal{N}) \\ (c, as) \mapsto \{r \mid \text{associates}(as) = \langle c_1, \dots, c_n \rangle \\ \quad \wedge \text{roles}(as) = \langle r_1, \dots, r_n \rangle \\ \quad \wedge \exists i, j \in \{1, \dots, n\} : (i \neq j \wedge c_i = c \wedge r_j = r)\} \end{cases} \\ & \text{navends}(c) : \begin{cases} \text{CLASS} \rightarrow \mathcal{P}(\mathcal{N}) \\ c \mapsto \bigcup_{as \in \text{participating}(c)} \text{navends}(c, as) \end{cases} \end{aligned}$$

DEFINITION A.6 (MULTIPLICITIES)

Let $as \in \text{ASSOC}$ be an association with $\text{associates}(as) = \langle c_1, \dots, c_n \rangle$. The function $\text{multiplicities}(as) = \langle M_1, \dots, M_n \rangle$ 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 $1 \leq i \leq n$. \square

Formal Semantics

GENERALIZATION

DEFINITION A.7 (GENERALIZATION HIERARCHY)

A generalization hierarchy \prec is a partial order on the set of classes 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. \square

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

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

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

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

Formal Semantics

DEFINITION A.9 (SYNTAX OF OBJECT MODELS)

The syntax of an object model is a structure

$$\mathcal{M} = (\text{CLASS}, \text{ATT}_c, \text{OP}_c, \text{ASSOC}, \text{associates}, \text{roles}, \text{multiplicities}, \prec)$$

where

- i. CLASS is a set of classes (Definition A.1).
- ii. ATT_c is a set of operation signatures for functions mapping an object of class c to an associated attribute value (Definition A.2).
- iii. OP_c is a set of signatures for user-defined operations of a class c (Definition A.3).
- iv. Assoc is a set of association names (Definition A.4).
 - (a) associates is a function mapping each association name to a list of participating classes (Definition A.4).
 - (b) roles is a function assigning each end of an association a role name (Definition A.5).
 - (c) multiplicities is a function assigning each end of an association a multiplicity specification (Definition A.6).
- v. \prec is a partial order on CLASS reflecting the generalization hierarchy of classes (Definitions A.7 and A.8).

Interpretation of Object Models (only some here)

DEFINITION A.10 (OBJECT IDENTIFIERS)

- i. The set of object identifiers of a class $c \in \text{CLASS}$ is defined by an infinite set $\text{oid}(c) = \{c_1, c_2, \dots\}$.
- ii. The domain of a class $c \in \text{CLASS}$ is defined as $I_{\text{CLASS}}(c) = \bigcup \{\text{oid}(c') \mid c' \in \text{CLASS} \wedge c' \preceq c\}$.

DEFINITION A.11 (LINKS)

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

DEFINITION A.12 (SYSTEM STATE)

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

- i. The finite sets $\sigma_{\text{CLASS}}(c)$ contain all objects of a class $c \in \text{CLASS}$ existing in the system state: $\sigma_{\text{CLASS}}(c) \subset \text{oid}(c)$.
- ii. Functions σ_{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 σ_{ASSOC} contain links connecting objects. For each $as \in \text{ASSOC}$: $\sigma_{\text{ASSOC}}(as) \subset I_{\text{ASSOC}}(as)$. 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):

$$\begin{aligned} & \forall i \in \{1, \dots, n\}, \forall l \in \sigma_{\text{ASSOC}}(as) : \\ & |\{l' \mid l' \in \sigma_{\text{ASSOC}}(as) \wedge (\bar{\pi}_i(l') = \bar{\pi}_i(l))\}| \in \pi_i(\text{multiplicities}(as)) \end{aligned}$$