Object Constraint Language
A definitive guide

Jordi Cabot – AtlanMod / EMN / INRIA /LINA/…
Martin Gogolla – University of Bremen
(with the invaluable help of slides of the BiG group - TUWien)

http://modeling-languages.com
@softmodeling

 1

Before we start…

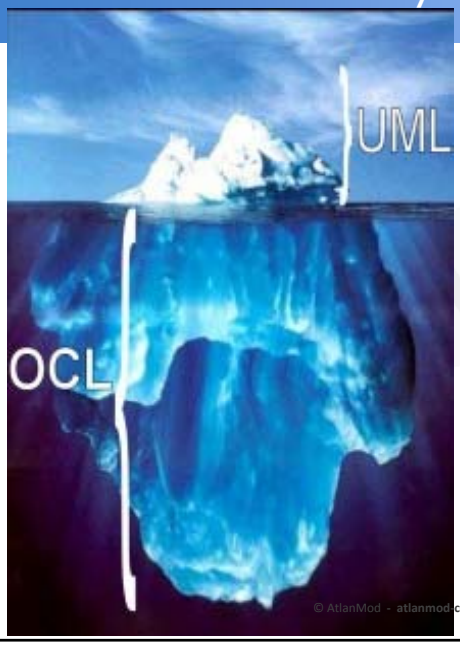
- OCL is not sexy and neither are these slides
- Interrupt and participate at any time – No place to hide

 2

WHY OCL?

3

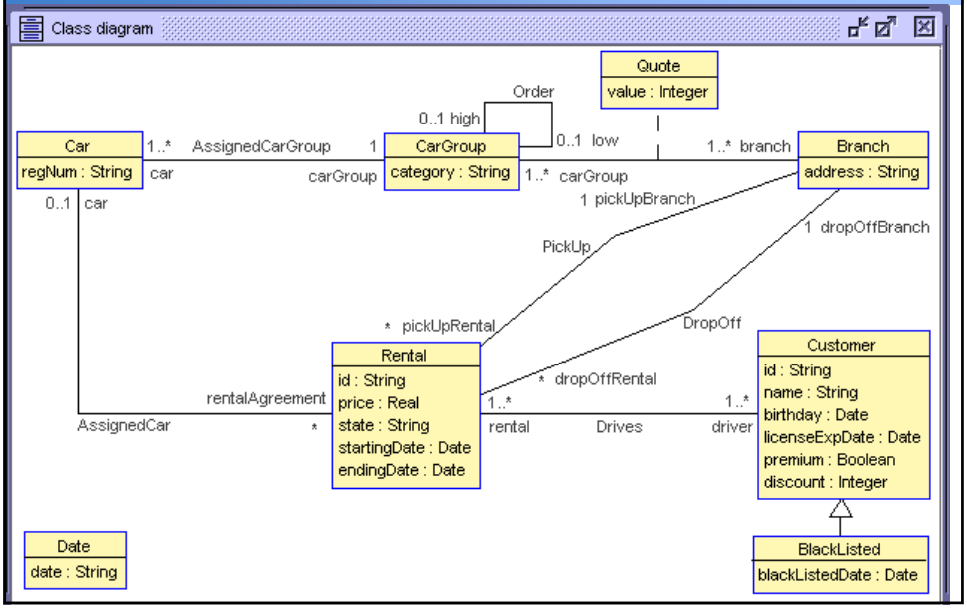
Why OCL



- *No serious use of UML without OCL!!! (limited expressiveness of ALL (graphical) notations*

4

What does the model NOT say



OCL is everywhere

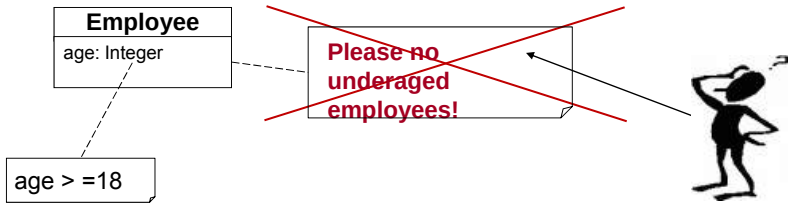
- Where can we find OCL?
  - Constraints
  - Well-formedness rules
  - Derivation rules
  - Pre/posconditions
  - Guards
  - Model to model transformations
  - Model to text transformations
  - …

Why OCL

- *Natural language is clearly not enough*

Motivation

- Example 1

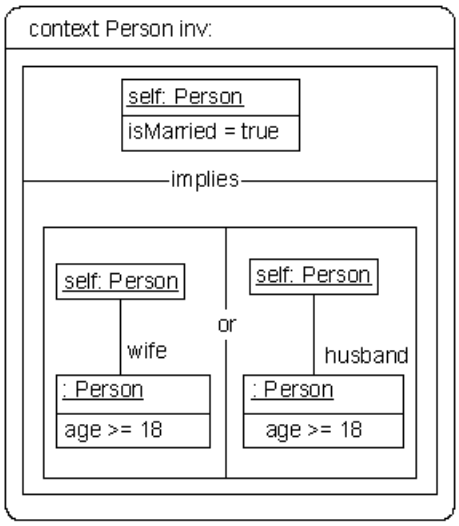


Why OCL - Alternatives to OCL

- NL
- FOL
- Alloy
- SQL
- DSLs for specific business rule patterns
- Visual OCL
- SBVR
- … others?

- You may not like it but right now there is nothing better than OCL!! (and so tightly integreated already with the other existing modeling standards)

Why OCL – Visual OCL



VS

Context Person inv:
self.isMarried implies
self.wife>=18 or
self.husband>=18

Why OCL - SBVR

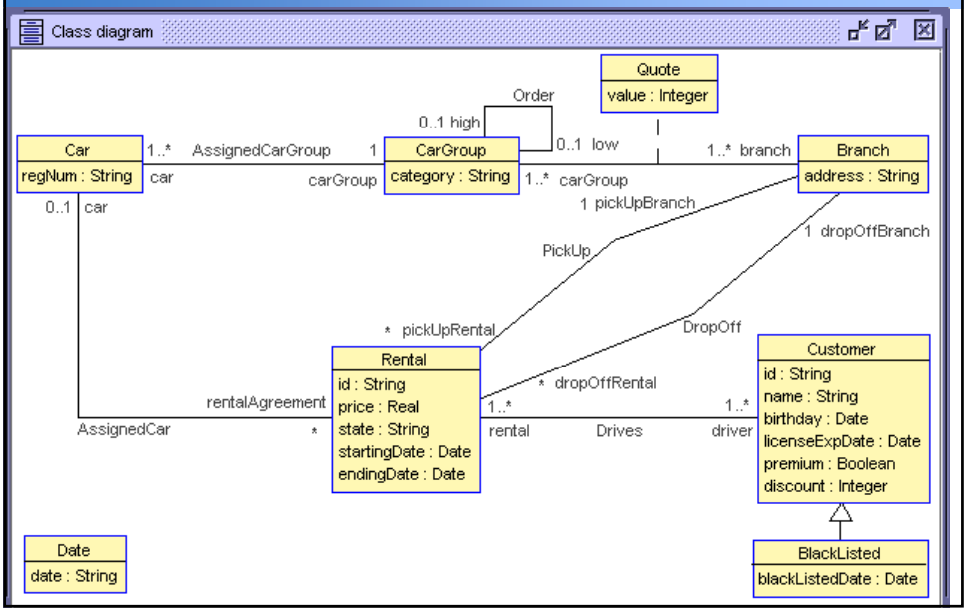
- 'each product belongs to exactly 1 family' is defined as:
- *The rule is a proposition meant by a necessity formulation.*
- *. .The necessity formulation embeds a universal quantification, u1.*
- *(3) . . . The universal quantification, u1, introduces a first variable, v1.*
- *(4) . . . . The variable v1 ranges over the concept 'product'.*
- *(5) . . . The universal quantification, u1, scopes over an exactly one quantification, e1.*
- *(6) . . . . The exactly one quantification, e1, introduces a second variable, v2.*
- *(7) . . . . . The variable, v2, ranges over the concept 'family'.*
- *(8) . . . . The exactly one quantification, e1, scopes over an atomic formulation, a1.*
- *(9) . . . . . The atomic formulation, a1, is based on the fact type 'product belongs to family'.*
- *(10) . . . . . The atomic formulation, a1, has the first role binding.*
- *(11) . . . . . The first role binding is of the role 'product'.*
- *(12) . . . . . The first role binding binds to the variable v1.*
- *(13) . . . . . The atomic formulation has the second role binding.*
- *(14) . . . . . The second role binding is of the role 'family.*
- *(15) . . . . . The second role binding binds to the variable v2.*

THE LANGUAGE

OCL: Basic Concepts

- OCL is a rich language that offers predefined mechanisms for:
  - Retrieving the values of an object
  - Navigating through a set of related objects,
  - Iterating over collections of objects (e.g., forAll, exists, select)
- OCL includes a predefined standard library: set of types + operations on them
  - Primitive types: Integer, Real, Boolean and String
  - Collection types: Set, Bag, OrderedSet and Sequence
  - Examples of operations: and, or, not (Boolean), +, −, , >, < (Real and Integer), union, size, includes, count and sum (Set).

Some first examples



Some first examples

```
context Quote inv OverZero: self.value > 0

context BlackListed inv NoRentalsBlackListed:
self.rental->forAll(r | r.startDate < self.blackListedDate)

context Customer::premium: boolean init: false
```

Some first examples

```
context Customer::discount: integer
derive:
if not self.premium then
   if self.rental.car.carGroup->
        select(c|c.category='high')->size()>=5
   then 15
   else 0 endif
else 30 endif
```

Some first examples

```
context Car::mostPopular(): boolean
body:
Car::allInstances()->forAll(c1|c1<>self implies
c1.rentalAgreement->size() <=
                    self.rentalAgreement->size())
```

Some first examples

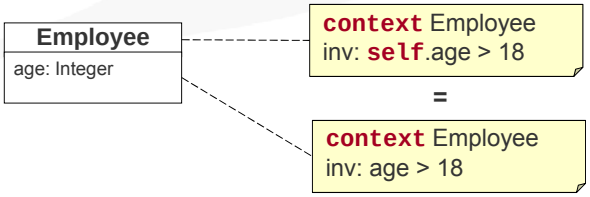
```
context Rental::newRental(id:Integer, price:Real,
   startingDate:Date, endingDate:Date, customer:Customer,
   carRegNum:String, pickupBranch: Branch, dropOffBranch:
   Branch)

pre: customer.licenseExpDate>endingDate

post: Rental.allInstances->one(r |
r.oclIsNew() and r.oclIsTypeOf(Rental) and
r.endingDate=endingDate and r.startingDate=startingDate and
r.driver=customer and r.pickupBranch=pickupBranch and
r.dropOffBranch=dropOffBranch and
r.car=Car.allInstances()->any(c | c.regNum=carRegNum))
```

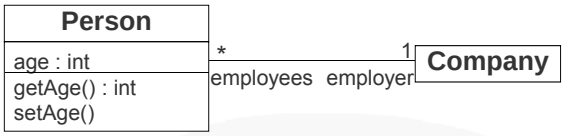
Language syntax: context

- A context has to be assigned to each OCL-statement
  - **Starting address –** which model element is the OCL-statement defined for
  - Specifies which model elements can be reached using path expressions
- The context is specified by the keyword **`context`** followed by the name of the model element (mostly class names)
- The keyword **`self`** specifies the current instance, which will be evaluated by the invariant (context instance).
  - **`self`** can be omitted if the context instance is unique



Language syntax: navigation

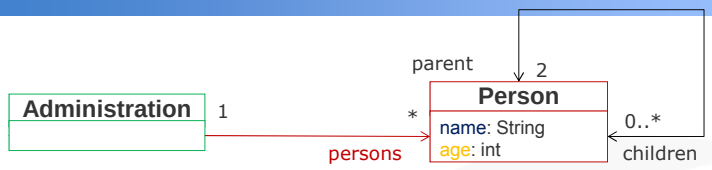
- Context instance
  - `context` **`Person`**
- Accessing attributes
  - `self.age`
- Calling operations
  - Operations must not have **side effects**
  - `self.getAge()`
- Association ends
  - Navigate to the opposite association end using role names
    `self.employer` – Return value is of type ***Company***
  - Navigation often results into a set of objects – Example
    `context` **`Company`**
    `self.employees` – Return value is of type **Set (Person)**

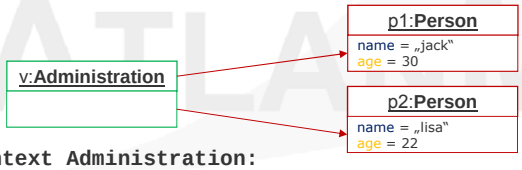


- ***What does c.employees.employer return?***

20

Example 1: Navigation (1)



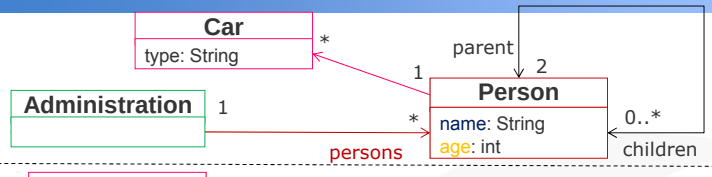


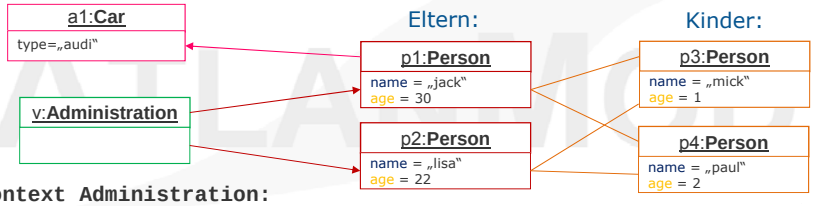

```
context Administration:
```

- `self.persons` → {Person p1, Person p2}
- `self.persons.name` → {jack, lisa}
- `self.persons.age` → {30, 22}

21

Example 1: Navigation (2)



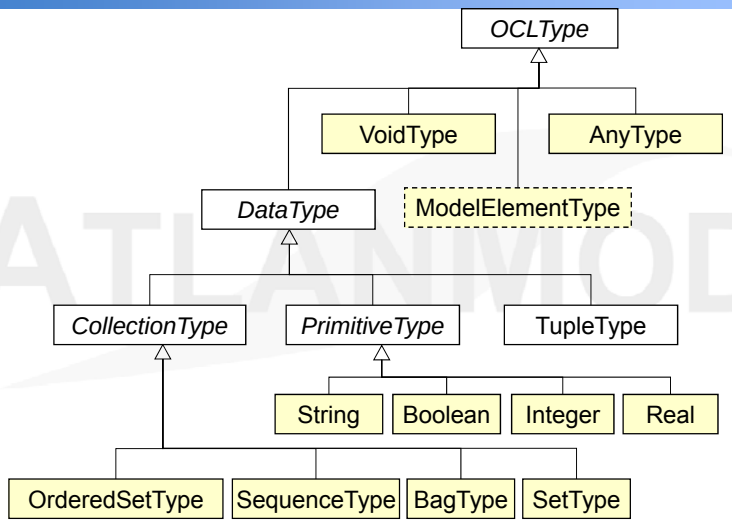



```
context Administration:
```

- `self.persons.children` → {p3, p4,p3, p4} // returned as flattened bag
- `self.persons.children.parent` → {p1, p2,p1, p2, ...}
- `self.persons.car.type` → {"audi"}

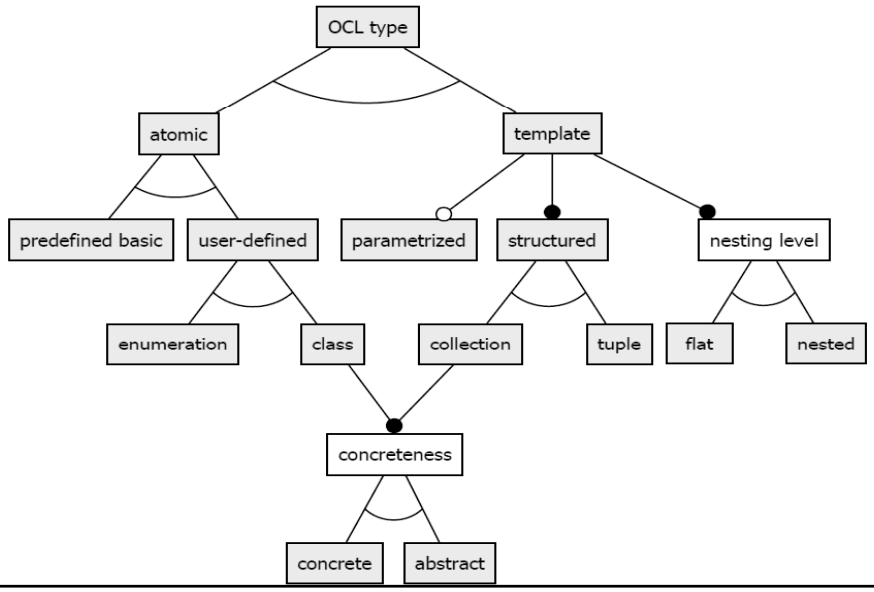
22

OCL Types (MM View)

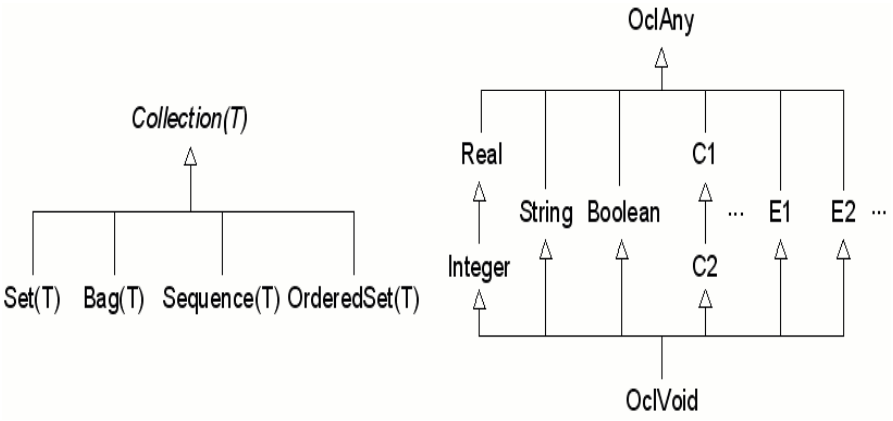


23

OCL Types (A feature model view)



Operations for primitive/predefined types



- ***Note that an OrderedSet is not a Set!***

25

Operations for primitive/predefined types

- **Predefined** simple types
  - Integer {Z}
  - Real {R}
  - Boolean {true, false}
  - String {ASCII, Unicode}

Simple type	Predefined operations
Integer	*, +, -, /, abs(), …
Real	*, +, -, /, floor(), …
Boolean	and, or, xor, not, implies
String	concat(), size(), substring(), …

26

Operations for OclAny

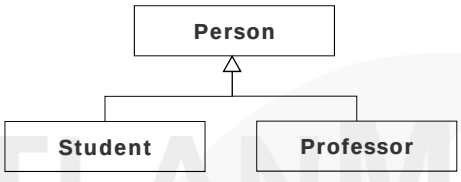
- *OclAny* - **Supertype** of all primitive types
  - **Operations** are **inherited** by all other types.
- **Operations** of*OclAny* (extract)
  - Receiving object is denoted by *obj*

Operation	Explanation of result
=(obj2:OclAny):Boolean	True, if *obj2* and *obj* reference the same object
oclIsTypeOf(type:OclType):Boolean	
oclIsKindOf(type:OclType): Boolean	
oclAsType(type:Ocltype): Type	The result is *obj* of type *type*, or *undefined*, if the current type of *obj* is not *type* or a direct or indirect subtype of it (casting).

27

Operations for OclAny

- ***oclIsKindOf vs. oclIsTypeOf***



context **Person**
self.oclIsKindOf(Person) : *true*
self.oclIsTypeOf(Person) : *true*
self.oclIsKindOf(Student) : *false*
self.oclIsTypeOf(Student) : *false*

context **Student**
self.oclIsKindOf(Person) : *true*
self.oclIsTypeOf(Person) : *false*
self.oclIsKindOf(Student) : *true*
self.oclIsTypeOf(Student) : *true*
self.oclIsKindOf(Professor) : *false*
self.oclIsTypeOf(Professor) : *false*

- ***Can you guess the difference?***

28

OCLVoid

- OCLVoid includes the null value (and therefore all supertypes of OCLVoid)
- Examples

null:OCLVoid
ada.discount=null : Boolean
branch42.address=null : Boolean
1/0=null : Boolean
(1/0).oclIsUndefined=true : Boolean
42.oclIsUndefined=false : Boolean
Set{'800-275-2273','800-694-7466',null} : Set(String)

3-valued logic for OCL

b	not(b)
null	null
false	true
true	false

b1 or b2	b2 = null	b2 = false	b2 = true
b1 = null	null	null	true
b1 = false	null	false	true
b1 = true	true	true	true

b1 and b2	b2 = null	b2 = false	b2 = true
b1 = null	null	null	false
b1 = false	false	false	false
b1 = true	null	false	true

b1 xor b2	b2 = null	b2 = false	b2 = true
b1 = null	null	null	null
b1 = false	null	false	true
b1 = true	null	true	false

b1 implies b2	b2 = null	b2 = false	b2 = true
b1 = null	null	null	true
b1 = false	true	true	true
b1 = true	null	false	true

b1 = b2	b2 = null	b2 = false	b2 = true
b1 = null	true	false	false
b1 = false	false	true	false
b1 = true	false	false	true

b1 <> b2	b2 = null	b2 = false	b2 = true
b1 = null	null	false	true
b1 = false	true	false	true
b1 = true	true	true	false

Operations for collections

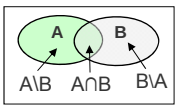
- Collection is an **abstract supertype** for all set types
  - Specification of the **mutual** operations
  - *Set, Bag, Sequence, OrderedSet* inherit these operations
- **Caution:** Operations with a return value of a set-valued type create a new collection (no side effects)
- Syntax: v -> op(…) – Example: {1, 2, 3} -> size()

Operation	Explanation of result
size():Integer	Number of elements in *coll*
includes(obj:OclAny):Boolean	True, if *obj* exists in *coll*
isEmpty:Boolean	True, if *coll* contains no elements
sum:T	Sum of all elements in *coll* Elements have to be of type Integer or Real

31

Operationen for Set/Bag

- *Set* and *Bag* define additional operations
  - Generally based on **theory of set concepts**
- **Operations** of **Set** (extract)
  - Receiving object is denoted by set



Operation	Explanation of result
union(set2:Set(T)):Set(T)	Union of *set* and *set2*
intersection(set2:Set(T)):Set(T)	Intersection of *set* and *set2*
difference(set2:Set(T)):Set()	Difference set; elements of *set*, which do not consist in *set2*
symmetricDifference(set2:Set(T)): Set(T)	Set of all elements, which are either in *set* or in *set2*, but do not exist in both sets at the same time

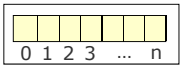
Operations of **Bag** (extract)

Operation	Explanation of result
union(bag2:Bag(T)):Bag(T)	Union of *bag* and *bag2*
intersection(bag2:Bag(T)): Bag(T)	Intersection of *bag* and *bag2*

32

Operations for OrderedSet/Sequence

- *OrderedSet* and *Sequences* define additional operations
  - Allow access or modification through an **Index**



- **Operations** of **OrderedSet** (extract)
  - Receiving object is denoted by *orderedSet*
- **Operations** of **Sequence**
  - Analogous to the operations of OrderedSet

Operation	Explanation of result
first:T	First element of *orderedSet*
last:T	Last element of *orderedSet*
at(i:Integer):T	Element on index i of *orderedSet*
subOrderedSet(lower:Integer, upper:Integer):OrderedSet(T)	Subset of *orderedSet,* all elements of *orderedSet* including the element on position *lower* and the element on position *upper*
insertAt(index:Integer,object:T) :OrderedSet(T)	Result is a copy of the *orderedSet*, including the element *object* at the position *index*

33

Iterator-based operations

- OCL defines operations for *Collections* using *Iterators*
  - **Projection** of new *Collections* out of existing ones
  - Compact **declarative specification** instead of imperative algorithms
- Predefined Operations
  - select(exp) : *Collection*
  - reject(exp) : *Collection*
  - collect(exp) : *Collection*
  - forAll(exp) : *Boolean*
  - exists(exp) : *Boolean*
  - isUnique(exp) : *Boolean*
- iterate(…) – Iterate over all elements of a *Collection*
  - Generic operation
  - Predefined operations are defined with iterate(…)

34

Select / Reject

- **Select** and **Reject** return subsets of collections
  - Iterate over the complete collection and collect elements
- Select
  - **Result**: Subset of collection, including elements where *booleanExpr* is **true**

```
collection -> select( v : Type | booleanExp(v) )
collection -> select( v | booleanExp(v) )
collection -> select( booleanExp )
```

- Reject
  - **Result**: Subset of collection, including elements where *booleanExpr* is **false**

```
collection-> reject(v : Type | booleanExp(v))
```

- ***Do we need a reject operation?***

35

Select operation

OCL

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

Java 5

```
List persons<Person> = new List();
for ( Iterator<Person> iter = comp.getEmployee();
iter.hasNext() ){
    Person p = iter.next();
    if ( p.age > 50 ){
        persons.add(p);
    }
}
```

36

Collect operation

- *Collect-Operation* returns a new collection from an existing one.
  - Result of *collect* always **Bag<T>.T** defines the type of the property to be collected

```
collection -> collect( v : Type | exp(v) )
collection -> collect( v | exp(v) )
collection -> collect( exp )
```

- Example
  - *self.employees -> collect(age)* – Return type: Bag(Integer)
- Short notation for collect: *self.employees.age*

37

Semantics of the Collect operation

OCL

```
context Company inv:
  self.employee -> collect(birthdate)  -> size() > 3
```

Java 5

```
List birthdate<Integer> = new List();
for ( Iterator<Person> iter = comp.getEmployee();
iter.hasNext() ){
    birthdate.add(iter.next().getBirthdate());
}
```

OCL

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

Set (without duplicates)

Bag (with duplicates)

38

Iterator-based operations

- **ForAll** checks, if all elements of a collection evaluate to true. **Example:**

```
collection -> forAll( v : Type | booleanExp(v) )
collection -> forAll( v | booleanExp(v) )
collection -> forAll( booleanExp )
```

- **Nesting** of forAll-Calls (*Cartesian Product*)

```
context Company inv:
self.employee->forAll (e1 | self.employee -> forAll (e2 |
        e1 <> e2 implies e1.id <> e2.id))
```

- ***Alternative***: Use of multiple iteratores

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

- **Exists** checks, if at least one element evaluates to true
  - Self.employees -> exists(e: Employee | e.isManager = true)

39

Iterate

- **Iterate** is the generic form of all iterator-based operations
- **Syntax**

  collection -> iterate(**elem** : Typ; **acc** : Typ =
  <initExp> | **exp(elem, acc)**)
  - Variable **elem** is a typed ***Iterator***
  - Variable **acc** is a typed ***Accumulator***
    - Gets assigned initial value initExp
  - **exp(elem, acc)** is a function to calculate **acc**

- collection -> collect(x : T | x.property)

  -- semantically equivalent to:

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

40

Iterate

OCL

```
collection -> iterate(x : T; acc : T2 = value | acc -> u(acc, x))
```

Java 5

```
iterate (coll : T, acc : T2 = value){
      acc=value;
      for( Iterator<T> iter =
coll.getElements(); iter.hasNext(); ){
           T elem = iter.next();
           acc = u(elem, acc);
      }
}
```

- Example
  - Set{1, 2, 3} -> iterate(i:Integer, a:Integer=0 | a+i)
  - Result: 6

41

Tuple types

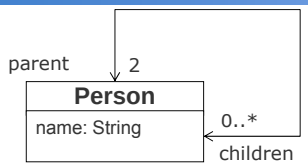
- Tuple types are structured created with the keyword Tuple and by additionally specifying part names and part values.
- Tuples and collections are orthogonal

Set{Tuple{name:'Ada',emails:Set{'ada@acm','ada@ibm'}},Tuple{name:'Bob',emails:Sequence{'bob@omg','bob@sun'}}} :

Set(Tuple(emails:Collection(String),name:String))

42

Example : Alternative, equivalent OCL-formulations

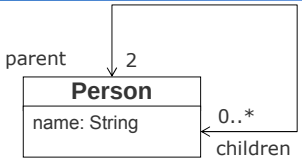


Constrain: A person is not its own child

- ***How many different formulations can you find?***

43

Example 5: Alternative, equivalent OCL-formulations (1)



Constrain: A person is not its own child

- `(self.children->select(k | k = self))->size() = 0`

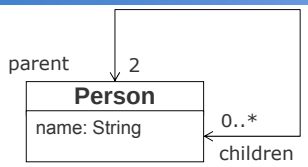
  The Number of children for each person „self", where the children are the person „self", have to be 0.

- `(self.children->select(k | k = self))->isEmpty()`

  The set of children for each person „self, where the children are the person „self", has to be empty.

44

Example 5: Alternative, equivalent OCL-formulations (2)



Constrain: A person is not its own child

- `not self.children->includes(self)`

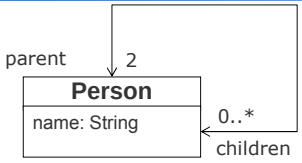
  It is not possible, that the set of children of each person „self" contains the person „self".

- `self.children->excludes(self)`

  `The set of children of each person „self" cannot contain „self".`

45

Example 5: Alternative, equivalent OCL-formulations (3)



Constrain: A person is not its own child

- `Set{self}->intersection(self.children)->isEmpty()`

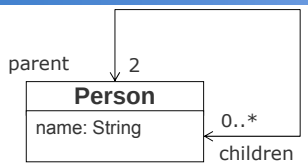
  The intersection between the one element set, which only includes one person „self" and the set of the children of „self" has to be empty.

- `(self.children->reject(k | k <> self))->isEmpty()`

  The set of children for each person „self", for whome does not apply, that they are not equal to the person „self", has to be empty.

46

Example 5: Alternative, equivalent OCL-formulations (4)



Constrain: A person is not its own child

- `self.children->forAll(k | k <> self)`

  Each child of the person „self" is not the person „self".

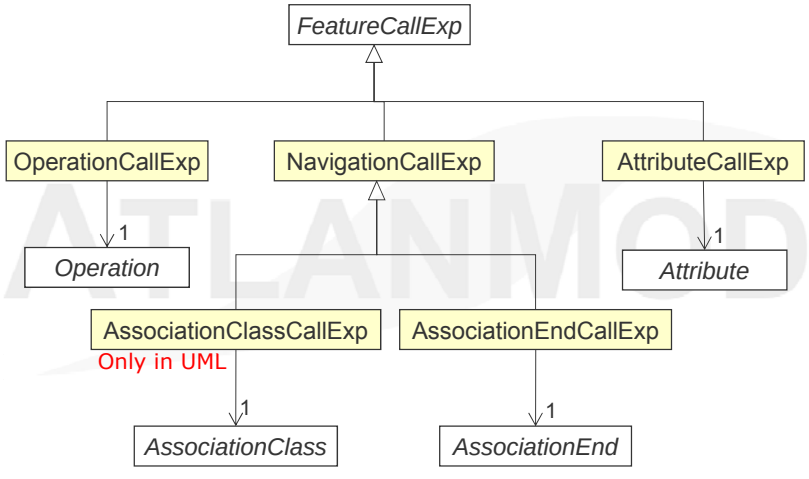
- `not self.children->exists(k | k = self)`

  There is no child for each person „self", which is the person „self"

  - ***Having so many ways to express each constraint is good or bad?***

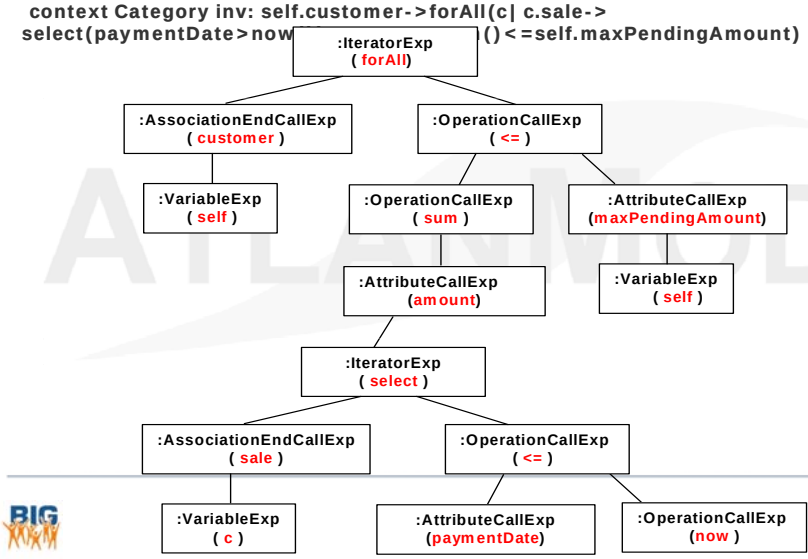
47

OCL Metamodel (expressions)



48

An OCL expression as an instance of the OCL MM



SO YOU THINK YOU KNOW OCL...

50

allInstances

- Car::allInstances () : Set(Car)
- String::allInstances() : Set(String) ???

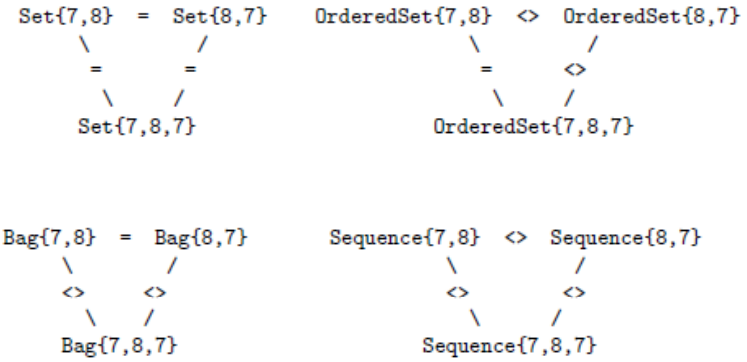
- ***Careful with the allInstances operation. Do not apply on infinite types!***

Null values

- 0<>null ?
- ''<>null ?
- 'null'<>null ?
- 'NULL'<>null ?
- 'Null'<>null ?

- ***All true, null is a value different from all the others***

Collections



Collections

- Sets are insensible to insertion order and frequency
- Ordered Stes are sensible to order, insensible to frequency
- Bags are sensible to frequency, insensible to order
- Sequence are sensible to both

Examples of Collections

- Set{7,8}=Set{}->including(8)->including(7)

  Bag{7,8,7}=Bag{8}->including(7)->including(7)

  Sequence{7,8,7}=Sequence{7,8}->including(7)

  OrderedSet{7,8}=OrderedSet{7}->including(8)->including(7)

  Set{7,8}->excluding(8)=Set{7}

  Bag{7,8,7}->excluding(7)=Bag{8}

  Sequence{7,8,7}->excluding(7)=Sequence{8}

  OrderedSet{9,6,7,8,6,7}->excluding(6)=OrderedSet{9,7,8}

Collection Casting

- Some ambiguos conversions
- Sequence{8,7,7,8}->asSet() = Set{7,8} :
- Sequence{8,7,7,8}->asBag() = Bag{7,7,8,8}
- Set{8,7,7,8}->asSequence()
  ? Sequence{7,8}
  ?Sequence{8,7}

  - *Implementator's decision!!*
  - *Also others like :*
    - *Set{7,8,9}->any(true)*

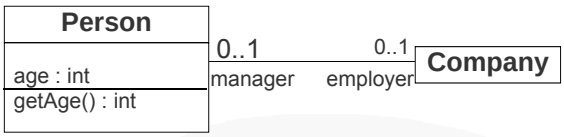
Collection Equality

- Set{7,8} = Set{8,7,7}
- Set{7,8} = Bag{7,8} ✖
- OrderedSet{8,9} = Sequence{8,9} ✖
- OrderedSet{7,8,8} = OrderedSet{7,8,7}
- OrderedSet{7,9,8} = OrderedSet{9,7,8} ✖

Invariants with undefined

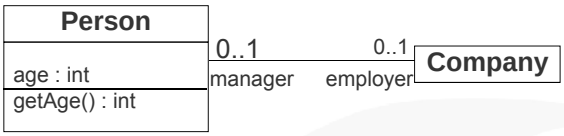
- ***Context Company inv:***

Self.manager.age>18



- Constraints are violated if they do not evaluate to true
  - Companies with a manager with age undefined will violate the constraint
  - What would happen in SQL?

Invariants with undefined (2)



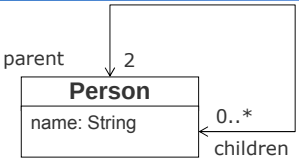
- ***Context Company inv: Self.manager.age>18***

- We need to check for undefinedness
  - If not self.manager.oclIsUndefiend then self.manager.age>18

- ***Or, Context Company inv: Self.manager->forAll(age>18)***
- ***Why??***

59

Recursive OCL queries



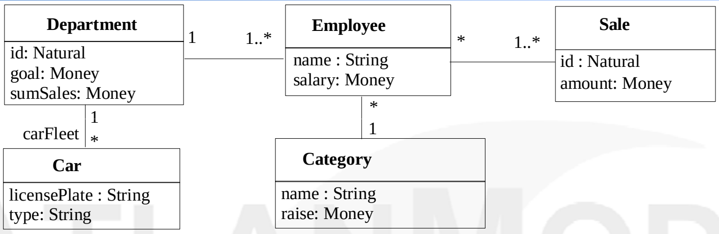
Constrain: A person cannot be his own ancestor

- Context Person inv: self->asSet()->closure(children)

- Closures added in the last version of OCL

60

OCL in postconditions (frame problem also in OCL)



If employees have sold enough to fulfill the goal their salary is increased as indicated by their category. Otherwise, employees are not longer allowed to use luxury cars

Or when we add/remove non-luxury cars to the department

(or remove all cars)

```
context Department::GoalEvaluation()
post: self.sumSales=self.employee.sales.amount->sum()  and
      if  self.sumSales>self.goal then
          self.employee->forAll(e| e.salary= e.salary@pre + e.category.raise)
      else self.carFleet->excludesAll(Car.allInstances->select(c| c.type='Luxury'))
      endif
```

(MISSING) OCL LIBRARIES

62

Extending OCL 1 – Aggregation functions

- Extension classified in three different groups of functions:
  - Distributive functions: can be defined by structural recursion
    - Max, min, sum, count, count distinct,…
  - Algebraic functions: finite algebraic expressions over distributive functions
    - Avg, variance, stddev, covariance, …
  - Holistic functions: the rest
    - Mode, descending rank, ascending rank, percentile, median

> ***These operations can be combined to provide more advanced ones (e.g. top(x) that is implemented using rank). See*** Jordi Cabot, Jose-Norberto Mazón, Jesús Pardillo, Juan Trujillo: Specifying Aggregation Functions in Multidimensional Models with OCL. ER 2010: 419-432

Some examples (1/3)

- MAX: Returns the element in a non-empty collection of objects of type *T* with the highest value.

```
context Collection::max():T
pre: self->notEmpty()
post: result = self->any(e | self->forAll(e2 | e >= e2))
```

- COUNT DISTINCT: Returns the number of different elements in a collection

```
context Collection::countDistinct(): Integer
post: result = self->asSet()->size()
```

Some examples (2/3)

- AVG: Returns the arithmetic average value of the elements in the non-empty collection.

```
context Collection::avg():Real
pre: self->notEmpty()
post: result = self->sum() / self->size()
```

- COVARIANCE: Returns the covariance value between two ordered sets

```
context OrderedSet::covariance(Y: OrderedSet):Real
pre: self->size() = Y->size() and self->notEmpty()
post:
  let avgY:Real = Y->avg() in
  let avgSelf:Real = self->avg() in
  result = (1/self->size()) *
  self->iterate(e; acc:Real=0 | acc +
  ((e - avgSelf) * (Y->at(self->indexOf(e)) - avgY))
```

Some examples (3/3)

- MODE: Returns the most frequent value in a collection.

```
context Collection::mode(): T
pre: self->notEmpty()
post: result = self->any(e | self->forAll(e2 |
                        self->count(e) >= self->count(e2))
```

- DESCENDING RANK: Returns the position (i.e., ranking) of an element within a Collection.

```
context Collection::rankDescending(e: T): Integer
pre: self->includes(e)
post: result = self->size() - self->select(e2 | e>=e2)
->size() + 1
```

Using our new aggregate functions

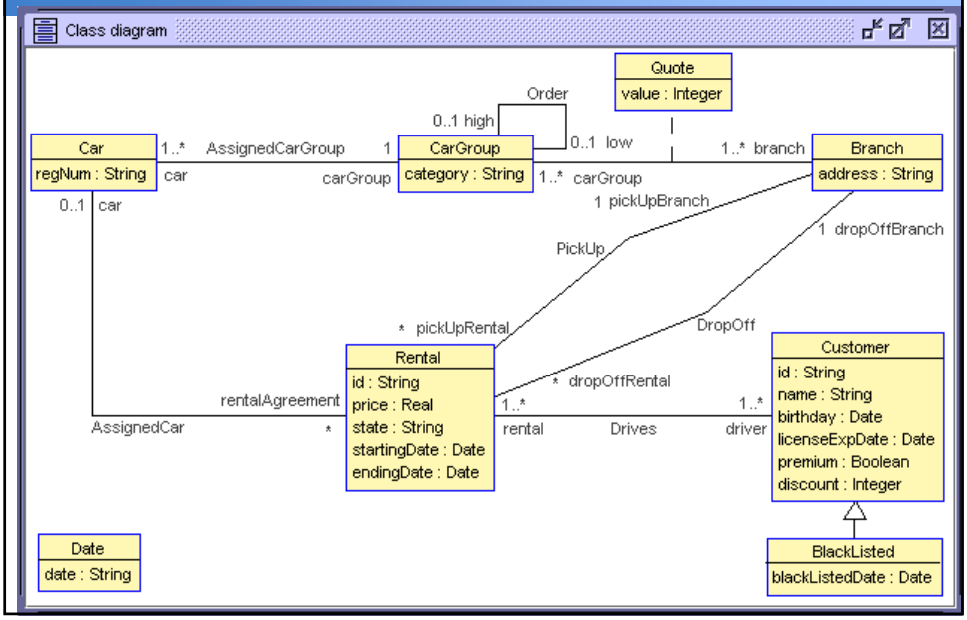
- Our functions can be used wherever a OCL standard function can be used
- They are called exactly in the same way
- Ex of use of the *avg* function to compute the average number of miles earned by a customer in each flight leg.

```
context Customer::avgPriceRental():Real
body: self.drives.price->avg()
```

OCL and Time

- OCL has serious limitations to express time-related constraints
  - Date and time datatypes are not even part of the OCL standard!!
- But time/date/calendar related constraints are very common in data-intensive applications
- Some works on extending OCL with temporal logic but we need much more than this

EU-Rent Example



OCL and Time

- Examples of useful constraints (can we express them in OCL?)
  - Rental ending date must be later or equal than starting date
  - On Mondays the agency is close (so no rentals starting on a Monday)
  - Minimum duration of a rental is 10 hours
  - Same customer cannot rent twice in the same week
  - Last Friday of each month the agency is closed

OCL and Time

- I'd like to be able to write things like:
  - Context Rental inv: self.startingdate<=seld.endingDate
  - Context Rental inv: not self.startingDate.dayOfTheWeek()<>DayOftheWeek::Monday
  - Context Rental inv: (self.endDate-self.startingDate).hours()>=10
  - Context rental inv : not self.startingdate=self.startingdate.getMonth().last('Friday')

Modularizing OCL

- OCL needs a modularization mechanism to extend the basic library with domain-specific ones created by experts

TOOLS

TOOLS – OCL IDES

Desired Features of a OCL tool IDE

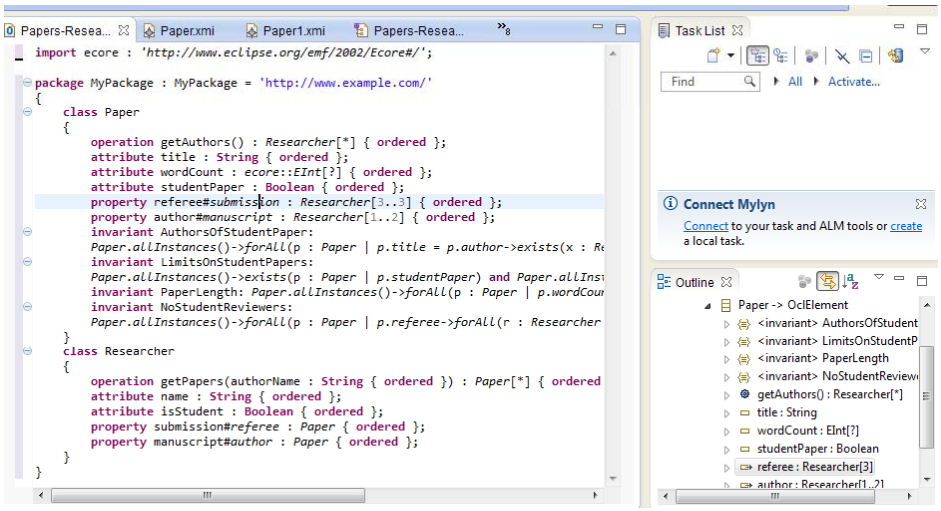
- List of features you'd like:

- *Joanna Dobroslawa Chimiak-Opoka, Birgit Demuth: A Feature Model for an IDE4OCL. ECEASST 36: (2010)*

Desired Features of a OCL tool IDE

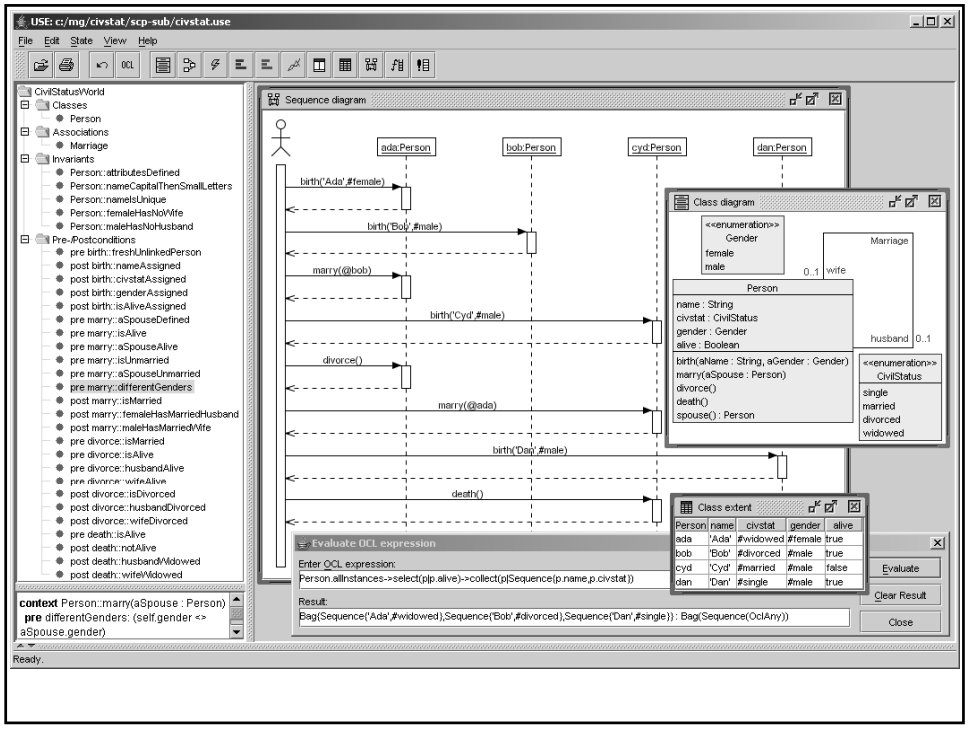
- Main active IDEs
  - MDT/OCL component
  - Dresden OCL
  - USE tool
- OCL support also in some UML tools like ArgoUML, Enterprise Architect, MagicDraw…

OCL Editor in Eclipse



USE tool

- USE allows to
  - Perform UML and OCL model animation, validation and verification
  - Evaluate OCL expressions
  - Get confidence in models (formal descriptions) by testing it with scenarios
  - Check consistency of models by constructing an object diagram
  - Show independency of invariants (no invariant follows from the others)
  - Check whether a property follows from other stated constraints
  - Construct a large amount of object diagrams through an ASSL program (A Snapshot Sequence Language)

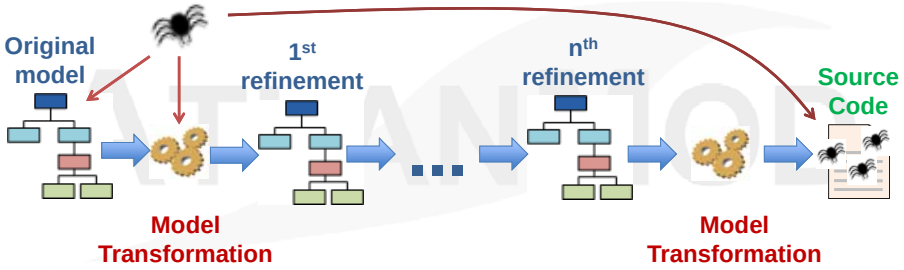


TOOLS – ANALYZING OCL

80

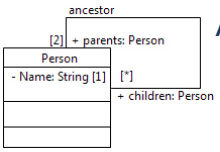
Why model verification?

MDE-based software development process

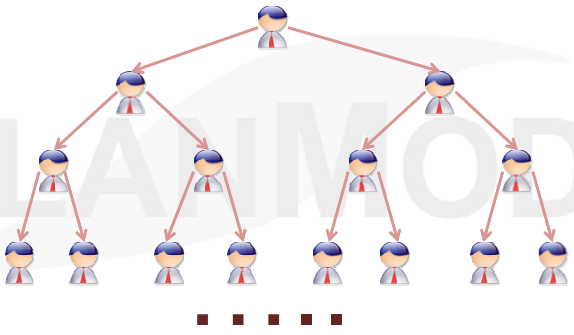


Errors in models will lead to errors in the resulting software

How models are verified?



A Person cannot be his own ancestor



Strong Satisfiability

What properties do you find useful?

- For static models?
- For dynamic ones? (including transformations)

Properties for static models

"It should be possible to instantiate…"

Strong SAT	≈	Every class and association
Weak SAT	≈	Some class of the model
Liveliness	≈	A specific class

"Given two constraints C1 and C2 …"

No subsumption	≈	C1 does not imply C2
No redundancy	≈	C1 & C2 are not redundant

And for dynamic ones

Applicabiliy ≈ Is it possible to execute the operation/transformation

(weak/strong) Executability ≈ Is the generated target model correct?

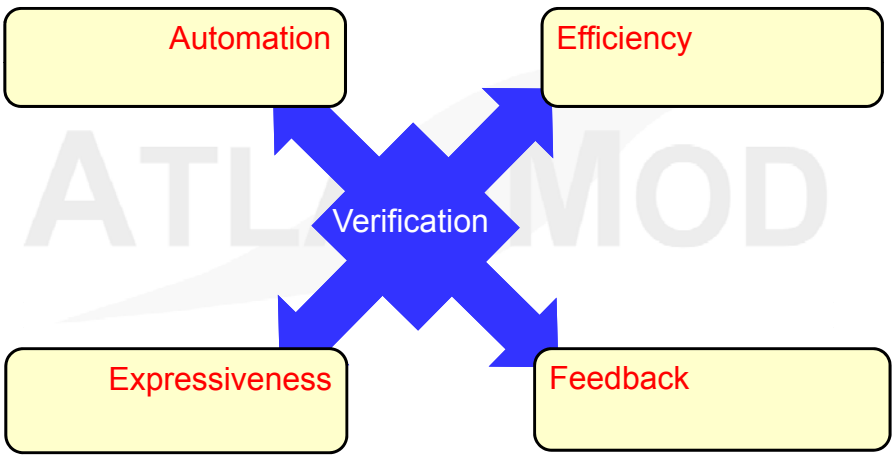
Determinism ≈ Given a source model there's only a possible target model?

Injective (and same with all other functional categories) ≈ Is each target model the result of a single source model?

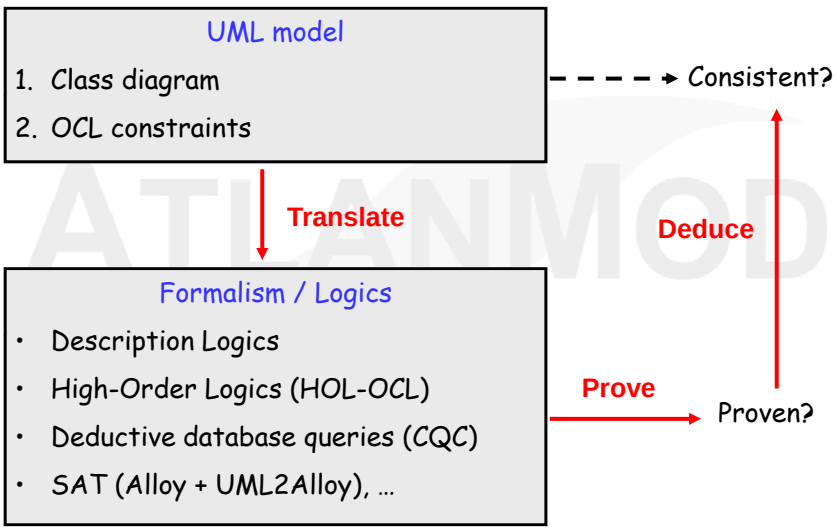
The problem of undecidability

- The expressiveness of modeling languages and the OCL represents an important setback at the time of checking correctness properties.
- Verification techniques usually run into decidability issues when checking correctness properties
- A typical example of this is checking strong satisfiability in UML class diagrams complemented with generic OCL constraints.

Our innovative approach



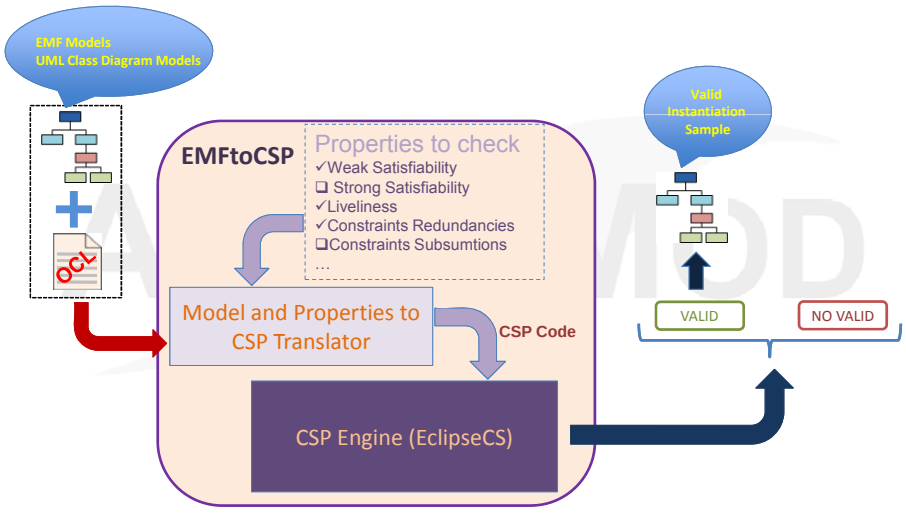
Summary of existing approaches



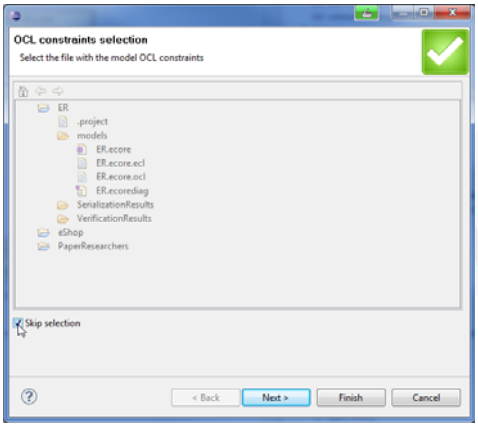
EMFtoCSP Features

- Supports
  - Verification of EMF models + OCL
  - Verification of UML Class Diagrams + OCL
- Pragmatic Approach
  - Model and constraints are converted into a CSP
  - Decidability
  - Bounded Verification
- Implementation
  - Made for Eclipse

EMFtoCSP: A General Picture



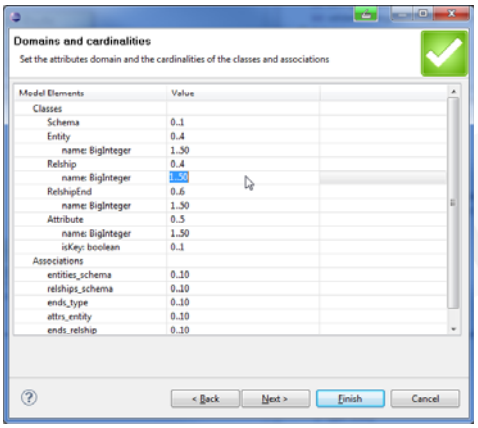
Using EMFtoCSP (II)



Step 1
OCL Selection

It allows to load OCL constraints from a separate ".ocl" file.

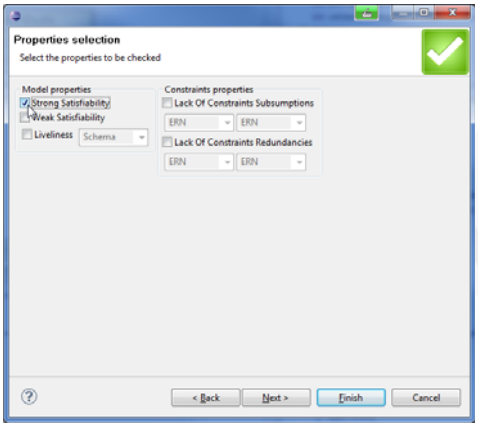
Using EMFtoCSP (III)



Step 2
Setting domains and cardinalities

To limit the search space where to look for a valid instance of the model.

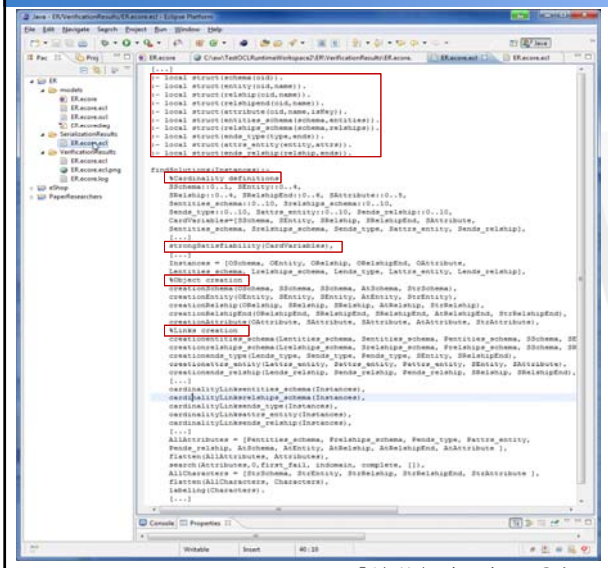
Using EMFtoCSP (IV)



Step 3
Selecting the properties to be checked

Some properties can only be checked when the model includes OCL constraints.

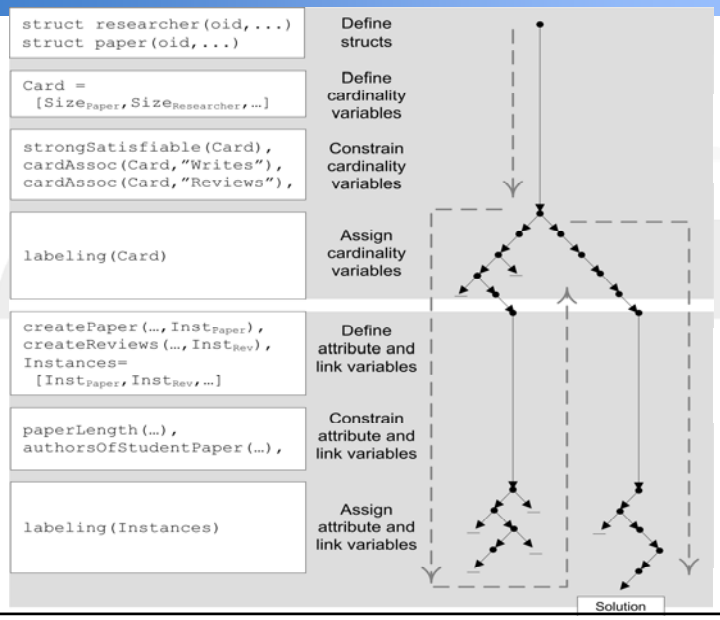
Using EMFtoCSP (VI)



Output 1
CSP code

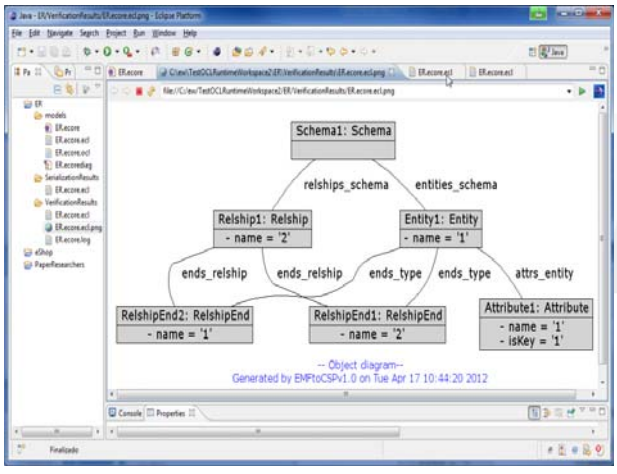
It is possible to review the autogenerated CSP code, once the verification process ends.

Resolution of the CSP



95

Using EMFtoCSP (and VII)



Output 2
Model instance

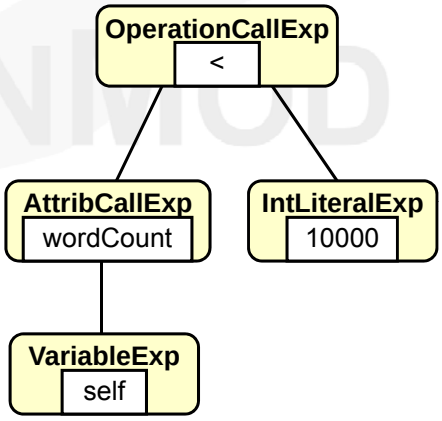
The instance proves that the model satisfies the correctness properties selected.

96

Translation of OCL invariants

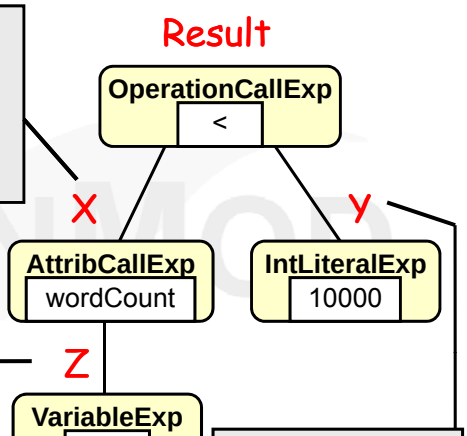
context Paper **inv**: self.wordCount < 10000

- OCL invariant = instance of OCL metamodel
- Invariant becomes a constraint of the CSP



Translation of OCL invariants

```
attribute(Vars, X) :-
  // Evaluate subexpressions
  self(Vars, Z),
  arg(Z, "wordCount", X).
```

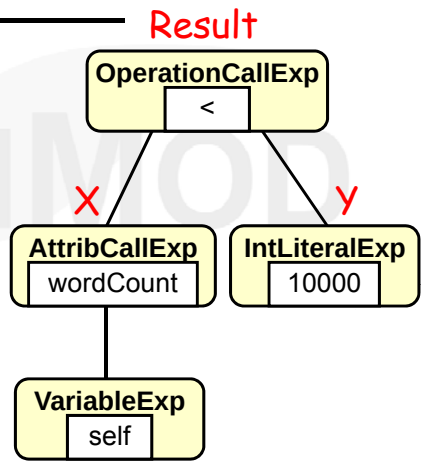



```
self(Vars, Z) :-
  // Z is the only visible var
  nth1(1, Vars, Z).
```

```
constant (Y):-
  Y = 10000.
```

Translation of OCL invariants

```
lessThan(…, Result):-
  // Evaluate subexpressions
  attribute(…, X),
  constant(…, Y),
  // Combine partial results
  #<(X,Y, Result).
```



EMFtoCSP Library

- All this with the help of Prolog library for OCL (2000 LOC)
- Key element of ECLiPSe (the constraint solver) -> support of a higher-order predicate

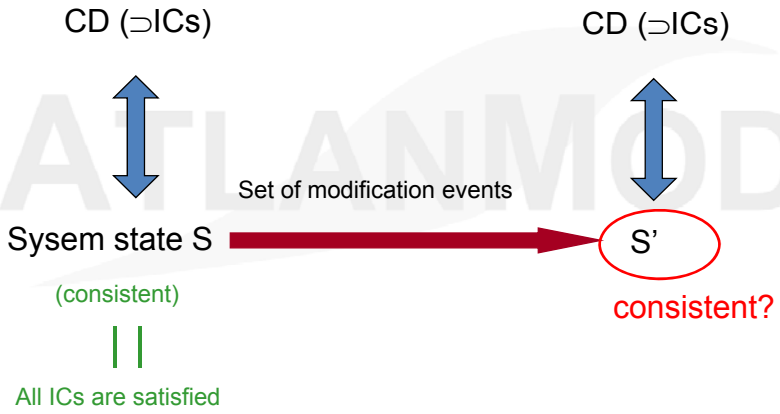
ocl_int_greater_than(Instances, Vars, Pred1, Pred2, Result) :-
apply(Pred1, [Instances, Vars, X]), % X is the result of evaluating Pred1
apply(Pred2, [Instances, Vars, Y]), % Y is the result of evaluating Pred2
Result::0..1, % Result is a boolean value
#>(X, Y, Result). % Result is true iff X > Y

TOOLS – CODE GENERATION

Code Generation for OCL expressions

- OCL expressions must be propagated to the system implementation
- Very very limited tool support for this (and not efficient – big scalability issues)
- Typically MDE tools limit validation to the input forms and/or provide a small DSL to express the rules they can enforce
- So, how would you implement OCL constraints?

The efficiency problem



Goal

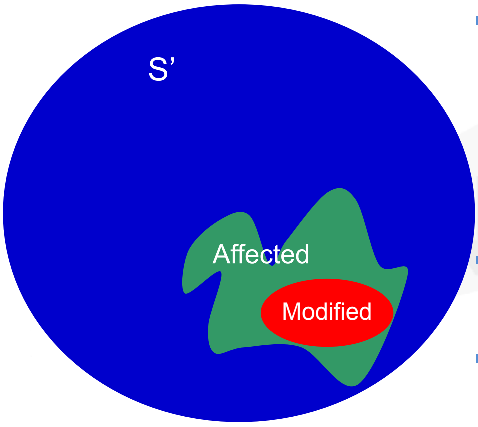
To propose a method to *efficiently* decide the consistency of S' wrt CD

Efficiency = Number of instances of State' that must be considered when checking its consistency

The method should be technologically independent

- Not bound to any technological assumption
- The results may be used to implement the CS in any platform

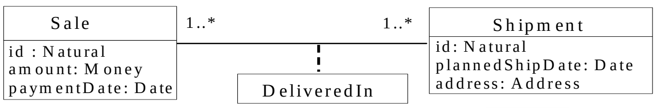
Efficiency (Incrementality) of a solution



- Since S was consistent, we may use the information about the applied events to avoid verifying the whole S' (checking *Affected* suffices)
- S' is consistent ⟺ *Affected* is consistent
- The affected instances are the modified ones plus some other instances related with them

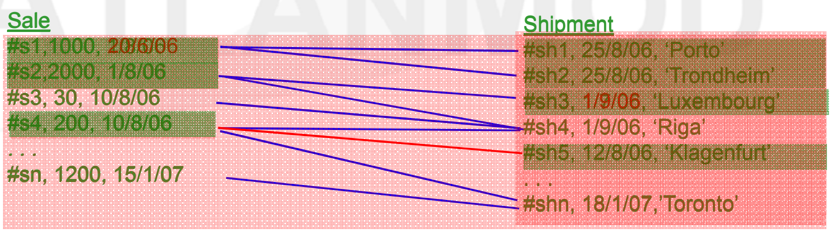
- *These kind of methods are referred as incremental methods (DB, as usual)*

A simple example



context Sale inv: self.shipment->forAll(s|
s.plannedShipDate<=self.paymentDate+30)

UpdateAttribute(Sale, paymentDate, #s1, 20/6/06) / InsertRT(DeliveredIn, #s4, #sh5) / UpdateAttribute(Shipment, plannedShipDate, #sh3, 1/9/06)



In OCL ICs must be satisfied by all instances of the context type / *Only affected sales must be considered*

Code Generation

- The generated expressions may be used for code-generation methods to efficiently check the constraints in the final platforms

UpdateAttr(paymentDate,Sale)
Constraint ValidShipDate

s.shipment->forAll(sh|
sh.plannedShipDate<=s.paymentDate+30)

```
MethodX(Sale s,…)
{  … s.paymentDate = value;  …
  //Verification of expression 3
  Iterator setsh = s.shipments.iterator();
  while ( setsh.hasNext() )
  {  Shipment sh = (Shipment) setsh.next();
     If (sh.plannedShipDate>s.paymentDate+30)
      throw new Exception("Invalid date");
   }
}
```

```
create trigger uPaymentDate
before update of PaymentDate on Sale for each row
Declare v_Error NUMBER;
        EInvalidDate Exception;
Begin   --Verification of expression 3
   SELECT count(*) into v_Error
   FROM DeliveredIn d, Shipment sh
   WHERE d.sale = :new.id and d.shipment = sh.id
      and sh.plannedShipDate>:new.paymentDate+30;
   If (v_Error>0) then raise EInvalidDate; end if;
End:
```

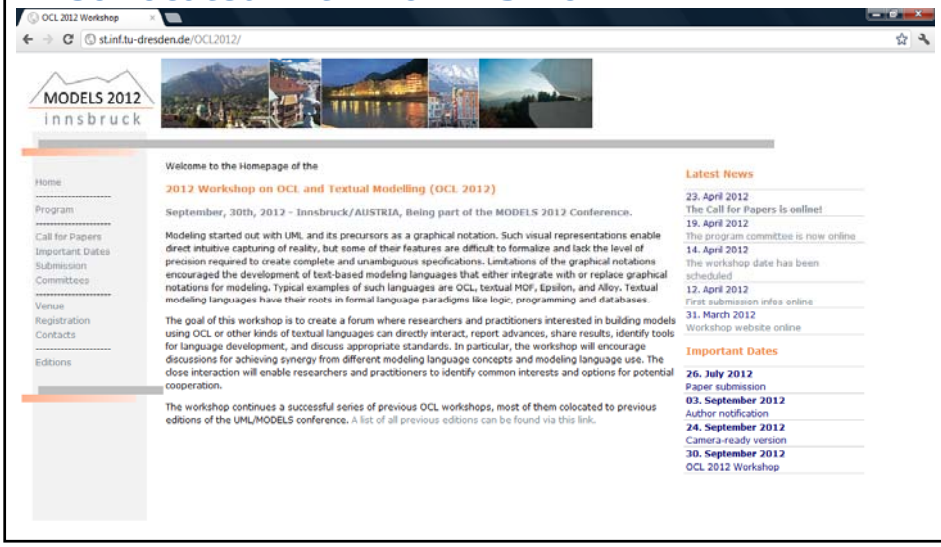
SHAMELESS PUBLICITY

The Modeling Portal

- Modeling-languages.com + @softmodeling

OCL Workshp 2012

- Co-located with MoDELS 2012



TO KEEP IN TOUCH:
JORDI.CABOT@INRIA.FR

Appendix

OCL Standard Library

- OCLAny operations
- String operations
- Integer/Real operations
- Boolean operations
- Collection operations
- Iterator-based Collection operations

Appendix

OCLAny operations

Operation	Return value type
=	Boolean
<>	Boolean
oclIsUndefined()	Boolean
oclAsType(type2)	Objecs of Type *type2* or *oclUndefined*
oclIsTypeOf()	Boolean
oclIsKindOf()	Boolean
allInstances()	Set(***T***)

Appendix

String operations

Operation	Notation	Return value type
concatenation	*s.concat(String)*	String
size	*s.size()*	Integer
to lower case	*s.toLower()*	String
to upper case	*s.toUpper()*	String
substring	*s.substring(Integer, Integer)*	String
equals	*s1 = s2*	Boolean
not equals	*s1 <> s2*	Boolean

Appendix

Integer/Real operations

Operation	Notation	Return value type
plus	*a + b*	Integer/Real
minus	*a – b*	Integer/Real
multiplication	*a * b*	Integer/Real
division	*a / b*	Real
modulus	*a.mod(b)*	Integer
integer division	*a.div(b)*	Integer
maximum	*a.max(b)*	Integer/Real
minimum	*a.min(b)*	Integer/Real
round	*a.round()*	Integer
floor	*a.floor()*	Integer
absolute value	*a.abs()*	Integer/Real
equals	*a = b*	Boolean
not equals	*a <> b*	Boolean
less	*a < b*	Boolean
more	*a > b*	Boolean
less or equal	*a <= b*	Boolean
more or equal	*a >= b*	Boolean

Appendix

Boolean operations

Operation	Notation	Return value type
or	*a or b*	Boolean
and	*a and b*	Boolean
exclusive or[1)]	*a xor b*	Boolean
negation	*not a*	Boolean
implication	*a implies b*	Boolean
if then else[2)]	*if a then b1 else b2 endif*	*typeOf*(b1) = *typeOf*(b2)
equals	*a = b*	Boolean
not equals	*a <> b*	Boolean

1) true, if either *a* or *b* is true, but nt both are true at the same time
2) else-part is mandatory

Appendix

Collection operations

Operation	Return value type
count(object)	Integer
excludes(object)	Boolean
excludesAll(collection)	Boolean
includes(object)	Boolean
includesAll(collection)	Boolean
isEmpty()	Boolean
notEmtpy()	Boolean
size()	Integer
sum()	Integer/Real

Appendix

Iterator-based operations for Collections

Operation	Return value type
any(exp)	Object oder *oclUndefined*
collect(exp)	Bag(***T***)
collectNested(exp)	Collection(Collection(***T***))
exists(exp)	Boolean
forAll(exp)	Boolean
isUnique(exp)	Boolean
iterate(…)	beliebig
one(exp)	Boolean
reject(exp)	Collection(***T***)
select(exp)	Collection(***T***)
sortedBy(exp)	Collection(***T***)