OCL – The Object Constraint Language

Based on presentations from:

Warmer, Kleppe, Selic, Gorman, Yong He, Amyot, Brugge and Dutoit, Bultan, Lings, Lieber.

Precise Models

- Specification of software is written down in one or more models.
  - Natural language text is used to explain the background and motivation of the models.

- The models are precise enough to have a direct link with the actual code.

- This is what the OMG calls MDA or Model Driven Architecture.
Models Only

- The models are precise and detailed enough to allow complete code-generation.

- The code is invisible (as assembler is today).

- Modeling language ➔ High level programming language.

- This is future technology (🤔).

---

Combining UML and OCL

- Without OCL expressions, the model would be severely underspecified;

- Without the UML diagrams, the OCL expressions would refer to non-existing model elements,
  - there is no way in OCL to specify classes and associations.

- Only when we combine the UML diagrams and the constraints can we completely specify the model.
UML diagrams don’t tell us everything

OCL makes constraints unambiguous

Context Person
Invariant: parents-->exclude(self) and children-->exclude(self)

Where to use OCL?

- Specify invariants for classes and types
- To specify derivation rules for attributes.
- Specify pre-conditions and post-conditions for methods.
- To specify constraints on operations.
- To describe guard conditions in UML diagrams.
What is OCL?

- A **textual specification language**
- Is a **formal language**
- Is **part of UML**
- Is **not a programming language**
- Based on set theory and **predicate logic**:  
  - has a formal mathematical semantics

Advantages of Formal Constraints

- **Better documentation**
  - Constraints add information about the model elements and their relationships to the visual models used in UML
  - It is way of documenting the model

- **More precision**
  - OCL constraints have formal semantics, hence, can be used to reduce the ambiguity in the UML models

- **Communication without misunderstanding**
  - UML models are used to communicate between developers,
  - Using OCL constraints modelers can communicate unambiguously.
OCL Constraints and Expressions

A constraint is a restriction on one or more values of (part of) an object model/system.

- Each OCL expression has a type and evaluates to a value, object or collection of objects within the system.
- A constraint is a OCL expression of type Boolean.

OCL Constraints

- OCL constraints are declarative
  - They specify what must be true, not what must be done
- OCL constraints have no side effects
  - Evaluating an OCL expression does not change the state of the system
- OCL constraints have formal syntax and semantics
  - their interpretation is unambiguous
Elements of an OCL expression that is associated with a UML model

- basic types: String, Boolean, Integer, Real.
- classes from the UML model and their attributes.
- enumeration types from the UML model.
- associations from the UML model.

Constraints, Contexts and Self

- Every constraint is bound to a specific type (class, association class, interface) in the UML model: its context.

- The context objects may be denoted within the expression using the keyword ‘self’.

- The context can be specified by:
  
  - Context <context name>
  - A dashed note line connecting to the context figure in the UML models

- A constraint might have a name following the keyword invariant.
Simple constraints on class attributes

<table>
<thead>
<tr>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: String</td>
</tr>
<tr>
<td>title: String</td>
</tr>
<tr>
<td>age: Integer</td>
</tr>
<tr>
<td>isMale: Boolean</td>
</tr>
</tbody>
</table>

Context Customer

Invariant: age >= 18 and age < 66

Invariant: title = if isMale then 'Mr.' else 'Ms.' endif

Invariant: name.size() < 100

Standard Operations for Real and Integer types

<table>
<thead>
<tr>
<th>Operation</th>
<th>Notation</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>equals</td>
<td>a = b</td>
<td>Boolean</td>
</tr>
<tr>
<td>not equals</td>
<td>a &lt;&gt; b</td>
<td>Boolean</td>
</tr>
<tr>
<td>less</td>
<td>a &lt; b</td>
<td>Boolean</td>
</tr>
<tr>
<td>more</td>
<td>a &gt; b</td>
<td>Boolean</td>
</tr>
<tr>
<td>less or equal</td>
<td>a &lt;= b</td>
<td>Boolean</td>
</tr>
<tr>
<td>more or equal</td>
<td>a &gt;= b</td>
<td>Boolean</td>
</tr>
<tr>
<td>plus</td>
<td>a + b</td>
<td>Integer or Real</td>
</tr>
<tr>
<td>minus</td>
<td>a - b</td>
<td>Integer or Real</td>
</tr>
<tr>
<td>multiply</td>
<td>a * b</td>
<td>Integer or Real</td>
</tr>
<tr>
<td>divide</td>
<td>a / b</td>
<td>Real</td>
</tr>
<tr>
<td>modulus</td>
<td>a.mod(b)</td>
<td>Integer</td>
</tr>
<tr>
<td>integer division</td>
<td>a.div(b)</td>
<td>Integer</td>
</tr>
<tr>
<td>absolute value</td>
<td>a.abs()</td>
<td>Integer or Real</td>
</tr>
<tr>
<td>maximum</td>
<td>a.max(b)</td>
<td>Integer or Real</td>
</tr>
<tr>
<td>minimum</td>
<td>a.min(b)</td>
<td>Integer or Real</td>
</tr>
<tr>
<td>round</td>
<td>a.round()</td>
<td>Integer</td>
</tr>
<tr>
<td>floor</td>
<td>a.floor()</td>
<td>Integer</td>
</tr>
</tbody>
</table>
### Standard Operations for the Boolean Type

<table>
<thead>
<tr>
<th>Operation</th>
<th>Notation</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>or</td>
<td>a or b</td>
<td>Boolean</td>
</tr>
<tr>
<td>and</td>
<td>a and b</td>
<td>Boolean</td>
</tr>
<tr>
<td>exclusive or</td>
<td>a xor b</td>
<td>Boolean</td>
</tr>
<tr>
<td>negation</td>
<td>not a</td>
<td>Boolean</td>
</tr>
<tr>
<td>equals</td>
<td>a = b</td>
<td>Boolean</td>
</tr>
<tr>
<td>not equals</td>
<td>a &lt;&gt; b</td>
<td>Boolean</td>
</tr>
<tr>
<td>implication</td>
<td>a implies b</td>
<td>Boolean</td>
</tr>
<tr>
<td>if then else</td>
<td>if a then b1 else b2 endif</td>
<td>type of b</td>
</tr>
</tbody>
</table>

### Standard Operations for the String Type

<table>
<thead>
<tr>
<th>Operation</th>
<th>Expression</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>concatenation</td>
<td>s.concat(string)</td>
<td>String</td>
</tr>
<tr>
<td>size</td>
<td>s.size()</td>
<td>Integer</td>
</tr>
<tr>
<td>to lower case</td>
<td>s.toLowerCase()</td>
<td>String</td>
</tr>
<tr>
<td>to upper case</td>
<td>s.toUpperCase()</td>
<td>String</td>
</tr>
<tr>
<td>substring</td>
<td>s.substring(int, int)</td>
<td>String</td>
</tr>
<tr>
<td>equals</td>
<td>s1 = s2</td>
<td>Boolean</td>
</tr>
<tr>
<td>not equals</td>
<td>s1 &lt;&gt; s2</td>
<td>Boolean</td>
</tr>
</tbody>
</table>
Invariants using Navigation over Association Ends or Roles

Navigation over associations is used to refer to associated objects, starting from the context object.

If the role name is missing: use the class name at the other end of the association, starting with a lowercase letter.

Preferred: Always give role names.

More Constraints Examples

- All players must be over 18.

  context Player invariant:
  self.age >= 18

- The number of guests in each room doesn’t exceed the number of beds in the room.

  context Room invariant:
  guests -> size <= numberOfBeds

  context Room invariant:
  self.numberOfBeds: Integer
Context of an OCL expression

\[\textbf{Vocabulary} \rightarrow^1 * \textbf{VocabElement} \rightarrow^1 0..5 \rightarrow \textbf{Hint}\]

\textbf{Context VocabElement}
\textbf{Invariant:} \(\text{self.hint} \rightarrow \text{size()} \geq 0 \land \text{self.hint} \rightarrow \text{size()} \leq 5\)

\textbf{Context VocabElement}
\textbf{Invariant:} \(\text{self.vocabulary} \rightarrow \text{size()} = 1\)

\textbf{Context Hint}
\textbf{Invariant:} \(\text{self.vocabElement} \rightarrow \text{size()} = 1\)

An example: constraints on Polygons

a LinearShape is any shape that can be constructed of line segments (in contrast with shapes that contain curves).

\{\text{edge} \rightarrow \text{forall(e1, e2 | e1 <> e2 implies e1.startPoint <> e2.startpoint and e1.endPoint <> e2.endpoint)\}

\{\text{edge} \rightarrow \text{size()} = 1\}

\{\text{length} = \text{edge.length} \rightarrow \text{sum()}\}

(Not OCL 2.0)
Constraints may be denoted within the UML model or in a separate document.

- the expression:
  context Flight
  inv: self.duration < 4

- is identical to:
  context Flight
  inv: duration < 4

- is identical to:
Model classes and attributes

- “Normal” attributes

  context Flight
  inv: self.maxNrPassengers <= 1000

- Class attributes

  context Passenger
  inv: age >= Passenger.minAge

Example: query operations

context Flight
inv: self.departTime.difference(self.arrivalTime).equals(self.duration)
Example: navigations

context Flight
inv: origin <> destination
inv: origin.name = ‘Amsterdam’

context Flight
inv: airline.name = ‘KLM’

OCL Collection types

- When a multiplicity is greater than one, we have to talk about collections of objects. These collections can allow duplicate elements and ordered elements.
- **Collection** is a predefined OCL type
- There are four different collection subtypes:
  - **Set**: no duplicates, no ordering
  - **Bag**: duplicates allowed, no ordering
  - **OrderedSet**: no duplicates, with ordering
  - **Sequence**: duplicates allowed, with ordering
    - Essentially an ordered Bag

Examples: Set { 1, 4, 3, 8 }; OrderedSet {1, 3, 4, 8 }, Bag { 1, 8, 4, 8, 3 }, Sequence { 1, 3, 4, 8, 8 }
Collection Types and Navigation in OCL Expressions

- If self is class C, with attribute a then self.a evaluates to the object stored in a.

- If C has a 1..* association called R to another class D
  self.R returns:
  - to a Set whose elements are of type D
  - if R is {ordered} then a Sequence is returned

- If D has attribute b then self.R.b evaluates to the set (or sequence if {ordered is used}) of all the b’s belonging to D.

Navigating associations

![Diagram of associations]

**Context Account**

**Invariant:** self.transaction returns a set of transactions

**Context Book**

**Invariant:** self.borrower returns a set of members
Navigating to ordered collections

Customer * Account * Transaction

**Context Account**
**Invariant:** `self.customer` produces a customer object

**Context Customer**
**Invariant:** `self.account` produces an ordered set of accounts

**Context Customer**
**Invariant:** `self.account.transaction` produces a sequence of transactions

---

OCL operations on Collection types

- With collection types, an OCL expression
  - states a fact about all objects in the collection, or
  - states a fact about the collection itself, e.g. the size of the collection.

- Syntax: `collection->operation`

  Use of the “->” (arrow) operator instead of the “.” (dot) operator
Specific operations for collection subtypes

- **Set**
  - `asSequence()`
  - `including(object)`
  - `asOrderedSet()`
  - `intersection(coll)`

- **Collection**
  - **OrderedSet**
    - `append(object)`
    - `=`, `<>`, `-`
    - `asSequence()`
    - `asBag()`
  - **Bag**
    - `flatten()`
    - `asSet()`
  - **Sequence**
    - `first()`
    - `last()`
    - `at(int)`
    - `append()`
    - `prepend()`
    - `asBag()`
    - `asSet()`

- Many more exist
- Some common operations have different meanings

---

Navigating associations

- **Customer**
  - *Account*
  - *Transaction*

**Context Customer**

**Invariant:** `self.account.transaction` produces a bag of transactions

If we want to use this result as a set, we can do the following

`self.account.transaction -> asSet`
### Standard operations on all Collection types

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size()</td>
<td>The number of elements in the collection</td>
</tr>
<tr>
<td>count(object)</td>
<td>The number of occurrences of object in the collection.</td>
</tr>
<tr>
<td>includes(object)</td>
<td>True if the object is an element of the collection.</td>
</tr>
<tr>
<td>includesAll(collection)</td>
<td>True if all elements of the parameter collection are present in the current collection.</td>
</tr>
<tr>
<td>excludes(object)</td>
<td>True if the object is <em>not</em> an element of the collection.</td>
</tr>
<tr>
<td>excludesAll(collection)</td>
<td>True if all elements of the parameter collection are <em>not</em> present in the current collection.</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>True if the collection contains no elements.</td>
</tr>
<tr>
<td>notEmpty()</td>
<td>True if the collection contains one or more elements.</td>
</tr>
</tbody>
</table>

### Standard loop operations on all Collection types

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iterate(expression)</td>
<td>Expression is evaluated for every element in the collection.</td>
</tr>
<tr>
<td>sum(collection)</td>
<td>The addition of all elements in the collection (type must support addition, e.g. Real or Integer).</td>
</tr>
<tr>
<td>isUnique(expr)</td>
<td>True if expr has a unique value for all elements in the source collection.</td>
</tr>
<tr>
<td>select(expr)</td>
<td>Returns a subcollection of the source collection containing all elements for which expr is true.</td>
</tr>
<tr>
<td>exists(expression)</td>
<td>True if expression is true for at least one element in the collection.</td>
</tr>
<tr>
<td>forAll(expression)</td>
<td>True if expression is true for all elements.</td>
</tr>
</tbody>
</table>

Other operations include:

- any, collect, collectNested, one, reject, and sortedBy
Example using `size()` and `notEmpty()`

![Diagram of Account and Customer relationship]

Each account can have at most 2 customers

context Account
inv: self.R -> size() <= 2

An account must be assigned to at least one customer

context Account
inv: self.R -> notEmpty()

Example: subset constraint

![Diagram of Flight and Person]

Context Flight
Inv: self.crew -> includes( self.pilot )

Context Flight
Inv: self.crew -> includesAll(self.flightAttendant)
Select, Reject, Collect, and Sum

- `collection->select(condition):` creates a subcollection that contains objects that satisfy the condition.

- `collection->reject(condition):` creates a subcollection that contains objects that do not satisfy the condition.

- `collection->collect(object.attribute):` creates a collection of equal size with objects containing the attribute specified.

- `collection->sum():` returns the sum of the elements of the collection; these elements must be numerical.

Example: iteration over collections

```
Account 1
  transactions * Transaction

Context Account
Inv: self.transactions -> select(value > 500 )

Context Account
Inv: self.transactions -> reject(not(value > 500 ))

Context Account
Inv: self.transactions -> sum(collect(value))

Context Account
Inv: self.transactions.value -> sum()
```
Iterate example

- Example iterate:
  context Airline
  inv: flights->select(maxNrPassengers > 150) -> notEmpty

- Is identical to:
  context Airline
  inv:
  flights->iterate (f : Flight;
  answer : Set(Flight) = Set{} |
  if f.maxNrPassengers > 150 then
    answer -> including(f)
  else
    answer endif ) -> notEmpty

forAll with two variables

- Considers each pair of employees

context Account
inv: self.R -> forAll( e1, e2 : Customer |
  e1 <> e2 implies e1.id <> e2.id)
Adding Preconditions and Postconditions to Operations

- **Preconditions:**
  - are predicates associated with a specific operation
  - must be true before the operation is invoked.

- **Postcondition:**
  - are predicates associated with a specific operation
  - must be true after an operation is invoked

About postconditions – initial values

- What is the problem with the following:

```
context Person::hasBirthday()
post: age = age + 1
```

- In OCL, we use the `@pre` suffix to indicate that we are referring to a value at the start of an operation:

```
context Person::hasBirthday()
post: age = age@pre + 1
```

- `@pre` can also be used on a query operation:

```
context Person::hasBirthday()
post: getAge() = getAge@pre() + 1
```
Enumerations in OCL

Example: abstract classes

The fact that the ClasseA is abstract can be expressed this way:

Context ClasseA

Inv: ClasseA.allInstances() ->
    select(oclType = ClasseA) -> isEmpty()

This constraint uses the meta facility oclType.
Example Model

Invariants on Classes

- **context Crate**
  -- the number of bottles may not exceed its capacity
  \( \text{inv: bottles->size() <= capacity} \)
  -- each bottle must fit in the crate
  \( \text{inv: bottles->forAll(diameter < maxDiameter)} \)
  -- the total weight must be less than the maximum
  \( \text{inv: totalWeight() <= maxWeight} \)

- **FORALL b : Bottle FROM bottles**
  \( \text{ISTRUE} \)
  \( b.\text{diameter < maxDiameter} \)
Preconditions on Operations

- context **Crate::addBottle( b : Bottle )**
  -- there must be room in the crate
  pre:  bottles->size() < capacity
- -- the maximum weight for the crate may not be exceeded
  pre:  totalWeight() + b.totalWeight() <= maxWeight
- -- the bottle is not in the crate
  pre:  not bottles->includes( b )
- -- ‘b’ is added to the collection of bottles
  post: bottles = bottles@pre->including(b)

Business Rules

- There may be at most as many caps with winner markers on a pallet as there are crates on the pallet:
  - context Pallet inv:
  - caps->select(winnerMark = true)->size() <= crates->size()
  - SIZEOF
    - SELECT c : Cap FROM self.caps
    - WHERE c.winnerMark = true
    - <=
    - SIZEOF self.crates
Initial values and derivations

-- The caps in a pallet are derived:
- context Pallet::cap
  derive: crates.bottles.cap->asSet()
- context Bottle::contents
  init: 0

Operation body

The body of a query operation can be specified in OCL

- context Bottle::totalWeight() : Integer
  body: weight + cap.weight

- context Crate::totalWeight() : Integer
  body: weight + bottles.totalWeight()->sum()

State invariants

context Bottle
state Closed inv: contents = capacity

Guards

open

close [ contents=capacity ]

closed
The start date for any mortgage must be before the end date.
context Mortgage
inv: startDate < endDate
(or startDate.before(endDate)???)

A person may have a mortgage on a house only if that house is owned by himself or herself (one cannot get a mortgage on the house of one’s neighbor or friend!)
context Mortgage
inv: security.owner = borrower
A person may have a mortgage on a house only if that house is owned by himself or herself (one cannot get a mortgage on the house of one’s neighbor or friend!). Second example, with Person as context.

context Person

inv: self.mortgages.security.owner -> --Coll of persons

forAll(p : Person | p = self)

The social security number of all persons must be unique.

context Person

inv: Person::allInstance() ->

isUnique(socSecNr)
A new mortgage will be allowed only when the person’s income is sufficient.
(assume that the yearly payment must be less than 30% of the salary)
context Person::getMortgage(sum:Money, security:house)
pre: self.mortgages.monthlyPayment->sum() < self.salary * 0.30 / 12
(this still does not consider the new sum requested...)

A new mortgage will be allowed only when the countervalue of the house is sufficient.
context Person::getMortgage(sum:Money, security:house)
pre: security.value >= (security.mortgages.principal->sum()+sum)
a) Modules can be taken iff they have more than seven students registered
   Note: when should such a constraint be imposed?

**context Module**
**invariant:** \( taken\_by \rightarrow size > 7 \)

b) The assessments for a module must total 100%

**context Module**
**invariant:**
\[
set\_work.weight \rightarrow \sum(\ ) = 100
\]
c) Students must register for 120 credits each year

\textbf{context} Student
\textbf{invariant: }\text{takes.credit} \rightarrow \text{sum( )} = 120

d) Students must take at least 90 credits of CS modules each year

\textbf{context} Student
\textbf{invariant: }
\text{takes} \rightarrow
\text{select}(\text{code.substring}(1,2) = 'CS').\text{credit} \rightarrow \text{sum( )} \geq 90

e) All modules must have at least one assessment worth over 50%

\textbf{context} Module
\textbf{invariant: }\text{set\_work} \rightarrow \text{exists}(\text{weight} > 50)

f) Students can only have assessments for modules which they are taking

\textbf{context} Student
\textbf{invariant: }\text{takes} \rightarrow \text{includesAll}(\text{submits\_for\_module})
Example of a static UML Model

Problem story:
A company handles loyalty programs (class LoyaltyProgram) for companies (class ProgramPartner) that offer their customers various kinds of bonuses. Often, the extras take the form of bonus points or air miles, but other bonuses are possible. Anything a company is willing to offer can be a service (class Service) rendered in a loyalty program. Every customer can enter the loyalty program by obtaining a membership card (class CustomerCard). The objects of class Customer represent the persons who have entered the program. A membership card is issued to one person, but can be used for an entire family or business. Loyalty programs can allow customers to save bonus points (class loyaltyAccount), with which they can “buy” services from program partners. A loyalty account is issued per customer membership in a loyalty program (association class Membership). Transactions (class Transaction) on loyalty accounts involve various services provided by the program partners and are performed per single card. There are two kinds of transactions: Earning and burning. Membership durations determine various levels of services (class serviceLevel).
Invariants on Attributes

context Customer
invariant ageRestriction: age >= 18

context CustomerCard
invariant correctDates: validFrom.isBefore(goodThru)

The type of validFrom and goodThru is Date. isBefore(Date):Boolean is a Date operation.

Invariants using Navigation over Association Ends

context CustomerCard
invariant printedName:
printedName =
owner.title.concat(' ').concat(owner.name)

Invariants using Navigation from Association Classes

Navigation from an association class can use the classes at the association class end, or the role names. The context object is the association class instance – a tuple.

“The owner of the card of a membership must be the customer in the membership”:

context Membership
invariant correctCard: card.owner = customer
Invariants using Navigation through Associations with “Many” Multiplicity

Navigation over associations roles with multiplicity greater than 1 yields a Collection type. Operations on collections are accessed using an arrow ->, followed by the operation name.

“A customer card belongs only to a membership of its owner”:

context CustomerCard
invariant correctCard:
  owner.Membership->includes(membership)

owner : a Customer instance.
owner.Membership : a set of Membership instances.
membership : a Membership instance.
includes is an operation of the OCL Collection type.

Navigation to Collections

“The partners of a loyalty program have at least one delivered service”:

context LoyaltyProgram
invariant minServices:
  partners.deliveredservices->size() >= 1

“The number of a customer’s programs is equal to that of his/her valid cards”:

context Customer
invariant sizesAgree:
  Programs->size() = cards->select(valid=true)->size()
“When a loyalty program does not offer the possibility to earn or burn points, the members of the loyalty program do not have loyalty accounts. That is, the loyalty accounts associated with the Memberships must be empty”:

context LoyaltyProgram
invariant noAccounts:
partners.deliveredServices->
    forAll(pointsEarned = 0 and pointsBurned = 0)
    implies Membership.account->isEmpty()

and, or, not, implies, xor are logical connectives.

Changing the context

<table>
<thead>
<tr>
<th>Customer</th>
<th>StoreCard</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: String</td>
<td>printName: String</td>
</tr>
<tr>
<td>title: String</td>
<td>points: Integer</td>
</tr>
<tr>
<td>golduser: Boolean</td>
<td>earn(p: Integer)</td>
</tr>
<tr>
<td>age(): Integer</td>
<td></td>
</tr>
</tbody>
</table>

context StoreCard
invariant: printName = owner.title.concat(owner.name)

context Customer
cards → forAll ( printName = owner.title.concat(owner.name) )

Note switch of context!
Invariants using Navigation through Cyclic Association Classes

- Navigation through association classes that are cyclic requires use of roles to distinguish between association ends:
  \[ \text{object.associationClass[role]} \]

- The accumulated score of an employee is positive:
  \[
  \text{context Person invariant: employeeRanking[employees].score->sum()>0}
  \]

- Every boss must give at least one 10 score:
  \[
  \text{context Person invariant: employeeRanking[bosses]->exists(score = 10)}
  \]

Classes and Subclasses

- Consider the following constraint
  \[
  \text{context LoyaltyProgram invariant: partners.deliveredServices.transaction.points->sum()<10,000}
  \]

- If the constraint applies only to the Burning subclass, we can use the operation oclType of OCL:
  \[
  \text{context LoyaltyProgram invariant: partners.deliveredServices.transaction}
  \]
  \[
  \hspace{1cm} \rightarrow \text{select(oclType = Burning).points->sum()<10,000}
  \]
Classes and Subclasses

“The target of a dependency is not its source”

context Dependency
invariant: self.source <> self

Is ambiguous:
Dependency is both
a ModelElement and an Association class.

context Dependency
invariant: self.oclAsType(Dependency).source <> self
invariant:
    self.oclAsType(ModelElement).source -> isEmpty()

OCL for other diagrams

OCL is not limited to class diagrams. Can be used for:

- **Statecharts**: guarding conditions [conditions] on transitions, target of actions, parameter values, restrictions on states

- **Sequence/collaboration/activity diagrams**: guarding conditions [conditions] on messages, constraints on instances, parameter values

- **Use cases**: pre/post conditions
Inheritance and constraints

- Liskov Substitution Principle
  - Whenever an instance of a class is expected, an instance of its subclass can be substituted.

- Implications
  - The invariant of a super-class is inherited by its subclasses. Hence a subclass may strengthen it, but not weaken it.
  - A precondition may be weakened, not strengthened, in the redefinition of an operation.
  - A postcondition may be strengthened, not weakened, in the redefinition of an operation.

References

- The Amsterdam Manifesto on OCL
  - In Object Modeling with the OCL (LNCS2263) p115-149
- Response to the UML 2.0 OCL RfP (ad/2000-09-03) Revised Submission, Version 1.6 January 6, 2003
- Some Shortcomings of OCL, the Object Constraint Language of UML
  - Mandana Vaziri and Daniel Jackson, 1999
- http://www.klasse.nl/english/uml/ UML CENTER
- Informal formality? The Object Constraint Language and its application in the UML metamodel
  - Anneke Kleppe, Jos Warmer, Steve Cook
- A Pratical Application of the Object Constraint Language OCL
  - Kjetil M’age
- The UML's Object Constraint Language: OCL Specifying Components, JAOO Tutorial – September 2000
  - Jos Warmer & Anneke Kleppe